On Higher Order Derivatives of Lyapunov Functions

Amir Ali Ahmadi and Pablo A. Parrilo

Abstract— **This note is concerned with a class of differential inequalities in the literature that involve higher order derivatives of Lyapunov functions and have been proposed to infer asymptotic stability of a dynamical system without requiring the first derivative of the Lyapunov function to be negative definite. We show that whenever a Lyapunov function satisfies these conditions, we can explicitly construct another (standard) Lyapunov function that is positive definite and has a negative definite first derivative. Our observation shows that a search for a standard Lyapunov function parameterized by higher order derivatives of the vector field is less conservative than the previously proposed conditions. Moreover, unlike the previous inequalities, the new inequality can be checked with a convex program. This is illustrated with an example where sum of squares optimization is used.**

I. Higher order derivatives of Lyapunov functions

Consider the dynamical system

$$\dot{x} = f(x), \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}^n$ has an equilibrium point at the origin (i.e., $f(0) = 0$), and satisfies the standard assumptions for existence and uniqueness of solutions; see e.g. [1, Chap. 3]. By higher order derivatives of a Lyapunov function $V(x) : \mathbb{R}^n \to \mathbb{R}$ we mean the time derivatives of V along the trajectories of (1) given by $\dot{V}(x) = \langle \frac{\partial V(x)}{\partial x}, f(x) \rangle, \ddot{V}(x) = \langle \frac{\partial \dot{V}(x)}{\partial x}, f(x) \rangle, \ldots, V^{(m)}(x) = \langle \frac{\partial V^{(m-1)}(x)}{\partial x}, f(x) \rangle$. In [2], Butz showed that existence of a three times continuously differentiable Lyapunov function $V(x)$ satisfying

$$\tau_2 \dddot{V}(x) + \tau_1 \ddot{V}(x) + \dot{V}(x) < 0 \tag{2}$$

for all $x \neq 0$ and for some nonnegative scalars τ_1, τ_2 implies global asymptotic stability of the origin of (1).[1] Note that unlike the standard condition $\dot{V}(x) < 0$, condition (2) is not jointly convex in the scalars τ_i and the parameters of the Lyapunov function $V(x)$. Therefore, computational techniques based on convex optimization cannot be used to search for a Lyapunov function satisfying (2). In [3], Heinen and Vidyasagar adapted the condition of Butz to establish a result on boundedness of the trajectories. More recently, Meigoli and Nikravesh [4], [5] have generalized the result of Butz to derivatives of higher order and to the case of time-varying systems. A simplified version of their result that is most relevant for our purposes deals with a differential inequality of the type

$$V^{(m)}(x) + \tau_{m-1} V^{(m-1)}(x) + \cdots + \tau_1 \dot{V}(x) < 0. \tag{3}$$

It is shown in [4] that if the corresponding characteristic polynomial

$$p(s) = s^m + \tau_{m-1} s^{m-1} + \cdots + \tau_1 s$$

is Hurwitz (and some additional standard assumptions hold), then the inequality in (3) proves global asymptotic stability. It is later shown in [5, Cor. 1] that this condition can be weakened to $p(s)$ having nonnegative coefficients. We will show that no matter what types of conditions on $V(x)$ and the scalars $\tau_{m-1}, \ldots, \tau_1$ are placed, if the system is globally asymptotically stable and the inequality (3) holds (which is in particular the case if inequality (3) is used to establish global asymptotic stability), then we can explicitly extract a standard Lyapunov function from it. This will follow as a corollary of the following simple and general fact.

Theorem 1.1: Consider a system $\dot{x} = f(x)$ that is known to be globally asymptotically stable. Suppose there exists a continuously differentiable function $W(x)$ whose derivative $\dot{W}(x)$ along the trajectories is negative definite and satisfies $W(0) = 0$. Then, $W(x)$ must be positive definite.

Proof: Assume by contradiction that there exists a nonzero point $\bar{x} \in \mathbb{R}^n$ such that $W(\bar{x}) \leq 0$. We evaluate the Lyapunov function $W(x)$ along the trajectory of the system starting from the initial condition $\bar{x}$. The value of the Lyapunov function is nonpositive to begin with and will strictly decrease because $\dot{W}(x) < 0$. Therefore, the value of the Lyapunov function can never become zero. On the other hand, since we know that the vector field is globally asymptotically stable, trajectories of the system must all go to the origin, where we have $W(0) = 0$. This gives us a contradiction. ■

Corollary 1.1: Consider a globally asymptotically stable dynamical system $\dot{x} = f(x)$. Suppose that the higher order differential inequality in (3) holds for some scalars $\tau_1, \ldots, \tau_{m-1}$ and for some m times continuously differentiable Lyapunov function $V(x)$ with $V(0) = 0$. Then,

$$W(x) = V^{(m-1)}(x) + \tau_{m-1} V^{(m-2)} + \cdots + \tau_2 \dot{V}(x) + \tau_1 V(x) \tag{4}$$

is continuously differentiable and positive definite and its derivative $\dot{W}(x)$ is negative definite.

Proof: Continuous differentiability of $W(x)$ and negative definiteness of $\dot{W}(x)$ follow from condition (3). Since $f(0) = 0$, we have that $\dot{V}^{(m-1)}(0) = \cdots = \dot{V}(0) = 0$. This together with the assumption that $V(0) = 0$ implies that $W(0) = 0$. Positive definiteness of $W(x)$ follows from Theorem 1.1. ■

Amir Ali Ahmadi and Pablo A. Parrilo are with the Laboratory for Information and Decision Systems, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. E-mail: a_a_a@mit.edu, parrilo@mit.edu.

[1] Throughout, when we state a condition involving $V^{(m)}(x)$, there is always the implicit assumption that the vector field $f(x)$ is $m-1$ times differentiable.

This corollary shows that instead of imposing the inequality in (3) along with conditions on $V(x)$ and the scalars $\tau_1, \ldots, \tau_{m-1}$ (such as positive definiteness of $V(x)$ and nonnegativity of τ_is) as proposed by the works in [2], [4], [5], we are better off imposing no conditions on $V(x)$ and the scalars τ_i individually but instead require

$$V^{(m-1)}(0)+\tau_{m-1}V^{(m-2)}(0)+\cdots+\tau_2\dot{V}(0)+\tau_1 V(0)=0, \quad (5)$$

and

$$V^{(m-1)}(x)+\tau_{m-1}V^{(m-2)}(x)+\cdots+\tau_2\dot{V}(x)+\tau_1 V(x)>0 \quad (6)$$

$$V^{(m)}(x)+\tau_{m-1}V^{(m-1)}(x)+\cdots+\tau_2\ddot{V}(x)+\tau_1 \dot{V}(x)<0 \quad (7)$$

for all $x \neq 0$. In other words, we simply impose the standard Lyapunov conditions on a Lyapunov function of the specific structure in (4).[2] By Corollary 1.1, the latter approach is always less conservative than the former.

Now to get around the issue of non-convexity of inequalities (5)-(7) in the decision variables τ_i and the parameters of $V(x)$, we can simply search for different functions $V_1(x), \ldots, V_m(x)$ (with no sign conditions on them individually), such that

$$V_m^{(m-1)}(0)+V_{m-1}^{(m-2)}(0)+\cdots+\dot{V}_2(0)+V_1(0)=0 \quad (8)$$

and

$$V_m^{(m-1)}(x)+V_{m-1}^{(m-2)}(x)+\cdots+\dot{V}_2(x)+V_1(x)>0 \quad (9)$$

$$V_m^{(m)}(x)+V_{m-1}^{(m-1)}(x)+\cdots+\ddot{V}_2(x)+\dot{V}_1(x)<0 \quad (10)$$

for all $x \neq 0$. These three conditions are convex and it should be clear that if conditions (5)-(7) are satisfied for some function $V(x)$ and some scalars τ_i, then conditions (8)-(10) are satisfied with $V_i = \tau_i V$ for $i = 1, \ldots, m-1$ and $V_m = V$.

We refer the reader to [6], [7] for more discussion and also for a discrete time analogue of these results.

II. An example

The following example shows the potential advantages of using higher order derivatives of Lyapunov functions. It also demonstrates the use of convex optimization for imposing constraints of the type (8)-(10). We assume the reader is familiar with the sum of squares (sos) relaxation of polynomial nonnegativity and its formulation as a linear matrix inequality. See [8], [9].

Example 2.1: Consider the following polynomial dynamics

$$\begin{array}{rcl} \dot{x_1} & = & -0.8x_1^3-1.5x_1x_2^2-0.4x_1x_2-0.4x_1x_3^2-1.1x_1 \\ \dot{x_2} & = & x_1^4+x_3^6+x_1^2x_3^4 \\ \dot{x_3} & = & -0.2x_1^2x_3-0.7x_2^2x_3-0.3x_2x_3-0.5x_3^3-0.5x_3. \end{array}$$

If we use SOSTOOLS [10] to search for a standard quadratic Lyapunov function $V(x)$ that is a sum of squares and for which $-\dot{V}(x)$ is a sum of squares, the search will be infeasible. If needed, this can be turned into a proof (using duality of semidefinite programming) that no such Lyapunov function exists. Instead, we search for $V_1(x)$ and $V_2(x)$ such that $\dot{V}_2(0)+V_1(0)=0$, and $\dot{V}_2(x)+V_1(x)$ and $-(\ddot{V}_2(x)+\dot{V}_1(x))$ are both sums of squares.[3] In principle, we can start with a linear parametrization for $V_1(x)$ and $V_2(x)$ since there is no positivity constraint on them directly. For this example, a linear parametrization will be infeasible. However, if we search for a linear function $V_2(x)$ and a quadratic function $V_1(x)$, SOSTOOLS and the semidefinite programming solver SeDuMi [11] find

$$\begin{array}{rcl} V_1(x) & = & 0.47x_1^2+0.89x_2^2+0.91x_3^2 \\ V_2(x) & = & 0.36x_2. \end{array}$$

Therefore, the origin is asymptotically stable. The standard Lyapunov function constructed from $V_1(x)$ and $V_2(x)$ will be the following sextic polynomial:

$$\begin{array}{l} W(x)=\dot{V}_2(x)+V_1(x)= \\ 0.36x_1^4+0.36x_1^2x_3^4+0.47x_1^2+0.89x_2^2+0.36x_3^6+0.91x_3^2. \end{array}$$

It is easy to see that $W(x)-\frac{1}{4}(x_1^2+x_2^2+x_3^2)$ is a sum of squares. This confirms positivity of $W(x)$ and also shows that it is radially unbounded. Therefore, the origin is in fact globally asymptotically stable.

One could of course forget about higher order derivatives all together and directly search for a standard degree six polynomial Lyapunov function using SOSTOOLS. A simple calculation shows however that this search would have had 68 more decision variables than our search for the quadratic and linear functions $V_1(x)$ and $V_2(x)$.

References

[1] H. Khalil. *Nonlinear systems.* Prentice Hall, 2002.

[2] A. R. Butz. Higher order derivatives of Liapunov functions. *IEEE Trans. Automatic Control*, AC-14:111–112, 1969.

[3] J. A. Heinen and M. Vidyasagar. Lagrange stability and higher order derivatives of Liapunov functions. *IEEE Trans. Automatic Control*, 58(7):1174, 1970.

[4] V. Meigoli and S. K. Y. Nikravesh. A new theorem on higher order derivatives of Lyapunov functions. *ISA Transactions*, 48:173–179, 2009.

[5] V. Meigoli and S. K. Y. Nikravesh. Applications of higher order derivatives of Lyapunov functions in stability analysis of nonlinear homogeneous systems. In *Proceedings of the International Multi Conference of Engineers and Computer Scientists*, 2009.

[6] A. A. Ahmadi and P. A. Parrilo. Non-monotonic Lyapunov functions for stability of discrete time nonlinear and switched systems. In *IEEE Conference on Decision and Control*, 2008.

[7] A. A. Ahmadi. Non-monotonic Lyapunov functions for stability of nonlinear and switched systems: theory and computation. Master's Thesis, Massachusetts Institute of Technology, June 2008. Available from `http://dspace.mit.edu/handle/1721.1/44206`.

[8] P. A. Parrilo. *Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization*. PhD thesis, California Institute of Technology, May 2000.

[9] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Math. Prog.*, 96(2, Ser. B):293–320, 2003.

[10] S. Prajna, A. Papachristodoulou, and P. A. Parrilo. *SOSTOOLS: Sum of squares optimization toolbox for MATLAB*, 2002-05. Available from `http://www.cds.caltech.edu/sostools` and `http://www.mit.edu/~parrilo/sostools`.

[11] J. Sturm. *SeDuMi version 1.05*, October 2001. Latest version available at `http://sedumi.ie.lehigh.edu/`.

[2] We can see from (4) that $W(x)$ is a Lyapunov function that has the vector field $f(x)$ and its derivatives in its parametrization. This is in some sense reminiscent of Krasovskii's method, where the vector field $f(x)$ is used in the parametrization of a Lyapunov function. See e.g. [1, p. 183].

[3] It is easy to check whether a sum of squares polynomial is strictly positive as opposed to merely nonnegative. This is the case throughout this example.

Higher-order Recursive Types

Working Notes, 21st March 2002

Gert Smolka

We are interested in a lambda calculus whose expressions describe types that are possibly higher-order and recursive. Types are modelled as trees, which can be seen as possibly infinite, normal lambda expressions in de Bruijn's notation.

In general, equivalence of type expressions is undecidable. We are interested in subclasses for which equivalence is efficiently decidable. We define a subclass whose equivalence problem can be solved in $O(n \log n)$ time.

1 Trees and Graphs

As usual, we define a *tree* as a function from a tree domain (possibly infinite) to a set of so-called labels. We consider nonempty, ordered trees whose domains are prefix-closed subsets of $\mathbb{N}^*$. A tree is called *regular* if it has only finitely many distinct subtrees.

Trees can be described by *graphs* whose edges are labeled with natural numbers. Each node of such a graph describes exactly one tree. The graph itself describes a set of trees that is closed under taking subtrees.

A graph together with one of its nodes can be seen as a kind of automaton, which accepts the elements of the tree the node (acting as initial state) describes (up to details concerning the final states).

The trees described by a finite graph are all regular. For every regular tree t there is a finite graph whose nodes describe exactly the subtrees of t.

A graph is *minimal* if different nodes of the graph describe different trees. Minimal graphs are unique up to renaming of nodes; that is, for a finite set of regular trees there is one and only one minimal graph representing it (up to renaming of nodes). A finite graph can be minimized in $O(n \log n)$ time by an algorithm known as graph partioning (originally devised by Hopcroft (1971) for minimizing automata).

2 Rho Calculus

Regular trees can be described by so-called *μ-expressions* (e.g., $\mu x.f(a,x)$). We define the *rho calculus* as the system that, given a set of labels $f \in F$ and a set of variables $x \in X$, consists of the expressions

$$M \;=\; fM_1 \dots M_n \mid x \mid \mu x.M \quad (M \text{ not a variable})$$

and the reduction rule

$$(\mu) \quad \mu x.M \;\rightarrow\; M[\mu x.M \,/\, x]$$

We need to define which tree is denoted by an expression (given an environment E mapping its free variables to trees). This can be done by first defining inductively an *acceptance relation*

$$(\pi, u), E \vdash M$$

and then defining the *denotation function* as

$$[\![M]\!]E = \{(\pi, u) \mid (\pi, u), E \vdash M\}$$

The definition of the acceptance relations involves μ-reductions that are applied when needed:

$$\frac{(\pi, u), E \vdash M[\mu x.M \,/\, x]}{(\pi, u), E \vdash \mu x.M}$$

The equation corresponding to the μ-rule is not a complete characterisation of the μ-operator. For instance, it fails to establish the equivalence of the expressions $\mu x.f(gx)$ and $f(\mu x.g(fx))$.

There is the minor complication that certain μ-expressions do not describe trees, for instance, $\mu x.x$ and $f(\mu x.x)$. This degenerate μ-expressions can be excluded by a straightforward syntactic condition, which requires that the M in $\mu x.M$ is not a variable.[1]

The natural computer representation for regular trees are graphs. Expressions are used as a convenient representation for input and output. In contrast to trees, graphs can

[1] Degenerate expressions could be taken as descriptions of the empty tree. However, excluding the empty tree and degenerate expressions seems to save a lot of complications.

express structure sharing. Minimal graphs are graphs with maximal structure sharing. Expressions with a let construct can also express structure sharing.

Chapter 21 of Benjamin Pierces's book [2002] studies a variant of the rho calculus.

Problem 1 *Define the acceptance relation for the rho calculus.*

Problem 2 *Show that the equivalence problem of the rho calculus can be decided in $O(n \log n)$ time by the graph partitioning algorithm mentioned in the previous section.*

Problem 3 *Define a denotational semantics for the rho calculus (i.e., a denotation function defined by structural induction on expressions).*

3 Lambda Mu Calculus

The canonical framework for the description of higher-order recursive objects is the simply typed lambda calculus with fixed point combinators:

$$\mathit{fix}_\sigma \colon (\sigma \to \sigma) \to \sigma$$
$$(\mathit{fix}) \quad \mathit{fix}_\sigma = \lambda f \colon \sigma \to \sigma.\ f(\mathit{fix}_\sigma f)$$

We assume the usual axioms (α), (β) and (η).

The μ-operator of the rho calculus can be expressed with fixed point combinators:

$$\mu x \colon \sigma.M \overset{\text{def}}{=} \mathit{fix}_\sigma(\lambda x \colon \sigma.M)$$

Moreover, the μ-rule from the rho calculus

$$(\mu) \quad \mu x \colon \sigma.M \ \to\ M[\mu x \colon \sigma.M \,/\, x]$$

can be simulated by one application of the fix-rule and two applications of the β-rule. On the other hand, the fixed point combinators can be expressed with the μ-operator:

$$\mathit{fix}_\sigma = \lambda f \colon \sigma \to \sigma.\ f(\mu x \colon \sigma.fx)$$

Moreover, from the equations corresponding to the μ-rule we can derive the axioms for the fixed point combinators. We conclude that the simply typed lambda fix calculus has the same expressivity and the same axiomatic equivalence as the simply typed *lambda mu calculus*, where the fixed point operators are replaced by the μ-operator. We prefer to work with the lambda mu calculus since it facilitates the formulation of constrained versions of recursion.

It is well-known that reduction in the simply typed lambda fix calculus is confluent and that *lambda reduction* (i.e., reduction with the β- and η-rule) is terminating (see, e.g., [Mitchell]). Hence equivalence of expressions not containing a fixed point operator is decidable. In general, equivalence of expressions in the lambda fix calculus is undecidable.

It is easy to see that the termination result for lambda reduction carries over to the lambda mu calculus. We also expect that the lambda mu calculus is confluent, but the proof of this result is not obvious since reduction in the lambda fix calculus cannot be fully simulated in the lambda mu calculus. Ariola and Klop [LICS 1994, Section 8] consider an untyped lambda mu calculus *without the η-rule*, which is confluent since it is an orthogonal, combinatory reduction system.

Problem 4 *Show that the lambda mu calculus is confluent.*

A head normal form is an expression that has the property that no reduction rule applies at the top level, even after preparatory reduction steps at lower levels. The set of *head normal forms* is defined inductively as follows:

1. $M_0 \dots M_n$ is a head normal form if M_0 is a constant or a variable.
2. $\lambda x.M_0 \dots M_n$ is a head normal form if M_0 is a constant or a variable and $M_n \neq x$.
3. $\lambda x.\lambda y.M$ is a head normal form if $\lambda y.M$ is a head normal form.

We say that M' is a *head normal form of M* if M' is a head normal form and $M \rightarrow^* M'$.

Problem 5 *Suppose M and M' are head normal forms of the same expression. Then either M and M' are both lambda abstractions, or there exist $n \in \mathbb{N}$ and expressions F, $M_1, \dots, M_n$ and $M'_1, \dots, M'_n$ such that $M = FM_1 \dots M_n$, $M' = FM'_1 \dots M'_n$, and F is a constant or a variable.*

Problem 6 *Show that the existence of head normal forms is undecidable.*

4 Tau Calculus and Lambda Trees

The *tau calculus* is the specialized simply typed lambda mu calculus that has only one base type $*$. For the expressions of the tau calculus we define an equivalence that is weaker than the canonical equivalence (defined by the axioms).[2] The need for weak equivalence was already motivated in the context of the rho calculus; for instance, we would like that the expressions $\mu x.f(gx)$ and $f(\mu x.g(fx))$, which are not equivalent,

[2] Weaker means, that expression that are equivalent are also weakly equivalent, and that weakly equivalent expressions are not necessarily equivalent.

be weakly equivalent. For μ-free expressions, *weak equivalence* should coincide with canonical equivalence.

We will define weak equivalence for the tau calculus in a similar way as it was done for the rho calculus. This time we take so-called *lambda trees* as denotations of expressions.

Finite lambda trees correspond to closed and normal expressions, where variable bindings are represented with de Bruijn's notation.[3] For instance, the expression

$$\lambda x.\lambda y.\lambda z.(xz)(yz)$$

corresponds to the lambda tree

$$\lambda(\lambda(\lambda(2@0)(1@0)))$$

where the labels λ and @ (written in infix notation) represent lambda abstraction and application. Nodes labeled with de Bruijn indices (i.e., natural numbers) are called *alpha nodes*. Constants appear as leaves of lambda trees.

Infinite lambda trees can be seen as infinite, closed, and normal expressions.

If an expression is in head normal form, its top level corresponds exactly to the top level of the denoted lambda tree.

A *convergence property* is a set S of expressions as follows:

1. S is closed under reduction.
2. S is closed under taking subexpressions.
3. Every expression in S has a head normal form.

The union of convergence properties is again a convergence property. Hence there exists a largest convergence property. An expression is called *convergent* if and only if it is an element of the largest convergence property.

Problem 7 *Show that convergence is undecidable.*

Nonconvergent expressions in the tau calculus correspond to degenerate expressions in the rho calculus. We will define weak equivalence only for convergent expressions.

Intuitively, convergent expressions have possibly infinite normal forms. We express this intuition by defining an acceptance relation

$$(\pi, u), E \vdash M$$

that relates lambda trees and convergent expressions.

[3] Note that a normal expression cannot contain the μ-operator.

Problem 8 *Define the acceptance relation for convergent expressions.*

5 Higher-order and Recursive Types

We model higher-order recursive types as possibly infinite lambda trees, and we use the tau calculus for describing them.[4] To avoid confusion, the simple types of the tau calculus are called *kinds*. There is only one base kind $*$. Types that can be described by closed expressions of kind $*$ are called *proper types*. Types that can be described by closed expressions of a functional kind are called *higher-order types*.

An expression is called *lambda normal* if it cannot be reduced with the β- or η-rule. An expression is called *weakly lambda normal* if it is lambda normal and lambda normality is preserved by iterated μ-reduction.

Consider the following higher-order type expressions:

$$\begin{aligned} A &= \lambda x:*.\ \mu y:*.\ 1+(x\times y) \\ B &= \mu f:*\to *.\ \lambda x:*.\ 1+(x\times fx) \end{aligned}$$

Both expressions describe the same higher-order type (a function mapping a type x to the type of lists over x). We consider the expressions $A1$ and $B1$, which describe the proper type of list over 1. A single β-reduction

$$\begin{aligned} A1 &\overset{\beta}{\to} \mu y:*.\ 1+(1\times y) \\ &\overset{\mu}{\to} 1+(1\times(\mu y:*.\ 1+(1\times y))) \\ &\overset{\beta}{\leftarrow} 1+(1\times A1) \end{aligned}$$

yields a weakly lambda normal expression that describes the proper type described by $A1$. The situation is more complicated for $B1$. We need a μ-reduction followed by a β-reduction

$$\begin{aligned} B1 &\overset{\mu}{\to} (\lambda x:*.\ 1+(x\times Bx))\ 1 \\ &\overset{\beta}{\to} 1+(1\times B1) \end{aligned}$$

[4] We follow the so-called *equi-recursive approach* to recursive types, which treats recursive types exactly as one would expect from the statement that a recursive type is an infinite tree. The rho calculus can be used to describe regular recursive types. In contrast, the *iso-recursive approach* to recursive types is a formal trick that takes a recursive type and its unfolding as different (see Mitchell's book, for instance). In the iso-recursive approach the two type expressions $\mu x.1+x$ and $1+(\mu x.1+x)$ describe different types; in the equi-recursive approach they describe the same type.

to unravel the top level of the described type. The lambda-abstraction is not removed, and we need infinitely many β-reductions to unravel the described type completely (which is a regular tree).

The tau calculus can describe nonregular types. For instance:

$$\begin{aligned} C &= \mu f : * \to *.\ \lambda x : *.\ 1 + (x \times f(x + x)) \\ C1 &= 1 + (1 \times C(1 + 1)) \\ &= 1 + (1 \times (1 + ((1 + 1) \times C((1 + 1) + (1 + 1))))) \end{aligned}$$

In languages with polymorphic recursion (not SML) there are useful applications of nonregular types (see Okasaki's book on functional data structures).

6 Decidability of Type Expression Equivalence

We are interested in formal systems where well-typedness is a decidable property. For this to be the case, equivalence of expressions describing types must be decidable.

Marvin Solomon [POPL 1978] has shown that equivalence is decidable for type expressions with first-order recursion (i.e., μ_σ occurs only with kinds $\sigma = * \to \cdots \to *$) and parameters of the base kind $*$. Solomon shows that the problem is equivalent to the deterministic push-down automata equivalence problem (which at the time was still open but has now been settled as decidable [Sénizergues, ICALP 1997]).

It seems that equivalence of expressions that use recursion only for the base kind (i.e., contain μ_σ only with kind $\sigma = *$) is decidable.

We are interested in a subclass of expressions for which equivalence is decidable. The first restriction, called *BR (basic recursion)*, requires that all μ-variables have kind $*$. BR yields that μ-reduction preserves β-normality (normality with respect to β-reduction).

It seems that equivalence of expressions restricted to BR is decidable.

We can now λ-normalize an expression as follows: First, we normalize with respect to β and η. Then we apply μ-reduction to a bottom-most subexpression

$$\lambda x.\ \mu y.\ M \qquad \text{where } x \text{ free in } M$$

and try to apply η-reduction. After finitely many steps we reach a λ-normal expression, for which μ-reduction preserves λ-normality.

7 Definition of Lambda Trees

Lambda trees are trees with variable bindings. Finite lambda trees correspond exactly to lambda terms in de Bruijn's Notation that are normal with respect to β- and η-reduction. We are interested in infinite lambda trees since they may serve as a semantics for recursive and higher-order types.

We assume a set X and two distinct objects @ and λ such that X, $\{@, \lambda\}$ and the set of positive integers are pairwise disjoint.

A *lambda tree* over X is a tree as follows:

1. Each node is labeled with one of the following: an element of X, a positive integer, @ or λ. Nodes labeled with @ are called *application nodes*, nodes labeled with λ are called λ-*nodes*, and nodes labeled with a positive integer are called α-*nodes*. Alpha nodes represent bound or unbound variables.

2. Nodes labeled with elements of X or positive integers must be leaves.

3. Application nodes must have exactly two and lambda nodes must have exactly one successor.

4. There are neither β- nor η-redexes.

5. There is a fixed $k \in \mathbb{N}$ such that every α-node labeled with n is below of at least $n - k$ different λ-nodes.

Regular trees always satisfy condition (5).

It is easy to see that subtrees of lambda trees are lambda trees.

An α-node labeled with n is bound by the n'th λ-node that is above it. This minor derivation from the de Bruijn Notation is convenient for the following definition.

The *degree* of a lambda tree is the least k such that condition (5) is satisfied. A lambda tree of degree 0 contains no unbound variable and is called *closed*. A lambda tree of positive degree contains unbound variables and is called *open*.

15-150 Lecture 11:
Higher-Order Functions

Lecture by Dan Licata

Februrary 16, 2012

1 Functions as Arguments

Next, we're going to talk about the thing that makes functional programming *functional*:

> functions are values that can be passed to, and returned from, other functions

Consider the following two functions:

```
fun double (x : int) : int = x * 2
fun doubAll (l : int list) : int list =
    case l of
        [] => []
      | x :: xs => double x :: doubAll xs

fun raiseBy (l : int list , c : int) : int list =
    case l of
        [] => []
      | x :: xs => (x + c) :: raiseBy (xs,c)
```

How do they differ? They both do something to each element of a list, but they do different things (adding one, multiplying by c).

We want to write one function that expresses the pattern that is common to both of them:

```
fun map (f : int -> int , l : int list) : int list =
    case l of
        [] => []
      | x :: xs => f x :: map (f , xs)
```

The idea with map is that it takes a function f : int -> int as an argument, which represents what you are supposed to do to each element of the list.

int -> int is a type and can be used just like any other type in ML. In particular, a function like map can take an argument of type int -> int; and, as we'll see next time, a function can return a function as a result.

For example, we can recover doubAll like this:

```
fun doubAll l = map (double , l)
```

If you substitute add1 into the body of map, you see that it results in basically the same code as before.

Anonymous functions Another way to instantiate `map` is with an anonymous function, which is written `fn x => 2 * x`:

```
fun doubAll l = map (fn x => 2 * x , l)
```

The idea is that `fn x => 2 * x` has type `int -> int` because assuming `x:int`, the body `2 * x : int`. To evaluate an anonymous function applied to an argument, you plug the value of the argument in for the variable. E.g. `(fn x => 2 * x) 3 |-> 2 * 3`. **Functions are values:** The value of `fn x => x + (1 + 1)` is `fn x => x + (1 + 1)`. You don't evaluate the body until you apply the function.

`doubAll` can be defined anonymously too:

```
val doubAll : int list -> int list = fn l => map (fn x => 2 * x , l)
```

Closures A somewhat tricky, but very very useful, fact is that anonymous functions can refer to variables bound in the enclosing scope. This gets used when we instantiate `map` to get `raiseBy`:

```
fun raiseBy (l , c) = map (fn x => x + c , l)
```

The function `fn x => x + c` adds `c` to its argument, where `c` bound as the argument to `raiseBy`. For example, in

```
    raiseBy ( [1,2,3] , 2)
|-> map (fn x => x + 2 , [1,2,3])
|-> (fn x => x + 2) 1 :: map (fn x => x + 2 , [2,3])
|-> 1 + 2 :: map (fn x => x + 2 , [2,3])
|-> 3 :: map (fn x => x + 2 , [2,3])
|-> 3 :: (fn x => x + 2) 2 :: map (fn x => x + 2 , [3])
 == 3 :: 4 :: map (fn x => x + 2 , [3])
 == 3 :: 4 :: (fn x => x + 2) 3 :: map (fn x => x + 2 , [])
 == 3 :: 4 :: 5 :: map (fn x => x + 2 , [])
 == 3 :: 4 :: 5 :: []
```

the `c` gets specialized to `2`. If you keep stepping, the function `fn x => x + 2` gets applied to each element of `[1,2,3]`. The important fact, which takes some getting used to, is that the function `fn x => x + 2` is *dynamically generated*: at run-time, we make up new functions, which do not appear anywhere in the program's source code!

Here's a puzzle: what does

```
let val x = 3
    val f = fn y => y + x
    val x = 5
 in
    f 10
 end
```

evaluate to?

Well, you know how to evaluate `let`: you evaluate the declarations in order, substituting as you go. So, you get

```
let val x = 3
    val f = fn y => y + 3
    val x = 5
 in
    f 10
 end
```

The fact that `x` is shadowed below is irrelevant; the result is `13`. This is one of the reasons why we've been teaching you the substitution model of evaluation all semester; it explains tricky puzzles like this in a natural way.

Polymorphism Another instance of the same pattern is:

```
fun showAll (l : int list) : string list =
    case l of
        [] => []
      | x :: xs => Int.toString x :: showAll xs
```

However, this is not an instance of `map`, because it transforms an `int list` into a `string list`.

We can fix this by giving `map` a polymorphic type:

```
fun map (f : 'a -> 'b , l : 'a list ) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map (f , xs)
```

Then both `doubAll` and `showAll` are instances:

```
fun doubAll l = map (fn x => 2 * x , l)
fun showAll l = map (Int.toString , l)
```

`map` is an example of what is called a *higher-order function*, a function that takes a function as an argument.

2 Exists

Here's another example of a higher-order function, which captures the pattern of "is there some element of a list such that ...":

```
fun evenP (x : int) : bool = (x mod 2) = 0

fun hasEven (l : int list) : bool =
    case l of
        [] => false
      | x :: xs => evenP x orelse hasEven xs

fun doctor (l : string list) : bool =
    case l of
        [] => false
      | x :: xs => (x = "doctor") orelse doctor xs

fun exists (p : 'a -> bool, l : 'a list) : bool =
    case l of
        [] => false
      | x :: xs => p x orelse exists (p, xs)

fun hasEven (l : int list) : bool = exists(evenP, l)
fun doctor (l : string list) : bool = exists((fn x => x = "doctor"), l)
```

3 Currying

Currying is the idea that a function with two arguments is roughly the same thing as a function with one argument, that returns a function as a result. For example, here is the curried version of `map`:

```
fun map (f : 'a -> 'b) : 'a list -> 'b list =
    fn l =>
     case l of
         [] => []
       | x :: xs => f x :: map f xs
```

Instead of taking two arguments, a function and a list, it takes one argument (a function), and returns a function that takes the list and computes the result. Note that function application left-associates, so `f x y` gets parsed as `(f x) y`. Thus, a curried function can be applied to multiple arguments just by listing them in sequence, as in `map f xs` above. This is really two function applications: one of `map` to `f`, producing `map f : 'a list -> 'b list`; and one of `map f` to `xs`.

Thus far, currying may seem undermotivated: is it just a trivial syntactic thing? We will eventually discuss several advantages of curried functions:

- Today: It is easy to *partially apply* a curried function, e.g. you can write `map f`, rather than `fn l => map (f,l)`, when you want the function that maps `f` across a list. This will come up when you want to pass `map f` as an argument to another higher-order function, or when you want to compose it with other functions, as we will see below. This is a difference in readability, but not functionality.

- Another motivation for currying is that it lets you write programs that you wouldn't otherwise be able to write. But we won't get to this for a couple of lectures, when we talk about something called *staged computation*. Another example is *function-local state*, but we won't get to that until much later. The general idea is that currying lets you do real computation between the arguments to a function.

Because of this, there is special syntax for curried functions, where you write multiple arguments to a function in sequence, with spaces between them:

```
fun map (f : 'a -> 'b) (l : 'a list) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map f xs
```

This means the same as the explicit `fn` binding in the body, as above.

3.1 Fusion

If you have two maps in succession, this transforms a list, and then immediately transforms the resulting list again (and does nothing else with it). Thus, you might as well just compute the third tree right away, in one pass. This saves time (one traversal instead of two) and space (no need to create and then throw away the second tree).

In general, when `f` and `g` are total, we have

```
map f o map g == map (f o g)
```

which is called *fusion*. You'll look at the proof of this in HW. It is a generalization of the theorem about `raiseBy` from the lecture introducing lists.

4 Reduce

Consider the following two functions:

```
     fun sum (l : int list) : int =
        case l of
             [] => 0
           | x :: xs => x + (sum xs)
     fun join (l : string list) : string =
         case l of
             [] => ""
           | x :: xs => x ^ join xs
```

The pattern is "give some answer for the empty list, and for a cons, somehow combine the first element with the recursive call on the rest of the list." Here's is a curried version of `reduce` from lab:

```
fun reduce (c : 'a * 'a -> 'a) (n : 'a) (l : 'a list) : 'a =
    case l of
        [] => n
      | x :: xs => c (x , reduce c n xs)

fun sum (l : int list) : int = reduce (fn (x,y) => x + y) 0 l
fun join (l : string list) : string = reduce (fn (x,y) => x ^ y) "" l
```

5 Functional Decomposition of Problems

In functional programming, one way to think about problems is to decompose them into a series of tasks, represented as functions, and to solve the problem by composing these functions together.

Function composition is a builtin `o`, which is defined as follows:

```
fun (g : 'b -> 'c) o (f : 'a -> 'b) : 'a -> 'c =
  fn x => g (f x)
```

Function composition makes things more succinct. For example, the theorem on HW3 about `zip` and `unzip` is saying that `unzip o zip` $\cong$ `fn x => x`.

5.1 wordcount and longestline

For example, let's count all the words in a string. We can represent this as three tasks:

Start with a string

```
"hi there"
```

First, divide it up into a list of words:

```
["hi","there"]
```

Second replace each element with `1`

```
[1,1]
```

Third, sum up the list:

```
+ (1, 1)
```

Let's assume a function `words : string -> string list` for the first task. We can implement the second using `map (fn _ => 1) : string list -> int list`. And we defined `sum` above for the third.

Thus, we can implement

```
val wordcount : string -> int = sum o map (fn _ => 1) o words
```

by composing these tasks together.

Suppose we want to know the length of the longest line in a file:

```
val longestlinelength : string -> int =
   reduce Int.max 0  o  map wordcount  o  lines
```

Divide the file into lines, compute the number of words in each, and then take the max.

5.2 Stocks

Functional programming makes it easy to express computations as compositions of tasks, where the tasks themselves can often be expressed concisely using higher-order functions like `map` and `reduce`. This leads to short and elegant solutions to problems.

Suppose we have the prices for stocks for a stock over time:

Day 1	Day 2	Day 3	Day 4
$20	$25	$24	$30

Let's write a function `bestGain : int list -> int` that computes the best profit you could have made on the stock, were you to buy and sell on any two days. Here's the algorithm, on the above example:

1. Form pairs (*buy*, *sells*), where *buy* is a price you could have bought at, and *sells* is the prices you could then sell at:

   ```
   (20, [25,24,30])
   (25, [24,30])
   (24, [30])
   (30, [])
   ```

We can do this by pairing each price with the suffix of the data after that point.

2. For each pair (*buy*, *sells*), for each element *sell* of *sells*, compute the difference *sell* − *buy*:

   ```
   [5,4,10]
   [~1,5]
   [6]
   []
   ```

3. Take the max of all of these. In this case, the answer is 10.

Note that the output of each step is the input to the subsequent step. This means we can implement this algorithm as a composition of three functions, one for each step.

We use the following helper functions:

```
(* See homework 3 *)
val zip : ('a list * 'b list) -> ('a * 'b) list

(* compute a list whose elements are all the suffixes of the input *)
val suffixes : 'a list -> ('a list) list
```

For step (1), we implement a function `withSuffixes` that pairs each price with the suffix of the list after that price.

```
fun withSuffixes (t : int list) : (int * int list) list = zip (t, suffixes t)
```

For step (3), we want to take the max of all of the `int`'s in a list of lists. This should be familiar from Lecture 1:

```
val maxL : int list -> int = reduce Int.max minint
val maxAll : (int list) list -> int = maxL o map maxL
```

Note that `minint` is an `int` that is ≤ all other `int`'s.

Now we can implement `bestGain` as a composition of three functions:

```
val bestGain : int list -> int =
      maxAll                                                   (* step 3 *)
    o (map (fn (buy,sells) => (map (fn sell => sell - buy) sells))) (* step 2 *)
    o withSuffixes                                             (* step 1 *)
```

For step (1), we use `withSuffixes`.

For step (2), the input is the `(int * int tree) tree` resulting from step (1). We use a `map` to do something to each pair `(buy,sells)` (e.g. the pairs `(20, <25,24,30>)` and `(25, <24,30>)` . . . in the above example). Note that this outer map is partially applied, because the argument to `o` is a function. The body of the outer `map` is again a call to `map`, time over `sells`, to do something to each `sell` price—in particular, to compute `sell - buy`. This leaves us with an `(int tree) tree`, whose leaves are all of the `sell - buy` differences.

For step (3), we use `maxAll`.

In case currying is confusing you, here is what the above example looks like if we uncurry everything:

```
fun map (f : 'a -> 'b, l : 'a list) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map (f,xs)

fun reduce (c : 'a * 'a -> 'a, n : 'a, l : 'a list) : 'a =
    case l of
        [] => n
      | x :: xs => c (x , reduce (c, n, xs))

fun maxL (l : int list) : int = reduce (Int.max , minint , l)
fun maxAll (l : (int list) list) : int = maxL (map (maxL, l))

fun withSuffixes (l : int list) : (int * int list) list =
    zip (l, suffixes l)

fun bestGain (l : int list) : int =
    maxAll (map (fn (buy,sells) => (map (fn sell => sell - buy, sells)),
                 withSuffixes l))
```

6 Functions as Data

Now that we know that functions can be returned as results from other functions, we can start to put this idea to use. For example, returning to the polynomial example from before, we can represent a polynomial

```
c0 + c1 x + c2 x^2 + ... cn x^n
```

How can we represent such a table? As a function that maps each exponent to its coefficient:

```
   fn x => case x of
      0 => c0
    | 1 => c1
    | 2 => c2
    | ...
    | n => cn
    | _ => 0
```

This carries the same information as the coefficient list representation, but it also scales to series, which may have infinitely many terms.

Here's an example polynomial:

```
type poly = int -> int

(* x^2 + 2x + 1 *)
val example : poly =
     fn 0 => 1
      | 1 => 2
      | 2 => 1
      | _ => 0
```

The function that adds two polynomials takes two functions as input, and produces a function as output:

```
fun add (p1 : poly, p2 : poly) : poly = fn e => p1 e + p2 e
```

This is like Jeopardy: the answer of `add(p1,p2)` has the form of a question—what exponent would you like the coefficient of? In the case of addition, the answer is that the coefficient of `e` is the sum of the coefficients in the summand. Note that the function we return refers to the arguments `p1` and `p2`—this is called a *closure*, and we say that the function *closes over* `p1` and `p2`. When you call `add`, you generate a new function that mentions whatever polynomials you want to sum.

For multiplication, we need to do a *convolution*: the coefficient of e is

$$\sum_{i=0}^{e} c_i d_{e-i}$$

We render this in SML by writing a simple local, recursive helper function, which loops from `e` to `0`:

```
fun mult (c , d) =
    fn e => let
                fun convolution i =
                    case i of
                        ~1 => 0
                      | _ => (c i) * (d (e - i)) + convolution (i - 1)
            in
                convolution e
            end
```

More generally, we could represent a dictionary mapping `key`s to `value`s as functions `key -> value`.

Focusing and Higher-Order Abstract Syntax

Noam Zeilberger
Carnegie Mellon University
noam@cs.cmu.edu

Abstract

Focusing is a proof-search strategy, originating in linear logic, that elegantly eliminates inessential nondeterminism, with one byproduct being a correspondence between focusing proofs and programs with explicit evaluation order. Higher-order abstract syntax (HOAS) is a technique for representing higher-order programming language constructs (e.g., λ's) by higher-order terms at the "meta-level", thereby avoiding some of the bureaucratic headaches of first-order representations (e.g., capture-avoiding substitution).

This paper begins with a fresh, judgmental analysis of focusing for intuitionistic logic (with a full suite of propositional connectives), recasting the "derived rules" of focusing as *iterated inductive definitions.* This leads to a uniform presentation, allowing concise, modular proofs of the identity and cut principles. Then we show how this formulation of focusing induces, through the Curry-Howard isomorphism, a new kind of higher-order encoding of abstract syntax: functions are encoded by maps from *patterns* to expressions. Dually, values are encoded as patterns together with *explicit substitutions.* This gives us pattern-matching "for free", and lets us reason about a rich type system with minimal syntactic overhead. We describe how to translate the language and proof of type safety almost directly into Coq using HOAS, and finally, show how the system's modular design pays off in enabling a very simple extension with recursion and recursive types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.4.1 [*Theory of Computation*]: Mathematical Logic—*Lambda calculus and related systems*

General Terms Languages

1. Introduction

The end result of this paper will be to show how so-called *focusing* proofs produce—through a careful judgmental analysis and the Curry-Howard isomorphism—an exceptionally compact presentation of a call-by-value language with a full suite of types. In the process, we hope to convince the reader of an aphorism: abstract syntax should be even *more* abstract.

The technique of focusing was originally invented by Andreoli (1992) as a refinement of bottom-up proof search in linear logic, to reduce an otherwise intractable amount of nondeterminism. Soon afterwards, it was promoted by Girard (1993) as a conceptual tool for finding unity in logic, as it turned out that also the classical and intuitionistic connectives could be classified by their focusing behavior, or *polarity*. Recently, focusing and polarity have seen a surge in interest as more and more surprising properties of focusing proofs are discovered, including one important example: it is slowly becoming clear that focusing proofs correspond to programs with explicit evaluation order (Herbelin 1995; Curien and Herbelin 2000; Selinger 2001; Laurent 2002; Wadler 2003; Laurent 2005; Dyckhoff and Lengrand 2006). In this paper we will demonstrate an additional fascinating fact about focusing proofs: they correspond to programs with *pattern-matching.* Moreover, it turns out that focusing can be given a uniform, higher-order formulation as an *iterated inductive definition,* and that this representation can be encoded naturally in Coq (Martin-Löf 1971; Coquand and Paulin-Mohring 1989). Combining these facts, we obtain the above aphorism: a new kind of higher-order abstract syntax that encodes "pattern-matching for free".

POPL '08 January 10–12, 2008, San Francisco, CA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001...$5.00

2. Focusing intuitionistic logic

2.1 Background

Before diving into the compact presentation of focusing and its Curry-Howard interpretation à la HOAS, let us start on more familiar ground with a standard intuitionistic sequent calculus, and describe how to obtain a "small-step" focusing system. Figure 1 gives the sequent calculus for intuitionistic logic in a slight variation of Kleene's **G3i** formulation (Kleene 1952; Troelstra and Schwichtenberg 1996). Formulas (P, Q, R) are built out of conjunction ($\times$) and disjunction ($+$) and their respective units (1 and 0), implication ($\rightarrow$), and logical atoms (X, Y, Z). Every logical connective has a pair of a left rule and right rule(s) (we omit the rules for the units to save space). The identity rule is restricted to atoms and there is no explicit cut rule, though both cut (from $\Gamma \vdash P$ and $\Gamma, P \vdash Q$ conclude $\Gamma \vdash Q$) and the general identity principle ($P \in \Gamma$ implies $\Gamma \vdash P$) are admissible.

Now, one way to conceive of the sequent calculus, as Gentzen (1935) originally suggested, is as a proof search procedure. Each rule can be read bottom-up as a prescription, "To prove the conclusion, try proving the premises". Starting from a goal sequent $\Gamma \vdash P$, one attempts to build a proof by invoking left- and right-rules provisionally to obtain a new set of goals until, hopefully, all goals can be discharged using rules with no premises (i.e., *id*, $1R$ or $0L$). Since there are only finitely many rules and each satisfies the *subformula property* (Troelstra and Schwichtenberg 1996), it is not hard to see that (so long as one checks saturation conditions to avoid repeatedly applying left rules) the sequent calculus gives a naive decision procedure for propositional intuitionistic logic.

The reason this decision procedure is naive, though, is because the order of application of rules is left entirely unspecified. For example, the following are two equally legitimate derivations of $X \times Y \vdash X \times Y$, that differ only in the order of $\times L$ and $\times R$:

Context $\quad \Gamma ::= \cdot \mid \Gamma, P$

$$\frac{X \in \Gamma}{\Gamma \vdash X}\,id \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\,{\to}R$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \times Q}\,{\times}R \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P + Q} \quad \frac{\Gamma \vdash Q}{\Gamma \vdash P + Q}\,{+}R$$

$$\frac{P \times Q \in \Gamma \quad \Gamma, P, Q \vdash R}{\Gamma \vdash R}\,{\times}L$$

$$\frac{P + Q \in \Gamma \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R}\,{+}L$$

$$\frac{P \to Q \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q \vdash R}{\Gamma \vdash R}\,{\to}L$$

Figure 1. Intuitionistic sequent calculus

$$\cfrac{\cfrac{\overline{X \times Y, X, Y \vdash X}\,id \quad \overline{X \times Y, X, Y \vdash Y}\,id}{X \times Y, X, Y \vdash X \times Y}\,{\times}R}{X \times Y \vdash X \times Y}\,{\times}L$$

$$\cfrac{\cfrac{\overline{X \times Y, X, Y \vdash X}\,id}{X \times Y \vdash X}\,{\times}L \quad \cfrac{\overline{X \times Y, X, Y \vdash Y}\,id}{X \times Y \vdash Y}\,{\times}L}{X \times Y \vdash X \times Y}\,{\times}R$$

However, it is not the case that order of application is *arbitrary.* For example, to prove $X + Y \vdash X + Y$, one must apply $+L$ first (from the bottom):

$$\cfrac{\cfrac{\overline{X + Y, X \vdash X}\,id}{X + Y, X \vdash X + Y}\,{+}R \quad \cfrac{\overline{X + Y, Y \vdash Y}\,id}{X + Y, Y \vdash X + Y}\,{+}R}{X + Y \vdash X + Y}\,{+}L$$

Applying either right-rule first will yield a failed proof attempt.

In these terms, focusing can be seen as exploiting properties about the connectives to implement a smarter bottom-up proof search. Figure 2 presents a focusing system for intuitionistic logic that implements the following strategy:

1. Decompose conjunctions and disjunctions greedily on the left, until the context contains only atoms and implications.
2. Given a stable sequent (i.e., one with no undecomposed hypotheses), "focus" on some proposition ($\Gamma \vdash [P]$), either the right side of the sequent or the antecedent of a hypothesis $P \to Q$.
3. A proposition in focus remains in focus (forcing us to keep applying right-rules) until either there are no more premises, or else we reach an implication, which "blurs" the sequent (and we go back to step 1).

Note that this is not the only possible focusing strategy for propositional intuitionistic logic. Most of the intuitionistic connectives have ambiguous *polarity,* in Girard's sense (Girard 1993). This is in contrast with the connectives of linear logic, which have fixed polarity. So whereas there is essentially only one way to focus linear logic, there are different possible strategies for intuitionistic logic, corresponding to different *polarizations.* Our strategy treats conjunction and disjunction as both *positive,* implication as *negative,* which turns out to correspond, via Curry-Howard, to the strict, call-by-value interpretation (Curien and Herbelin 2000; Selinger 2001; Laurent 2005). To emphasize this fact, we adopt linear logic notation for positive conjunction and disjunction ($\otimes$, $\oplus$, 1, 0), and write $\xrightarrow{v}$ for implication.[1] Very similar focusing systems based on the same polarizations are presented in (Girard 2001, §9.2.3) and (Dyckhoff and Lengrand 2006). Some examples of focusing systems derived from alternative (lazy, call-by-name) polarizations of intuitionistic logic are in (Herbelin 1995; Howe 1998; Miller and Liang 2007).

Stable context $\quad \Gamma \;::=\; \cdot \mid \Gamma, X \mid \Gamma, P \xrightarrow{v} Q$
Active context $\quad \Omega \;::=\; \cdot \mid P, \Omega$

$$\boxed{\Gamma \vdash [P]}$$

$$\frac{X \in \Gamma}{\Gamma \vdash [X]} \qquad \frac{\Gamma; P \vdash Q}{\Gamma \vdash [P \xrightarrow{v} Q]}$$

$$\frac{\Gamma \vdash [P] \quad \Gamma \vdash [Q]}{\Gamma \vdash [P \otimes Q]} \qquad \frac{\Gamma \vdash [P]}{\Gamma \vdash [P \oplus Q]} \quad \frac{\Gamma \vdash [Q]}{\Gamma \vdash [P \oplus Q]}$$

$$\boxed{\Gamma; \Omega \vdash R}$$

$$\frac{\Gamma, X; \Omega \vdash R}{\Gamma; X, \Omega \vdash R} \qquad \frac{\Gamma, P \xrightarrow{v} Q; \Omega \vdash R}{\Gamma; P \xrightarrow{v} Q, \Omega \vdash R}$$

$$\frac{\Gamma; P, Q, \Omega \vdash R}{\Gamma; P \otimes Q, \Omega \vdash R} \qquad \frac{\Gamma; P, \Omega \vdash R \quad \Gamma; Q, \Omega \vdash R}{\Gamma; P \oplus Q, \Omega \vdash R}$$

$$\frac{\Gamma \vdash R}{\Gamma; \cdot \vdash R}$$

$$\boxed{\Gamma \vdash R}$$

$$\frac{\Gamma \vdash [P]}{\Gamma \vdash P} \qquad \frac{P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash [P] \quad \Gamma; Q \vdash R}{\Gamma \vdash R}$$

Figure 2. Focused intuitionistic sequent calculus

Of course, from the point of view of proof-search, it is crucial that any focusing strategy be *complete,* i.e., if a sequent is provable in the ordinary sequent calculus, then the focusing strategy will succeed in finding *some* derivation. We will not give a completeness proof for this system here (the reader could refer to (Dyckhoff and Lengrand 2006)), and instead move on to describe an alternative presentation of focusing.

2.2 A higher-order formulation

Let us begin with some observations about derived rules in the focused system. These observations are not new (Andreoli 2001; Girard 2001)—but the system we obtain from these observations will be.

Consider proving the proposition $X \otimes (Y \oplus (P \xrightarrow{v} Q))$ in focus. The derivation must begin in one of the following two ways, before losing focus:

$$\cfrac{\Gamma \vdash [X] \quad \cfrac{\Gamma \vdash [Y]}{\Gamma \vdash [Y \oplus (Q \xrightarrow{v} P)]}}{\Gamma \vdash [X \otimes (Y \oplus (Q \xrightarrow{v} P))]} \qquad \cfrac{\Gamma \vdash [X] \quad \cfrac{\Gamma \vdash [P \xrightarrow{v} Q]}{\Gamma \vdash [Y \oplus (P \xrightarrow{v} Q)]}}{\Gamma \vdash [X \otimes (Y \oplus (P \xrightarrow{v} Q))]}$$

Once in focus, atomic propositions can only be proven by assumption, while implications initiate a decomposition phase. The set of derived rules

$$\frac{X \in \Gamma \quad Y \in \Gamma}{\Gamma \vdash [X \otimes (Y \oplus (P \xrightarrow{v} Q))]} \qquad \frac{X \in \Gamma \quad \Gamma; P \vdash Q}{\Gamma \vdash [X \otimes (Y \oplus (P \xrightarrow{v} Q)))]}$$

[1] Describing the polarity of call-by-value implication is actually a bit more subtle. Technically, one can identify an underlying negative implication $P \multimap N$ which takes positive antecedent and negative consequent, and then analyze $P \xrightarrow{v} Q$ with implicit polarity "shifts" (Girard 2001, §3.3.2), i.e., either as $P \multimap \uparrow Q$ (as a negative hypothesis) or $\Downarrow(P \multimap \uparrow Q)$ (as a positive conclusion).

is therefore *complete,* in the sense that it covers all possible derivations of the formula in right-focus.

Similarly, consider decomposing $X \otimes (Y \oplus (P \xrightarrow{v} Q))$ on the left of the sequent:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, X, Y \vdash R}{\Gamma, X, Y; \cdot \vdash R}}{\Gamma, X; Y \vdash R} \quad \dfrac{\dfrac{\Gamma, X, P \xrightarrow{v} Q \vdash R}{\Gamma, X, P \xrightarrow{v} Q; \cdot \vdash R}}{\Gamma, X; P \xrightarrow{v} Q \vdash R}}{\Gamma, X; Y \oplus (P \xrightarrow{v} Q) \vdash R}}{\Gamma; X, Y \oplus (P \xrightarrow{v} Q) \vdash R}}{\Gamma; X \otimes (Y \oplus (P \xrightarrow{v} Q)) \vdash R}$$

Again, the following derived rule is complete:

$$\dfrac{\Gamma, X, Y \vdash R \quad \Gamma, X, P \xrightarrow{v} Q \vdash R}{\Gamma; X \otimes (Y \oplus (P \xrightarrow{v} Q)) \vdash R}$$

In general for a proposition P, we can give a complete set (possibly empty) of derived rules for establishing $\Gamma \vdash [P]$, each containing a set (possibly empty) of premises of the form $X \in \Gamma$ or $\Gamma; Q \vdash R$. Likewise, we can give a single, complete derived rule for establishing $\Gamma; P \vdash R$, with a set (possibly empty) of premises of the form $\Gamma, \Gamma' \vdash R$.

Both kinds of derived rules for a formula P can be generated from a single description, which we can gloss as the possible "recipes" for a focused proof. To derive $\Gamma \vdash [P]$, we must provide (using Γ) all of the "ingredients" for *some* recipe. To derive $\Gamma; P \vdash R$, we must show how to derive R given (Γ and) the ingredients for *any* of the recipes. As we are trying to suggest by using culinary language (Wadler 1993), the method for constructing both kinds of derived rules can be expressed in terms of linear entailment.

More precisely, a "list of ingredients" Δ is a linear context of atoms and implications, and we write $\Delta \Rightarrow P$ when Δ exactly describes the focused premises in a possible focused proof of P. The rules for $\Delta \Rightarrow P$ (top of Figure 3) are just the usual right-rules for the positive connectives of linear logic together with axioms $X \Rightarrow X$ and $P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q$. By way of example, we have $X, Y \Rightarrow X \otimes (Y \oplus (P \xrightarrow{v} Q))$ and $X, P \xrightarrow{v} Q \Rightarrow X \otimes (Y \oplus (P \xrightarrow{v} Q))$. Note that the judgment $\Delta \Rightarrow P$ obeys a subformula property.

Proposition (Subformula property)**.** *If $\Delta \Rightarrow P$, then Δ contains only subformulas of P.*

The generic instructions for proving a proposition in focus, which we described informally above, can now be written formally:

$$\dfrac{\Delta \Rightarrow P \quad \Gamma \vdash \Delta}{\Gamma \vdash [P]}$$

The judgment $\Gamma \vdash \Delta$ is interpreted conjunctively:[2] from the hypotheses in Γ we must prove everything in Δ. Thus the rule asks for some choice of recipe (i.e., $\Delta \Rightarrow P$), and a proof that we have all the ingredients (i.e., $\Gamma \vdash \Delta$). Note that although this rule leaves Δ unspecified, it still obeys the usual subformula property, by the subformula property for $\Delta \Rightarrow P$.

Likewise, we can write the generic rule for decomposing a proposition on the left:

$$\dfrac{\forall(\Delta \Rightarrow P): \quad \Gamma, \Delta \vdash Q}{\Gamma; P \vdash Q}$$

Here the rule quantifies over *all* Δ such that $\Delta \Rightarrow P$, showing that from any such Δ (together with Γ), Q is derivable. This sort of quantification over derivations might seem like a risky form of definition, but it is simply an *iterated* inductive definition (Martin-Löf 1971)—since we already established what $\Delta \Rightarrow P$ means, there is no circularity in treating it as an assumption here. Indeed, for any particular P built out of the connectives we have considered, there will only ever be finitely many derivations $\Delta \Rightarrow P$, so this rule will just have a finite list of premises (as in the example above). However, we hope to make the case that this higher-order formulation should be taken at face value—interpreted constructively, it demands a *mapping* from derivations of $\Delta \Rightarrow P$ to unfocused sequents $\Gamma, \Delta \vdash Q$. This idea will play a central role in our Curry-Howard interpretation.

Linear context $\quad \Delta \quad ::= \ \cdot \mid X, \Delta \mid P \xrightarrow{v} Q, \Delta$

$$\boxed{\Delta \Rightarrow P}$$

$$\dfrac{}{X \Rightarrow X} \qquad \dfrac{}{P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q}$$

$$\dfrac{}{\cdot \Rightarrow 1} \qquad \dfrac{\Delta_1 \Rightarrow P \quad \Delta_2 \Rightarrow Q}{\Delta_1, \Delta_2 \Rightarrow P \otimes Q}$$

$$\text{(no rule for 0)} \qquad \dfrac{\Delta \Rightarrow P}{\Delta \Rightarrow P \oplus Q} \qquad \dfrac{\Delta \Rightarrow Q}{\Delta \Rightarrow P \oplus Q}$$

....

Stable context $\quad \Gamma \quad ::= \ \cdot \mid \Gamma, \Delta$

$$\boxed{\Gamma \vdash [P]}$$

$$\dfrac{\Delta \Rightarrow P \quad \Gamma \vdash \Delta}{\Gamma \vdash [P]}$$

$$\boxed{\Gamma; P \vdash Q}$$

$$\dfrac{\forall(\Delta \Rightarrow P): \quad \Gamma, \Delta \vdash Q}{\Gamma; P \vdash Q}$$

$$\boxed{\Gamma \vdash \Delta}$$

$$\dfrac{}{\Gamma \vdash \cdot} \qquad \dfrac{X \in \Gamma \quad \Gamma \vdash \Delta}{\Gamma \vdash X, \Delta} \qquad \dfrac{\Gamma; P \vdash Q \quad \Gamma \vdash \Delta}{\Gamma \vdash P \xrightarrow{v} Q, \Delta}$$

$$\boxed{\Gamma \vdash R}$$

$$\dfrac{\Gamma \vdash [P]}{\Gamma \vdash P} \qquad \dfrac{P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash [P] \quad \Gamma; Q \vdash R}{\Gamma \vdash R}$$

Figure 3. Large-step focusing

The entire "large-step" focusing system is given in Figure 3, with all of the rules for all of the connectives (including the units 1 and 0). Observe that the *only* rules that explicitly mention the positive connectives are those for the $\Delta \Rightarrow P$ judgment, and we can take the latter as literally *defining* the positive connectives. While the system is relatively sparse in rules, it is "rich in judgments". The idea of the *judgmental method* (Martin-Löf 1996; Pfenning and Davies 2001) in general is that by distinguishing between different kinds of reasoning as different judgments (and not merely between different logical connectives or type constructors), one can clarify the structure of proofs. This becomes very vivid under a Curry-Howard interpretation, as the proofs of different judgments are internalized by different syntactic categories of a programming language. We will find that the five judgments of large-step focusing all correspond to very natural programming constructs. First, though, let us see how the identity and cut principles work in this new logical setting. Because of the additional judgmental machinery, identity is refined into three different principles.

Principle (Identity)**.** $\Gamma; P \vdash P$

Principle (Context identity)**.** *If* $\Gamma \supseteq \Delta$ *then* $\Gamma \vdash \Delta$

Principle (Arrow identity)**.** *If* $P \xrightarrow{v} Q \in \Gamma$ *then* $\Gamma; P \vdash Q$

[2] And so is *not* like a "multiple conclusion sequent" in Gentzen's LK.

Proof. These three principles are proven simultaneously—we give the proof first, and then explain its inductive structure.

- (Identity) The following derivation reduces identity to context identity:

$$\dfrac{\forall(\Delta \Rightarrow P) : \quad \dfrac{\dfrac{\Delta \Rightarrow P \quad \Gamma, \Delta \vdash \Delta}{\Gamma, \Delta \vdash [P]}}{\Gamma, \Delta \vdash P}}{\Gamma; P \vdash P}$$

  Note the first premise can be discharged since the derivation quantifies over Δ such that $\Delta \Rightarrow P$.
- (Context Identity) We apply a side-induction on the length of Δ. The interesting case is $\Delta = P \xrightarrow{v} Q, \Delta'$. By arrow identity we have $\Gamma; P \vdash Q$, and by the side-induction we have $\Gamma \vdash \Delta'$, letting us build the derivation:

$$\dfrac{\Gamma; P \vdash Q \quad \Gamma \vdash \Delta'}{\Gamma \vdash P \xrightarrow{v} Q, \Delta'}$$

- (Arrow Identity) Consider the following derivation:

$$\dfrac{\forall(\Delta \Rightarrow P) : \quad \dfrac{P \xrightarrow{v} Q \in \Gamma \quad \Gamma, \Delta \vdash [P] \quad \Gamma, \Delta; Q \vdash Q}{\Gamma, \Delta \vdash Q}}{\Gamma; P \vdash Q}$$

  The first premise $P \xrightarrow{v} Q \in \Gamma$ is by assumption. The second premise reduces (as in the proof of identity above) to context identity. The third premise is by identity.

The above argument can be seen to be well-founded so long as the relationship of being a proper subformula is well-founded. We reason as follows: The proof of identity appealed to context identity, which in turn appealed to arrow identity, and which finally appealed back to both context identity and identity. The first cycle (id on $P \rightsquigarrow$ context id on $\Delta \Rightarrow P \rightsquigarrow$ arrow id on $P_1 \xrightarrow{v} P_2 \in \Delta \rightsquigarrow$ id on P_2), takes P to a proper subformula P_2. The second cycle (context id on $\Delta \rightsquigarrow$ arrow id on $P_1 \xrightarrow{v} P_2 \in \Delta \rightsquigarrow$ context id on $\Delta' \Rightarrow P_1$), takes Δ to a proper *subcontext* Δ' (i.e., Δ' contains only proper subformulas of formulas in Δ). Both cycles cannot continue indefinitely if the proper subformula relationship is well-founded—as indeed it is for the propositional connectives. □

We can also distinguish between three different kinds of principles that would ordinarily be called "cuts". The first is where we have a derivation of $\Gamma, \Delta \vdash J$ (in which J stands for an arbitrary concluding judgment, i.e., $\Gamma, \Delta \vdash [P]$ or $\Gamma, \Delta; P \vdash Q$ or $\Gamma, \Delta \vdash R$ or $\Gamma, \Delta \vdash \Delta'$), and we want to substitute another derivation $\Gamma \vdash \Delta$ for the hypotheses Δ. The second is where we have a coincidence between a right-focused derivation $\Gamma \vdash [P]$, and a derivation $\Gamma; P \vdash Q$, which we can transform into an unfocused derivation $\Gamma \vdash Q$. In the third, we combine an unfocused derivation $\Gamma \vdash P$ together with $\Gamma; P \vdash Q$ to obtain $\Gamma \vdash Q$. We call the first cut principle *substitution,* the second *reduction,* and the third *composition.* In the usual proof-theoretic terminology, these correspond to *right-commutative*, *principal,* and *left-commutative cuts,* respectively.

Principle (Substitution). *If* $\Gamma, \Gamma' \vdash \Delta$ *and* $\Gamma, \Delta, \Gamma' \vdash J$ *then* $\Gamma, \Gamma' \vdash J$

Principle (Reduction). *If* $\Gamma \vdash [P]$ *and* $\Gamma; P \vdash Q$ *then* $\Gamma \vdash Q$

Principle (Composition). *If* $\Gamma \vdash P$ *and* $\Gamma; P \vdash Q$ *then* $\Gamma \vdash Q$

To prove these we need a weakening lemma, which is immediate.

Proposition (Weakening). *If* $\Gamma \vdash J$*, then* $\Gamma, \Delta \vdash J$.

Proof of substitution, reduction and composition. Again, the proof is simultaneous.

- (Substitution) We examine the derivation of $\Gamma, \Delta, \Gamma' \vdash J$. Almost all cases (there are seven total) are immediate, simply applying substitution (and possibly weakening) to the premises and reconstructing the derivation. The one interesting case is the following:

$$\dfrac{P \xrightarrow{v} Q \in \Delta \quad \Gamma, \Delta, \Gamma' \vdash [P] \quad \Gamma, \Delta, \Gamma'; Q \vdash R}{\Gamma, \Delta, \Gamma' \vdash R}$$

  By substitution on the premises we have $\Gamma, \Gamma' \vdash [P]$ and $\Gamma, \Gamma'; Q \vdash R$. Moreover $\Gamma, \Gamma' \vdash \Delta$ and $P \xrightarrow{v} Q \in \Delta$ imply (by inversion) that $\Gamma, \Gamma'; P \vdash Q$. We cut $\Gamma, \Gamma' \vdash [P]$ and $\Gamma, \Gamma'; P \vdash Q$ using reduction to obtain $\Gamma, \Gamma' \vdash Q$, and the latter with $\Gamma, \Gamma'; Q \vdash R$ using composition to obtain $\Gamma, \Gamma' \vdash R$.
- (Reduction) By inversion on $\Gamma \vdash [P]$, there exists some $\Delta \Rightarrow P$ such that $\Gamma \vdash \Delta$, and by inversion on $\Gamma; P \vdash Q$ we have $\Gamma, \Delta \vdash Q$. Hence $\Gamma \vdash Q$ by substitution.
- (Composition) We examine the derivation of $\Gamma \vdash P$. If it was derived from $\Gamma \vdash [P]$, we immediately apply reduction. Otherwise the derivation must look like so:

$$\dfrac{P_1 \xrightarrow{v} P_2 \in \Gamma \quad \Gamma \vdash [P_1] \quad \dfrac{\forall(\Delta \Rightarrow P_2) : \quad \Gamma, \Delta \vdash P}{\Gamma; P_2 \vdash P}}{\Gamma \vdash P}$$

  For any $\Delta \Rightarrow P_2$, we can weaken the derivation $\Gamma; P \vdash Q$ to $\Gamma, \Delta; P \vdash Q$, and then apply composition to obtain $\Gamma, \Delta \vdash Q$. Thus $\Gamma; P_2 \vdash Q$, and we can reconstruct the derivation concluding $\Gamma \vdash Q$.

The above defines a cut-elimination procedure, which we can easily see is terminating by a nested induction. First on the cut formula/context, then on the second derivation for substitution, and on the first derivation for composition. Again, this uses the fact that the proper subformula relationship is well-founded. □

We gave the proofs of identity and cut in such explicit detail in part to emphasize that there actually *isn't* very much detail. For example, we did not have to give the case of one "typical" positive connective and sweep the others under the rug, because both proofs do not even *mention* particular positive connectives—instead they reason modularly about derivations of $\Delta \Rightarrow P$. And modularity is a powerful tool: it gives us license to introduce new types almost arbitrarily, so long as we define them purely through the $\Delta \Rightarrow P$ judgment.

3. Focusing the λ-calculus

In the previous section, we saw how combining the technique of focusing with a judgmental and higher-order analysis of derived rules led to a sequent calculus "rich in judgments". Now, we will show how these different judgments correspond precisely, through the Curry-Howard isomorphism, to natural programming language constructs. We start with a type system containing all the propositional connectives described above, though for simplicity leaving out atomic types—so the language will have strict products and sums, and call-by-value function spaces. After giving it an operational semantics corresponding to cut-elimination and proving type safety, we will show how our informal use of higher-order abstract syntax can be formalized in Coq. Finally, we will try to give a demonstration of the aforementioned modularity principle, by showing the ease with which recursion and recursive types can be added to the language.

Focusing	Typing	Syntactic category
$\Delta \Rightarrow P$	$\Delta \Rightarrow p : P$	patterns
$\Gamma \vdash [P]$	$\Gamma \vdash V : [P]$	values
$\Gamma; P \vdash Q$	$\Gamma \vdash F : P > Q$	(CBV) functions
$\Gamma \vdash R$	$\Gamma \vdash E : R$	expressions
$\Gamma \vdash \Delta$	$\Gamma \vdash \sigma : \Delta$	substitutions

Figure 4. The Curry-Howard isomorphism

3.1 Type system

Let us begin by examining the $\Delta \Rightarrow P$ judgment, which lies at the heart of our formulation of focusing. Previously we described $\Delta \Rightarrow P$ as holding when the context Δ is an exact list of focused premises needed for a focused proof of P. For a particular P, there need not be a unique Δ such that $\Delta \Rightarrow P$, and indeed there might not be any such Δ (e.g., when $P = 0$). What then do derivations of $\Delta \Rightarrow P$ look like? Abstractly, they describe the different *shapes* a focused proof of P can have, up to the point where either the derivation ends or focus is lost. Thus for example we only have the axiomatic derivation $P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q$, because the first step in a focused proof of $P \xrightarrow{v} Q$ is to immediately lose focus. On the other hand, there are two rules for disjunction:

$$\frac{\Delta \Rightarrow P}{\Delta \Rightarrow P \oplus Q} \quad \frac{\Delta \Rightarrow Q}{\Delta \Rightarrow P \oplus Q}$$

because a focused proof of $P \oplus Q$ can continue by focusing on either P or Q.

Now, let us label the hypotheses in Δ with variables—since we are ignoring atomic hypotheses, there are only function variable hypotheses $f : P \xrightarrow{v} Q$. Then we can annotate $\Delta \Rightarrow P$ as a *pattern-typing* judgment:

$$\frac{}{f : P \xrightarrow{v} Q \Rightarrow f : P \xrightarrow{v} Q}$$

$$\frac{}{\cdot \Rightarrow () : 1} \qquad \frac{\Delta_1 \Rightarrow p_1 : P \quad \Delta_2 \Rightarrow p_2 : Q}{\Delta_1, \Delta_2 \Rightarrow (p_1, p_2) : P \otimes Q}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta \Rightarrow p : P}{\Delta \Rightarrow \mathsf{inl}\, p : P \oplus Q} \quad \frac{\Delta \Rightarrow p : Q}{\Delta \Rightarrow \mathsf{inr}\, p : P \oplus Q}$$

A programmer might now get an intuition for why the context Δ must be linear: it corresponds to the usual restriction that *patterns cannot bind a variable more than once.* Likewise why $P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q$ is an axiom: it corresponds to a *primitive pattern.*

If $\Delta \Rightarrow P$ represents pattern-typing, what can we conclude about the other judgments of the focusing system? As we will describe, these correspond to typing judgments for *values, functions, expressions,* and *substitutions* (see Figure 4). Since these judgments are defined by mutual recursion, we will have to work our way through the system to convince ourselves that these names for the different syntactic categories were not chosen arbitrarily. We begin with $\Gamma \vdash [P]$, which will be annotated with a value V. Recall that the judgment is defined by a single rule:

$$\frac{\Delta \Rightarrow P \quad \Gamma \vdash \Delta}{\Gamma \vdash [P]}$$

The first premise is now annotated $\Delta \Rightarrow p : P$, giving us a pattern p binding some function variables with types given by Δ. The second premise is annotated with a simultaneous substitution $\sigma = (F_1/f_1, \ldots, F_n/f_n)$, where $f_1, \ldots, f_n$ are the variables in Δ and $F_1, \ldots, F_n$ are functions. Thus the annotated rule becomes:

$$\frac{\Delta \Rightarrow p : P \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash [\sigma]p : [P]}$$

What exactly is this curious value $[\sigma]p$, which combines a pattern together with an explicit substitution? We can think of this notation as simply internalizing a trivial factorization lemma about values in the ordinary sense. For example the ML value

```
(fn x => x*x, fn x => x-3)
```

can be factored as the pattern `(f,g)` composed with a substitution [`(fn x => x*x)/f`, `(fn x => x-3)/g`]. As we shall see, the utility of this factorization is that values are given a uniform representation.

What about functions? Again, let us look at the unannotated rule for $\Gamma; P \vdash Q$:

$$\frac{\forall(\Delta \Rightarrow P) : \quad \Gamma, \Delta \vdash Q}{\Gamma; P \vdash Q}$$

Recall this is a higher-order rule, which can be interpreted constructively as demanding a map from derivations of $\Delta \Rightarrow P$ to derivations of unfocused sequents $\Gamma, \Delta \vdash Q$. The former, we know, correspond to patterns with types for their free variables. The latter correspond to "expressions" (the precise sense of which will be explained below). Therefore, a function is *a map from patterns to expressions.* In other words, functions are defined using higher-order abstract syntax (Pfenning and Elliott 1988).

Formally, we will assume the existence of partial maps ϕ, defined inductively over patterns. Thus for any pattern p, $\phi(p)$ is either undefined or denotes a unique expression, possibly mentioning variables bound by p, and moreover this mapping respects renaming of pattern variables. Concretely, partial maps may be specified by a finite list of branches:

$$\phi ::= (p_1 \mapsto E_1 \mid \ldots \mid p_n \mapsto E_n)$$

with the proviso that the p_i do not overlap. In Section 3.3 we will describe how to encode the HOAS representation explicitly in Coq, using the function space $\mathsf{pat} \to \mathsf{exp}$.

Now to build a function, we simply wrap a ϕ with a λ. The annotated rule for function-typing becomes:

$$\frac{\forall(\Delta \Rightarrow p : P) : \quad \Gamma, \Delta \vdash \phi(p) : Q}{\Gamma \vdash (\lambda\phi) : P > Q}$$

We should emphasize that we are still only defining the abstract *syntax* of functions, not their evaluation semantics—although the two aspects are indeed closely related. For instance, the syntax forces a call-by-value interpretation, since functions are defined by pattern-matching over fully-expanded patterns.[3] Moreover, a well-typed function $(\lambda\phi) : P > Q$ is necessarily *exhaustive* (since the typing rule forces $\phi(p)$ to be defined for all p and Δ such that $\Delta \Rightarrow p : P$) and *non-redundant* (since ϕ is defined as a map), in the usual sense of pattern-matching.

Finally, the two rules for deriving unfocused sequents are now annotated as typing expressions:

$$\frac{\Gamma \vdash V : [P]}{\Gamma \vdash V : P} \quad \frac{g : P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash V : [P] \quad \Gamma \vdash F : Q; R}{\Gamma \vdash F(g(V)) : R}$$

The first rule creates an expression directly from a value, the second by feeding a value to a named function variable, and composing the result with another function. From these two rules, we can intuit that "expressions" really do correspond closely to expressions in the ML sense—that is to *computations* (Moggi 1991). However, our expressions have a more rigid syntax, with an explicit sequencing of evaluation that resembles A-normal form (Flanagan et al. 1993).

[3] Of course, Haskell has pattern-matching too, so the emphasis is on "fully-expanded". In Haskell, there is a semantic difference between a function defined using wildcard/variable patterns, and the one obtained by replacing the wildcards/variables with expanded patterns. For example `\x -> ()` and `\() -> ()` both can be given type `() -> ()`, but the latter is strict. This difference does not exist in ML since all functions are strict.

Indeed, as with A-normal form, we seem to encounter the problem that substitution requires a "re-normalization" step: for how do we express the result of substituting G/g into $F(g(V))$?

Another way of looking at this is that the expression $F(g(V))$ corresponds to the "one interesting case" in the proof of the substitution principle from Section 2.2, wherein we appealed back to the reduction and composition principles. Consequently, to make the language closed under ordinary substitution, we *internalize* these principles as additional rules for forming expressions:

$$\frac{\Gamma \vdash V : [P] \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(V) : Q} \qquad \frac{\Gamma \vdash E : P \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(E) : Q}$$

We can likewise internalize identity principles to let us take shortcuts when building terms:

$$\frac{}{\Gamma \vdash \mathsf{id} : P > P} \qquad \frac{f : P \xrightarrow{v} Q \in \Gamma}{\Gamma \vdash f : P > Q}$$

id is the polymorphic identity function, while arrow identity allows us to treat a function variable directly as a function.

The complete type system is summarized in Figure 5, defining this Curry-Howard interpretation for large-step focusing, which we call focused λ-calculus. The figure visually quarantines identity and cut principles, to highlight their special status. The reader may wonder why we have not also internalized the substitution and context identity principles. Such steps are possible—for example, internalizing substitution would give us a calculus in which explicit substitutions are evaluated incrementally (Abadi et al. 1991)—but we forgo them here, choosing instead to define these as meta-theoretic operations. Substitution is defined in Section 3.2; we state context identity here:

Principle (Context identity)**.** *Suppose $\Gamma \supseteq \Delta$, and that $f_1, \ldots, f_n$ are the variables in Δ. Then $\Gamma \vdash (f_1/f_1, \ldots, f_n/f_n) : \Delta$.*

Proof. Trivial (now that we can directly appeal to arrow identity). □

Let us consider some examples—but to make these more palatable, we first develop some syntactic sugar. Without danger of ambiguity, we can write values in "unfactorized" form:

$$V ::= F \mid () \mid (V_1, V_2) \mid \mathsf{inl}(V) \mid \mathsf{inr}(V)$$

It is always possible to recover a unique factorization, i.e., σ and p such that $V = [\sigma]p$. As a special case, every pattern p can also be seen as the value $[(f_1/f_1, \ldots, f_n/f_n)]p$, where $f_1, \ldots, f_n$ are the variables bound by p. Because the syntax is higher-order, we can use *meta*-variables to build maps by quantifying over (all or some subset of) patterns. So for example $p \mapsto ()$ is a constant map which sends any pattern to $()$, while the map $f \mapsto ()$ is only defined on function variable patterns. When a function is defined by a single pattern-branch, we use the more conventional notation $\lambda p.E$ instead of $\lambda(p \mapsto E)$. Finally, we let $2 = 1 \oplus 1$ be the type of booleans, write $\mathsf{t} = \mathsf{inl}()$, $\mathsf{f} = \mathsf{inr}()$ for boolean patterns, and use b as a meta-variable quantifying over these two patterns.

EXAMPLE 1. We define boolean functions and and not:

$$\mathsf{and} = \lambda((\mathsf{t},\mathsf{t}) \mapsto \mathsf{t} \mid (\mathsf{t},\mathsf{f}) \mapsto \mathsf{f} \mid (\mathsf{f},\mathsf{t}) \mapsto \mathsf{f} \mid (\mathsf{f},\mathsf{f}) \mapsto \mathsf{f})$$

$$\mathsf{not} = \lambda(\mathsf{t} \mapsto \mathsf{f} \mid \mathsf{f} \mapsto \mathsf{t})$$

It is easy to check that $\mathsf{and} : 2 \otimes 2 > 2$ and $\mathsf{not} : 2 > 2$. In this simple case there is a bijective correspondence between patterns and values, and so the syntax basically mimics the standard mathematical definitions. ■

Linear context	Δ	$::=$	$\cdot \mid f : P \xrightarrow{v} Q, \Delta$
Pattern	p	$::=$	$f \mid () \mid (p_1, p_2) \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p$

$$\boxed{\Delta \Rightarrow p : P}$$

$$\frac{}{f : P \xrightarrow{v} Q \Rightarrow f : P \xrightarrow{v} Q}$$

$$\frac{}{\cdot \Rightarrow () : 1} \qquad \frac{\Delta_1 \Rightarrow p_1 : P \quad \Delta_2 \Rightarrow p_2 : Q}{\Delta_1, \Delta_2 \Rightarrow (p_1, p_2) : P \otimes Q}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta \Rightarrow p : P}{\Delta \Rightarrow \mathsf{inl}\, p : P \oplus Q} \qquad \frac{\Delta \Rightarrow p : Q}{\Delta \Rightarrow \mathsf{inr}\, p : P \oplus Q}$$

Stable context	Γ	$::=$	$\cdot \mid \Gamma, \Delta$
Value	V	$::=$	$[\sigma]p$
Function	F	$::=$	$\lambda\phi \mid \mathsf{id} \mid f$
			where $\phi ::= (p_1 \mapsto E_1 \mid \ldots \mid p_n \mapsto E_n)$
Substitution	σ	$::=$	$\cdot \mid (F/f, \sigma)$
Expression	E	$::=$	$V \mid F(g(V)) \mid F(V) \mid F(E)$

$$\boxed{\Gamma \vdash V : [P]}$$

$$\frac{\Delta \Rightarrow p : P \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash [\sigma]p : [P]}$$

$$\boxed{\Gamma \vdash F : P > Q}$$

$$\frac{\forall(\Delta \Rightarrow p : P) :\ \Gamma, \Delta \vdash \phi(p) : Q}{\Gamma \vdash (\lambda\phi) : P > Q} \qquad \frac{}{\Gamma \vdash \mathsf{id} : P > P} \qquad \frac{f : P \xrightarrow{v} Q \in \Gamma}{\Gamma \vdash f : P > Q}$$

$$\boxed{\Gamma \vdash \sigma : \Delta}$$

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash F : P > Q \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash (F/f, \sigma) : (f : P \xrightarrow{v} Q, \Delta)}$$

$$\boxed{\Gamma \vdash E : R}$$

$$\frac{\Gamma \vdash V : [P]}{\Gamma \vdash V : P} \qquad \frac{g : P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash V : [P] \quad \Gamma \vdash F : Q; R}{\Gamma \vdash F(g(V)) : R}$$

$$\frac{\Gamma \vdash V : [P] \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(V) : Q} \qquad \frac{\Gamma \vdash E : P \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(E) : Q}$$

identity principles

cut principles

Figure 5. Focused λ-calculus (type system)

EXAMPLE 2. We define $\mathsf{table1} : 2 \xrightarrow{v} 2 > 2 \otimes 2$, a higher-order function taking a unary boolean operator as input, and returning its truth table as output:

$$\mathsf{table1} = \lambda f.(\lambda b_1.(\lambda b_2.(b_1, b_2))(f\ \mathsf{f}))(f\ \mathsf{t})$$

Here f is a function variable, while b_1 and b_2 are meta-variables quantifying over boolean patterns. Observe that the syntax forces us to choose a sequential order for the calls to f (we evaluate $f(\mathsf{t})$ first, then $f(\mathsf{f})$). ■

3.2 Operational semantics

The substitution principle of Section 2.2, translated to the language of proof terms, says that for any substitution $\Gamma, \Gamma' \vdash \sigma : \Delta$ and arbitrary term $\Gamma, \Delta, \Gamma' \vdash t : J$ (i.e., a value $V : [P]$, function $F : P > Q$, substitution $\sigma' : \Delta'$, or expression $E : R$), there should be a term $[\sigma]t$ such that $\Gamma, \Gamma' \vdash [\sigma]t : J$. Rather than internalizing this principle in the syntax, we define $[\sigma]t$ as an operation, namely the usual simultaneous, capture-avoiding substitution. The definition

$$\boxed{E \rightsquigarrow E'}$$

$$\frac{\phi(p) \text{ defined}}{(\lambda\phi)([\sigma]p) \rightsquigarrow [\sigma]\phi(p)} \qquad \mathsf{id}(V) \rightsquigarrow V \qquad \frac{E \rightsquigarrow E'}{F(E) \rightsquigarrow F(E')}$$

$$\boxed{[\sigma]t}$$

$$[\sigma]f = \begin{cases} F & (F/f) \in \sigma \\ f & f \notin \mathrm{dom}(\sigma) \end{cases}$$

$$[\sigma]([\sigma']p) = [[\sigma]\sigma']p \qquad [\sigma](\lambda\phi) = \lambda p.[\sigma]\phi(p) \quad [\sigma]\mathsf{id} = \mathsf{id}$$

$$[\sigma]\cdot = \cdot \qquad [\sigma](F/f, \sigma') = ([\sigma]F/f, [\sigma]\sigma')$$

$$[\sigma](F(g(V))) = [\sigma]F([\sigma]g([\sigma]V))$$
$$[\sigma](F(V)) = [\sigma]F([\sigma]V) \qquad [\sigma](F(E)) = [\sigma]F([\sigma]E)$$

Figure 6. Focused λ-calculus (operational semantics)

of $[\sigma]t$ (given in Figure 6) is completely unsurprising, but a couple cases worth mention. Applying $[\sigma]$ to a function $\lambda\phi$ defines a new function by composing ϕ with the substitution:

$$[\sigma](\lambda\phi) = \lambda p.[\sigma]\phi(p)$$

Moreover, as we observed above, if σ maps G/g, then applying σ to the irreducible expression $F(g(V))$ converts it into two cuts: the expression $G([\sigma]V)$ composed with $[\sigma]F$.

The annotated version of the substitution principle is proven easily by induction, as in Section 2.2.

Lemma (Substitution)**.** *If* $\Gamma, \Gamma' \vdash \sigma : \Delta$ *and* $\Gamma, \Delta, \Gamma' \vdash t : J$ *and then* $\Gamma, \Gamma' \vdash [\sigma]t : J$.

The operational semantics is then given by a transition relation $E \rightsquigarrow E'$ on closed expressions, with three rules:

$$\frac{\phi(p) \text{ defined}}{(\lambda\phi)([\sigma]p) \rightsquigarrow [\sigma]\phi(p)} \qquad \mathsf{id}(V) \rightsquigarrow V \qquad \frac{E \rightsquigarrow E'}{F(E) \rightsquigarrow F(E')}$$

All of the complexity of pattern-matching is implemented by the one rule on the left, so let's unpack it: a function $F = \lambda\phi$ is defined (syntactically) as a partial map from patterns to open expressions; a value $V = [\sigma]p$ is a pattern together with an explicit substitution for its variables; thus to *apply* F to V, we find the expression $\phi(p)$ corresponding to p (assuming one exists), and apply the substitution σ.

Preservation and progress are stated in the usual way.

Theorem (Preservation)**.** *If* $\Gamma \vdash E : P$ *and* $E \rightsquigarrow E'$*, then* $\Gamma \vdash E' : P$.

Proof. Immediate by induction on (the derivation of) $E \rightsquigarrow E'$, using the substitution lemma in the case of a reduction $(\lambda\phi)([\sigma]p) \rightsquigarrow [\sigma]\phi(p)$ (like in the proof of reduction from Section 2.2). □

Theorem (Progress)**.** *If* $\vdash E : P$*, then either* $E = V$ *or else there exists* E' *such that* $E \rightsquigarrow E'$.

Proof. Immediate by induction on $\vdash E : P$, using the fact that well-typed functions are exhaustive. □

EXAMPLE 3. Recall the functions and, not, and table1 from Examples 1 and 2. The reader can verify the following calculation:

$\mathsf{and}(\mathsf{table1}(\mathsf{not}))$
$\rightsquigarrow \mathsf{and}((\lambda b_1.(\lambda b_2.(b_1, b_2))(\mathsf{not}\,\mathsf{f}))(\mathsf{not}\,\mathsf{t}))$
$\rightsquigarrow \mathsf{and}((\lambda b_1.(\lambda b_2.(b_1, b_2))(\mathsf{not}\,\mathsf{f}))\ \mathsf{f})$
$\rightsquigarrow \mathsf{and}((\lambda b_2.(\mathsf{f}, b_2))(\mathsf{not}\,\mathsf{f}))$
$\rightsquigarrow \mathsf{and}((\lambda b_2.(\mathsf{f}, b_2))\ \mathsf{t})$
$\rightsquigarrow \mathsf{and}(\mathsf{f}, \mathsf{t})$
$\rightsquigarrow \mathsf{f}$

■

3.3 Representation in Coq

Yet the reader may still have lurking suspicions about our language definition. Aren't we overlooking Reynolds' lesson about the pitfalls of higher-order definitions of higher-order programming languages (Reynolds 1972)? Isn't there a circularity in our appeal to a "meta-level" notion of maps while defining functions? Here, we will attempt to rest these concerns by giving an encoding of focused λ-calculus in Coq, a proof assistant based on the Calculus of Inductive Constructions (Coquand and Huet 1988; Coquand and Paulin-Mohring 1989; Coq Development Team 2006).

But our first step will be to try to explain how in a paper with "higher-order abstract syntax" in the title, we will have the chutzpah to use de Bruijn indexes in this encoding (de Bruijn 1972). To be clear, we are proposing a new kind of higher-order abstract syntax. In its usual application, HOAS refers to representing object-language variables by meta-language variables (Pfenning and Elliott 1988). This allows object-language binding constructs to be encoded by corresponding meta-language constructs, and thereby eliminates the need for dealing explicitly with tricky notions such as variable-renaming, parametric quantification and capture-avoiding substitution. The logical framework Twelf is very well-suited for this kind of representation technique (Twelf 2007). In contrast, the novelty of our approach is encoding object-language *induction* by meta-level induction. The Coq proof assistant, it turns out, is well-suited for this kind of representation technique. Ideally, we would be able to combine both forms of HOAS, as we did above at a pre-formal level. But although there have been some attempts at encoding standard HOAS in Coq (Despeyroux et al. 1995), and some work on incorporating induction principles into LF (Schürmann et al. 2001), these are still at experimental stages. We therefore use Coq to highlight the novel aspects of our higher-order encoding, but accept the limitations of a first-order representation of variables.

With that apology out of the way, let us move on to the formalization.[4] As we did throughout the above discussion, we will define the focused λ-calculus in "Curry-style", that is, with typing rules for type-free terms, and a type-free operational semantics. An alternative "Church-style" approach would be to directly encode the logical rules of Section 2.2, and then simply extract the language, with *typed* terms being derivations of the logical judgments.[5]

We begin by defining $\mathsf{tp} : \mathsf{Set}$ as a standard algebraic datatype with constructors $0, 1 : \mathsf{tp}$ and $\otimes, \oplus, \xrightarrow{v} : \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}$ (we will use infix notation for the latter). For convenience, we also add $2 : \mathsf{tp}$ to directly represent booleans. The type of hypotheses $\mathsf{hyp} : \mathsf{Set}$ is defined by one constructor of type $\mathsf{tp} \to \mathsf{tp} \to \mathsf{hyp}$, but we will simply write $P \xrightarrow{v} Q : \mathsf{hyp}$, overloading the tp constructor (it will always be clear from context which constructor we really mean).

Now, linear contexts $\Delta : \mathsf{linctx}$ are lists of hyps, while stable contexts $\Gamma : \mathsf{ctx}$ are lists of linctxs. We write $[]$ for the empty list, $[a]$ for a singleton, $a :: l$ for the "cons" operation, and $l_1 +\!\!+ \, l_2$ for concatenation. Since contexts are lists of *lists*, de Bruijn indexes are given by *pairs* of natural numbers, written $i.j : \mathsf{index}$. It is quite reasonable to think of these using machine intuitions: if Γ represents a stack of frames Δ, then a de Bruijn index $i.j$ specifies a "frame pointer" i plus an "offset" j. We write $\#j(\Delta)$ for the

[4] The full Coq source code for the encoding described here is available at: `http://www.cs.cmu.edu/~noam/research/focusing.tar`

[5] See `http://www.cs.cmu.edu/~noam/research/focus-church.v`.

jth element of Δ, and $\#i.j(\Gamma)$ for the jth element of the ith linear context in Γ. These are both partial operations, returning options in Coq, but we will abuse notation and write $\#i.j(\Gamma) = H$ meaning $\#i.j(\Gamma) = \mathsf{Some}\ H$, and similarly with $\#j(\Delta)$. In general, we will stray slightly from concrete Coq syntax so as to improve readability.

We define pat as another algebraic datatype, built using constructors $(), \mathsf{t}, \mathsf{f} : \mathsf{pat}$ and $(-,-) : \mathsf{pat} \rightarrow \mathsf{pat} \rightarrow \mathsf{pat}$, $\mathsf{inl}, \mathsf{inr} : \mathsf{pat} \rightarrow \mathsf{pat}$, and $\mathsf{fvar} : \mathsf{pat}$. The latter stands for a pattern binding a function variable—since we are using a de Bruijn representation, patterns do not actually name any variables. The pattern-typing judgment $\Delta \Rightarrow p : P$ is encoded by an inductive type family $\mathsf{pat\_tp} : \mathsf{linctx} \rightarrow \mathsf{pat} \rightarrow \mathsf{tp} \rightarrow \mathsf{Prop}$. We omit the names of the constructors for pat_tp, but give their types below (also leaving implicit the $\forall$-quantification over all free variables):

$\_ : \mathsf{pat\_tp}\ [P \xrightarrow{v} Q]\ \mathsf{fvar}\ P \xrightarrow{v} Q$
$\_ : \mathsf{pat\_tp}\ []\ ()\ 1$
$\_ : \mathsf{pat\_tp}\ \Delta_1\ p_1\ P \rightarrow \mathsf{pat\_tp}\ \Delta_2\ p_2\ Q \rightarrow$
$\qquad \mathsf{pat\_tp}\ (\Delta_1 +\!\!+ \Delta_2)\ (p_1, p_2)\ P \otimes Q$
$\_ : \mathsf{pat\_tp}\ \Delta\ p\ P \rightarrow \mathsf{pat\_tp}\ \Delta\ (\mathsf{inl}\ p)\ P \oplus Q$
$\_ : \mathsf{pat\_tp}\ \Delta\ p\ Q \rightarrow \mathsf{pat\_tp}\ \Delta\ (\mathsf{inr}\ p)\ P \oplus Q$

$\_ : \mathsf{pat\_tp}\ []\ \mathsf{t}\ 2$
$\_ : \mathsf{pat\_tp}\ []\ \mathsf{f}\ 2$

Now, the syntax of the language is defined through four mutually inductive types val, fnc, sub, and exp, with the following constructors:

$\mathsf{Value} : \mathsf{pat} \rightarrow \mathsf{sub} \rightarrow \mathsf{val}$

$\mathsf{Lam} : (\mathsf{pat} \rightarrow \mathsf{exp}) \rightarrow \mathsf{fnc}$
$\mathsf{Id} : \mathsf{fnc}$
$\mathsf{IdVar} : \mathsf{index} \rightarrow \mathsf{fnc}$

$\mathsf{Subst} : \mathsf{list\ fnc} \rightarrow \mathsf{sub}$

$\mathsf{Return} : \mathsf{val} \rightarrow \mathsf{exp}$
$\mathsf{Comp} : \mathsf{fnc} \rightarrow \mathsf{index} \rightarrow \mathsf{val} \rightarrow \mathsf{exp}$
$\mathsf{AppV} : \mathsf{fnc} \rightarrow \mathsf{val} \rightarrow \mathsf{exp}$
$\mathsf{AppE} : \mathsf{fnc} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp}$
$\mathsf{Fail} : \mathsf{exp}$

As promised, fnc contains maps from patterns to expressions, embedded through the constructor $\mathsf{Lam} : (\mathsf{pat} \rightarrow \mathsf{exp}) \rightarrow \mathsf{fnc}$. Note that this is a positive definition (and thus acceptable in Coq) because the type pat was already defined—as opposed to, say, the definition $\mathsf{Lam}' : (\mathsf{val} \rightarrow \mathsf{exp}) \rightarrow \mathsf{fnc}$ (which would be illegal in Coq). On the other hand, Coq requires maps $\mathsf{pat} \rightarrow \mathsf{exp}$ to be *total,* so to simulate partial maps we add an expression $\mathsf{Fail} : \mathsf{exp}$, which can be read as "undefined" or "stuck".

Following the representation of linear contexts as unlabelled lists of hypotheses, a substitution is just an unlabelled list of functions, while the expression $\mathsf{Return}\ V$ makes explicit the implicit inclusion of values into expressions. Otherwise, the constructors are all straightforward transcriptions of terms of focused λ-calculus.

In the following examples, we abbreviate $\mathsf{Value}\ p\ (\mathsf{Subst}\ [])$ by $\ulcorner p \urcorner$, and $\mathsf{Return}\ (\ulcorner p \urcorner)$ by $\lceil p \rceil$.

EXAMPLE 4. The Coq encodings of $\mathsf{and}, \mathsf{not} : \mathsf{fnc}$ are:

$$\mathsf{and} = \mathsf{Lam}\begin{pmatrix} (\mathsf{t},\mathsf{t}) & \mapsto & \lceil \mathsf{t} \rceil \\ (\mathsf{t},\mathsf{f}) & \mapsto & \lceil \mathsf{f} \rceil \\ (\mathsf{f},\mathsf{t}) & \mapsto & \lceil \mathsf{f} \rceil \\ (\mathsf{f},\mathsf{f}) & \mapsto & \lceil \mathsf{f} \rceil \\ \_ & \mapsto & \mathsf{Fail} \end{pmatrix}$$

$$\mathsf{not} = \mathsf{Lam}(\mathsf{t} \mapsto \lceil \mathsf{f} \rceil \mid \mathsf{f} \mapsto \lceil \mathsf{t} \rceil \mid \_ \mapsto \mathsf{Fail})$$

These definitions make use of Coq's built-in pattern-matching facilities to in order to pattern-match on pats. ■

EXAMPLE 5. The encoding of $\mathsf{table1} : \mathsf{fnc}$ makes careful use of de Bruijn indexes:

$$\mathsf{table1} = \mathsf{Lam}\begin{pmatrix} \mathsf{fvar} \mapsto & \mathsf{Comp}\ (\mathsf{Lam}(b_1 \mapsto \\ & \mathsf{Comp}\ (\mathsf{Lam}(b_2 \mapsto \lceil (b_1, b_2) \rceil)) \\ & 1.0\ \ulcorner \mathsf{f} \urcorner))\ 0.0\ \ulcorner \mathsf{t} \urcorner \\ \_ \quad\ \mapsto & \mathsf{Fail} \end{pmatrix}$$

In the first call (with value t), the function argument is (the first and only entry) on the top of the stack, so we reference it by 0.0. In the second call, a frame (coincidentally empty) has been pushed in front of the function, so we reference it by 1.0. ■

Now we build the four typing-judgments as mutually inductive type-families, defined as follows (again omitting constructors for the typing rules, and outermost $\forall$-quantifiers):

$\mathsf{val\_tp} : \mathsf{ctx} \rightarrow \mathsf{tp} \rightarrow \mathsf{Prop}$
$\_ : \mathsf{pat\_tp}\ \Delta\ p\ P \rightarrow \mathsf{sub\_tp}\ \Gamma\ \sigma\ \Delta \rightarrow \mathsf{val\_tp}\ \Gamma\ (\mathsf{Value}\ p\ \sigma)\ P$

$\mathsf{fnc\_tp} : \mathsf{ctx} \rightarrow \mathsf{tp} \rightarrow \mathsf{tp} \rightarrow \mathsf{Prop}$
$\_ : (\forall p \forall \Delta. \mathsf{pat\_tp}\ \Delta\ p\ P \rightarrow \mathsf{exp\_tp}\ (\Delta :: \Gamma)\ \phi(p)\ Q)$
$\qquad \rightarrow \mathsf{fnc\_tp}\ \Gamma\ (\mathsf{Lam}\ \phi)\ P\ Q$
$\_ : \mathsf{fnc\_tp}\ \mathsf{Id}\ P\ P$
$\_ : (\#i.j(\Gamma) = (P \xrightarrow{v} Q)) \rightarrow \mathsf{fnc\_tp}\ \Gamma\ (\mathsf{IdVar}\ i.j)\ P\ Q$

$\mathsf{sub\_tp} : \mathsf{ctx} \rightarrow \mathsf{linctx} \rightarrow \mathsf{Prop}$
$\_ : \mathsf{sub\_tp}\ \Gamma\ (\mathsf{Subst}\ [])\ []$
$\_ : \mathsf{fnc\_tp}\ \Gamma\ F\ P\ Q \rightarrow \mathsf{sub\_tp}\ \Gamma\ (\mathsf{Subst}\ \sigma)\ \Delta$
$\qquad \rightarrow \mathsf{sub\_tp}\ \Gamma\ (\mathsf{Subst}\ (F :: \sigma))\ (P \xrightarrow{v} Q :: \Delta)$

$\mathsf{exp\_tp} : \mathsf{ctx} \rightarrow \mathsf{tp} \rightarrow \mathsf{Prop}$
$\_ : \mathsf{val\_tp}\ \Gamma\ V\ P \rightarrow \mathsf{exp\_tp}\ \Gamma\ (\mathsf{Return}\ V)\ P$
$\_ : (\#i.j(\Gamma) = (P \xrightarrow{v} Q)) \rightarrow \mathsf{val\_tp}\ \Gamma\ V\ P \rightarrow \mathsf{fnc\_tp}\ \Gamma\ F\ Q\ R$
$\qquad \rightarrow \mathsf{exp\_tp}\ \Gamma\ (\mathsf{Comp}\ F\ i.j\ V)\ R$
$\_ : \mathsf{val\_tp}\ \Gamma\ V\ P \rightarrow \mathsf{fnc\_tp}\ \Gamma\ F\ P\ Q \rightarrow \mathsf{exp\_tp}\ \Gamma\ (\mathsf{AppV}\ F\ V)\ Q$
$\_ : \mathsf{exp\_tp}\ \Gamma\ E\ P \rightarrow \mathsf{fnc\_tp}\ \Gamma\ F\ P\ Q \rightarrow \mathsf{exp\_tp}\ \Gamma\ (\mathsf{AppE}\ F\ E)\ Q$

Again, these definitions are a direct transcription of the typing rules in Figure 5, including the higher-order rule for function-typing.

Finally, to encode the operational semantics, we first define the different substitution operations:

$\mathsf{sub\_val} : \mathsf{nat} \rightarrow \mathsf{sub} \rightarrow \mathsf{val} \rightarrow \mathsf{val}$
$\mathsf{sub\_fnc} : \mathsf{nat} \rightarrow \mathsf{sub} \rightarrow \mathsf{fnc} \rightarrow \mathsf{fnc}$
$\mathsf{sub\_sub} : \mathsf{nat} \rightarrow \mathsf{sub} \rightarrow \mathsf{sub} \rightarrow \mathsf{sub}$
$\mathsf{sub\_exp} : \mathsf{nat} \rightarrow \mathsf{sub} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp}$

These are defined by (mutual) structural induction on the term being substituted into, essentially as in Figure 6, but with a bit of extra reasoning about de Bruijn indices. The extra nat argument is a frame pointer to the linear context Δ being substituted for, and is used as follows in the IdVar case (and analogously in the Comp case):

$$\mathsf{sub\_fnc}\ i\ \sigma\ (\mathsf{IdVar}\ i'.j) = \begin{cases} \#j(\sigma) & i = i' \\ \mathsf{IdVar}\ i'.j & i > i' \\ \mathsf{IdVar}\ (i'-1).j & i < i' \end{cases}$$

We can then define the transition relation as an inductive family $\mathsf{step} : \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{Prop}$, with the following rules:

$\_ : \mathsf{step}\ (\mathsf{AppV}\ (\mathsf{Lam}\ \phi)\ (\mathsf{Value}\ p\ \sigma))\ (\mathsf{sub\_exp}\ 0\ \sigma\ \phi(p))$
$\_ : \mathsf{step}\ (\mathsf{AppV}\ \mathsf{Id}\ V)\ (\mathsf{Return}\ V)$
$\_ : \mathsf{step}\ E\ E' \rightarrow \mathsf{step}\ (\mathsf{AppE}\ F\ E)\ (\mathsf{AppE}\ F\ E')$
$\_ : \mathsf{step}\ (\mathsf{AppE}\ F\ (\mathsf{Return}\ V))\ (\mathsf{AppV}\ F\ V)$

These mirror the rules in Figure 6, with one additional rule for the (formerly implicit) transition from composition to reduction after the expression argument has been reduced to a value.

Finally, we define a predicate $\mathsf{terminal} : \mathsf{exp} \to \mathsf{Prop}$ and assert $\_ : \mathsf{terminal}\ (\mathsf{Return}\ V)$. Given these definitions, we can state the preservation and progress theorems:

$\mathsf{preservation} : \mathsf{exp\_tp}\ \Gamma\ E\ P \to \mathsf{step}\ E\ E' \to \mathsf{exp\_tp}\ \Gamma\ E'\ P$
$\mathsf{progress} : \mathsf{exp\_tp}\ []\ E\ P \to (\mathsf{terminal}\ E \vee \exists E'.\mathsf{step}\ E\ E')$

Both theorems have short proofs in Coq, constructed using the tactic language. As in the paper proof, the preservation theorem relies on the substitution principle, which in turn relies on weakening. Both substitution and weakening require establishing a few trivial facts about arithmetic, lists, and de Bruijn indices. This (about 140 lines to prove the trivial lemmas, followed by about 230 lines to prove weakening and substitution, much of it dealing simply with the coding of mutual induction principles in Coq) is the main source of bureaucracy in the Coq formalization, which otherwise follows our informal presentation very closely.

3.4 Recursion and recursive types

We have seen how focusing the λ-calculus gives logical explanations for notions such as pattern-matching and evaluation order, which are typically seen as extra-logical. Once we have this analysis, we can extend the language in a fairly open-ended way without modifying the logical core. In this section, we will consider two particularly easy extensions: recursion and recursive types. For recursive functions, we add one typing rule and one evaluation rule:

$$\frac{\Gamma, f : P \overset{v}{\to} Q \vdash F : P > Q}{\Gamma \vdash \mathsf{fix}\, f.F : P > Q} \qquad (\mathsf{fix}\, f.F)(V) \leadsto ([\mathsf{fix}\, f.F/f]F)(V)$$

These rules can be transcribed directly (modulo de Bruijn indices) into Coq:

$\_ : \mathsf{fnc\_tp}\ (\Gamma, [P \overset{v}{\to} Q])\ F\ P\ Q \to \mathsf{fnc\_tp}\ \Gamma\ (\mathsf{fix}\ F)\ P\ Q$
$\_ : \mathsf{step}\ (\mathsf{AppV}\ (\mathsf{fix}\ F)\ V)$
$\qquad\qquad (\mathsf{AppV}\ (\mathsf{sub\_fnc}\ 0\ (\mathsf{Subst}\ [\mathsf{fix}\ F])\ F)\ V)$

To verify the safety of this extension, we need only localized checks: one extra case each in the proofs of weakening, substitution, preservation, and progress.

EXAMPLE 6. We define $\mathsf{loop} : 1 > 1 = \mathsf{fix}\ f.\lambda().f()$. Then $\mathsf{loop}() \leadsto (\lambda().\mathsf{loop}())() \leadsto \mathsf{loop}() \leadsto \ldots$. ■

For recursive types, we add a single *pattern*-typing rule:

$$\frac{\Delta \Rightarrow p : [\mu X.P/X]P}{\Delta \Rightarrow \mathsf{fold}(p) : \mu X.P}$$

To add general μ-types to our Coq formalization, we would have to introduce the additional bureaucracy of type substitution. On the other hand, for particular recursive types (such as those considered below) we can directly transcribe their pattern-typing rules. And these rules suffice: we do not have to extend or modify any other aspect of the type system or operational semantics. The machinery of focusing and higher-order abstract syntax gives us the value-forming rules and pattern-matching on recursive types "for free". In particular, our proof of type safety (both on paper and in the Coq formalization) needs absolutely no modification, since it references the pattern-typing judgment uniformly.

EXAMPLE 7. In this example, we consider natural numbers $\mathsf{Nat} = \mu X.1 \oplus X$, defined by two pattern-typing rules:

$$\frac{}{\cdot \Rightarrow \mathsf{Z} : \mathsf{Nat}} \qquad \frac{\cdot \Rightarrow p : \mathsf{Nat}}{\cdot \Rightarrow \mathsf{S}\, p : \mathsf{Nat}}$$

We can encode the plus function like so:

$$\mathsf{plus} = \mathsf{fix}\, f.\lambda \begin{pmatrix} (m, \mathsf{Z}) & \mapsto & m \\ (m, \mathsf{S}\, n) & \mapsto & (\lambda n'.\mathsf{S}\, n')f(m,n) \end{pmatrix}$$

For instance, $\mathsf{plus}\ (\mathsf{S(S\ Z)}, \mathsf{S\ Z}) \leadsto^* \mathsf{S(S(S\ Z))}$. To verify that $\mathsf{plus} : \mathsf{Nat} \otimes \mathsf{Nat} > \mathsf{Nat}$, we must check that for any $\mathsf{Nat} \otimes \mathsf{Nat}$ pattern, there is a corresponding Nat-typed branch of the function. This is easily seen to be true, since all $\mathsf{Nat} \otimes \mathsf{Nat}$ patterns have the form (m, Z) or $(m, \mathsf{S}\, n)$. ■

EXAMPLE 8. Consider a domain $\mathsf{D} = \mu X.1 \oplus \mathsf{Nat} \oplus (X \overset{v}{\to} X)$:

$$\frac{}{\cdot \Rightarrow \mathsf{U} : \mathsf{D}} \qquad \frac{\cdot \Rightarrow p : \mathsf{Nat}}{\cdot \Rightarrow \mathsf{N}\, p : \mathsf{D}} \qquad \frac{}{f : \mathsf{D} \overset{v}{\to} \mathsf{D} \Rightarrow \mathsf{F}\, f : \mathsf{D}}$$

We define a function $\mathsf{app} : \mathsf{D} \otimes \mathsf{D} > \mathsf{D}$, which tries to apply the first argument to the second (and returns U if the first argument is not a function):

$$\mathsf{app} = \lambda \begin{pmatrix} (F\, f, d) & \mapsto & \mathsf{id}(f(d)) \\ (\_, d) & \mapsto & \mathsf{U} \end{pmatrix}$$

For instance, $\mathsf{app}\ (\mathsf{F\ id}, V) \leadsto \mathsf{id}(\mathsf{id}(V)) \leadsto \mathsf{id}(V) \leadsto V$. ■

While these examples illustrate the simplicity of higher-order syntax for pattern-matching on recursive types, they also raise some subtle theoretical questions. A careful reader might have noticed that there is another way of defining the plus function in Coq: rather than explicitly using the fix operator, we could use Coq's built-in Fixpoint mechanism to define a map $\mathsf{plus\_pat} : \mathsf{pat} \to \mathsf{pat} \to \mathsf{pat}$ computing the sum of two Nat patterns, and then define

$$\mathsf{plus}^* : \mathsf{fnc} = \mathsf{Lam}((m,n) \mapsto \ulcorner \mathsf{plus\_pat}\ m\ n \urcorner \mid \_ \mapsto \mathsf{Fail})$$

Strictly speaking, plus^* is an "exotic term", i.e., does not represent a term of concrete syntax (Despeyroux et al. 1995), since it corresponds to a function defined by infinitely many pattern-branches. Operationally, it computes the sum of two numbers in a single step of evaluation, whereas plus computes it in multiple steps (linear in n). Nonetheless, plus and plus^* are observationally equivalent. We conjecture that this is always the case, and that any term definable in the Coq encoding of focused λ-calculus with recursive types is observationally equivalent to a term of concrete syntax using explicit recursion. Yet, even if this conjecture holds, it is an interesting question whether there is a principled way to adapt the HOAS encoding to eliminate terms such as plus^* altogether.

Proof-theoretically speaking, we can put it this way: for some recursive types P, establishing $\Gamma; P \vdash Q$ requires an infinitely wide derivation in the focusing system.[6] Conversely, for other recursive types, the identity principle $\Gamma; P \vdash P$ requires a derivation that is infinitely *deep*. In particular, the subformula relationship may not be well-founded (e.g., $\mathsf{D} \overset{v}{\to} \mathsf{D}$ is a subformula of D). This is not really an issue for our programming language: rather than attempting an infinite derivation, we can simply invoke the internalized identity principle. More fundamentally, though, we can give these derivations a coinductive reading—this becomes particularly significant if we want to extend the language and incorporate *subtyping* through an identity-coercion interpretation, as explored by Brandt and Henglein (1998).

[6] E.g., for Nat we essentially have the ω-rule (Buchholz et al. 1981).

4. Related work

This paper is by no means the first to propose a logical explanation for pattern-matching or explicit substitutions. Recently, Nanevski et al. (2007) and Pientka (2008) offer a judgmental explanation for explicit substitutions in a modal type theory. Methodologically their work is close to ours, but their development is rather different since they seek to understand the connection between explicit substitutions and *meta-variables* (as used, e.g., in logical frameworks and staged computation), rather than pattern-matching. Cerrito and Kesner (2004) give an interpretation for both nested patterns and explicit substitutions in sequent calculus. It seems the difficulty with taking the unfocused sequent calculus as a starting point, though, is that it suffers from a "lack of judgments"—to explain pattern-matching Cerrito and Kesner must introduce additional scaffolding beyond the Curry-Howard isomorphism. For example, to obtain a well-behaved language with substitution and subject reduction, they must annotate the single cut rule of sequent calculus as three different typing rules, and add another typing rule (*app*) with little proof-theoretic motivation. In contrast, every typing rule we gave in Section 3.1 was either a direct annotation of a logical rule in Section 2.2, or else internalized one of the cut or identity principles.

An additional byproduct of our use of focusing as a logical foundation is that the extracted language has explicit evaluation order. As mentioned in the Introduction, this connection has been explored before by various people, at first with only a loose tie to linear logic (Curien and Herbelin 2000; Selinger 2001; Wadler 2003), but later with an explicit appeal to polarity and focusing (Laurent 2005; Dyckhoff and Lengrand 2006). From this line of work, our main technical innovation is the uniform treatment of the positive connectives through pattern-matching, which considerably simplifies previous formalisms while allowing us to consider a rich set of connectives. Our approach is loosely inspired by that of Girard (2001).

In a short but prescient paper, Coquand (1992) examines pattern-matching as an alternative to the usual elimination rules in the framework of Martin-Löf's type theory, and concludes with an offhand remark, "From a proof-theoretic viewpoint, our treatment can be characterized as fixing the meaning of a logical constant by its introduction rules". We have seen how this interpretation arises naturally out of focusing for the positive connectives, although how to extend our approach to the dependently-typed case remains an important open question. Elsewhere, we explore the dual interpretation for the negative connectives (and lazy evaluation), tying the unified analysis to Michael Dummett's examination of the justification of logical laws (Dummett 1991; Zeilberger 2007).

Acknowledgments

Special thanks to Frank Pfenning for his invaluable guidance during the development of this work, and for suggestions on improving its presentation. I would also like to thank Bob Harper, Neel Krishnaswami, Peter Lee, William Lovas, Dan Licata, Jason Reed, Rob Simmons, and other members of the CMU PL group for lively discussions, and the POPL reviewers for their helpful comments. Finally, I am grateful to the `coq-club` mailing list for useful advice on Coq, and particularly to Xavier Leroy for explaining a cool trick for coding mutual induction principles.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1):131–163, 2001.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equaility and subtyping. *Fundamenta Informaticae*, 20:1–24, 1998.

W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*. Springer-Verlag, 1981.

Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323(1-3):71–127, 2004.

The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.1*. INRIA, 2006. `http://coq.inria.fr/doc/main.html`.

Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.

Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *LNCS 389*. Springer-Verlag, 1989.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP '00: Proceedings of the SIGPLAN International Conference on Functional Programming*, pages 233–243. 2000.

Nicolaas G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

Michael Dummett. *The Logical Basis of Metaphysics*. The William James Lectures, 1976. Harvard University Press, Cambridge, Massachusetts, 1991. ISBN 0-674-53785-8.

Roy Dyckhoff and Stephane Lengrand. LJQ: A strongly focused calculus for intuitionistic logic. In *Proceedings of the Second Conference on Computability in Europe*, 2006.

Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

Jean-Yves Girard. On the unity of logic. *Annals of pure and applied logic*, 59(3):201–217, 1993.

Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *CSL '94: Proceedings of the 8th International Workshop on Computer Science Logic*, 1995.

Jacob M. Howe. *Proof search issues in some non-classical logics*. PhD thesis, University of St Andrews, December 1998. URL `http://www.cs.kent.ac.uk/pubs/1998/946`. Available as University of St Andrews Research Report CS/99/1.

Steven C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, NJ, 1952.

Olivier Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, March 2002.

Olivier Laurent. Classical isomorphisms of types. *Mathematical Structures in Computer Science*, 15(5):969–1004, October 2005.

Per Martin-Löf. *Hauptsatz* for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, Amsterdam, 1971. North Holland.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. URL `http://www.hf.uio.no/filosofi/njpl/vol1no1/meaning/meaning.html`.

Dale Miller and Chuck Liang. Focusing and polarization in intuitionistic logic. In *CSL '07: Proceedings of the 21st International Workshop on Computer Science Logic*. 2007.

Eugenio Moggi. Notions of computation and monads. *Information and Compution*, 93(1):55–92, 1991.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI '88: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL '08: Proceedings of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume Cambridge Tracts in Theoretical Computer Science 43. Cambridge University Press, 1996.

Twelf 2007. *The Twelf Project*, 2007. `http://twelf.plparty.org/`.

Philip Wadler. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the SIGPLAN International Conference on Functional Programming*, pages 189–201, 2003.

Philip Wadler. A taste of linear logic. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1993.

Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 2007. To appear in a special issue on "Classical Logic and Computation".

An Introduction to Higher Order Logic

Conden Chao*

Logic Frontier Problems Seminar II, February 28, 2017

Abstract

We mainly talked about the following topics: syntax and Tarskian semantics of (monadic) second order logic; advantages and disadvantages of (monadic) second order logic; concepts of k'th(finite and transfinite) order logic; formalisms for second(k'th and finite) order logic; some ontology considerations about higher order logic; some meta-theory but non-logical applications of higher order logic in practice.

Key Words: second order logic, higher order logic, syntax, semantics

1 Introduction

Higher order logics, long considered by many to be an esoteric subject, are increasingly recognized for their foundational importance and practical usefulness, notably in Theoretical Computer Science.

We'll try to cover most aspects of higher order logic which are attractive to most of us. We'll mainly focus on the following questions:

- What is higher order logic?
- What advantages and disadvantages does higher order logic have contrast with first order logic?
- What effects does higher order logic have in common(possibly, mathematical and logical) practice?
- Also some other questions people may concern.

2 Second Order Logic(SOL)

2.1 Syntax for SOL

Definition 2.1. The symbols for second order language $\mathscr{L}_2$ consists of two parts.

(1) Logical symbols are consists of a set $\mathfrak{V} = \mathfrak{V}_1 \cup \mathfrak{V}_2 \cup \mathfrak{V}_3$ of variables where $\mathfrak{V}_1 = \{x, y, z, \cdots\}$, $\mathfrak{V}_2 = \{f, g, h, \cdots\}$ and $\mathfrak{V}_3 = \{p, q, r, \cdots\}$; a set $\{=\}$ of equality symbol; a set $\{\neg, \wedge\}$ of connectives and a set $\{\forall\}$ of quantifier.

*2014-2018, Department of Philosophy, Peking University. Supervisor: Zhuanghu Liu.

(2) Non-logical symbols are consist of three sets $\mathfrak{C}, \mathfrak{P}, \mathfrak{F}$, where $\mathfrak{C}, \mathfrak{P}, \mathfrak{F}$ are the sets of constant, predicate and function symbols respectively; an arity function $\pi_1 : \mathfrak{P} \cup \mathfrak{F} \to \omega$; a variable arity function $\pi_2 : \mathfrak{V}_2 \cup \mathfrak{V}_3 \to \omega$. Further, we call the set $\text{sig}\mathscr{L}_2 = \mathfrak{C} \cup \mathfrak{P} \cup \mathfrak{F}$ the signature for $\mathscr{L}_2$.

Definition 2.2. The set of terms for $\mathscr{L}_2$ is the smallest set T such that

(1) for any variable $x \in \mathfrak{V}_1$ and $c \in \mathfrak{C}$, the expressions $x, c \in T$;

(2) for any n-ary $f \in \mathfrak{V}_2$, if $\tau_0, \cdots, \tau_{n-1}$ are terms, then $f(\tau_0, \cdots, \tau_{n-1}) \in T$;

(3) for any n-ary $F \in \mathfrak{F}$, if $\tau_0, \cdots, \tau_{n-1}$ are terms, then $F(\tau_0, \cdots, \tau_{n-1}) \in T$.

Definition 2.3. The set of formulas for $\mathscr{L}_2$ is the smallest set L satisfying

(1) if τ_0, τ_1 are terms, then $\tau_0 = \tau_1 \in L$;

(2) for any n-ary $p \in \mathfrak{V}_3$, if $\tau_0, \cdots, \tau_{n-1}$ are terms, then $p(\tau_0, \cdots, \tau_{n-1}) \in L$;

(3) for any n-ary $P \in \mathfrak{P}$, if $\tau_0, \cdots, \tau_{n-1}$ are terms, then $P(\tau_0, \cdots, \tau_{n-1}) \in L$;

(4) if $\phi \in L$, then $\neg\phi \in L$;

(5) if $\phi, \psi \in L$, then $\phi \wedge \psi \in L$;

(6) if $\phi \in L$ and $x, f, p \in \mathfrak{V}$, then $\forall x\phi, \forall f\phi, \forall p\phi \in L$.

Remark 2.4. Under the usual (classical, extensional) reading of functions and relations, these two concepts are of course reducible to each other: k-ary functions can be viewed as a special kind of $k+1$-ary relations, and k-ary relations can be interpreted by their k-ary characteristic functions. For instance, a formula $\exists f(\cdots X(f(\tau))\cdots)$ is interpreted by $\exists F(\forall x \exists! y F(x,y) \wedge (\cdots(\exists y(F(\tau, y) \wedge X(y))\cdots))$ (F fresh). Conversely, a formula $\forall X(\cdots X(t)\cdots)$ can be interpreted by $\forall f(\cdots(f(\tau) = c)\cdots)$, where c is a fixed constant symbol (and f is fresh). It therefore suffices to formulate second order logic using only relation-variables, or only function variables.

2.2 Tarskian Semantics for SOL

$\mathscr{L}_2$ structures $\mathcal{M} = (M, I)$ are defined the same as first order structures.

Definition 2.5. For any $\mathscr{L}_2$ structure $\mathcal{M} = (M, I)$, the $\mathcal{M}$ assignment is a function $\nu : \mathfrak{V} \to \bigcup_{m<\omega} M^m$ such that $\nu(x) \in M$ for all $x \in \mathfrak{V}_1$, $\nu(f)$ is a function from M^m into M for all $f \in \mathfrak{V}_2$, and $\nu(p) \subseteq M^m$ for all $p \in \mathfrak{V}_3$.

To define the semantics for second order logic, we also need to give the interpretations for second order languages terms.

Definition 2.6. For any term τ and any $\mathcal{M}$-assignment ν, $\overline{\nu}(\tau)$ is defined by recursion:

(1) if $\tau = x$ for some $x \in \mathfrak{V}_1$, then $\overline{\nu}(\tau) = \overline{\nu}(x) = \nu(x)$;

(2) if $\tau = (f(\tau_0, \cdots, \tau_{n-1}))$, then $\overline{\nu}(\tau) = \nu(f)(\overline{\nu}(\tau_0), \cdots, \overline{\nu}(\tau_{n-1}))$.

(3) if $\tau = (F(\tau_0, \cdots, \tau_{n-1}))$, then $\overline{\nu}(\tau) = I(F)(\overline{\nu}(\tau_0), \cdots, \overline{\nu}(\tau_{n-1}))$.

Definition 2.7. For any formula ϕ, any $\mathscr{L}_2$ structure $\mathcal{M} = (M, I)$ and any $\mathcal{M}$-assignment ν, $(\mathcal{M}, \nu) \vDash \phi$ is defined by recursion:

(1) if $\phi = (\tau_0 = \tau_1)$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $\overline{\nu}(\tau_0) = \overline{\nu}(\tau_1)$;

(2) if $\phi = p(\tau_0, \cdots, \tau_{n-1})$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\overline{\nu}(\tau_0), \cdots, \overline{\nu}(\tau_{n-1})) \in \nu(p)$;

(3) if $\phi = P(\tau_0, \cdots, \tau_{n-1})$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\overline{\nu}(\tau_0), \cdots, \overline{\nu}(\tau_{n-1})) \in I(P)$;

(4) if $\phi = \neg\psi$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\mathcal{M}, \nu) \nvDash \psi$;

(5) if $\phi = \psi \wedge \theta$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\mathcal{M}, \nu) \vDash \psi$ and$(\mathcal{M}, \nu) \vDash \theta$;

(6) if $\phi = \forall x\psi$, then $(\mathcal{M}, \nu) \vDash \phi$ iff for all $a \in M$ we have $(\mathcal{M}, \nu_{x\to a}) \vDash \psi$;

(7) if $\phi = \forall f\psi$ where f has arity n, then $(\mathcal{M}, \nu) \vDash \phi$ iff for all n-ary function F from M^n into M, we have $(\mathcal{M}, \nu_{f\to F}) \vDash \psi$;

(8) if $\phi = \forall p\psi$ where p has arity n, then $(\mathcal{M}, \nu) \vDash \phi$ iff for all n-ary predicate P on M^n, we have $(\mathcal{M}, \nu_{p\to P}) \vDash \psi$.

2.3 Expressive Power

Second order languages have a stronger expressive power than first order languages, for example:

Theorem 2.8. *The collection of finite structures is not first order definable (by a set of sentences or a single sentence), but it is definable already by a single second order sentence.*

Proof. (1) By Compactness.

(2) A statement that holds exactly of the finite structures is 'every injection is a surjection', i.e.,

$$\sigma_{\text{finite}} =_{df} \forall f(\text{Inj}(f) \to \text{Surj}(f)),$$

where $\text{Inj}(f) =_{df} \forall x, y(f(x) = f(y) \to x = y)$ and $\text{Surj}(f) =_{df} \forall y \exists x f(x) = y$. □

Another example is

Theorem 2.9. $\text{Th}(\mathcal{N})$ *isn't definable in first order arithmetic language but definable in second order arithmetic language.*

3 Monadic Second Order Logic(MSOL)

3.1 Syntax for MSOL

Definition 3.1. The symbols for second order language $\mathscr{L}_{2m}$ consists of two parts.

(1) Logical symbols are consists of a set $\mathfrak{V} = \mathfrak{V}_1 \cup \mathfrak{V}_2$ of variables where $\mathfrak{V}_1 = \{x, y, z, \cdots\}$ and $\mathfrak{V}_2 = \{X, Y, Z, \cdots\}$; a set $\{=\}$ of equality symbol; a set $\{\neg, \wedge\}$ of connectives and a set $\{\forall\}$ of quantifier.

(2) Non-logical symbols are consist of three sets $\mathfrak{C}, \mathfrak{P}, \mathfrak{F}$, where $\mathfrak{C}, \mathfrak{P}, \mathfrak{F}$ are the sets of constant, predicate and function symbols respectively; an arity function $\pi : \mathfrak{P} \cup \mathfrak{F} \to \omega$. Further, we call the set $\text{sig}\mathscr{L}_{2\text{m}} = \mathfrak{C} \cup \mathfrak{P} \cup \mathfrak{F}$ the signature for $\mathscr{L}_{2m}$.

Definition 3.2. The set of terms for $\mathscr{L}_{2m}$ is the smallest set T such that

(1) for any variable $x \in \mathfrak{V}_1$ and $c \in \mathfrak{C}$, the expressions $x, c \in T$;

(2) for any n-ary $F \in \mathfrak{F}$, if $\tau_0, \cdots, \tau_{n-1}$ are terms, then $F(\tau_0, \cdots, \tau_{n-1}) \in T$.

Definition 3.3. The set of formulas for $\mathscr{L}_{2m}$ is the smallest set L satisfying

(1) if τ_0, τ_1 are terms, then $\tau_0 = \tau_1 \in L$;

(2) for any $X \in \mathfrak{V}_2$ and term τ, the expression $X\tau \in L$;

(3) for any n-ary $P \in \mathfrak{P}$, if $\tau_0, \cdots, \tau_{n-1}$ are terms, then $P(\tau_0, \cdots, \tau_{n-1}) \in L$;

(4) if $\phi \in L$, then $\neg\phi \in L$;

(5) if $\phi, \psi \in L$, then $\phi \wedge \psi \in L$;

(6) if $\phi \in L$ and $x, X \in \mathfrak{V}$, then $\forall x\phi, \forall X\phi \in L$.

3.2 Tarskian Semantics for MSOL

$\mathscr{L}_{2m}$ structures $\mathcal{M} = (M, I)$ are defined the same as first order structures.

Definition 3.4. For any $\mathscr{L}_{2m}$ structure $\mathcal{M} = (M, I)$, the $\mathcal{M}$ assignment is a function $\nu : \mathfrak{V} \to M \cup \wp(M)$ such that $\nu(x) \in M$ for all $x \in \mathfrak{V}_1$ and $\nu(X) \in \wp(M)$ for all $X \in \mathfrak{V}_2$.

To define the semantics for monadic second order logic, we also need to give the interpretations for monadic second order languages terms.

Definition 3.5. For any term τ and any $\mathcal{M}$-assignment ν, $\overline{\nu}(\tau)$ is defined by recursion:

(1) if $\tau = x$ for some $x \in \mathfrak{V}_1$, then $\overline{\nu}(\tau) = \overline{\nu}(x) = \nu(x)$;

(2) if $\tau = (F(\tau_0, \cdots, \tau_{n-1}))$, then $\overline{\nu}(\tau) = I(F)(\overline{\nu}(\tau_0), \cdots, \overline{\nu}(\tau_{n-1}))$.

Definition 3.6. For any formula ϕ, any $\mathscr{L}_{2m}$ structure $\mathcal{M} = (M, I)$ and any $\mathcal{M}$-assignment ν, $(\mathcal{M}, \nu) \vDash \phi$ is defined by recursion:

(1) if $\phi = (\tau_0 = \tau_1)$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $\overline{\nu}(\tau_0) = \overline{\nu}(\tau_1)$;

(2) if $\phi = X\tau$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $\overline{\nu}(\tau) \in \nu(X)$;

(3) if $\phi = P(\tau_0, \cdots, \tau_{n-1})$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\overline{\nu}(\tau_0), \cdots, \overline{\nu}(\tau_{n-1})) \in I(P)$;

(4) if $\phi = \neg\psi$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\mathcal{M}, \nu) \nvDash \psi$;

(5) if $\phi = \psi \wedge \theta$, then $(\mathcal{M}, \nu) \vDash \phi$ iff $(\mathcal{M}, \nu) \vDash \psi$ and$(\mathcal{M}, \nu) \vDash \theta$;

(6) if $\phi = \forall x\psi$, then $(\mathcal{M}, \nu) \vDash \phi$ iff for all $a \in M$ we have $(\mathcal{M}, \nu_{x \to a}) \vDash \psi$;

(7) if $\phi = \forall X\psi$, then $(\mathcal{M}, \nu) \vDash \phi$ iff for all $A \in \wp(M)$, we have $(\mathcal{M}, \nu_{X \to A}) \vDash \psi$.

3.3 Meta-theoretical Properties for MSOL

Fact 3.7. *Monadic second order logic has no compactness property.*

Example 3.8. Let $\mathscr{L}^1_{2m}$ be the language with one binary relation symbol $<$ and countable constant symbols $c_i (i < \omega)$, and let σ_{SLO} be the sentence which says that $<$ is a strict linear ordering. Define

$$\sigma_{\text{WF}} =_{df} \forall X(\exists y\, Xy \to \exists y[Xy \wedge \neg \exists z(Xz \wedge z < y)]),$$

and let $\sigma_{\text{WO}} =_{df} \sigma_{\text{SLO}} \wedge \sigma_{\text{WF}}$. Consider the set

$$\Delta = \{\sigma_{\text{WO}}\} \cup \{c_{i+1} < c_i \mid i < \omega\}.$$

Clearly every finite subset of Δ has a model, i.e., $(m, <, 0, \cdots, m-1)$ for some $m < \omega$, but Δ has no models.

Fact 3.9. *Monadic second order logic has no downward Löwenheim-Skolem property.*

Example 3.10. Let $\mathscr{L}^2_{2m}$ be the language with only one relation symbols $<$, and let σ_{DLO} be the sentence which says $<$ is a dense linear ordering without endpoints. Define

$$\sigma_{\text{CP}} =_{df} \forall X\Big[\exists y\, Xy \wedge \exists z \forall y(Xy \to y \leq z) \to \exists z\big(\forall y(Xy \to y \leq z) \wedge \forall w[\forall y(Xy \to y \leq w) \to z \leq w]\big)\Big].$$

Then set $\sigma_{\text{CDLO}} =_{df} \sigma_{\text{DLO}} \wedge \sigma_{\text{CP}}$. It's easy to see that models of σ_{CDLO} are exactly structures in which $<$ is a *complete dense linear ordering without endpoints*, and that σ_{CDLO} has models of size 2^ω and larger but none smaller models.

Fact 3.11. *Monadic second order logic has no upward Löwenheim-Skolem property.*

Example 3.12. Let $\mathscr{L}^3_{2m}$ be any language which includes a unary symbol S and a constant symbol 0. Define

$$\sigma_{\text{IA}} =_{df} \forall X[(X0 \wedge \forall y(Xy \to XSy)) \to \forall y\, Xy]$$

Clearly σ_{IA} has models with any finite or countable size, but has no uncountable models.

Further,

- the set of valid first order sentences is recursively enumerable, while the set of valid second order sentences isn't;
- the unification problem is decidable for first order logic, while not for second order logic;
- first order logic satisfies *Beth's definability property*, *Keisler-Shelah Theorem* and *0-1 law*, while second order logic doesn't.

More about this see [1, pp.29-31]. Therefore, many people insist on avoiding higher order logic.

4 Higher Order Logic(HOL)

Higher order logic is an extension of second order logic in which quantification is used over higher order ≥ 2 predicates and functions.

4.1 Types

Definition 4.1. Types are syntactic expressions recursively generated by:

(1) ι is a type;

(2) if $\upsilon_0, \cdots, \upsilon_{k-1}$ $(k \geq 0)$ are types then $\upsilon = (\upsilon_0, \cdots, \upsilon_{k-1})$ is a type.

In particular, the type $()$ is denoted as $\varnothing$, and if $\upsilon_0 = \cdots = \upsilon_{k-1} = \eta$ then $(\upsilon_0, \cdots, \upsilon_{k-1})$ is denoted as η^k.

The language of higher order logic, for each type υ, variables of type υ and quantifiers over them.

We intend ι to denote the set of individuals, i,e., structures elements, and $(\upsilon_0, \cdots, \upsilon_{k-1})$ the set of k-ary relations between k objects of types $\upsilon_0, \cdots, \upsilon_{k-1}$. Exactly,

Definition 4.2. For a set M and any type υ, the set M_υ is defined by recursion

$$\begin{aligned} M_\iota &= M, \\ M_{(\upsilon_0, \cdots, \upsilon_{k-1})} &= \wp(\textstyle\prod_{i=0}^{k-1} M_{\upsilon_i}) = \wp(M_{\upsilon_0} \times \cdots \times M_{\upsilon_{k-1}}). \end{aligned}$$

Definition 4.3. Higher order logic has υ-terms for each type υ as follows

(1) a variable of type υ is a term of type υ;

(2) if F is a k-ary function symbol, and $\tau_0, \cdots, \tau_{k-1}$ are terms of type ι, then $F(\tau_0, \cdots, \tau_{k-1})$ is a term of type ι;

(3) if P is a k-ary predicate symbol, then P is a term of type ι^k;

(4) if t is a term of type $(\tau_0, \cdots, \tau_{k-1})$, and $t_0, \cdots, t_{k-1}$ are terms of types $\upsilon_0, \cdots, \upsilon_{k-1}$ respectively, then $t(t_0, \cdots, t_{k-1})$ is a term of type $\varnothing$.

The terms of type $\varnothing$ are the atomic formulas, and compound formulas are generated as in first order and second order logic.

The semantics of formulas of higher order logic is defined formally like for second order logic, except that an assignment over a structure with universe M maps variables of type υ to objects in M_υ.

4.2 Finite Order Logic

Definition 4.4. Types naturally fall into *orders* which could be define by recursion:

$$\begin{aligned} \text{order}(\iota) &= 1; \\ \text{order}((\upsilon_0, \cdots, \upsilon_{k-1})) &= 1 + \max\{\text{order}(\upsilon_0), \cdots, \text{order}(\upsilon_{k-1})\}. \end{aligned}$$

Definition 4.5. The fragment of higher order logic in which of types variables are of order $\leq k$ is dubbed *k'th order logic*, and The fragment of higher order logic in which of types variables are of finite order is dubbed *finite order logic*.

The construction of types can be extended to allow types whose orders are transfinite ordinals. For example, let ω consist of all objects of finite type. Then (ω, ω) is the type of binary relations between objects of finite types.

5 Formalisms for Higher Order Logic

5.1 Formalisms have no Gödel's completeness

Theorem 5.1. *The set of valid second order formulas is not definable (under canonical numeric coding) by a second order formula in the language of arithmetic. It is therefore not recursively enumerable; in particular, there is no effective formalism whose theorems are precisely the valid second order formulas.*

So second order logic does not have a complete deductive calculus. But it does have sound deductive calculi that are logically natural, powerful, of metamathematical interest, and suitable for the formalization of much of mathematics, of computer science, and of cognitive science.

5.2 Basic Formalisms

A natural formalism for the relational variant of second order logic uses the usual axioms for first order logic, with quantifier rules applying to relational variables as well as individual variables (we give a precise formulation momentarily), with the stipulation that the range of relation variables includes at least all the relations definable by the formulas of the language(in which there are only predicate variables but nor fucntion variables).

- Quantification rules for the predicate variables are:
  - the axiom schema of universal instantiation: $(\forall p\phi) \to [q/p]\phi$ where $\pi_2(p) = \pi_2(q)$.
  - the inference rule of universal generalization: if $\psi \to [q/p]\phi$, then $\psi \to \forall p\phi$ where q is not free in ψ.
- This stipulation is a form of the Comprehension Principle of Set Theory, and is formally rendered by the schema: $\exists p \forall x_1 \cdots x_n (p(x_1, \cdots, x_n) \leftrightarrow \phi)$ where $n \geq 0$, p is an n-ary predicate variable, and ϕ is a second order formula in which p isn't free.

Now we give an outline for the formalisms for second order logic for which the language has function predicate variables at the same time. For the axioms, there are five kinds in total:

- Propositional Axioms:
  - $\varphi \to (\psi \to \varphi)$;
  - $(\varphi \to \psi \to \vartheta) \to (\varphi \to \psi) \to (\varphi \to \vartheta)$;
  - $(\neg\varphi \to \psi) \to (\neg\varphi \to \neg\psi) \to \varphi$.
- Substitution Axioms:
  - $\forall x\varphi \to [\tau/x]\varphi$ where $\varphi(x;\tau)$ is a free substitution;
  - $\forall f\varphi \to [g/f]\varphi$ where $\pi_2(f) = \pi_2(g)$;
  - $\forall p\varphi \to [q/p]\varphi$ where $\pi_2(f) = \pi_2(g)$.
- Distribution Axioms:
  - $\forall x(\varphi \to \psi) \to \forall x\varphi \to \forall x\psi$;

- $\forall f(\varphi \to \psi) \to \forall f\varphi \to \forall f\psi$;
- $\forall p(\varphi \to \psi) \to \forall p\varphi \to \forall p\psi$.

- Equality Axioms:
  - $\tau{=}\tau$;
  - $\tau_0{=}\sigma_0 \to \cdots \to \tau_{n-1}{=}\sigma_{n-1} \to f(\tau_0, \cdots, \tau_{n-1}){=}f(\sigma_0, \cdots, \sigma_{n-1})$;
  - $\tau_0{=}\sigma_0 \to \cdots \to \tau_{n-1}{=}\sigma_{n-1} \to F(\tau_0, \cdots, \tau_{n-1}){=}F(\sigma_0, \cdots, \sigma_{n-1})$;
  - $\tau_0{=}\sigma_0 \to \cdots \to \tau_{n-1}{=}\sigma_{n-1} \to p(\tau_0, \cdots, \tau_{n-1}) \to p(\sigma_0, \cdots, \sigma_{n-1})$;
  - $\tau_0{=}\sigma_0 \to \cdots \to \tau_{n-1}{=}\sigma_{n-1} \to P(\tau_0, \cdots, \tau_{n-1}) \to P(\sigma_0, \cdots, \sigma_{n-1})$.
- Comprehension Axioms:
  - $\varphi \to \forall x\varphi$ where x is not free in φ;
  - $\varphi \to \forall f\varphi$ where f is not free in φ;
  - $\varphi \to \forall p\varphi$ where p is not free in φ;
  - $\exists f\forall x_1 \cdots x_n\exists! y(f(x_1, \cdots, x_n) = y \leftrightarrow \phi)$ where $n \geq 0$, f is an n-ary predicate variable, and ϕ is a second order formula in which f isn't free.
  - $\exists p\forall x_1 \cdots x_n(p(x_1, \cdots, x_n) \leftrightarrow \phi)$ where $n \geq 0$, p is an n-ary predicate variable, and ϕ is a second order formula in which p isn't free.
  - $\forall x_0 \cdots \forall x_{n-1}\varphi$ where φ is an axiom with one of **all the above forms.**

For the inference rules we have

- Generalization Rules:
  - if $\psi \to [\tau/x]\phi$, then $\psi \to \forall x\phi$ where $[\tau/x]\phi$ is a free substitution;
  - if $\psi \to [g/f]\phi$, then $\psi \to \forall f\phi$ where g is not free in ψ;
  - if $\psi \to [q/p]\phi$, then $\psi \to \forall p\phi$ where q is not free in ψ;
- Modus Ponens: if $\vdash \varphi$ and $\vdash \varphi \to \psi$, then we have $\vdash \psi$.

We note that for the generalization rules are not necessary.

Formalisms for k-th order logic are the same as second order logic except the comprehension principle which is called comprehension for all types of order $\leq k$.

If $x_0, \cdots, x_{k-1}$ are variable of type $\upsilon_0, \cdots, \upsilon_{k-1}$ respectively, $\upsilon = (\upsilon_0, \cdots, \upsilon_{k-1})$ and p is a variable of type υ, then comprehension at type υ is the schema: $\exists p\forall x_1 \cdots x_n(p(x_1, \cdots, x_n) \leftrightarrow \phi)$ where p isn't free.

Formalisms for finite order logic are the same as second order logic except the comprehension principle which is called comprehension for all types.

6 Ontology Considerations

6.1 Is second order logic truly a logic?

Is second order logic truly a logic?

On a technical level the answer is trivially positive. From a model-theoretic viewpoint second order logic is merely one of many possible logics since its syntax and semantics are similar to other logics such as first order logic.

From a philosophically ontological angle the answer is less clear.

Quine(1970): it's a mathematical theory rather than a logic.

- From a philosophical viewpoint, we might wish to reserve the term 'logic' to a priori concepts and truths, ones that do not depend on experience and observation.
- Quine's test: the demarcation between logic and mathematics is determined by *ontological neutrality*, that is, on not assuming the existence of certain objects and structures.In particular, if the notion of *infinity* is delineated by a formalism, then that formalism is mathematical rather than logical. Notably that as above 'infinity' could be defined by $\neg\sigma_{\text{finite}}$ a second order sentence.
- Lindström's test: compactness and downward Löwenheim-Skolem property. A landmark theorem of Lindström states, roughly, that Out of all logics, first order logic is characterized as the maximal logic that is both compact and satisfies the downward Löwenheim-Skolem property.
- From Quine's viewpoint, these two characteristics of first order logic are indeed litmus tests for being a logic: compactness is the failure to distinguish between the finite and the infinite, and downwards Löwenheim-Skolem is the failure to distinguish between different infinities.

Hazen objects Quine.

- David Lewis (quoted by Hazen, 1989) has argued that even monadic third order logic is ontologically neutral.
- Hazen showed that that formalism suffices to interpret all of second order logic.

Girard supports Quine.

- Girard's proof-theoretic criterion: a true logic must be amenable to a cut-free sequential calculus without axioms, which satisfies the subformula property.
- The underlying intuition is that neither axioms nor rules should allow 'communication' between formulas other than by rules that explicate the logical constants in isolation.
- This criterion is clearly related to Quine's ontological neutrality. Indeed, even relatively weak fragments of second order logic fail Girard's criterion.

6.2 Slipping first order to second order logic

Meta-mathematics of first order logic are in fact second order supported by the following two points:

- Hilbert and Ackermann discovered that the most basic notions of the meta-mathematics of first order logic are second order.
  - To say that a formula ϕ is valid is to say that it is true in all interpretations, a statement involving a universal quantification over universes as well as over the relations interpreting predicate letters in ϕ.
  - So the notion of 'arbitrary relation' is implicit already in first order logic (though not in first order languages used to describe and prove properties of particular first order structures).
- Moreover, there is a logical construct that seems even more ontologically benign than relational quantification, and which yields nonetheless the full expressive power of second order logic, namely *partially order quantifiers*.
  - For instance, consider the formula $\forall x \exists y \forall u \exists v \phi$. It states that for all x and u, ϕ can be made true by suitably choosing y depending on the value of x, and choosing v depending on u.
  - More generally, we may define a partially ordered quantifier to be a triple $Q = (\vec{x}; \vec{y}; \beta)$, where $\vec{x}$, $\vec{y}$ are tuples of variables, all distinct, and is a function assigning to each variable in $\vec{y}$ a sublist of $\vec{x}$.
  - If ϕ is a formula whose semantics is defined, then the semantics of $Q\phi$ is: for all $\vec{x}$ one can find values for each variable y among $\vec{y}$, depending only on the values of the variables in $\beta(y)$ (i.e. invariant with respect to changes in values of the remaining x's), which make ϕ true.

6.3 Henkin's Semantics and Henkin's Completeness

By developing some new kind of Henkin's semantics which allows non-canonical interpretations, Henkin shows that finite order logic satisfies some Henkin's completeness which is different from Gödel's completeness.

Theorem 6.1 (Henkin's Completeness). *A finite order formula ϕ is true in all Henkin-prestructures iff ϕ^T is provable in first order logic from Θ_ω.*

More about this see [1, pp.34-38]. Clearly, Henkin's construction also reduces finite order logic to a first order theory.

6.4 Finite order logic as a second order logic

In fact, without allowing non-cannonical interpretations we can also reduce finite order logic to a second order logic.

Theorem 6.2. *A formula ϕ of finite order logic is valid iff the corresponding Σ_1^1 formula $\chi_\phi \to \phi^T$ is valid.*

More about this see [1, pp.39-40]. So it's enough to have second order logic other than k'th order and finite order logic.

7 Some Other Topics

7.1 Second order logic and reverse mathematics

- As we know, second order arithmetic supplies a frame for reverse mathematics.

7.2 Higher order aspects of set theory

- Whether the widespread insistence on avoiding direct reference to higher order constructs is justified?

- (Whitehead and Russell: *Ramified Type Theory*) Recall the construction of cumulative universe of sets: $V_0 = \varnothing$, $V_{\alpha+1} = \wp(V_\alpha)$, $V_\delta = \bigcup_{\alpha<\delta} V_\alpha$ for all limit ordinal δ, and $V = \bigcup_{\alpha \text{ an ordinal}} V_\alpha$.

- As known to deal with the 3rd fundamental conflict of mathematics, second order versions of ZF emerged as being of interest both as frameworks for formalizing set theory and other parts of mathematics, and as relevant to the meta-theory of set theory itself.

- The main rationale of second order set theories is to permit at least reference to arbitrary collections, even when these are of 'unmanageable size'. Because both first order and second order objects are 'classes of sets,' it is customary to refer to second order extensions of ZF and to related theories as *theories of classes*.

- Among the theories of classes there are variants of second order ZF with comprehension restricted to first order formulas, and variants with full comprehension.

- (von Neumann, 1925; Bernays, 1937 & 1958; Gödel, 1940; Mostowski, 1939) NBG: the theory is rephrased as a first order theory of classes, in which the notion of a set is definable (x is a set iff $x \in a$ for some class a), and comprehension is restricted to formulas where all quantifiers are bounded to sets. Note that in NBG we may state the principle of global choice, asserting the existence of a class acting as a global choice function for all sets in the universe: $\exists C \forall x (x \neq \varnothing \rightarrow \exists ! y ((x, y) \in C \wedge y \in x))$.

- (Wang, 1949; Quine, 1951; Tarski, 1981; Kelley, 1955; Morse, 1965) KM: second order ZF with full comprehension. The relation of KM to ZF is analogous to the relation of full analysis to first order arithmetic: KM proves reflection for ZF, and is therefore not a conservative extension thereof.

- Higher order logic has also impacted the development of set theory itself, notably concerning the status of strong infinity axioms.

- The most important results here were inspired by the Montague-Lévy reflection principle: ϕ is true in V iff ϕ is true in V_κ for some cardinal κ.

- The principle is interiorized as an axiom schema in a very powerful extension of KM, due to Bernays(1976). Various forms of this principle imply the existence of ever larger cardinals, including measurable cardinals, which do not seem to have a clear justification on the basis of first order closure conditions.

7.3 Higher order logic in relation to computing and programing

Higher order logic's fundamental effects on computing and programming are mainly as follows:

- the very use of higher order data;
- the computational nature of natural deduction for higher order logic;
- the use of higher order logic in the meta-theory of formal systems;
- the relations between second and higher order logic and computational complexity.

Most notable is our omission of a legion of uses of higher order constructs in computer science, whose relation to logic is less direct.

References

[1] J. Van Benthem and K. Doets. *Higher-Order Logic.* In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, Volume 164 of the series Synthese Library, pp. 275-329, Springer, 1983.

[2] P. G. Hinman. *Fundamentals of Mathematical Logic.* pp. 276-293, A K Peters, Ltd, 2005.

[3] D. Leivant. *Higher Order Logic.* In D. Gabbay, C. J. Hogger and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Volume 1 of Logical Foundations, Oxford Science Publications, 1993.

Recursive Functions with Higher Order Domains

Ana Bove[1] and Venanzio Capretta[2]

[1] Department of Computing Science, Chalmers University of Technology,
412 96 Göteborg, Sweden
telephone: +46-31-7721020, fax: +46-31-165655
bove@cs.chalmers.se

[2] Department of Mathematics and Statistics, University of Ottawa,
585 King Edward, Ottawa, Canada
telephone: +1-613-562-5800 extension 2103, fax: +1-613-562-5776
venanzio.capretta@mathstat.uottawa.ca

Abstract. In a series of articles, we developed a method to translate general recursive functions written in a functional programming style into constructive type theory. Three problems remained: the method could not properly deal with functions taking functional arguments, the translation of terms containing λ-abstractions was too strict, and partial application of general recursive functions was not allowed. Here, we show how the three problems can be solved by defining a type of partial functions between given types. Every function, including arguments to higher order functions, λ-abstractions and partially applied functions, is then translated as a pair consisting of a domain predicate and a function dependent on the predicate. Higher order functions are assigned domain predicates that inherit termination conditions from their functional arguments. The translation of a λ-abstraction does not need to be total anymore, but generates a local termination condition. The domain predicate of a partially applied function is defined by fixing the given arguments in the domain of the original function. As in our previous articles, simultaneous induction-recursion is required to deal with nested recursive functions. Since by using our method the inductive definition of the domain predicate can refer globally to the domain predicate itself, here we need to work on an impredicative type theory for the method to apply to all functions. However, in most practical cases the method can be adapted to work on a predicative type theory with type universes.

1 Introduction

In functional programming, functions can be defined by recursive equations where the arguments of the recursive calls are not required to be smaller than the input, hence allowing the definition of general recursive functions. Thus, the termination of a program is not guaranteed by its structure. On the other hand, in type theory, only structurally recursive functions are allowed, that is, functions where the recursive calls are performed only on arguments structurally smaller than the input. Thus, some functional programs have no direct translation into type theory.

P. Urzyczyn (Ed.): TLCA 2005, LNCS 3461, pp. 116–130, 2005.

In a series of articles, we have developed a method to translate functional programs into constructive type theory. Given a general recursive function, the method consists in defining an inductive predicate that characterises the inputs on which the function terminates. We can think of this predicate as the domain of the function. The type-theoretic version of the function can then be defined by structural recursion on the proof that the input values satisfy this predicate. A similar method was independently developed by Dubois and Viguié Donzeau-Gouge [DDG98], however they treat nested recursion in a different way and they do not consider the issues we tackle in the present article.

Given a general recursive function $\mathtt{f}$ from σ to τ, its formalisation in type theory following our method consists of an inductive predicate fAcc and a function f with types

$$\mathsf{fAcc}\colon \widehat{\sigma} \to \mathsf{Prop}$$
$$\mathsf{f}\colon (x\colon \widehat{\sigma}; \mathsf{fAcc}\ x) \to \widehat{\tau}$$

where $\widehat{\sigma}$ and $\widehat{\tau}$ are the type-theoretic translations of σ and τ, respectively, and f is defined by structural recursion on its second argument.

Intuitively, if the ith recursive equation of the original program is

$$\mathtt{f}(p) = \cdots \mathtt{f}(a_1) \cdots \mathtt{f}(a_n) \cdots$$

calling itself recursively on the arguments $a_1, \ldots, a_n$, then the ith constructor of fAcc has type

$$\mathsf{facc}_i\colon (\ldots; \mathsf{fAcc}\ a_1; \ldots; \mathsf{fAcc}\ a_n)(\mathsf{fAcc}\ p)$$

and the ith structural recursive equation of f is

$$\mathsf{f}\ p\ (\mathsf{facc}_i\ \cdots\ h_1\ \cdots\ h_n) = \cdots (\mathsf{f}\ a_1\ h_1) \cdots (\mathsf{f}\ a_n\ h_n) \cdots .$$

In practise, the types of the constructors of fAcc and the structure of the equations of f may be more complex, but the idea remains the same: fAcc is inductively defined so that proving fAcc on an input p requires proofs of fAcc for the arguments of all the recursive calls that $\mathtt{f}$ performs when applied to p.

The method was introduced by Bove [Bov01] to formalise simple general recursive algorithms in constructive type theory (by simple we mean non-nested and non-mutually recursive). It was extended by Bove and Capretta [BC01] to treat nested recursion by using Dybjer's simultaneous induction-recursion, and by Bove [Bov02a] to treat mutually recursive algorithms, nested or not. A formal description of the method is given in [BC04] where we also prove a soundness and a weak completeness theorem. The first three papers mentioned above and a previous version of [BC04] have been put together into the first author's Ph.D. thesis [Bov02b]. A tutorial on the method can be found in [Bov03].

The method of [BC04] separates the computational and logical parts of the type-theoretic versions of the functional programs. An immediate consequence is that it allows the formalisation of partial functions: proving that a certain function is total amounts to proving that the corresponding domain predicate is satisfied by every input. Another consequence is that the resulting type-theoretic algorithms are clear, compact and easy to understand. They are as simple as their counterparts in a functional programming language. However, our method has some problems and limitations which we have already mentioned in [BC04].

The first problem concerns higher order functions, that is, functions that take other functions as arguments. For example, let us consider the function `map`: $(\sigma \to \tau) \to [\sigma] \to [\tau]$, as defined in the Haskell [Jon03] prelude, which has a first argument in a functional type. Since `map` is structurally recursive on its list argument, in [BC04] we translate it into a structurally recursive function map in type theory with type $(\widehat{\sigma} \to \widehat{\tau}) \to [\widehat{\sigma}] \to [\widehat{\tau}]$. This means that the first argument of map can only be instantiated to a total function of type $\widehat{\sigma} \to \widehat{\tau}$. But in functional programming `map` could be applied to potentially non-terminating functions. Therefore, even if `map` is structurally recursive, it is still liable to non-termination since its functional argument f might be undefined on some (or all) of the elements to which it will be applied. In other words, `map` inherits the termination conditions from its functional argument. In [BC04] we had no way to express this fact. Therefore, our translation was too restrictive with respect to the original functional program.

Another problem is that whenever there is a λ-abstraction in the right-hand side of an equation, the method of [BC04] translates it as a total function in type theory. This interpretation is too strict, since the corresponding function could diverge on arguments to which it is not actually applied during execution, without jeopardising the termination behaviour of the program. More specifically, if under the scope of the λ-abstraction there is a call to one of the general recursive functions we are defining in our program, let us say `f`, the method inductively requires every instantiation of the argument of the λ-abstraction to be in the domain of f. However, this constraint might be stronger than actually needed since the λ-abstraction may just be applied to a proper subset of all the instantiations. This problem is already present in a classical setting in the work of Finn, Fourman, and Longley [FFL97].

The third problem is that partial applications of general recursive functions are not allowed in [BC04]. When applying a recursive function f taking m arguments $a_1, \ldots, a_m$, we must also provide a proof h of the accessibility of $a_1, \ldots, a_m$. If f is applied to an insufficient number of arguments $a_1, \ldots, a_k$ with $k < m$, then the accessibility condition cannot even be formulated and it is not possible to prove that the result of the application converges. Therefore, we barred partial applications of general recursive functions.

Here, we introduce a type of partial functions in type theory and we present a new method to translate functional programs into their type-theoretic equivalents. The method is based on the one presented in [BC04] but, now, it translates every function in the functional side into an element of the new type of partial functions. With this new approach, the problems we mention above disappear.

For our new method to be applicable to any function, we need to work in a type theory with an impredicative universe Set with inductive-recursive definitions à la Dybjer [Dyb00]. Both datatypes and propositions are represented as elements of Set. Therefore, we use the Calculus of Construction [CH88] extended with a schema for simultaneous inductive-recursive definitions. A justification of the soundness of inductive-recursive definitions in the Calculus of Constructions is given by [Bla03] and [Cap04]. The method works also in the slightly different type architecture used in the proof assistant Coq [Coq02]. There are in Coq two impredicative universes, Set and Prop, both being elements of the predicative universe Type. For the purpose of this work, it is possible to use Prop for the domain predicate and Set for the functions. Usually, elimination of a proposition over a set (essential in our method) is not allowed in Coq, a feature intended to prevent mixing logical information with pure computational content. However, Christine Paulin [Pau] devised a way around this difficulty, consisting in defining structural deconstruction functions on propositions that have the property of uniqueness of proofs, as is the case of our domain predicates. Therefore, we use the notation Set for datatypes and Prop for propositions, to be interpreted as either the same impredicative universe or two distinct impredicative universes, in which case Paulin's method is to be used in place of structural recursion on the proofs of accessibility. As we point out later, it would be possible to adapt the method to work on a predicative type theory. However, we would loose generality since then the method could not be used to translate all functions (see function `itz` in section 4).

An extension of type theory with a constructor $A \rightsquigarrow B$ for partial functions from A to B was already proposed by Constable and Mendler in [CM85]. Together with the type $A \rightsquigarrow B$ they introduce a new form of canonical elements, which is not the case in our partial functions type. They are of the form $fix(f, x.F)$, to be intended as the functions with definition $f(x) = F$. From the definition of f one can construct its domain $dom(f)(x)$, essentially as in our method, as a recursive predicate generated structurally by the recursive calls to f in F. The roles of both domain predicates are however quite different. While our functions are defined by structural recursion on their domain predicates, the predicates in [CM85] serve only as a way to characterise the valid inputs and the functions in [CM85] are defined independently of their domain predicates.

This paper is mainly intended for readers with some knowledge in type theory. This said, what follows is just intended to fix the notation.

A *context* Γ is a sequence of assumptions $\Gamma \equiv x_1 \colon \alpha_1; \ldots ; x_n \colon \alpha_n$ where $x_1, \ldots , x_n$ are distinct variables and each α_i is a type that can contain occurrences of the variables that precede it. We call a sequence of variable assumptions Δ a *context extension* of the context Γ if $\Gamma; \Delta$ is a context.

If α is a type and β is a family of types over α, we write $(x \colon \alpha)\beta(x)$ for the type of dependent functions from α to β. If β does not depend on values of type α we might simply write $\alpha \to \beta$ for the type of functions from α to β. Functions have abstractions as canonical values, which we write $[x \colon \alpha]e$. Consecutive dependent function types and abstractions are written $(x_1 \colon \alpha_1; \ldots ; x_n \colon \alpha_n)\beta(x_1, \ldots , x_n)$ and $[x_1 \colon \alpha_1, \ldots , x_n \colon \alpha_n]e$, respectively. In either case, each α_i can contain occurrences of the variables that precede it. If β doest not depend on the last assumption x_n, then we can write $(x_1 \colon \alpha_1; \ldots ; x_{n-1} \colon \alpha_{n-1}; \alpha_n) \to \beta(x_1, \ldots , x_{n-1})$.

For the sake of simplicity, we will use the same notation as much as possible both in the functional programming side and in the type-theoretic side. In addition, names in the functional programming side will be written with `typewriter` font while names in the type-theoretic side will be written with Sans Serif font.

The article has the following organisation. Section 2 briefly presents the method of [BC04]. Section 3 defines the type of partial functions and gives a formal definition of the new translation. Section 4 illustrates the translation on some examples. Finally, Section 5 discusses advantages and disadvantages of the new method.

2 Brief Summary of the Original Translation

We start from a Haskell-like functional programming language $\mathcal{FP}$. The types allowed in $\mathcal{FP}$ are: variable types, inductive data types, and function types.

Elements of inductive types are generated by constructors which must always be used fully applied.

There are two kinds of functions, that is, of elements in the functional type: those defined by structural recursion and those defined by general recursion. Since the two kinds need to be translated differently, we distinguish them in the functional programming notation. Structurally recursive functions acquire the usual Haskell-like functional types, $\sigma \to \tau$. On the other hand, general recursive functions must always be used fully applied. We reflect this requirement in the syntax by assigning them a *specification* $\sigma_1, \ldots, \sigma_m \Rightarrow \tau$ rather than a proper functional type.

The two kinds of functions give rise to two kinds of applications: those dealing with proper functional types and those dealing with specifications.

The form of the definition of a general recursive function is:

$$\begin{array}{l} \texttt{fix f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau \\ \qquad \texttt{f}(p_{11}, \ldots, p_{1m}) = e_1 \\ \qquad\qquad \vdots \\ \qquad \texttt{f}(p_{l1}, \ldots, p_{lm}) = e_l \end{array}$$

where the p_{ij}'s are exclusive linear patterns of the corresponding types and the e_i's are valid terms of type τ (see Definition 1 below). We also allow guarded equations in the definition of a function, where the condition in the equation must be a valid term of type `Bool`. In any case, the equations must satisfy the exclusivity condition: for every particular argument, at most one equation can apply. It is possible that, for some argument, no equation applies, in which case the function is undefined on the given input.

Since the definition of the set of valid terms of a certain type (definition 4 in [BC04]) is important for the understanding of this work, we transcribe it below.

Let us call $\mathcal{F}$ the set of all structurally recursive functions together with their types. Then, the valid terms that we allow in the definition of a recursive function depend on two components: (a) the set $\mathcal{X}$ of variables that can occur free in the terms, and (b) the set $\mathcal{SF}$ of functions that are being defined (we might define several mutually recursive functions simultaneously), which can be used in the recursive calls.

Definition 1. *Let $\mathcal{X}$ be a set of variables together with their types. Let $\mathcal{SF}$ be a set of function names together with their specifications. Let the set of names of the variables in $\mathcal{X}$, the set of names of the functions in $\mathcal{SF}$, and the set of names of the functions in $\mathcal{F}$ be disjoint. We say that t* is a valid term of type τ *with respect to $\mathcal{X}$ and $\mathcal{SF}$, if the judgement $\mathcal{X};\mathcal{SF} \vdash t\colon \tau$ can be derived from the rules in Figure 1.* □

$$\frac{x\colon \sigma \in \mathcal{X}}{\mathcal{X};\mathcal{SF} \vdash x\colon \sigma} \qquad \frac{f\colon \sigma \to \tau \in \mathcal{F}}{\mathcal{X};\mathcal{SF} \vdash f\colon \sigma \to \tau}$$

$$\frac{\mathtt{f}\colon \sigma_1, \dots, \sigma_m \Rightarrow \tau \in \mathcal{SF} \quad \mathcal{X};\mathcal{SF} \vdash a_i\colon \sigma_i \text{ for } 1 \leqslant i \leqslant m}{\mathcal{X};\mathcal{SF} \vdash f(a_1, \dots, a_m)\colon \tau}$$

$$\frac{\begin{array}{c}\mathtt{c}\colon \tau_1, \dots, \tau_k \Rightarrow \mathtt{T} \qquad \mathtt{c} \text{ constructor of } \mathtt{T} \\ \mathcal{X};\mathcal{SF} \vdash a_i\colon \tau_i \text{ for } 1 \leqslant i \leqslant k\end{array}}{\mathcal{X};\mathcal{SF} \vdash \mathtt{c}(a_1, \dots, a_k)\colon \mathtt{T}}$$

$$\frac{(\mathcal{X}\backslash x) \bigcup \{x\colon \sigma\};\mathcal{SF} \vdash b\colon \tau}{\mathcal{X};\mathcal{SF} \vdash [x]b\colon \sigma \to \tau} \qquad \frac{\mathcal{X};\mathcal{SF} \vdash f\colon \sigma \to \tau \quad \mathcal{X};\mathcal{SF} \vdash a\colon \sigma}{\mathcal{X};\mathcal{SF} \vdash (f\ a)\colon \tau}$$

Fig. 1. Rules for deriving valid terms judgements

Next, we briefly explain the translation of programs into type theory presented in [BC04]. Below, we call $\widehat{\sigma}$ the translation of the type σ.

Variable types and variables, and inductive data types and their constructors are translated straightforwardly. A function type $\sigma \to \tau$ is translated as the total function type $\widehat{\sigma} \to \widehat{\tau}$ in type theory. Structurally recursive functions are directly translated as structurally recursive functions in type theory with the same functional type (except for some possible changes in the notation).

As we have already mentioned in Section 1, a function $\mathtt{f}\colon \sigma_1, \dots, \sigma_m \Rightarrow \tau$ is translated as a pair

$$\begin{array}{l}\mathsf{fAcc}\colon \widehat{\sigma_1} \to \dots \to \widehat{\sigma_m} \to \mathsf{Prop} \\ \mathsf{f}\colon (x_1\colon \widehat{\sigma_1}; \dots; x_m\colon \widehat{\sigma_m}; \mathsf{fAcc}\ x_1 \cdots x_m) \to \widehat{\tau}\end{array}$$

In order to complete the translation of $\mathtt{f}$ we need to give the types of the constructors of fAcc and the equations defining f.

For each equation in $\mathtt{f}$ (i.e., in the functional side) we define a constructor for fAcc and an equation for f (i.e., in the type-theoretic side) as follows. Let

$$\mathtt{f}(p_1, \dots, p_m) = e \quad \mathtt{if}\ c$$

be a guarded equation of $\mathtt{f}$, let Γ be the context of variables occurring in the patterns $p_1, \dots, p_m$ and let $\widehat{\Gamma}$ be its type-theoretic translation. Given the term

e, we define a type-theoretic context Φ_e which extends $\widehat{\Gamma}$, and a type-theoretic translation $\widehat{e}$ of e whose free variables are included in $\widehat{\Gamma}; \Phi_e$. Similarly, we define Φ_c and $\widehat{c}$.

The type of the constructor of fAcc corresponding to the above equation is

$$\mathsf{facc}\colon (\widehat{\Gamma}; \Phi_c; q\colon \widehat{c} = \mathsf{true}; \Phi_e)(\mathsf{fAcc}\ \widehat{p_1} \cdots \widehat{p_m})$$

and the corresponding equation of f is

$$\mathsf{f}\ \widehat{p_1} \cdots \widehat{p_m}\ (\mathsf{facc}\ \overline{x}\ \overline{y}\ q\ \overline{z}) = \widehat{e}$$

where $\overline{x}$, $\overline{y}$ and $\overline{z}$ are the variables defined in $\widehat{\Gamma}$, Φ_c and Φ_e, respectively, and $\widehat{p_i}$ is the type-theoretic version of p_i. For non-conditional equations, we simply omit all the parts related with the condition c.

The translation of the application of a function to an argument must consider two cases. If the function has a regular function type $\sigma \to \tau$, that is, it is defined without using general recursion, then the application is translated straightforwardly. When we translate the application of a general recursive function to *all* its arguments, we have to make sure that the arguments are in the domain of the function. Hence, we must add a constraint expressing this fact in the translation context of the application. Concretely, when translating a term of the form $\mathtt{f}(a_1, \ldots, a_m)$, we must add an assumption $(h\colon \mathsf{fAcc}\ \widehat{a_1} \cdots\ \widehat{a_m})$ to the translation context of the application, where $\widehat{a_i}$ is the type-theoretic translation of a_i. The application itself is translated as $(\mathsf{f}\ \widehat{a_1} \cdots\ \widehat{a_m}\ h)$. The argument h is needed to make sure that f is only applied to arguments in its domain.

The crucial part of the method in [BC04] is the definition of the context Φ_a and the type-theoretic term $\widehat{a}$ associated with a term a. Both Φ_a and $\widehat{a}$ are defined simultaneously by recursion over the structure of a.

For a formal description of the language $\mathcal{FP}$ and the translation of its programs into type theory, the reader can refer to [BC04].

We finish this section with the definition of the partial function `my_mod` and its translation into type theory following the method we have just described. This function is such that $\mathtt{my\_mod}(m, n) = n\ \mathtt{mod}\ m$ whenever $m \neq 0$, where `mod` is the standard modulo function defined as in the Haskell prelude.

We define the functional version of `my_mod` as:

$$\begin{array}{ll}
\mathtt{fix}\ \mathtt{my\_mod}\colon \mathtt{N}, \mathtt{N} \Rightarrow \mathtt{N} & \\
\quad \mathtt{my\_mod}(m, n) = n & \mathtt{if}\ m \neq \mathsf{0} \wedge n < m \\
\quad \mathtt{my\_mod}(m, n) = \mathtt{my\_mod}(m, n - m) & \mathtt{if}\ m \neq \mathsf{0} \wedge n \geqslant m
\end{array}$$

where $-$, $<$, $\geqslant$, $\neq$ and $\wedge$ are defined as expected.

This function is translated into type theory as follows:

$$\begin{array}{l}
\mathsf{my\_modAcc}\colon \mathsf{N} \to \mathsf{N} \to \mathsf{Prop} \\
\quad \mathsf{my\_mod\_acc}_{<}\colon (m\colon \mathsf{N}; n\colon \mathsf{N}; q\colon (m \neq \mathsf{0} \wedge n < m) = \mathsf{true})(\mathsf{my\_modAcc}\ m\ n) \\
\quad \mathsf{my\_mod\_acc}_{\geqslant}\colon (m\colon \mathsf{N}; n\colon \mathsf{N}; q\colon (m \neq \mathsf{0} \wedge n \geqslant m) = \mathsf{true}; \\
\qquad\qquad\qquad\qquad h\colon \mathsf{my\_modAcc}\ m\ (n - m))(\mathsf{my\_modAcc}\ m\ n)
\end{array}$$

$$\begin{array}{l}
\mathsf{my\_mod}\colon (m\colon \mathsf{N}; n\colon \mathsf{N}; \mathsf{my\_modAcc}\ m\ n) \to \mathsf{N} \\
\quad \mathsf{my\_mod}\ m\ n\ (\mathsf{my\_mod\_acc}_{<}\ m\ n\ q) = n \\
\quad \mathsf{my\_mod}\ m\ n\ (\mathsf{my\_mod\_acc}_{\geqslant}\ m\ n\ q\ h) = \mathsf{my\_mod}\ m\ (n - m)\ h
\end{array}$$

Observe that we will never be able to apply the function $\mathsf{my\_mod}$ to the arguments $\mathsf{0}$ and $i\colon \mathsf{N}$ since we cannot construct a proof of $(\mathsf{my\_modAcc}\ \mathsf{0}\ i)$.

3 New Translation of Functional Programs

We now introduce the type of *partial functions* in our impredicative type theory. If $\alpha\colon \mathsf{Set}$ and $\beta\colon \mathsf{Set}$, then we define the type of partial functions as

$$\alpha \rightharpoonup \beta \equiv \Sigma D\colon \alpha \to \mathsf{Prop}.(x\colon \alpha; D\ x) \to \beta\colon \mathsf{Set}.$$

Thus, a partial function $f\colon \alpha \rightharpoonup \beta$ is actually a pair consisting of the domain of the function and the function itself, which depends on a proof that the input value is in the domain of the function. The definition can be extended to consider partial functions of several arguments. If $\alpha_1, \ldots, \alpha_m\colon \mathsf{Set}$ and $\beta\colon \mathsf{Set}$, we define

$$\begin{aligned}\alpha_1, \ldots, \alpha_m \rightharpoonup \beta \equiv\ & \Sigma D\colon \alpha_1 \to \cdots \to \alpha_m \to \mathsf{Prop}.\\ & (x_1\colon \alpha_1; \ldots; x_m\colon \alpha_m; D\ x_1\ \cdots\ x_m) \to \beta.\end{aligned}$$

If $f\colon \alpha_1, \ldots, \alpha_m \rightharpoonup \beta$, we write dom_f for $(\pi_1\ f)$ and, if $a_i\colon \alpha_i$ for $1 \leqslant i \leqslant m$ and $(h\colon \mathsf{dom}_f\ a_1\ \cdots\ a_m)$, we use the notation $f_{[h]}(a_1, \ldots, a_m)$ for $(\pi_2\ f\ a_1\ \cdots\ a_m\ h)$.

In the case of functions of many arguments, we may partially apply the function to only $k < m$ arguments. Then we write:

$$\begin{aligned}& f(a_1, \ldots, a_k) = \langle D', f'\rangle\colon \alpha_{k+1}, \ldots, \alpha_m \rightharpoonup \beta\\ & \text{where } D'\ x_{k+1}\ \cdots\ x_m = \mathsf{dom}_f\ a_1\ \cdots\ a_k\ x_{k+1}\ \cdots\ x_m\\ & \qquad\quad f'\ x_{k+1}\ \cdots\ x_m\ h = f_{[h]}(a_1, \ldots, a_k, x_{k+1}, \ldots, x_m)\end{aligned} \qquad (*)$$

This definition amounts to an outline of a proof of the left-to-right direction of the following equivalence:

$$\alpha_1, \ldots, \alpha_m \rightharpoonup \beta \cong \alpha_1 \to \cdots \to \alpha_k \to (\alpha_{k+1}, \ldots, \alpha_m \rightharpoonup \beta).$$

The right-to-left direction is straightforward.

In what follows, we give a modification of the method of [BC04] that uses the type of partial functions in place of the standard type of total functions.

First, we apply the translation method also to structurally recursive functions since we want all the functions to have a partial function type. Hence, structurally and general recursive functions are now all assigned specifications, so now the set $\mathcal{SF}$ contains also the functions previously in $\mathcal{F}$. As a consequence, the second rule in Definition 1 of valid terms simply disappears.

In addition, the definition of valid terms required every occurrence of a general recursive function to be fully applied. Now we lift this restriction and we replace the third rule in Definition 1 by the following rule:

$$\frac{\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau \in \mathcal{SF} \quad \mathcal{X}; \mathcal{SF} \vdash a_i\colon \sigma_i \text{ for } 1 \leqslant i \leqslant k \text{ and } k \leqslant m}{\mathcal{X}; \mathcal{SF} \vdash f(a_1, \ldots, a_k)\colon \sigma_{k+1}, \ldots, \sigma_m \Rightarrow \tau}$$

where if $k = m$, $\sigma_{k+1}, \ldots, \sigma_m \Rightarrow \tau$ is understood simply as τ.

A similar modification has to be made to the fourth rule of the same definition, which deals with constructors, since they must also be used fully applied in [BC04].

Next, we modify the definition of the translation into type theory.

Variable types and inductive datatypes are translated as before. The function type $\sigma \to \tau$ is now translated into the partial function type $\widehat{\sigma} \rightharpoonup \widehat{\tau}$. In our previous work, specifications were not translated into a type. Now, given the specification $\sigma_1, \ldots, \sigma_m \Rightarrow \tau$ we define its translation as the type of partial functions $\widehat{\sigma_1}, \ldots, \widehat{\sigma_m} \rightharpoonup \widehat{\tau}$.

Constructors are assigned specifications in [BC04] rather than types. With the translation we give of specifications, a constructor is now expected to have a domain predicate. However, the application of a constructor to arguments of the corresponding types should always be defined. Let $\mathtt{c}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \mathtt{T}$ be one of the constructors of the inductive type $\mathtt{T}$. In type theory, the corresponding constructor would have type $\mathsf{c}\colon \widehat{\sigma_1} \to \cdots \to \widehat{\sigma_m} \to \mathsf{T}$. The translation of $\mathtt{c}$ is then defined as $\mathsf{C} = \langle \mathsf{cAcc}, \mathsf{c}' \rangle \colon \widehat{\sigma_1}, \ldots, \widehat{\sigma_m} \rightharpoonup \mathsf{T}$ with

$$\begin{aligned} &\mathsf{cAcc} = [x_1 \colon \widehat{\sigma_1}; \ldots ; x_m \colon \widehat{\sigma_m}]\mathbb{T} \colon \widehat{\sigma_1} \to \cdots \to \widehat{\sigma_m} \to \mathsf{Prop} \\ &\mathsf{c}'_{[h]}(a_1, \ldots, a_m) = \mathsf{c}(a_1, \ldots, a_m) \end{aligned}$$

where $\mathbb{T}$ is a set containing only the element $\mathtt{tt}$.

Below we present the definition of the context Φ_a and the type-theoretic expression $\widehat{a}$ associated with a term a. The cases of function calls and constructors are now split into two cases, according to whether the function or the constructor is fully or partially applied; this distinction was not relevant in [BC04].

Definition 2. *Given a term a and a context Γ containing type assumptions for the free variables in a, we define the context extension Φ_a and the type-theoretic term $\widehat{a}$ by recursion on the structure of a. Note that, since Φ_a extends Γ, we should only introduce fresh variables in Φ_a.*

$a \equiv z$**:** *If the term a is the variable z, then $\Phi_a \equiv ()$ and $\widehat{a} \equiv z$.*

$a \equiv \mathtt{f}(a_1, \ldots, a_m)$**:** *Here, $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$, and a_1, ... ,a_m are arguments of the appropriate types. Hence, the function is fully applied. First, we determine $\Phi_{a_1}, \ldots, \Phi_{a_m}$ and $\widehat{a_1} \ldots, \widehat{a_m}$ by structural recursion. We then define*

$$\Phi_a \equiv \Phi_{a_1}; \ldots ; \Phi_{a_m}; (h\colon \mathsf{fAcc}\ \widehat{a_1}\ \cdots\ \widehat{a_m}) \quad \textit{and} \quad \widehat{a} \equiv \mathsf{f}_{[h]}(\widehat{a_1}, \ldots, \widehat{a_m})$$

$a \equiv \mathtt{f}(a_1, \ldots, a_k)$**:** *Let $\mathtt{f}$ be as above and $k < m$. In this case the function is not fully applied. Under similar assumptions as in the previous case, we define $\Phi_a \equiv \Phi_{a_1}; \ldots ; \Phi_{a_k}$ and $\widehat{a} \equiv \mathsf{f}(\widehat{a_1}, \ldots, \widehat{a_k})$. See (*) for the meaning of the partial application of f to only k of its m arguments.*

$a \equiv \mathsf{c}(a_1, \ldots, a_m)$**:** *Let c be a constructor fully applied to arguments of the appropriate types. Under similar assumptions as in the case of fully applied functions, we define $\Phi_a \equiv \Phi_{a_1}; \ldots ; \Phi_{a_m}$ and $\widehat{a} \equiv \mathsf{c}(\widehat{a_1}, \ldots, \widehat{a_m})$. Notice that this is equal to $\mathsf{c}'_{[\mathtt{tt}]}(\widehat{a_1}, \ldots, \widehat{a_m})$, so the translation is consistent with that of recursive functions.*

$a \equiv \mathsf{c}(a_1, \dots, a_k)$**:** *Let* c *be a constructor applied to only* k *of its* m *arguments. This case is similar to the case of partially applied functions.*

$a \equiv [z]b$**:** *Let* σ *be the type of* z*. We start by calculating* Φ_b *and* $\widehat{b}$ *recursively. Notice that now, the context with the assumptions for the free variables in* b *is* $(\Gamma; z\colon \widehat{\sigma})$*. We define* $\Phi_{[z]b} \equiv ()$ *and* $\widehat{[z]b} \equiv \langle \mathsf{gAcc}, \mathsf{g}\rangle$ *with*

$$\begin{aligned}\mathsf{gAcc} &= [z\colon \widehat{\sigma}]\Sigma\Phi_b\\ \mathsf{g} &= [z\colon \widehat{\sigma}; h\colon \mathsf{gAcc}\ z]\mathsf{Cases}\ h\ \mathsf{of}\ \left\{ \langle\overline{y_{\Phi_b}}\rangle \mapsto \widehat{b}\right.\end{aligned}$$

where $\Sigma\Phi_b$ *is a big* Σ*-type defining the conjunction of all the preconditions contained in* Φ_b *and* $\overline{y_{\Phi_b}}$ *is the sequence of variables in* Φ_b*. If* Φ_b *is empty then* $\Sigma\Phi_b$ *is simply understood as* $\mathbb{T}$ *and the whole case-expression can just be replaced by* $\widehat{b}$*. On the other hand, if* Φ_b *contains only one assumption then* $\Sigma\Phi_b$ *is simply* Φ_b*. Moreover, the case-expression can just be replaced by* $\widehat{b}[y_b := h]$*, where* y_b *is the variable assumed in* Φ_b*.*

$a \equiv (g\ b)$**:** *Here* g *stands for any function with a functional type (not a specification). As usual, we define* Φ_a *and* $\widehat{a}$ *in terms of* $\Phi_g, \widehat{g}, \Phi_b$ *and* $\widehat{b}$*. Since* g *is (potentially) a partial function, it has a partial function type in type theory, so we have to make sure that it is only applied to elements in its domain. Hence, we have* $\Phi_a = \Phi_g; \Phi_b; (h\colon \mathsf{dom}_{\widehat{g}}\ \widehat{b})$ *and* $\widehat{a} = \widehat{g}_{[h]}(\widehat{b})$*.* □

Theorem 1 of [BC04] can be strengthened: now, it states that the functional program `f` and its type-theoretic translation f denote the same partial recursive function. The proof is similar to those of Theorems 1 and 2 in [BC04] and it relies on a correspondence between computation of terms in functional programming and reduction of their translations in type theory. Given the application of a general recursive function, this correspondence depends, in turn, on the correspondence between the trace of the computation of the application in functional programming and a normal form of the proof that the type-theoretic versions of the arguments satisfy the corresponding domain predicate.

Lemma 1. *Let* $e\colon \tau$ *be a valid term in* $\mathcal{FP}$ *with respect to* Γ *and* $\mathcal{SF}$*, and let* $\widehat{\Gamma}$ *be the type-theoretic translation of* Γ*. Let* Φ_e *be the translation context generated by the method,* $\overline{z}$ *the sequence of variables in* Φ_e*, and* $\overline{d}$ *an instantiation of* Φ_e *depending only on variables in* $\widehat{\Gamma}$*. Then the computation of* e *in* $\mathcal{FP}$ *terminates with a term* r *whose type-theoretic translation* $\widehat{r}$ *is convertible with* $\widehat{e}[\overline{z := d}]$*.*

Proof. We do induction on the pair (l, e) where l is the maximum length of a normalisation path of $\widehat{e}[\overline{z := d}]$ (notice that the path is finite because type theory enjoys strong normalisation) and the order $<$ is the lexicographic order on pairs, that is: $(l', e') < (l, e)$ iff either $l' < l$ or $l' \leqslant l$ and e' is structurally smaller than e. This part of the proof does not present any substantial change with respect to our previous work. □

Lemma 2. *Let* e *and* Φ_e *be as in the previous lemma. If the computation of* e *terminates in* $\mathcal{FP}$ *then there is an instantiation* $\overline{d}$ *of* Φ_e*.*

Proof. We do induction on the pair (l, e) where l is the length of the trace of the computation of e and the order is the lexicographic order on pairs as in lemma 1.

The fact that we assign domain predicates also to local functions generated by λ-abstraction entails that, when computing a local function on a specific argument, we can generate a proof of the local termination predicate whenever the application terminates in the functional side. This is an improvement on [BC04], where we needed to generate a proof of totality for the local function. □

Theorem 1. *Let* $\mathtt{f}\colon \sigma_1, \ldots, \sigma_m \Rightarrow \tau$ *be a function in* $\mathcal{FP}$*. Let* fAcc *and* f *be the domain predicate for* $\mathtt{f}$ *and the type-theoretic version of* $\mathtt{f}$*, respectively. Then, for every sequence of values* $v_1\colon \sigma_1, \ldots, v_m\colon \sigma_m$ *we have that*

$$(\mathsf{fAcc}\ \widehat{v_1}\ \cdots\ \widehat{v_m}) \textit{ is provable} \quad\Longleftrightarrow\quad \mathtt{f} \textit{ is defined on } v_1, \ldots, v_m$$

and if $(h\colon \mathsf{fAcc}\ \widehat{v_1}\ \cdots\ \widehat{v_m})$ *is a closed proof, then*

$$\mathsf{f}_{[h]}(\widehat{v_1}, \ldots, \widehat{v_m}) = \mathtt{f}(\widehat{v_1, \ldots,} v_m).$$

Proof. Immediate by applying the previous lemmas to $e \equiv \mathtt{f}(v_1, \ldots, v_m)$. □

4 Illustration of the Method

We illustrate the advantages of the new method on some examples containing the features that were problematic in our previous work. The function `map` shows how to deal with functional arguments, the function `sumdel` illustrates how λ-abstractions are treated, and the function `is_div2` shows how to deal with partial applications.

In addition, we demonstrate that the impredicativity of the sort Prop is necessary for the generality of the method, by giving an example `itz` that gives rise to a polymorphic domain predicate. The example `itz` is also the only one with nested recursion, so it is the only one that needs induction-recursion.

Functional Arguments: `map`. Functional version:

$$\begin{aligned}
&\mathtt{fix\ map}\colon \gamma \to \delta, \mathtt{List}\ \gamma \Rightarrow \mathtt{List}\ \delta\\
&\qquad \mathtt{map}(f, \mathtt{nil}) = \mathtt{nil}\\
&\qquad \mathtt{map}(f, \mathtt{cons}(x, xs)) = \mathtt{cons}(f\ x, \mathtt{map}(f, xs))
\end{aligned}$$

Type-theoretic version: $\mathsf{Map} = \langle \mathsf{mapAcc}, \mathsf{map}\rangle \colon (\gamma \rightharpoonup \delta), \mathsf{List}\ \gamma \rightharpoonup \mathsf{List}\ \delta$ with:

$$\begin{aligned}
&\mathsf{mapAcc}\colon (\gamma \rightharpoonup \delta) \to \mathsf{List}\ \gamma \to \mathsf{Prop}\\
&\quad \mathsf{mapacc}_{\mathsf{nil}}\colon (f\colon \gamma \rightharpoonup \delta)(\mathsf{mapAcc}\ f\ \mathsf{nil})\\
&\quad \mathsf{mapacc}_{\mathsf{cons}}\colon (f\colon \gamma \rightharpoonup \delta; x\colon \gamma; xs\colon \mathsf{List}\ \gamma; h\colon \mathsf{dom}_f\ x; h_1\colon \mathsf{dom}_{\mathsf{Map}}\ f\ xs)\\
&\qquad\qquad\qquad (\mathsf{mapAcc}\ f\ \mathsf{cons}(x, xs))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{map}\colon (f\colon \gamma \rightharpoonup \delta; ys\colon \mathsf{List}\ \gamma; \mathsf{mapAcc}\ f\ ys) \to \mathsf{List}\ \delta\\
&\quad \mathsf{map}\ f\ \mathsf{nil}\ (\mathsf{mapacc}_{\mathsf{nil}}\ f) = \mathsf{nil}\\
&\quad \mathsf{map}\ f\ \mathsf{cons}(x, xs)\ (\mathsf{mapacc}_{\mathsf{cons}}\ f\ x\ xs\ h\ h_1) = \mathsf{cons}(f_{[h]}(x), \mathsf{Map}_{[h_1]}(f, xs))
\end{aligned}$$

Recall that $(\mathsf{dom}_{\mathsf{Map}}\ f\ xs)$ and $\mathsf{Map}_{[h_1]}(f, xs)$ reduce to $(\mathsf{mapAcc}\ f\ xs)$ and $(\mathsf{map}\ f\ xs\ h_1)$, respectively. In addition, when map is applied to a concrete partial function $\langle\mathsf{fAcc}, \mathsf{f}\rangle$, $(\mathsf{dom}_f\ x)$ and $f_{[h]}(x)$ reduce to $(\mathsf{fAcc}\ x)$ and $(\mathsf{f}\ x\ h)$, respectively. When checking the validity of the inductive-recursive definitions, we have to expand such terms.

Abstractions: `sumdel`. Here, the functions +, `sum` and `delete` are all structurally recursive. The function + is defined as expected, and `sum` and `delete` are as in the Haskell prelude. Below, we assume that the mentioned functions have already been translated into type theory. Moreover, for simplicity reasons, we use the structurally recursive versions of these functions rather than the formal translations we would obtain with our new method.

Functional version:

$$
\begin{array}{l}
\mathtt{fix\ sumdel}\colon \mathtt{List\ N} \Rightarrow \mathtt{N}\\
\quad \mathtt{sumdel}(\mathtt{nil}) = 0\\
\quad \mathtt{sumdel}(\mathtt{cons}(n, l)) = n + \mathtt{sum}(\mathtt{map}([x]\mathtt{sumdel}(\mathtt{cons}(n, \mathtt{delete}(x, l))), l))
\end{array}
$$

The actual value computed by the function is $\mathtt{sumdel}(n_1, \ldots, n_k) = n_1\mathtt{sd}(k)$ where $\mathtt{sd}(0) = 0$ and $\mathtt{sd}(k+1) = k\mathtt{sd}(k) + 1$.

Type-theoretic version: $\mathsf{Sumdel} = \langle\mathsf{sumdelAcc}, \mathsf{sumdel}\rangle\colon \mathsf{List\ N} \rightharpoonup \mathsf{N}$ with:

$$
\begin{array}{l}
\mathsf{sumdelAcc}\colon \mathsf{List\ N} \to \mathsf{Prop}\\
\quad \mathsf{sumdelacc_{nil}}\colon (\mathsf{sumdelAcc\ nil})\\
\quad \mathsf{sumdelacc_{cons}}\colon (n\colon \mathsf{N}; l\colon \mathsf{List\ N}; h\colon \mathsf{mapAcc\ G}\ l)(\mathsf{sumdelAcc\ cons}(n, l))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{sumdel}\colon (l\colon \mathsf{List\ N}; \mathsf{sumdelAcc}\ l) \to \mathsf{N}\\
\quad \mathsf{sumdel\ nil\ sumdelacc_{nil}} = 0\\
\quad \mathsf{sumdel\ cons}(n, l)\ (\mathsf{sumdelacc_{cons}}\ n\ l\ h) = n + \mathsf{sum}\ \ \mathsf{Map}_{[h]}(\mathsf{G}, l)
\end{array}
$$

with $\mathsf{G} \equiv \langle\mathsf{gAcc}, \mathsf{g}\rangle\colon \mathtt{N} \rightharpoonup \mathtt{N}$ where

$$
\begin{array}{l}
\mathsf{gAcc} = [x\colon \mathtt{N}](\mathsf{sumdelAcc\ cons}(n, (\mathsf{delete}\ x\ l)))\colon \mathsf{N} \to \mathsf{Prop}\\
\mathsf{g} = [x\colon \mathtt{N}; h\colon \mathsf{gAcc}\ x]\mathsf{Sumdel}_{[h]}(\mathsf{cons}(n, (\mathsf{delete}\ x\ l)))\colon (x\colon \mathsf{N}; \mathsf{gAcc}\ x) \to \mathsf{N}
\end{array}
$$

Notice that G is local to Sumdel and hence, n and l are known while defining G.

The important feature of this translation, not possible in the old one, is that we can assign a precise domain predicate gAcc to the local function generated by the λ-abstraction. Notice that in the body of the main function, the local function is applied only to arguments that satisfy gAcc, so termination is ensured.

Partial Application: `is_div2`. Functional version:

$$
\begin{array}{l}
\mathtt{fix\ is\_div2}\colon \mathtt{List\ N} \Rightarrow \mathtt{List\ N}\\
\qquad \mathtt{is\_div2}(xs) = \mathtt{map}(\mathtt{my\_mod}(2), xs)
\end{array}
$$

where the function `my_mod` is the one defined at the end of Section 2. Given a list of numbers, this function returns a list of 0's and 1's depending on whether the numbers in the list are divisible by 2 or not, respectively.

Observe that the type-theoretic version of `my_mod` is the same as before since this function does not present any of the problematic aspects on which the method has been changed. Let $\mathsf{My\_Mod} = \langle \mathsf{my\_modAcc}, \mathsf{my\_mod} \rangle \colon \mathsf{N}, \mathsf{N} \rightharpoonup \mathsf{N}$ with $\mathsf{my\_modAcc}$ and $\mathsf{my\_mod}$ as defined on page 122.

Type-theoretic version: $\mathsf{Is\_Div2} = \langle \mathsf{is\_div2Acc}, \mathsf{is\_div2} \rangle \colon \mathsf{List\ N} \rightharpoonup \mathsf{List\ N}$ with:

$$\begin{array}{l} \mathsf{is\_div2Acc} \colon \mathsf{List\ N} \to \mathsf{Prop} \\ \quad \mathsf{is\_div2acc} \colon (xs \colon \mathsf{List\ N}; h \colon \mathsf{mapAcc\ My\_Mod}(2)\ xs)(\mathsf{is\_div2Acc}\ xs) \end{array}$$

$$\begin{array}{l} \mathsf{is\_div2} \colon (xs \colon \mathsf{List\ N}; \mathsf{is\_div2Acc}\ xs) \to \mathsf{List\ N} \\ \quad \mathsf{is\_div2}\ xs\ (\mathsf{is\_div2acc}\ xs\ h) = \mathsf{Map}_{[h]}(\mathsf{My\_Mod}(2), xs) \end{array}$$

Notice that the translation of the partial application `my_mod`(2) does not introduce any domain constrains in the type of the constructor $\mathsf{is\_div2acc}$. Given an element x in the list xs, the application of the function $\mathsf{My\_Mod}(2)$ to the argument x will only be possible if we can find a proof of $(\mathsf{my\_modAcc}\ 2\ x)$. This is taken care of in the definition of mapAcc.

Necessity of Impredicativity: `itz`. The following example shows that it is necessary to have an impredicative type theory if we want our method to apply to every functional program. Functional version:

$$\begin{array}{l} \mathtt{fix\ itz} \colon \mathtt{N} \to \mathtt{N}, \mathtt{N} \Rightarrow \mathtt{N} \\ \qquad \mathtt{itz}(f, 0) = f\ 0 \\ \qquad \mathtt{itz}(f, \mathtt{succ}(n)) = f\ \ \mathtt{itz}(\mathtt{itz}(f), n) \end{array}$$

Type-theoretic version: $\mathsf{Itz} = \langle \mathsf{itzAcc}, \mathsf{itz} \rangle \colon (\mathsf{N} \rightharpoonup \mathsf{N}), \mathsf{N} \rightharpoonup \mathsf{N}$ with:

$$\begin{array}{l} \mathsf{itzAcc} \colon (\mathsf{N} \rightharpoonup \mathsf{N}) \to \mathsf{N} \to \mathsf{Prop} \\ \quad \mathsf{itzacc}_0 \colon (f \colon \mathsf{N} \rightharpoonup \mathsf{N}; h \colon \mathsf{dom}_f\ 0)(\mathsf{itzAcc}\ f\ 0) \\ \quad \mathsf{itzacc}_{\mathsf{succ}} \colon (f \colon \mathsf{N} \rightharpoonup \mathsf{N}; n \colon \mathsf{N}; h_1 \colon \mathsf{itzAcc\ Itz}(f)\ n; \\ \qquad\qquad h_2 \colon \mathsf{dom}_f\ \mathsf{Itz}_{[h_1]}(\mathsf{Itz}(f), n))(\mathsf{itzAcc}\ f\ \mathsf{succ}(n)) \end{array}$$

$$\begin{array}{l} \mathsf{itz} \colon (f \colon \mathsf{N} \rightharpoonup \mathsf{N}; n \colon \mathsf{N}; \mathsf{itzAcc}\ f\ n) \to \mathsf{N} \\ \quad \mathsf{itz}\ f\ 0\ (\mathsf{itzacc}_0\ f\ h) = f_{[h]}(0) \\ \quad \mathsf{itz}\ f\ \mathsf{succ}(n)\ (\mathsf{itzacc}_{\mathsf{succ}}\ f\ n\ h_1\ h_2) = f_{[h_2]}(\mathsf{Itz}_{[h1]}(\mathsf{Itz}(f), n)) \end{array}$$

We see that impredicativity is essential when we follow our method to formalise this example. When defining itzAcc, the constructor $\mathsf{itzacc}_{\mathsf{succ}}$ quantifies over all partial functions $f \colon \mathsf{N} \rightharpoonup \mathsf{N}$ (and, therefore, over the domain predicates of all those functions) and in the body of the constructor $\mathsf{itzacc}_{\mathsf{succ}}$ the function $\mathsf{Itz}(f)$ is itself an argument of itzAcc.

The alternative to use a predicative hierarchy of type universes $\mathsf{U}_0, \mathsf{U}_1, \mathsf{U}_2, \ldots$ does not work in this example. Function spaces would need to be stratified too, according to the universe in which the domain predicate lives, so we would have

$$A \rightharpoonup_i B = \Sigma P \colon A \to \mathsf{U}_i.(x \colon A; P\ x) \to B$$

Since we are quantifying over $A \to \mathsf{U}_i$, predicatively it must be $A \rightharpoonup_i B \colon \mathsf{U}_j$ with $j > i$. That is, we have at least $A \rightharpoonup_i B \colon \mathsf{U}_{i+1}$.

In the case of Itz, if we try to assign universe levels, that is, if we try to give it the type $\mathsf{Itz}\colon (\mathsf{N} \rightharpoonup_i \mathsf{N}), \mathsf{N} \rightharpoonup_j \mathsf{N}$ for some i and j, we reach a contradiction regardless of what i and j are. To start with, we have $\mathsf{itzAcc}\colon (\mathsf{N} \rightharpoonup_i \mathsf{N}) \to \mathsf{N} \to \mathsf{U}_j$. The first constructor itzacc_0 contains a quantification on $\mathsf{N} \rightharpoonup_i \mathsf{N}$, so its type must be at least in U_{i+1}. Thus, also the universe of itzAcc must be at least U_{i+1}, that is, $j \geqslant i+1$. Now, in the constructor $\mathsf{itzacc}_{\mathsf{succ}}$ we have the subterm $\mathsf{Itz}(\mathsf{Itz}(f), n)$, but this term does not type-check because $\mathsf{Itz}\colon (\mathsf{N} \rightharpoonup_i \mathsf{N}), \mathsf{N} \rightharpoonup_j \mathsf{N}$ and $\mathsf{Itz}(f)\colon \mathsf{N} \rightharpoonup_j \mathsf{N}$ with $j \geqslant i+1$. Hence, Itz cannot be applied to $\mathsf{Itz}(f)$ because the type is not correct: it expects an argument of type $\mathsf{N} \rightharpoonup_i \mathsf{N}$ and it gets one of type $\mathsf{N} \rightharpoonup_j \mathsf{N}$ with $j \geqslant i+1$.

5 Conclusions

This article presents a method to translate functional programs into type theory based on the one previously presented in [BC04]. The new approach relies on a type of partial functions whose elements are pairs consisting of a domain predicate and a function depending on a proof of the predicate. The problems that were left open in [BC04] are now solved: functional arguments are dealt with by lifting their domain conditions to the main call; λ-abstractions denote partial functions with domain condition generated locally; partial application is interpreted by just fixing the given parameters both in the domain predicate and the function.

These results are obtained at the cost of two disadvantages. First of all, we need an impredicative type theory for the method to be applicable to all functional programs (see example `itz`). This is indispensable to obtain a general result, but in most practical cases the method could be adapted to work on a predicative type theory with type universes.

As we have already mentioned, this paper is the last in a series of articles [Bov01, BC01, Bov02a, Bov02b, Bov03, BC04] aimed at representing general recursive functions in type theory. See the section on related work in [BC04] for a thorough discussion of the literature regarding representations of recursive functions in logical frameworks.

Acknowledgements. We would like to thank Thierry Coquand for his constructive criticism on earlier versions of this article and for useful comments on the type of partial functions and on the use of impredicativity. We are also grateful to Christine Paulin-Mohring for giving us a clarifying explanation of her method to extract computational content from the accessibility predicates.

References

[BC01] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 121–135, September 2001.

[BC04] A. Bove and V. Capretta. Modelling general recursion in type theory. To appear in *Mathematical Structures in Computer Science*. Available on the WWW: `http://www.cs.chalmers.se/~bove/Papers/general_presentation.ps.gz`, May 2004.

[Bla03] Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. In M. Hofmann, editor, *Typed Lambda Calculi and Applications: 6th International Conference, TLCA 2003*, volume 2701 of *LNCS*, pages 46–59. Springer-Verlag, 2003.

[Bov01] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

[Bov02a] A. Bove. Mutual general recursion in type theory. Technical Report, Chalmers University of Technology. Available on the WWW: `http://www.cs.chalmers.se/~bove/Papers/mutual_rec.ps.gz`, May 2002.

[Bov02b] Ana Bove. *General Recursion in Type Theory.* PhD thesis, Chalmers University of Technology, Department of Computing Science, November 2002. Available on the WWW: `http://cs.chalmers.se/~bove/Papers/phd_thesis.ps.gz`.

[Bov03] A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, number 2646 in Lecture Notes in Computer Science, pages 39–58, March 2003.

[Cap04] Venanzio Capretta. A polymorphic representation of induction-recursion. Draft paper. Available from `http://www.science.uottawa.ca/~vcapr396/`, 2004.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CM85] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 1985.

[Coq02] Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 7.4.* INRIA, 2002.

[DDG98] C. Dubois and V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: Domains as inductive predicates. In M. Kerber, editor, *CADE-15, The 15th International Conference on Automated Deduction*, pages 53–62, July 1998. WORKSHOP Mechanization of Partial Functions.

[Dyb00] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

[FFL97] S. Finn, M.P. Fourman, and J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.

[Jon03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, April 2003.

[Pau] Christine Paulin. How widely applicable is Coq? Contribution to the Coq mailing list, 19 Aug 2002, `http://pauillac.inria.fr/bin/wilma_hiliter/coq-club/200208/msg00003.html`.

Higher-Order Recursion Abstraction: How to Make Ackermann, Knuth and Conway Look Like a Bunch of Primitives, Figuratively Speaking

Baltasar Trancón y Widemann
Ilmenau University of Technology, DE
baltasar.trancon@tu-ilmenau.de

September 17, 2018

Abstract

The Ackermann function is a famous total recursive binary function on the natural numbers. It is the archetypal example of such a function that is not primitive recursive, in the sense of classical recursion theory. However, and in seeming contradiction, there are generalized notions of total recursion, for which the Ackermann function is in fact primitive recursive, and often featured as a witness for the additional power gained by the generalization. Here, we investigate techniques for finding and analyzing the primitive form of complicated recursive functions, namely also Knuth's and Conway's arrow notations, in particular by recursion abstraction, in a framework of functional program transformation.

1 Introduction

The Ackermann function is a famous total binary function on the natural numbers, given by the recursive equations:

$$\begin{array}{rlr} A(0,n) &= n+1 & \\ A(m,0) &= A(m-1,1) & (m>0) \\ A(m,n) &= A\big(m-1, A(m,n-1)\big) & (m,n>0) \end{array}$$

It is not a particularly useful function in practice; its fame (and ubiquitous appearance in undergraduate computer science curricula) stems from the fact that it is one of the simplest examples of a total, recursive (that is, recursively computable) function which is not *primitive* recursive. The class of primitive recursive functions is defined inductively, characteristically involving a recursion operator that, simply speaking, allows the value of a function at n depend on its value at $n-1$, without explicit self-reference.

Intuitively (and provably), the doubly nested self-reference of the Ackermann function in the third equation refuses to be adapted to the primitive recursion scheme. But one may find seemingly contradictory statements: the notion of primitive recursion is relative to its underlying datatype $\mathbb{N}$; the Ackermann function *does* have a representation in terms of analogous recursion operators for suitable, more powerful datatypes [6, 2]. A systematic introduction to recursion operators on lists [3] gives the Ackermann function (disguised as a list function) as a high-end example, but does not discuss the generalization to related functions. Here, we start off where they concluded. The purpose of the present article is threefold:

1. Recall from [3] how to obtain a "primitive" representation of Ackermann's and related functions by means of systematic *recursion abstraction* in a higher-order functional programming language.

2. Pinpoint how exactly these representations are "cheating" with respect to the original, first-order definition of primitive recursion.

3. Generalize the approach to (apparently) even more complicated recursive functions.

We begin by summarizing formal details about primitive recursion in mathematical theory and in functional programming. The expert reader is welcome to safely skip the next, preliminary section. The remainder of the article is about three concrete examples, in increasing order of logical complexity: the Ackermann function (archetypal), Knuth's up-arrows (slightly more powerful), Conway's chained arrows (significantly more powerful). They form a well-known hierarchy of operators for the arithmetics of "insanely large" numbers.

Even the non-expert reader might enjoy a quick glance ahead at Figures 1 and 2 on page 4, where the classical and primitive definition forms are contrasted in synopsis. With a little contemplation, all elements of the latter, except the *fold* operators, can be found verbatim in the former. The remainder of this article is a thorough exploration of the formal details that constitute their respective equivalence.

The program code scattered throughout this article is a valid literate Haskell program. It makes use of standard Haskell'98 features only, with the exception of some `Rank2Types` in meta-level analyses. Note that, for better fit with the mathematical notation, the first-fit rule of Haskell pattern matching is shunned; all overlapped patterns are explicitly guarded.

2 Primitive Recursion

2.1 Primitive Recursion in Mathematics

The conventional definition of the class of primitive recursive functions is inductive. It contains the nullary *zero* function and the unary *successor* function, and the projections, is closed under composition and under a certain recursion operator:

Let e and g be k-ary and $(k{+}2)$-ary primitive recursive functions, respectively. Then the unique $(k{+}1)$-ary function h given by the recursive equations

$$\begin{aligned} h(x_1, \dots, x_k; 0) &= e(x_1, \dots, x_k) \\ h(x_1, \dots, x_k; n) &= g\big(x_1, \dots, x_k; n-1, h(x_1, \dots, x_k; n-1)\big) \qquad (n > 0) \end{aligned}$$

is also primitive recursive. Then the self-referential definition of h can be replaced equivalently by a "primitive" one, specifying e and g instead. It has been shown by Ackermann that a precursor to the function now named after him is *not* of this form. Double recursion [7] has been introduced ad-hoc for Ackermann's and related functions, but obviously a nesting depth of two is no natural complexity limit—why not have triple recursion, and so forth? Less arbitrary limits are given by type systems that allow recursive computations on other datatypes than just natural numbers; see below.

2.2 Primitive Recursion in Functional Programming

The algebraic approach to functional programming [5, 1] generalizes the notion of a recursion operator to arbitrary algebraic datatypes (and with some effort, beyond). In a functional programming context, one can do away with the auxiliary arguments $x_1, \dots, x_k$ to the generating functions e and g, by having them implicitly as free variables in the defining expression. One can also do away with the $(k+1)$-th argument without losing essential power [4], thus arriving at a scheme called variously *structural recursion*, *iteration*, *fold* or *catamorphism*.

In the base case of the natural numbers (represented here by the Haskell datatype *Integer*, tacitly excluding negative numbers and $\bot$), the operator is:

$$\begin{aligned} &\mathit{foldn}\ g\ e\ 0 &&= e \\ &\mathit{foldn}\ g\ e\ n \mid n > 0 &&= g\ (\mathit{foldn}\ g\ e\ (n-1)) \end{aligned}$$

This form, as well as the constructor operations *zero* and *succ*, can be deduced from the category-theoretic representation of the natural numbers as the initial algebra of a certain functor. We do not repeat the details here, but one consequence is highly relevant: Fold operators have the nice universal property of *uniqueness*. On one hand, we can of course retrieve the preceding definition, where the generated function is named h:

From $h \equiv \mathit{foldn}\ g\ e$ we can conclude

$$\begin{aligned} &h\ 0 && \equiv e \\ &h\ n \mid n > 0 && \equiv g\ (h\ (n-1)) \end{aligned}$$

But, on the other hand, the converse does also hold: From

$$\begin{aligned} &h\ 0 && \equiv e \\ &h\ n \mid n > 0 && \equiv g\ (h\ (n-1)) \end{aligned}$$

we may deduce that necessarily $h \equiv \mathit{foldn}\ g\ e$.

This will become our tool of recursion abstraction in the following sections. As pointed out in [3] for the Ackermann example, the characteristic feature of "morally primitive" recursive functions is that this abstraction needs to applied more than once.

The resulting recursive functions do not look particular powerful at first glance: mathematically, $\mathit{foldn}\ g\ e\ n$ is simply $g^n(e)$. Their hidden power comes from the polymorphic nature of the fold operator, which can define functions from the natural numbers to arbitrary underlying datatypes.

$$\begin{aligned} &\mathbf{type}\ \mathit{FoldN}\ a = (a \to a) \to a \to \mathit{Integer} \to a \\ &\mathit{foldn} :: \mathit{FoldN}\ a \end{aligned}$$

The fold operator for lists is arguably the most popular one; see [3]. This is not only due to the fact that lists are the workhorse of datatypes: List folding is both reasonably simple regarding type and universal properties, and gives to nontrivial, useful recursive algorithms already from very simple generator functions.

$$\begin{aligned} &\mathit{foldr}\ g\ e\ [\,] && = e \\ &\mathit{foldr}\ g\ e\ (a : \mathit{xs}) && = g\ a\ (\mathit{foldr}\ g\ e\ \mathit{xs}) \end{aligned}$$

The type scheme for list folding is

$$\begin{aligned} &\mathbf{type}\ \mathit{FoldR}\ a\ b = (a \to b \to b) \to b \to [a] \to b \\ &\mathit{foldr} :: \mathit{FoldR}\ a\ b \end{aligned}$$

where we now have two type parameters; namely a for the list elements to be consumed, and b for the underlying datatype in which to perform the computation.

The associated universal property is that for a recursive function h on lists, the equations

$$\begin{aligned} &h\ [\,] \equiv e \\ &h\ (a : \mathit{xs}) \equiv g\ a\ (h\ \mathit{xs}) \end{aligned}$$

hold if and only if $h \equiv \mathit{foldr}\ g\ e$.

The trick behind apparently primitive representations of Ackermann and friends is to instantiate the fold type schemes with an underlying datatype that is strictly more powerful than the natural numbers, that is, that cannot be Gödel-encoded and -decoded by primitive recursive functions in the original sense.

Consider a type system that has some algebraic types. Then the class of primitive recursive functions *relative* to that type system is defined inductively as containing some elementary functions, and being closed under composition and the fold operators for all algebraic datatypes, instantiated with all valid types.

The classical definition can be retrieved as the special case of the type system whose only datatype is $\mathbb{N}$, and whose function types are all of the first-order form $\mathbb{N}^k \to \mathbb{N}$. The programme of the present article is to consider functions that are not primitive recursive in that classical sense, and to study the derivations of their respective primitive forms in a more powerful type system, namely one with higher-order function types, as usual in functional programming languages.

Our chief interest lies in the program transformation techniques necessary to take a *self-referential definition*, in terms of generally recursive equations, to a *primitive definition*, that is an expression without self-references, and composed only of legal building blocks for primitive recursive functions.

$$\begin{aligned} A(0,n) &= n+1 && \\ A(m,0) &= A(m-1,1) && (m>0) \\ A(m,n) &= A(m-1,A(m,n-1)) && (m,n>0) \end{aligned}$$

$$\begin{aligned} a \uparrow^{0} b &= a\,b && \\ a \uparrow^{n} 0 &= 1 && \\ a \uparrow^{n} b &= a \uparrow^{n-1} (a \uparrow^{n} (b-1)) && (n,b>0) \end{aligned}$$

$$\begin{aligned} \langle p \to q\rangle &= (p+1)^{(q+1)} && \\ \langle X \to p \to 0\rangle &= \langle X \to p\rangle && \\ \langle X \to 0 \to q\rangle &= \langle X \to 0\rangle && \\ \langle X \to p \to q\rangle &= \langle X \to (\langle X \to (p-1) \to q\rangle - 1) \to (q-1)\rangle && (p,q>0) \end{aligned}$$

Figure 1: Exercise—Ackermann, Knuth and Conway in traditional appearance

2.3 Elimination of Recursion

The skeptical reader might wonder what qualifies the fold operators as particularly primitive, seeing that their definitions make blatant use of self-reference. But that is an artifact of the way algebraic datatypes are commonly defined. An alternative definition technique, namely *Church encoding*, does not have this property. A Church-encoded version of the natural numbers is the following non-recursive type definition:

$$\mathbf{type}\ \mathit{Nat} = \forall\, a.\,(a \to a) \to a \to a$$

It allows the construction of numbers by the usual operations, by strategic deferral of their respective interpretations z and s:

$$\begin{aligned} &\mathit{zero} :: \mathit{Nat} \qquad\qquad && \mathit{succ} :: \mathit{Nat} \to \mathit{Nat} \\ &\mathit{zero}\ s\ z = z && \mathit{succ}\ n\ s\ z = s\ (n\ s\ z) \end{aligned}$$

A trivial coercion to standard Haskell numbers can be given by providing the obvious interpretations.

$$\begin{aligned} &\mathit{toInteger} :: \mathit{Nat} \to \mathit{Integer} \\ &\mathit{toInteger}\ n = n\ (+1)\ 0 \end{aligned}$$

But, less trivially, every Church-encoded number implements the action of the fold operator on itself, by substituting g for s and e for z, respectively, which can thus be defined without self-reference as

$$\begin{aligned} &\mathit{foldn}' :: (a \to a) \to a \to \mathit{Nat} \to a \\ &\mathit{foldn}'\ g\ e\ n = n\ g\ e \end{aligned}$$

and is easily seen to be equivalent to the original:

$$\mathit{foldn}'\ g\ e \equiv \mathit{foldn}\ g\ e \circ \mathit{toInteger}$$

For a general discussion of this construction for arbitrary algebraic datatype signatures, and its equivalence to the initial algebra approach, see [8].

$$\begin{aligned}
&\mathit{ack} \;\;\;= \qquad\qquad\quad \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f \qquad (f\ 1)\ \ )\qquad\qquad (+1)\\
&\mathit{knuth} = \qquad\qquad\quad \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f \qquad\ 1 \qquad)\qquad\qquad\ \circ (*)\\
&\mathit{cback} = \mathit{foldr}\ (\lambda o\ k \to \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ (f \circ (-\ 1))\ (k\ 0\ o))\ (\lambda p \to k\ p\ o))\ \mathit{cpow}\\
&\quad \textbf{where}\ \mathit{cpow}\ q\ p = (p+1)\ \hat{}\ (q+1)
\end{aligned}$$

Figure 2: Solution—Ackermann, Knuth and Conway looking like a bunch of primitives

3 The Ackermann Function

3.1 Derivation

We start with the Haskell version of the original self-referential definition

$$\begin{aligned}
&\mathit{ack}_0\ 0\ \ n && = n+1\\
&\mathit{ack}_0\ m\ 0 \mid m > 0 && = \mathit{ack}_0\ (m-1)\ 1\\
&\mathit{ack}_0\ m\ n \mid m > 0 \wedge n > 0 && = \mathit{ack}_0\ (m-1)\ (\mathit{ack}_0\ m\ (n-1))
\end{aligned}$$

and proceed by successive application of elementary transformation steps. Each new version is indexed differently, such that all can coexist in a single literate Haskell source file for this whole article.

In a first transformation step, we exploit currying to eliminate the second function argument n and thus unify the second and third equation in an auxiliary function aux.

$$\begin{aligned}
&\mathit{ack}_1\ 0 && = (+1)\\
&\mathit{ack}_1\ m \mid m > 0 && = \mathit{aux}\\
&\quad \textbf{where}\ \mathit{aux}\ 0 && = \mathit{ack}_1\ (m-1)\ 1\\
&\qquad\qquad\ \mathit{aux}\ n \mid n > 0 && = \mathit{ack}_1\ (m-1)\ (\mathit{ack}_{1\mathrm{a}}\ m\ (n-1))
\end{aligned}$$

We factor out the common term $\mathit{ack}\ (m-1)$ by beta expansion to an argument f. What seems like a simple matter of abbreviation here will prove a very useful preparation step later.

$$\begin{aligned}
&\mathit{ack}_{1\mathrm{a}}\ 0 && = (+1)\\
&\mathit{ack}_{1\mathrm{a}}\ m \mid m > 0 && = \mathit{aux}\ (\mathit{ack}_{1\mathrm{a}}\ (m-1))\\
&\quad \textbf{where}\ \mathit{aux}\ f\ 0 && = f\ 1\\
&\qquad\qquad\ \mathit{aux}\ f\ n \mid n > 0 && = f\ (\mathit{ack}_{1\mathrm{a}}\ m\ (n-1))
\end{aligned}$$

The single most tricky point is to get rid of the back-reference from aux to ack. Note that aux is only ever applied to $f = \mathit{ack}\ (m-1)$, hence we may substitute[1]

$$\mathit{ack}\ m \equiv \mathit{aux}\ (\mathit{ack}\ (m-1)) \equiv \mathit{aux}\ f$$

and obtain:

$$\begin{aligned}
&\mathit{ack}_{1\mathrm{b}}\ 0 && = (+1)\\
&\mathit{ack}_{1\mathrm{b}}\ m \mid m > 0 && = \mathit{aux}\ (\mathit{ack}_{1\mathrm{b}}\ (m-1))\\
&\quad \textbf{where}\ \mathit{aux}\ f\ 0 && = f\ 1\\
&\qquad\qquad\ \mathit{aux}\ f\ n \mid n > 0 && = f\ (\mathit{aux}\ f\ (n-1))
\end{aligned}$$

Now we have ack in a form generated by the fold operator from a function aux whose definition is independent from ack. We may invoke the universal property and conclude:

$$\begin{aligned}
&\mathit{ack}_2 = \mathit{foldn}\ \mathit{aux}\ (+1)\\
&\quad \textbf{where}\ \mathit{aux}\ f\ 0 && = f\ 1\\
&\qquad\qquad\ \mathit{aux}\ f\ n \mid n > 0 && = f\ (\mathit{aux}\ f\ (n-1))
\end{aligned}$$

[1] In case of doubt about the validity of this slightly simplified argument, see section 6 for details.

It remains to be established that the self-referential definition of the generator *aux* can be reduced to a primitive form. As it happens, it is already of the right shape.

$$\begin{aligned}&\mathit{ack}_3 = \mathit{foldn}\ \mathit{aux}\ (+1)\\&\quad \textbf{where}\ \mathit{aux}\ f = \mathit{foldn}\ f\ (f\ 1)\end{aligned}$$

So, in summary, we have the Ackermann function given by an entirely non-self-referential expression that invokes two nested instances of the fold operator.

$$\mathit{ack}_4 = \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ (f\ 1))\ (+1)$$

3.2 Verification

Since the final version of the Ackermann function admittedly looks very different from the original one, it is instructive to expand it back and demonstrate the inductive equivalence to the original recursive equations. As the base case for n we have

$$\begin{aligned}\mathit{ack}_4\ 0\ n &\equiv \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ (f\ 1))\ (+1)\ 0\ n\\&\equiv (+1)\ n\\&\equiv n+1\end{aligned}$$

and as the inductive case we have

$$\begin{aligned}\mathit{ack}_4\ m \mid m>0 &\equiv \qquad\qquad\qquad\qquad \mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ (f\ 1))\ (+1)\ \ m\\&\equiv (\lambda f \rightarrow \mathit{foldn}\ f\ (f\ 1))\ (\mathit{foldn}\ (\lambda f \rightarrow \mathit{foldn}\ f\ (f\ 1))\ (+1)\ (m-1))\\&\equiv (\lambda f \rightarrow \mathit{foldn}\ f\ (f\ 1))\ (\mathit{ack}_4 \qquad\qquad\qquad\qquad\qquad\quad (m-1))\\&\equiv \mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\end{aligned}$$

which cannot be unfolded yet. Proceeding to the second argument, we have the base case

$$\begin{aligned}\mathit{ack}_4\ m\ 0 &\equiv \mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\ 0\\&\equiv \qquad\qquad\qquad\qquad\quad \mathit{ack}_4\ (m-1)\ 1\end{aligned}$$

and the inductive case

$$\begin{aligned}\mathit{ack}_4\ m\ n \mid n>0 &\equiv \qquad\qquad\qquad \mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\ \ n\\&\equiv \mathit{ack}_4\ (m-1)\ (\mathit{foldn}\ (\mathit{ack}_4\ (m-1))\ (\mathit{ack}_4\ (m-1)\ 1)\ (n-1))\\&\equiv \mathit{ack}_4\ (m-1)\ (\mathit{ack}_4\ m \qquad\qquad\qquad\qquad\qquad\qquad\quad (n-1))\end{aligned}$$

by back-substitution. □

3.3 Analysis

So, in a sense, the Ackermann function is primitive after all! As discussed above, the polymorphic fold operator glosses over the fact that underlying datatypes of different, incommensurable power are used here. To make things more explicit, consider monomorphic instances of *foldn*:

$$\begin{aligned}&\mathit{foldn}_1 :: \mathit{FoldN}\ \mathit{Integer}\\&\mathit{foldn}_1 = \mathit{foldn}\end{aligned}$$

$$\begin{aligned}&\mathit{foldn}_2 :: \mathit{FoldN}\ (\mathit{Integer} \rightarrow \mathit{Integer})\\&\mathit{foldn}_2 = \mathit{foldn}\end{aligned}$$

Using these, we can make the different roles explicit:

$$\mathit{ack}_5 = \mathit{foldn}_2\ (\lambda f \rightarrow \mathit{foldn}_1\ f\ (f\ 1))\ (+1)$$

Contradiction is avoided by the fact that the higher-order datatype underlying the outer instance cannot be converted to and from the traditional first-order datatype underlying the inner one by means of primitive recursive conversion functions.

4 Knuth's Up-Arrows

A natural next candidate function to investigate is Knuth's up-arrow, which is known to have quite similar but mildly more flexible behavior than the Ackermann function. The usual recursive definition is, for $a, b \geqslant 0; n > 0$:

$$\begin{array}{rll} a \uparrow^1 b &= a^b & \\ a \uparrow^n 0 &= 1 & \\ a \uparrow^n b &= a \uparrow^{n-1} \big(a \uparrow^n (b-1)\big) & \qquad (n > 1; b > 0) \end{array}$$

This can be canonically extended to $n = 0$:

$$\begin{array}{rll} a \uparrow^0 b &= a\, b & \\ a \uparrow^n 0 &= 1 & \qquad (n > 0) \\ a \uparrow^n b &= a \uparrow^{n-1} \big(a \uparrow^n (b-1)\big) & \qquad (n, b > 0) \end{array}$$

which is also a neat case of consistent abuse of notation, seeing that

$$a \uparrow^n b = a \underbrace{\uparrow \cdots \uparrow}_{n} b$$

4.1 Derivation

This definition translates straightforwardly to Haskell:

$$\begin{array}{lcl} \mathit{knuth}_0\ a\ 0\ \ b & = & a * b \\ \mathit{knuth}_0\ a\ n\ 0 \mid n > 0 & = & 1 \\ \mathit{knuth}_0\ a\ n\ b \mid n > 0 \wedge b > 0 & = & \mathit{knuth}_0\ a\ (n-1)\ (\mathit{knuth}_0\ a\ n\ (b-1)) \end{array}$$

In analogy to the Ackermann example, we curry argument b.

$$\begin{array}{l} \mathit{knuth}_1\ a\ 0 \qquad\qquad = (a*) \\ \mathit{knuth}_1\ a\ n \mid n > 0 = \mathit{aux} \\ \quad \textbf{where}\ \mathit{aux}\ 0 \qquad\quad = 1 \\ \qquad\qquad\ \ \mathit{aux}\ b \mid b > 0 = \mathit{knuth}_1\ a\ (n-1)\ (\mathit{knuth}_1\ a\ n\ (b-1)) \end{array}$$

If we had tried this example first, we would probably have gotten stuck at this point! But following the procedure for the Ackermann function, we beta-expand the recursive subterm $\mathit{knuth}\ a\ (n-1)$, even if there is only one occurrence, and the step does not simplify anything at first glance.

$$\begin{array}{l} \mathit{knuth}_{1\mathrm{a}}\quad a\ 0 \qquad\qquad = (a*) \\ \mathit{knuth}_{1\mathrm{a}}\quad a\ n \mid n > 0 = \mathit{aux}\ (\mathit{knuth}_{1\mathrm{a}}\ a\ (n-1)) \\ \quad \textbf{where}\ \mathit{aux}\ f\ 0 \qquad\quad = 1 \\ \qquad\qquad\ \ \mathit{aux}\ f\ \ b \mid b > 0 = f\ (\mathit{knuth}_{1\mathrm{a}}\ a\ n\ (b-1)) \end{array}$$

We find that, in the context of the last equation, we can again substitute

$$\mathit{knuth}\ a\ n \equiv \mathit{aux}\ (\mathit{knuth}\ a\ (n-1)) \equiv \mathit{aux}\ f$$

and arrive at:

$$\begin{array}{l} \mathit{knuth}_{1\mathrm{b}}\quad a\ 0 \qquad\qquad = (a*) \\ \mathit{knuth}_{1\mathrm{b}}\quad a\ n \mid n > 0 = \mathit{aux}\ (\mathit{knuth}_{1\mathrm{b}}\ a\ (n-1)) \\ \quad \textbf{where}\ \mathit{aux}\ f\ 0 \qquad\quad = 1 \\ \qquad\qquad\ \ \mathit{aux}\ f\ \ b \mid b > 0 = f\ (\mathit{aux}\ f\ (b-1)) \end{array}$$

Now we may invoke the universal property of the fold operator to abstract from the outer recursion first, as in the previous example.

$$\begin{array}{l}
\mathit{knuth}_{2\mathrm{a}}\ a = \mathit{foldn}\ \mathit{aux}\ (a*) \\
\quad \textbf{where}\ \begin{array}[t]{lll} \mathit{aux}\ f\ 0 & & = 1 \\ \mathit{aux}\ f\ b & \mid b > 0 & = f\ (\mathit{aux}\ f\ (b-1)) \end{array}
\end{array}$$

Or, alternatively, we may abstract from the inner recursion first.

$$\begin{array}{lll}
\mathit{knuth}_{2\mathrm{b}}\ a\ 0 & & = (a*) \\
\mathit{knuth}_{2\mathrm{b}}\ a\ n & \mid n > 0 & = (\lambda f \to \mathit{foldn}\ f\ 1)\ (\mathit{knuth}_{2\mathrm{b}}\ a\ (n-1))
\end{array}$$

Either way, double recursion abstraction leads to:

$$\mathit{knuth}_3\ a = \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ 1)\ (a*)$$

For the friends of point-free humor this is:

$$\mathit{knuth}_4 = \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ 1) \circ (*)$$

Visual type inference tells us that the nesting pattern is the same as for the Ackermann function, a first-order instance of fold nested within a second-order instance.

$$\mathit{knuth}_5 = \mathit{foldn}_2\ (\lambda f \to \mathit{foldn}_1\ f\ 1) \circ (*)$$

4.2 Verification

Although the mode of derivation has paralleled the, already verified, Ackermann function quite closely, it might be assuring to repeat the verification process for Knuth's arrows. As the base case for n we have

$$\begin{aligned}
\mathit{knuth}_3\ a\ 0\ b &\equiv \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ 1)\ (a*)\ 0\ b \\
&\equiv (a*)\ b \\
&\equiv a * b
\end{aligned}$$

and as the inductive case we have:

$$\begin{aligned}
\mathit{knuth}_3\ a\ n \mid n > 0 &\equiv \mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ 1)\ (a*)\ \ n \\
&\equiv (\lambda f \to \mathit{foldn}\ f\ 1)\ (\mathit{foldn}\ (\lambda f \to \mathit{foldn}\ f\ 1)\ (a*)\ (n-1)) \\
&\equiv (\lambda f \to \mathit{foldn}\ f\ 1)\ (\mathit{knuth}_3\ a\ (n-1)) \\
&\equiv \mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1
\end{aligned}$$

As the base case for b we have

$$\begin{aligned}
\mathit{knuth}_3\ a\ n\ 0 &\equiv \mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1\ 0 \\
&\equiv 1
\end{aligned}$$

and as the inductive case we have:

$$\begin{aligned}
\mathit{knuth}_3\ a\ n\ b \mid b > 0 &\equiv \mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1\ \ b \\
&\equiv \mathit{knuth}_3\ a\ (n-1)\ (\mathit{foldn}\ (\mathit{knuth}_3\ a\ (n-1))\ 1\ (b-1)) \\
&\equiv \mathit{knuth}_3\ a\ (n-1)\ (\mathit{knuth}_3\ a\ n\ (b-1))
\end{aligned}$$

□

5 Conway's Chained Arrows

Knuth's up-arrow notation was conceived as an iterative generalization of the step from addition to multiplication and from there to exponentiation, remaining in the realm of binary operators. Conway's chained arrow notation releases the latter restriction to encode even huger numbers. It defines a single operator on a list of arbitrarily many numbers, traditionally called "chains" and written with interspersed arrows (hence the name). A typical self-referential definition is the following system of equations, where p, q are numbers and X is a nonempty subsequence:

$$\begin{aligned}
\langle p \rangle &= p \\
\langle p \to q \rangle &= p^q \\
\langle X \to p \to 1 \rangle &= \langle X \to p \rangle \\
\langle X \to 1 \to q \rangle &= \langle X \to 1 \rangle \\
\langle X \to p \to q \rangle &= \big\langle X \to \langle X \to (p-1) \to q \rangle \to (q-1) \big\rangle \qquad (p, q > 1) \\
&\text{with the canonical extension} \\
\langle\, \rangle &= 1
\end{aligned}$$

We have added angled brackets around chains to emphasize the relevance of bracketing: since a chain denotes a number, chains may be nested (as in the last equation); but unlike for many other arithmetic operators, a chain of more than two elements is not *merely* an abbreviation for the nested iteration of a binary operator $\to$. The full situation is complicated: even though generally $\langle o \to p \to q \rangle$ is different from both $\langle \langle o \to p \rangle \to q \rangle$ and $\langle o \to \langle p \to q \rangle \rangle$, this does not imply that the overall operation *cannot* be expressed by a fold operator acting binarily on the list of numbers—that will be exactly our next move.

5.1 Derivation

The above equations, read left-to-right, specify a total evaluation function. It can be translated to a Haskell function on lists of strictly positive numbers

```
conway₀ []                          = 1
conway₀ [p]                         = p
conway₀ [q, p]                      = p ^ q
conway₀ (1 : p : xs)                = conway₀ (p : xs)
conway₀ (q : 1 : xs)                = conway₀ (1 : xs)
conway₀ (q : p : xs) | p > 1 ∧ q > 1 = conway₀ ((q − 1) : p'       : xs)
  where p'                          = conway₀ (q       : (p − 1) : xs)
```

where the order is reversed, because Conway's notation matches from the right, but Haskell lists decompose from the left. In order to apply recursion abstraction successfully to this messy function, its arguments need to be regularized and reordered in a particular way. Firstly, we note that the first two cases, for lists of length less than two, are special, in the sense that they are never reached by recursion; the third case, for lists of length two exactly, is the actual recursive base case.

Secondly, we notice that the remaining cases all involve two or more numbers, and the nested recursion occurs quite inconveniently in second position in the list. Hence we separate a non-recursive front-end function for the first two cases

```
cfront₁ _ []           = 1
cfront₁ _ [p]          = p
cfront₁ b (q : p : xs) = b xs q p
```

and a recursive back-end function, with the first two list elements expanded and reordered, for the other cases

```
cback₁ []       q p        = p ^ q
cback₁ (o : ys) 1 p        = cback₁ ys p      o
cback₁ (o : ys) q 1        = cback₁ ys 1      o
```

$$\begin{array}{llll}
\mathit{cback}_1\;\mathit{xs} & q\;p \mid p > 1 \land q > 1 & = \mathit{cback}_1\;\mathit{xs}\;(q-1) & p' \\
\quad\textbf{where}\;p' & & = \mathit{cback}_1\;\mathit{xs}\;q & (p-1)
\end{array}$$

such that

$$\mathit{conway} \equiv \mathit{cfront}\;\mathit{cback}$$

Note that the reordering of arguments, however unintuitive at first glance, moves the nested recursion p' from the awkward middle to a more manageable final position.

Conway's arrows are traditionally defined starting from one rather than zero. But unlike for Knuth's arrows, there is no simple extension to zero arguments. In order to make the recursion pattern more similar to the previous ones, we consider a variant where all argument numbers are reduced by one in the front-end.[2]

$$\begin{array}{ll}
\mathit{cfront}_2\;\_\;[\,] & = 1 \\
\mathit{cfront}_2\;\_\;[p] & = p \\
\mathit{cfront}_2\;b\;\mathit{xs} \mid \mathit{length}\;\mathit{xs} > 1 & = \mathit{cfront}_1\;b\;(\mathit{map}\;(-\;1)\;\mathit{xs})
\end{array}$$

This gives us a slightly modified back-end:

$$\begin{array}{llll}
\mathit{cback}_2\;[\,] & q\;p & = (p+1)\;\hat{}\;(q+1) & \\
\mathit{cback}_2\;(o:\mathit{ys}) & 0\;p & = \mathit{cback}_2\;\mathit{ys}\;p & o \\
\mathit{cback}_2\;(o:\mathit{ys}) & q\;0 & = \mathit{cback}_2\;\mathit{ys}\;0 & o \\
\mathit{cback}_2\;\mathit{xs} & q\;p \mid p > 0 \land q > 0 & = \mathit{cback}_2\;\mathit{xs}\;(q-1) & (p'-1) \\
\quad\textbf{where}\;p' & & = \mathit{cback}_2\;\mathit{xs}\;q & (p\;\;-1)
\end{array}$$

Note the two corrections (+1) where arguments flow to results, and the single correction (− 1) where a recursive result flows back to an argument.

The first argument is the technical novelty of this exercise, because it is of list type. We tackle it by double currying:

$$\begin{array}{lll}
\mathit{cback}_3\;[\,] & = \mathit{cpow} & \\
\mathit{cback}_3\;\mathit{xs}@(o:\mathit{ys}) & = \mathit{aux} & \\
\quad\textbf{where}\;\mathit{aux}\;0\;p & = \mathit{cback}_3\;\mathit{ys}\;p & o \\
\qquad\quad\;\;\mathit{aux}\;q\;0 & = \mathit{cback}_3\;\mathit{ys}\;0 & o \\
\qquad\quad\;\;\mathit{aux}\;q\;p \mid p > 0 \land q > 0 & = \mathit{cback}_3\;\mathit{xs}\;(q-1) & (p'-1) \\
\qquad\qquad\textbf{where}\;p' & = \mathit{cback}_3\;\mathit{xs}\;q & (p\;\;-1) \\
\mathit{cpow}\;q\;p = (p+1)\;\hat{}\;(q+1) & &
\end{array}$$

Note the auxiliary function *cpow*, and the use of *en passant* pattern matching to unify the different heads of the three recursive cases. The fold operator for lists takes generators with two arguments, a non-recursive one for the head and a recursive one for the tail. We shape the auxiliary function accordingly by beta expansion,

$$\begin{array}{llll}
\mathit{cback}_{3\mathrm{a}}\;[\,] & = \mathit{cpow} & & \\
\mathit{cback}_{3\mathrm{a}}\;\mathit{xs}@(o:\mathit{ys}) & = \mathit{aux}\;o\;(\mathit{cback}_{3\mathrm{a}}\;\mathit{ys}) & & \\
\quad\textbf{where}\;\mathit{aux}\;o\;k\;0\;p & = k & p & o \\
\qquad\quad\;\;\mathit{aux}\;o\;k\;q\;0 & = k & 0 & o \\
\qquad\quad\;\;\mathit{aux}\;o\;k\;q\;p \mid p > 0 \land q > 0 & = \mathit{cback}_3\;\mathit{xs} & (q-1) & (p'-1) \\
\qquad\qquad\textbf{where}\;p' & = \mathit{cback}_3\;\mathit{xs} & q & (p\;\;-1)
\end{array}$$

make the, already familiar, context-sensitive substitution

$$\mathit{cback}\;\mathit{xs}@(o:\mathit{ys}) \equiv \mathit{aux}\;o\;(\mathit{cback}\;\mathit{ys}) \equiv \mathit{aux}\;o\;k$$

and eliminate the now unused variable *xs* to obtain:

[2] The notation (− 1) is actually (*subtract* 1) in Haskell.

```
cback3b []       = cpow
cback3b (o : ys) = aux o (cback3b ys)
  where aux o k  0 p                 = k          p        o
        aux o k  q 0                 = k          0        o
        aux o k  q p | p > 0 ∧ q > 0 = aux o k (q − 1) (p′ − 1)
          where p′                   = aux o k q       (p  − 1)
```

We readily read off the universal property of list folding and deduce:

```
cback4 = foldr aux cpow
  where aux o k 0 p                 = k          p        o
        aux o k q 0                 = k          0        o
        aux o k q p | p > 0 ∧ q > 0 = aux o k (q − 1) (p′ − 1)
          where p′                  = aux o k q       (p  − 1)
```

The remaining steps are business as usual. Eliminate p from *aux* by currying,

```
cback5 = foldr aux cpow
  where
    aux o k 0                = λp → k p o
    aux o k q | q > 0        = aux2
      where aux2 0           = k          0        o
            aux2 p | p > 0   = aux o k (q − 1) (p′ − 1)
              where p′       = aux o k q       (p  − 1)
```

abstract from the “productive” self-reference by beta expansion,

```
cback5a = foldr aux cpow
  where
    aux o k 0                  = λp → k p o
    aux o k q | q > 0          = aux2 (aux o k (q − 1))
      where aux2 f 0           = k          0 o
            aux2 f p | p > 0   = f              (p′ − 1)
              where p′         = aux o k q (p  − 1)
```

and substitute

$$aux\ o\ k\ q \equiv aux2\ (aux\ o\ k\ (q-1)) \equiv aux2\ f$$

to obtain:

```
cback5b = foldr aux cpow
  where
    aux o k      0             = λp → k p o
    aux o k      q | q > 0     = aux2 (aux o k (q − 1))
      where aux2 f 0           = k 0       o
            aux2 f p | p > 0   = f         (p′ − 1)
              where p′         = aux2 f (p  − 1)
```

Perform recursion abstraction in q,

```
cback6 = foldr aux cpow
  where
    aux o k = foldn aux2 (λp → k p o)
      where aux2 f 0           = k 0       o
            aux2 f p | p > 0   = f         (p′ − 1)
              where p′         = aux2 f (p  − 1)
```

beta-expand the auxiliary expression p',

```
cback7 = foldr aux cpow
  where
    aux o k = foldn aux2 (λp → k p o)
      where
        aux2 f 0         = k 0 o
        aux2 f p | p > 0 = (λp' → f (p' − 1)) (aux2 f (p − 1))
```

and perform recursion abstraction in p:

```
cback8 = foldr aux cpow
  where
    aux o k = foldn aux2 (λp → k p o)
      where aux2 f = foldn (λp' → f (p' − 1)) (k 0 o)
```

Final cosmetic translation for improved point-freeness:

```
cback9 = foldr aux cpow
  where
    aux o k = foldn aux2 (flip k o)
      where aux2 f = foldn (f ∘ (− 1)) (k 0 o)
```

5.2 Verification

Verification is a more complex issue here because of the sheer number of cases in indirections. Since no significant new insights are to be gained, we give just one case, corresponding to the fourth equation of $conway_0$, for illustration purposes:

```
cfront2 cback8 (1 : p : x) ≡ cfront1 cback8 (map (− 1) (1 : p : x))
                           ≡ cfront1 cback8 (0 : p − 1 : x')
                               where x' = map (− 1) x
                           ≡ cback8 x' 0 (p − 1)
                           ≡          foldr aux cpow x' 0 (p − 1)
                           ≡ aux o (foldr aux cpow y) 0 (p − 1)
                               where (o : y) = x'
                           ≡ foldn aux2 (λp → foldr aux cpow y p o) 0 (p − 1)
                           ≡            (λp → foldr aux cpow y p o)   (p − 1)
                           ≡ foldr aux cpow y (p − 1) o
                           ≡ cback8 y (p − 1) o
                           ≡ cfront1 cback8 (p − 1 : o : y)
                           ≡ cfront1 cback8 (p − 1 : x')
                           ≡ cfront1 cback8 (map (− 1) (p : x))
                           ≡ cfront2 cback8 (p : x)
```

□

5.3 Analysis

The type analysis of the recursion scheme of Conway's chained arrows is analogous to the preceding examples. The third, outermost layer of folding has yet a more complex underlying datatype.

```
foldr3 :: FoldR Integer (Integer → Integer → Integer)
foldr3 = foldr

cback10 = foldr3 aux cpow
  where
    aux o k = foldn2 aux2 (flip k o)
      where aux2 f = foldn1 (f ∘ (− 1)) (k 0 o)
```

6 Conclusion

Recursion abstraction is a sophisticated program transformation technique. Its straightforward applications are at the heart of the algebra-of-programs approach to functional programming. It is a matter of taste whether definitions in terms of recursion operators are more intuitive and readable than definitions in terms of self-reference. But without doubt, the primitive forms lends themselves more easily to formal, in particular equational, reasoning.

Iterated recursion abstraction is notably more tricky than just a single step. Self-references need to be eliminated by clever context-sensitive beta expansion. The Ackermann function is, as for many other questions, just the right introductory example to teach the technique. In particular, its argument order leads the way naturally. Conway's chained arrow notation, on the other hand, is a significantly more difficult nut to crack, and is rather at the high end of demonstrations for the potential of the technique.

The central trick has now been used four times in three exercises, so the general pattern should have become apparent. In summary, a multiply nested self-referential cycle in a definition is broken by bringing the function into the form

$$\begin{aligned}
&h\ 0 && = e \\
&h\ n \mid n > 0 && = g\ (h\ (n-1)) \\
&\quad \textbf{where}\ g\ f = i\ (h\ n)
\end{aligned}$$

where the auxiliary higher-order function i typically arises "virtually" by beta abstraction from a given expression in which $h\ n$ occurs. Then one proceeds to

$$\begin{aligned}
&h\ 0 && = e \\
&h\ n \mid n > 0 && = g'\ (h\ (n-1)) \\
&\quad \textbf{where}\ g'\ f = i\ (g'\ f)
\end{aligned}$$

Globally, g and g' are quite different functions, but we find for $n > 0$:

$$\begin{aligned}
&g\ \ (h\ (n-1)) \equiv i\ (h\ n) \equiv i\ (g\ \ (h\ (n-1))) \\
&g'\ (h\ (n-1)) \qquad\qquad\ \equiv i\ (g'\ (h\ (n-1)))
\end{aligned}$$

That is, both $g\ f$ and $g'\ f$, where $f = h\ (n-1)$, are fixed points of i. Since we are in a functional programming language with unique fixed point semantics as the very foundation of self-referential definitions, we may substitute one for the other. Then the outer recursion can be reduced to primitive form

$$\begin{aligned}
&h = \mathit{foldn}\ e\ g' \\
&\quad \textbf{where}\ g'\ f = i\ (g'\ f)
\end{aligned}$$

and the inner self-reference of g' can be processed further by recursion abstraction, using either the same or more basic techniques.

It seems plausible that this technique will work for a large class of multiply nested recursive functions. The auxiliary tactics that we have employed, namely moving nested recursion to final argument position and refactoring deep pattern matches, are possibly also more general heuristics, and merit further investigation.

6.1 Outlook

We close by posing several open problems as challenges to the reader:

1. Assess whether the typical, relative ease of inductive reasoning about recursive functions in primitive form carries over to well-known, albeit non-trivial identities concerning our example functions, such as:

   $$\mathit{ack}\ (m+2)\ n \equiv \mathit{knuth}\ 2\ m\ (n+3) - 3$$

2. As an important special case of equational reasoning, demonstrate program simplifications, such as *fusion* rules, for expressions involving our example functions.

3. Use the higher-order primitive recursion framework as a toolkit for synthesizing novel functions with interesting, recursive equational definitions.

Acknowledgments

Anonymous referees of a previous manuscript have made valuable contributions to the evolution of this article.

References

[1] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[2] R. Cockett and T. Fukushima. About Charity. Technical report, University of Calgary, 1992.

[3] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Functional Programming*, 9(4):355–372, 1999.

[4] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[5] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proc. International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[6] J. C. Reynolds. Three approaches to type structure. In *Proc. International Joint Conference on Theory and Practice of Software Development*, volume 185 of *Lecture Notes in Computer Science*. Springer, 1985.

[7] R. M. Robinson. Recursion and double recursion. *Bull. AMS*, 54(10):987–993, 1948.

[8] P. Wadler. Recursive types for free! Online manuscript, 1990.

Athina P. Petropulu. “Higher-Order Spectral Analysis.”
2000 CRC Press LLC. <http://www.engnetbase.com>.

Higher-Order Spectral Analysis

Athina P. Petropulu
Drexel University

76.1 Introduction

The past 20 years witnessed an expansion of power spectrum estimation techniques, which have proved essential in many applications, such as communications, sonar, radar, speech/image processing, geophysics, and biomedical signal processing [13, 11, 7]. In power spectrum estimation the process under consideration is treated as a superposition of statistically uncorrelated harmonic components. The distribution of power among these frequency components is the power spectrum. As such, phase relations between frequency components are suppressed. The information in the power spectrum is essentially present in the autocorrelation sequence, which would suffice for the complete statistical description of a Gaussian process of known mean. However, there are applications where one would need to obtain information regarding deviations from the Gaussianity assumption and presence of nonlinearities. In these cases power spectrum is of little help, and one would have to look beyond the power spectrum or autocorrelation domain. Higher-Order Spectra (HOS) (of order greater than 2), which are defined in terms of higher-order cumulants of the data, do contain such information [16]. The third-order spectrum is commonly referred to as bispectrum, the fourth-order one as trispectrum, and in fact, the power spectrum is also a member of the higher-order spectral class; it is the second-order spectrum.

HOS consist of higher-order moment spectra, which are defined for deterministic signals, and cumulant spectra, which are defined for random processes. In general, there are three motivations behind the use of HOS in signal processing: (1) to suppress Gaussian noise of unknown mean and variance; (2) to reconstruct the phase as well as the magnitude response of signals or systems; and (3) to detect and characterize nonlinearities in the data.

The first motivation stems from the property of Gaussian processes to have zero higher-order spectra. Due to this property, HOS are high signal-to-noise ratio domains, in which one can perform detection, parameter estimation, or even signal reconstruction even if the time domain noise is spatially correlated. The same property of cumulant spectra can provide means of detecting and characterizing deviations of the data from the Gaussian model.

The second motivation is based on the ability of cumulant spectra to preserve the Fourier-phase of signals. In the modeling of time series, second-order statistics (autocorrelation) have been heavily used because they are the result of least-squares optimization criteria. However, an accurate phase reconstruction in the autocorrelation domain can be achieved only if the signal is minimum phase. Nonminimum phase signal reconstruction can be achieved only in the HOS domain, due to the HOS ability to preserve phase. Figure 76.1 shows two signals, a nonminimum phase and a minimum phase, with identical magnitude spectra but different phase spectra. Although power spectrum cannot distinguish between the two signals, the bispectrum that uses phase information can.

Being nonlinear functions of the data, HOS are quite natural tools in the analysis of nonlinear systems operating under a random input. General relations for arbitrary stationary random data passing through an arbitrary linear system exist and have been studied extensively. Such expression, however, are not available for nonlinear systems, where each type of nonlinearity must be studied separately. Higher-order correlations between input and output can detect and characterize certain nonlinearities [34], and for this purpose several higher-order spectra-based methods have been developed.

The organization of this chapter is as follows. First the definitions and properties of cumulants and higher-order spectra are introduced. Then two methods for the estimation of HOS from finite length data are outlined and the asymptotic statistics of the obtained estimates are presented. Following that, parametric and nonparametric methods for HOS-based identification of linear systems are described, and the use of HOS in the identification of some particular nonlinear systems is briefly discussed. The chapter concludes with a section on applications of HOS and available software.

76.2 Definitions and Properties of HOS

In this chapter we will consider random one-dimensional processes only. The definitions can be easily extended to the two-dimensional case [15].

The joint moments of order r of the random variables $x_1, \ldots, x_n$ are given by [22]

$$\begin{aligned} Mom\left[x_1^{k_1}, \ldots, x_n^{k_n}\right] &= E\{x_1^{k_1}, \ldots, x_n^{k_n}\} \\ &= (-j)^r \frac{\partial^r \Phi(\omega_1, \ldots, \omega_n)}{\partial \omega_1^{k_1} \ldots \partial \omega_n^{k_n}}|_{\omega_1 = \cdots = \omega_n = 0} \,, \end{aligned} \tag{76.1}$$

where $k_1 + \cdots + k_n = r$, and $\Phi()$ is their joint characteristic function. The joint cumulants are defined as

$$Cum[x_1^{k_1}, \ldots, x_n^{k_n}] = (-j)^r \frac{\partial^r ln\Phi(\omega_1, \ldots, \omega_n)}{\partial \omega_1^{k_1} \ldots \partial \omega_n^{k_n}}\}|_{\omega_1 = \cdots = \omega_n = 0} \,. \tag{76.2}$$

For a stationary discrete time random process $X(k)$, (k denotes discrete time), the *moments* of order n are given by

$$m_n^x(\tau_1, \tau_2, \ldots, \tau_{n-1}) = E\{X(k)X(k+\tau_1) \cdots X(k+\tau_{n-1})\} \,, \tag{76.3}$$

where $E\{.\}$ denotes expectation. The nth order *cumulants* are functions of the moments of order up to n, i.e.,

1st order cumulants:

$$c_1^x = m_1^x = E\{X(k)\} \quad \text{(mean)} \tag{76.4}$$

2nd order cumulants:

$$c_2^x(\tau_1) = m_2^x(\tau_1) - \left(m_1^x\right)^2 \quad \text{(covariance)} \tag{76.5}$$

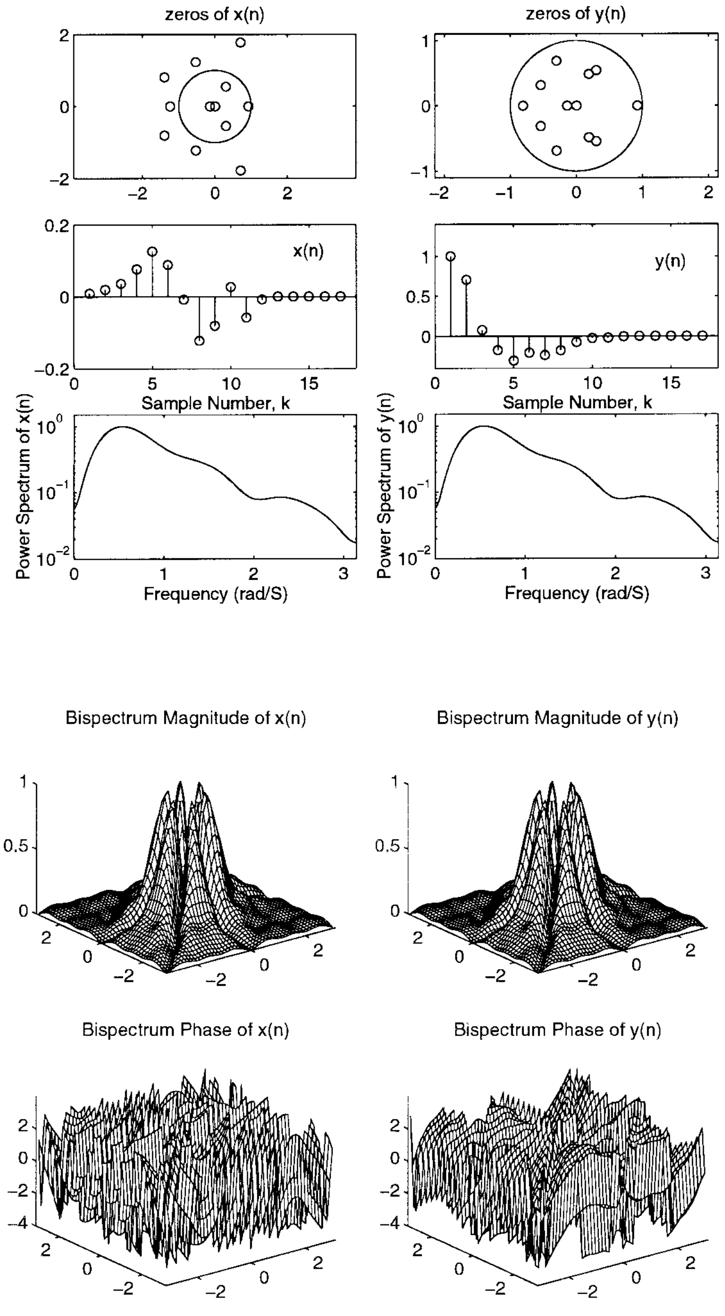


FIGURE 76.1: $x(n)$ is a nonminimum phase signal and $y(n)$ is a minimum phase one. Although their power spectra are identical, their bispectra are different because they contain phase information.

3rd order cumulants:

$$c_3^x\left(\tau_1, \tau_2\right) = m_3^x\left(\tau_1, \tau_2\right) - \left(m_1^x\right)\left[m_2^x\left(\tau_1\right) + m_2^x\left(\tau_2\right) + m_2^x\left(\tau_2 - \tau_1\right)\right] + 2\left(m_1^x\right)^3 \tag{76.6}$$

4th order cumulants:

$$\begin{aligned} c_4^x\left(\tau_1, \tau_2, \tau_3\right) \quad = \quad & m_4^x\left(\tau_1, \tau_2, \tau_3\right) - m_2^x\left(\tau_1\right) m_2^x\left(\tau_3 - \tau_2\right) - m_2^x\left(\tau_2\right) m_2^x\left(\tau_3 - \tau_1\right) \\ & - m_2^x\left(\tau_3\right) m_2^x\left(\tau_2 - \tau_1\right) \\ & - m_1^x\left[m_3^x\left(\tau_2 - \tau_1, \tau_3 - \tau_1\right) + m_3^x\left(\tau_2, \tau_3\right) + m_3^x\left(\tau_2, \tau_4\right) + m_3^x\left(\tau_1, \tau_2\right)\right] \\ & + \left(m_1^x\right)^2\left[m_2^x\left(\tau_1\right) + m_2^x\left(\tau_2\right) + m_2^x\left(\tau_3\right) + m_2^x\left(\tau_3 - \tau_1\right) + m_2^x\left(\tau_3 - \tau_2\right)\right. \\ & \left. + \, m_1^x\left(\tau_2 - \tau_1\right)\right] - 6\left(m_1^x\right)^4 \end{aligned} \tag{76.7}$$

where $m_3^x(\tau_1, \tau_2)$ is the 3rd order moment sequence, and m_1^x is the mean. The general relationship between cumulants and moments can be found in [16].

Some important properties of moments and cumulants are summarized next.

[P1] If $X(k)$ is Gaussian, the $c_n^x(\tau_1, \tau_2, \ldots, \tau_{n-1}) = 0$ for $n > 2$. In other words, all the information about a Gaussian process is contained in its first and second-order cumulants. This property can be used to suppress Gaussian noise, or as a measure for non-Gaussianity in time series.

[P2] If $X(k)$ is symmetrically distributed, then $c_3^x(\tau_1, \tau_2) = 0$. Third-order cumulants suppress not only Gaussian processes, but also all symmetrically distributed processes, such as uniform, Laplace, and Bernoulli-Gaussian.

[P3] For cumulants additivity holds. If $X(k) = S(k) + W(k)$, where $S(k),\ W(k)$ are stationary and statistically independent random processes, then $c_n^x(\tau_1, \tau_2, \ldots, \tau_{n-1}) = c_n^s(\tau_1, \tau_2, \ldots, \tau_{n-1}) + c_n^w(\tau_1, \tau_2, \ldots, \tau_{n-1})$. It is important to note that additivity does not hold for moments.

If $W(k)$ is Gaussian representing noise which corrupts the signal of interest, $S(k)$, then by means of (P2) and (P3), we get that $c_n^x(\tau_1, \tau_2, \ldots, \tau_{n-1}) = c_n^s(\tau_1, \tau_2, \ldots, \tau_{n-1})$, for $n > 2$. In other words, in higher-order cumulant domains the signal of interest propagates noise free. Property (P3) can also provide a measure of statistical dependence of two processes.

[P4] if $X(k)$ has zero mean, then $c_n^x(\tau_1, \ldots, \tau_{n-1}) = m_n^x(\tau_1, \ldots, \tau_{n-1})$, for $n \leq 3$.

Higher-order spectra are defined in terms of either cumulants (e.g., cumulant spectra) or moments (e.g., moment spectra).

Assuming that the nth order cumulant sequence is absolutely summable, the nth order **cumulant spectrum** of $X(k)$, $C_n^x(\omega_1, \omega_2, \ldots, \omega_{n-1})$, exists, and is defined to be the $(n-1)$-dimensional Fourier transform of the nth order cumulant sequence. In general, $C_n^x(\omega_1, \omega_2, \ldots, \omega_{n-1})$ is complex, i.e., it has magnitude and phase. In an analogous manner, **moment spectrum** is the multi-dimensional Fourier transform of the moment sequence.

If $v(k)$ is a stationary non-Gaussian process with zero mean and nth order cumulant sequence

$$c_n^v(\tau_1, \ldots, \tau_{n-1}) = \gamma_n^v \delta(\tau_1, \ldots, \tau_{n-1}) \ , \tag{76.8}$$

where $\delta(.)$ is the delta function, $v(k)$ is said to be nth order white. Its nth order cumulant spectrum is then flat and equal to γ_n^v.

Cumulant spectra are more useful in processing random signals than moment spectra since they posses properties that the moment spectra do not share: (1) the cumulants of the sum of two independent random processes equals the sum of the cumulants of the process; (2) cumulant spectra of order > 2 are zero if the underlying process in Gaussian; (3) cumulants quantify the degree of statistical dependence of time series; and (4) cumulants of higher-order white noise are multidimensional impulses, and the corresponding cumulant spectra are flat.

76.3 HOS Computation from Real Data

The definitions of cumulants presented in the previous section are based on expectation operations, and they assume infinite length data. In practice we always deal with data of finite length; therefore, the cumulants can only be approximated. Two methods for cumulants and spectra estimation are presented next for the third-order case.

Indirect Method :

Let $X(k)$, $k = 1, \ldots, N$ be the available data.

1. Segment the data into K records of M samples each. Let $X^i(k)$, $k = 1, \ldots, M$, represent the ith record.
2. Subtract the mean of each record.
3. Estimate the moments of each segments $X^i(k)$ as follows:

$$m_3^{x_i}(\tau_1, \tau_2) = \frac{1}{M} \sum_{l=l_1}^{l_2} X^i(l) X^i(l+\tau_1) X^i(l+\tau_2) \ ,$$

$$l_1 = max(0, -\tau_1, -\tau_2),\ l_2 = min(M-1, M-2), \\ |\tau_1| < L,\ |\tau_2| < L,\ i = 1, 2, \ldots, K \ . \tag{76.9}$$

   Since each segment has zero mean, its third-order moments and cumulants are identical, i.e., $c_3^{x_i}(\tau_1, \tau_2) = m_3^{x_i}(\tau_1, \tau_2)$.

4. Compute the average cumulants as:

$$\hat{c}_3^x(\tau_1, \tau_2) = \frac{1}{K} \sum_{i=1}^{K} m_3^{x_i}(\tau_1, \tau_2) \tag{76.10}$$

5. Obtain the third-order spectrum (bispectrum) estimate as

$$\hat{C}_3^x(\omega_1, \omega_2) = \sum_{\tau_1=-L}^{L} \sum_{\tau_2=-L}^{L} \hat{c}_3^x(\tau_1, \tau_2) e^{-j(\omega_1\tau_1 + \omega_2\tau_2)} w(\tau_1, \tau_2) \ , \tag{76.11}$$

   where $L < M - 1$, and $w(\tau_1, \tau_2)$ is a two-dimensional window of bounded support, introduced to smooth out edge effects. The bandwidth of the final bispectrum estimate is $\Delta = 1/L$.

A complete description of appropriate windows that can be used in (76.11) and their properties can be found in [16]. A good choice of cumulant window is:

$$w(\tau_1, \tau_2) = d(\tau_1) d(\tau_2) d(\tau_1 - \tau_2) \ , \tag{76.12}$$

where

$$d(\tau) = \begin{cases} \frac{1}{\pi} |\sin \frac{\pi\tau}{L}| + (1 - \frac{|\tau|}{L}) \cos \frac{\pi\tau}{L} & |\tau| \le L \\ 0 & |\tau| > L \end{cases} \tag{76.13}$$

which is known as the minimum bispectrum bias supremum [17].

Direct Method

Let $X(k)$, $k = 1, \ldots, N$ be the available data.

1. Segment the data into K records of M samples each. Let $X^i(k),\ k = 1, \ldots, M$, represent the ith record.
2. Subtract the mean of each record.
3. Compute the Discrete Fourier Transform $F_x^i(k)$ of each segment, based on M points, i.e.,

$$F_x^i(k) = \sum_{n=0}^{M-1} X^i(n) e^{-j\frac{2\pi}{M}nk},\ \ k = 0, 1, \ldots, M-1,\ i = 1, 2, \ldots, K\ . \tag{76.14}$$

4. The third-order spectrum of each segment is obtained as

$$C_3^{x_i}(k_1, k_2) = \frac{1}{M} F_x^i(k_1) F_x^i(k_2) F_x^{i^*}(k_1 + k_2),\ i = 1, \ldots, K\ . \tag{76.15}$$

   Due to the bispectrum symmetry properties, $C_3^{x_i}(k_1, k_2)$ need to be computed only in the triangular region $0 \leq k_2 \leq k_1,\ k_1 + k_2 < M/2$.
5. In order to reduce the variance of the estimate additional smoothing over a rectangular window of size $(M_3 \times M_3)$ can be performed around each frequency, assuming that the third-order spectrum is smooth enough, i.e.,

$$\tilde{C}_3^{x_i}(k_1, k_2) = \frac{1}{M_3^2} \sum_{n_1=-M_3/2}^{M_3/2-1} \sum_{n_2=-M_3/2}^{M_3/2-1} C_3^{x_i}(k_1 + n_1, k_2 + n_2)\ . \tag{76.16}$$

6. Finally, the third-order spectrum is given as the average over all third-order spectra, i.e.,

$$\hat{C}_3^x(\omega_1, \omega_2) = \frac{1}{K} \sum_{i=1}^{K} \tilde{C}_3^{x_i}(\omega_1, \omega_2)\ ,\ \omega_i = \frac{2\pi}{M} k_i,\ \ i = 1, 2\ . \tag{76.17}$$

   The final bandwidth of this bispectrum estimate is $\Delta = M_3/M$, which is the spacing between frequency samples in the bispectrum domain.

For large N, and as long as

$$\Delta \to 0,\ and\ \Delta^2 N \to \infty \tag{76.18}$$

[32], both the direct and the indirect methods produce asymptotically unbiased and consistent bispectrum estimates, with real and imaginary part variances:

$$\text{var}\left(\text{Re}\left[\hat{C}_3^x(\omega_1, \omega_2)\right]\right) = \text{var}\left(\text{Im}\left[\hat{C}_3^x(\omega_1, \omega_2)\right]\right) \tag{76.19}$$

$$= \frac{1}{\Delta^2 N} C_2^x(\omega_1) C_2^x(\omega_2) C_2^x(\omega_1 + \omega_2) = \begin{cases} \frac{VL^2}{MK} C_2^x(\omega_1) C_2^x(\omega_2) C_2^x(\omega_1 + \omega_2) & \text{indirect} \\ \frac{M}{KM_3^2} C_2^x(\omega_1) C_2^x(\omega_2) C_2^x(\omega_1 + \omega_2) & \text{direct}\ , \end{cases}$$

where V is the energy of the bispectrum window.

From the above expressions, it becomes apparent that the bispectrum estimate variance can be reduced by increasing the number of records, or reducing the size of the region of support of the window in the cumulant domain (L), or increasing the size of the frequency smoothing window (M_3), etc. The relation between the parameters $M,\ K,\ L,\ M_3$ should be such that (76.18) is satisfied.

76.4 Linear Processes

Let $x(k)$ be generated by exciting a linear time-invariant (LTI) system with frequency response $H(\omega)$ with a non-Gaussian process $v(k)$. Its nth order spectrum can be written as

$$C_n^x(\omega_1, \omega_2, \ldots, \omega_{n-1}) = C_n^v(\omega_1, \omega_2, \ldots, \omega_{n-1}) H(\omega_1) \cdots H(\omega_{n-1}) H^*(\omega_1 + \cdots + \omega_{n-1}) \ . \tag{76.20}$$

If $v(k)$ is nth order white then (76.20) becomes

$$C_n^x(\omega_1, \omega_2, \ldots, \omega_{n-1}) = \gamma_n^v H(\omega_1) \cdots H(\omega_{n-1}) H^*(\omega_1 + \cdots + \omega_{n-1}) \ , \tag{76.21}$$

where γ_n^v is a scalar constant and equals the nth order spectrum of $v(k)$. For a linear non-Gaussian random process $X(k)$, the nth order spectrum can be factorized as in (76.21) for every order n, while for a nonlinear process such a factorization might be valid for some orders only (it is always valid for $n = 2$).

If we express $H(\omega) = |H(\omega)| \exp\{j\phi_h(\omega)\}$, then (76.21) can be written as

$$\left|C_n^x(\omega_1, \omega_2, \ldots, \omega_{n-1})\right| = \gamma_n^v |H(\omega_1)| \cdots |H(\omega_{n-1})| \left|H^*(\omega_1 + \cdots + \omega_{n-1})\right| \ , \tag{76.22}$$

and

$$\left|\psi_n^x(\omega_1, \omega_2, \ldots, \omega_{n-1})\right| = \phi_h(\omega_1) + \cdots + \phi_h(\omega_{n-1}) - \phi_h(\omega_1 + \cdots + \omega_{n-1}) \ , \tag{76.23}$$

where $\psi_n^x()$ is the phase of the nth order spectrum.

It can be shown easily that the cumulant spectra of successive orders are related as follows:

$$C_n^x(\omega_1, \omega_2, \ldots, 0) = C_{n-1}^x(\omega_1, \omega_2, \ldots, \omega_{n-2}) H(0) \frac{\gamma_n^v}{\gamma_{n-1}^v} \ . \tag{76.24}$$

As a result, the power spectrum of a Gaussian linear process can be reconstructed from the bispectrum up to a constant term, i.e.,

$$C_3^x(\omega, 0) = C_2^x(\omega) \frac{\gamma_3^v}{\gamma_2^v} \ . \tag{76.25}$$

To reconstruct the phase $\phi_h(\omega)$ from the bispectral phase $\psi_3^x(\omega_1, \omega_2)$ several algorithms have been suggested. A description of different phase estimation methods can be found in [14] and also in [16].

76.4.1 Nonparametric Methods

Consider $x(k)$ generated as shown in Fig. 76.2. The system transfer function can be written as

$$H(z) = cz^{-r} I\left(z^{-1}\right) O(z) = cz^{-r} \frac{\Pi_i (1 - a_i z^{-1})}{\Pi_i (1 - b_i z^{-1})} \Pi_i (1 - c_i z), \ |a_i|, |b_i|, |c_i| < 1 \ , \tag{76.26}$$

where $I(z^{-1})$ and $O(z)$ are the minimum and maximum phase parts of $H(z)$, respectively; c is a constant; and r is an integer. The output nth order cumulant equals [2]

$$\begin{aligned} c_n^x(\tau_1, \ldots, \tau_{n-1}) &= c_n^y(\tau_1, \ldots, \tau_{n-1}) + c_n^w(\tau_1, \ldots, \tau_{n-1}) \\ &= c_n^y(\tau_1, \ldots, \tau_{n-1}) && (76.27) \\ &= \gamma_n^v \sum_{k=0}^{\infty} h(k) h(k + \tau_1) \cdots h(k + \tau_{n-1}) \ , \ n \geq 3 && (76.28) \end{aligned}$$

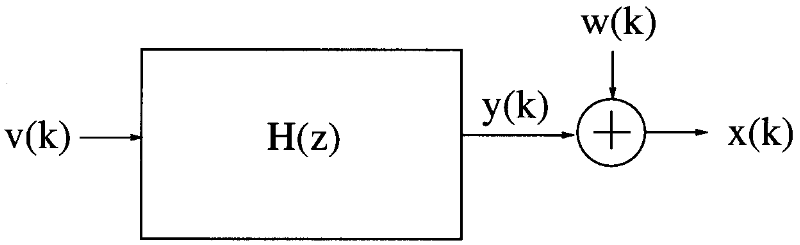


FIGURE 76.2: Single channel model.

where the noise contribution in (76.27) was zero due to the Gaussianity assumption. The Z-domain equivalent of (76.28) for $n = 3$ is

$$C_3^x\left(z_1, z_2\right) = \gamma_3^v H\left(z_1\right) H\left(z_2\right) H\left(z_1^{-1} z_2^{-1}\right) . \tag{76.29}$$

Taking the logarithm of $C_3^x(z_1, z_2)$ followed by an inverse 2-D Z-transform we obtain the output bicepstrum $b_x(m, n)$. The bicepstrum of linear processes is nonzero only along the axes ($m = 0,\ n = 0$) and the diagonal $m = n$ [21]. Along these lines the bicepstrum is equal to the complex cepstrum, i.e.,

$$b_x(m, n) = \begin{cases} \hat{h}(m) & m \neq 0,\ n = 0 \\ \hat{h}(n) & n \neq 0,\ m = 0 \\ \hat{h}(-n) & m = n,\ m \neq 0 \\ \ln(c\gamma_n^v) & m = n = 0, \\ 0 & \text{elsewhere} \end{cases} \tag{76.30}$$

where $\hat{h}(n)$ denotes complex cepstrum [20]. From (76.30), the system impulse response $h(k)$ can be reconstructed from $b_x(m, 0)$ (or $b_x(0, m)$, or $b_x(m, m)$), within a constant and a time delay, via inverse cepstrum operations. The minimum and maximum phase parts of $H(z)$ can be reconstructed by applying inverse cepstrum operations on $b_x(m, 0)u(m)$ and $b_x(m, 0)u(-m)$, respectively, where $u(m)$ is the unit step function.

To avoid phase unwrapping with the logarithm of the bispectrum which is complex, the bicepstrum can be estimated using the group delay approach:

$$b_x(m, n) = \frac{1}{m} F^{-1}\{\frac{F\left[\tau_1 c_3^x(\tau_1, \tau_2)\right]}{C_3^x(\omega_1, \omega_2)}\},\ m \neq 0 \tag{76.31}$$

with $b_x(0, n) = b_x(n, 0)$, and $F\{\cdot\}$ and $F^{-1}\{\cdot\}$ denoting 2-D Fourier transform operator and its inverse, respectively.

The cepstrum of the system can also be computed directly from the cumulants of the system output based on the equation [21]:

$$\sum_{k=1}^{\infty} k\hat{h}(k)\left[c_3^x(m-k, n) - c_3^x(m+k, n+k)\right] + k\hat{h}(-k)\left[c_3^x(m-k, n-k) - c_3^x(m+k, n)\right] = mc_3^x(m, n) \tag{76.32}$$

If $H(z)$ has no zeros on the unit circle its cepstrum decays exponentially, thus (76.32) can be truncated to yield an approximate equation. An overdetermined system of truncated equations can be formed for different values of m and n, which can be solved for $\hat{h}(k),\ k = \ldots, -1, 1, \ldots$. The system response $h(k)$ then can be recovered from its cepstrum via inverse cepstrum operations.

The bicepstrum approach for system reconstruction described above led to estimates with smaller bias and variance than other parametric approaches at the expense of higher computational complexity [21]. The analytic performance evaluation of the bicepstrum approach can be found in [25].

The inverse Z-transform of the logarithm of the trispectrum (fourth-order spectrum), or otherwise tricepstrum, $t_x(m, n, l)$, of linear processes is also zero everywhere except along the axes and the diagonal $m = n = l$. Along these lines it equals the complex cepstrum, thus $h(k)$ can be recovered from slices of the tricepstrum based on inverse cepstrum operations.

For the case of nonlinear processes, the bicepstrum will be nonzero everywhere [4]. The distinctly different structure of the bicepstrum corresponding to linear and nonlinear processes has led to tests of linearity [4].

A new nonparametric method has been recently proposed in [1, 26] in which the cepstrum $\hat{h}(k)$ is obtained as:

$$\hat{h}(-k) = \frac{\hat{p}_n^x\left(k;\ e^{j\beta_1}\right) - \hat{p}_n^x\left(k; e^{j\beta_2}\right)}{e^{j(n-2)\beta_1 k} - e^{j(n-2)\beta_2 k}},\ k \neq 0,\ n > 2 \tag{76.33}$$

where $p_n^x\left(k;\ e^{jb_i}\right)$ is the time domain equivalent of the nth order spectrum slice defined as:

$$P_n^x\left(z;\ e^{j\beta_i}\right) = C_n^x\left(z,\ e^{j\beta_i}, \cdots, e^{j\beta_i}\right) . \tag{76.34}$$

The denominator of (76.33) is nonzero if

$$|\beta_1 - \beta_2| \neq \frac{2\pi l}{k(n-2)},\ \text{for every integer } k \text{ and } \ l\ . \tag{76.35}$$

This method reconstructs a complex system using two slices of the nth order spectrum. The slices, defined as shown above, can be selected arbitrarily as long as their distance satisfy (76.35). If the system is real, one slices is sufficient for the reconstruction. It should be noted that the cepstra appearing in (76.33) require phase unwrapping. The main advantage of this method is that the freedom to choose the higher-order spectra areas to be used in the reconstruction allows one to avoid regions dominated by noise or finite data length effects. Also, corresponding to different slice pairs various independent representations of the system can be reconstructed. Averaging out these representations can reduce estimation errors [26].

Along the lines of system reconstruction from selected HOS slices, another method has been proposed in [28, 29] where the log $H(k)$ is obtained as a solution to a linear system of equations. Although logarithimc operation is involved, no phase unwrapping is required and the principal argument can be used instead of real phase. It was also shown that, as long as the grid size and the distance between the slices are coprime, reconstruction is always possible.

76.4.2 Parametric Methods

One of the popular approaches in system identification has been the construction of a white noise driven, linear time invariant model from a given process realization.

Consider the real autoregressive moving average (ARMA) stable process $y(k)$ given by:

$$\sum_{i=0}^{p} a(i)y(k-i) = \sum_{j=0}^{q} b(j)v(k-j) \tag{76.36}$$

$$x(k) = y(k) + w(k) \tag{76.37}$$

where $a(i),\ b(j)$ represent the AR and MA parameters of the system, $v(k)$ is an independent identically distributed random process, and $w(k)$ represents zero-mean Gaussian noise.

Equations analogous to the Yule-Walker equations can be derived based on third-order cumulants of $x(k)$, i.e.,

$$\sum_{i=0}^{p} a(i) c_3^x(\tau - i, j) = 0, \;\; \tau > q \; , \tag{76.38}$$

or

$$\sum_{i=1}^{p} a(i) c_3^x(\tau - i, j) = -c_3^x(\tau, j), \;\; \tau > q \; , \tag{76.39}$$

where it was assumed $a(0) = 1$. Concatenating (76.39) for $\tau = q+1, \ldots, q+M, \;\; M \geq 0$ and $j = q-p, \ldots, q$, the matrix equation

$$\underline{C}\underline{a} = \underline{c} \tag{76.40}$$

can be formed, where $\underline{C}$ and $\underline{c}$ are a matrix and a vector, respectively, formed by third-order cumulants of the process according to (76.39), and the vector $\underline{a}$ contains the AR parameters. If the AR order p is unknown and (76.40) is formed based on an overestimate of p, the resulting matrix $\underline{C}$ always has rank p. In this case, the AR parameters can be obtained using a low-rank approximation of $\underline{C}$ [5].

Using the estimated AR parameters, $\hat{a}(i), \;\; i = 1, \ldots, p$, a pth order filter with transfer function $\hat{A}(z) = 1 + \sum_{i=1}^{p} \hat{a}(i) z^{-1}$ can be constructed. Based on the filtered through $\hat{A}(z)$ process $x(k)$, i.e., $\tilde{x}(k)$, or otherwise known as the residual time series [5], the MA parameters can be estimated via any MA method [15], for example:

$$b(k) = \frac{c_3^{\tilde{x}}(q, k)}{c_3^{\tilde{x}}(q, 0)}, \;\; k = 0, 1, \ldots, q \tag{76.41}$$

known as the $c(q, k)$ formula [6].

Practical problems associated with the described approach are sensitivity to model order mismatch, and AR estimation errors that propagate in the estimation of the MA parameters. A significant amount of research has been devoted to the ARMA parameter estimation problem. A thorough review of existing ARMA system identification methods can be found in [15, 16]; a more recent method can be found in [24].

76.5 Nonlinear Processes

Despite the fact that progress has been established in developing the theoretical properties of nonlinear models, only a few statistical methods exist for detection and characterization of nonlinearities from a finite set of observations. In this section, we will consider nonlinear Volterra systems excited by Gaussian stationary inputs. Let $y(k)$ be the response of a discrete time invariant pth order Volterra filter whose input is $x(k)$. Then,

$$y(k) = h_0 + \sum_{i} \sum_{\tau_1, \ldots, \tau_i} h_i\left(\tau_1, \ldots, \tau_i\right) x\left(k - \tau_1\right) \cdots x\left(k - \tau_i\right) \; , \tag{76.42}$$

where $h_i(\tau_1, \ldots, \tau_i)$ are the Volterra kernels of the system, which are symmetric functions of their arguments; for causal systems $h_i(\tau_1, \ldots, \tau_i) = 0$ for any $\tau_i < 0$.

The output of a second-order Volterra system when the input is zero-mean stationary is

$$y(k) = h_0 + \sum_{\tau_1} h_1(\tau_1) x(k - \tau_1) + \sum_{\tau_1} \sum_{\tau_2} h_2\left(\tau_1, \tau_2\right) x\left(k - \tau_1\right) x\left(k - \tau_2\right) \; . \tag{76.43}$$

Equation (76.43) can be viewed as a parallel connection of a linear system $h_1(\tau_1)$ and a quadratic system $h_2(\tau_1, \tau_2)$ as illustrated in Fig. 76.3. Let

$$c_2^{xy}(\tau) = E\left\{x(k+\tau)\left[y(k) - m_1^y\right]\right\} \tag{76.44}$$

be the cross-covariance of input and output, and

$$c_3^{xxy}(\tau_1, \tau_2) = E\left\{x\left(k+\tau_1\right) x\left(k+\tau_2\right)\left[y(k) - m_1^y\right]\right\} \tag{76.45}$$

be the third-order cross-cumulant sequence of input and output.

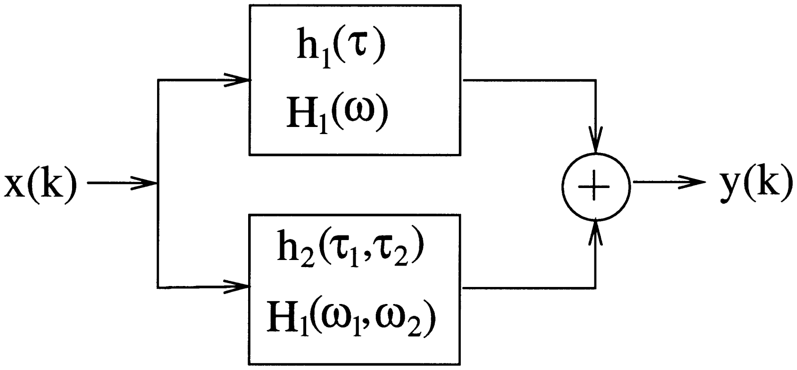


FIGURE 76.3: Second-order Volterra system. Linear and quadratic parts are connected in parallel.

It can be shown that the system's linear part can be identified by

$$H_1(-\omega) = \frac{C_2^{xy}(\omega)}{C_2^{x}(\omega)} , \tag{76.46}$$

and the quadratic part by

$$H_2\left(-\omega_1, -\omega_2\right) = \frac{C_3^{xxy}\left(\omega_1, \omega_2\right)}{2C_2^x\left(\omega_1\right) C_2^x\left(\omega_2\right)} , \tag{76.47}$$

where $C_2^{xy}(\omega)$ and $C_3^{xxy}(\omega_1, \omega_2)$ are the Fourier transforms of $c_2^{xy}(\tau)$ and $c_3^{xxy}(\tau_1, \tau_2)$, respectively. It should be noted that the above equations are valid only for Gaussian input signals. More general results assuming non-Gaussian input have been obtained in [9, 27]. Additional results on particular nonlinear systems have been reported in [3, 33].

An interesting phenomenon caused by a second-order nonlinearity is the quadratic phase coupling. There are situations where nonlinear interaction between two harmonic components of a process contribute to the power of the sum and/or difference frequencies. The signal

$$x(k) = A \cos\left(\lambda_1 k + \theta_1\right) + B \cos\left(\lambda_2 k + \theta_2\right) \tag{76.48}$$

after passing through the quadratic system:

$$z(k) = x(k) + \epsilon x^2(k), \quad \epsilon \neq 0 . \tag{76.49}$$

contains cosinusoidal terms in (λ_1, θ_1), (λ_2, θ_2), $(2\lambda_1, 2\theta_1)$, $(2\lambda_2, 2\theta_2)$, $(\lambda_1 + \lambda_2, \theta_1 + \theta_2)$, $(\lambda_1 - \lambda_2, \theta_1 - \theta_2)$. Such a phenomenon that results in phase relations that are the same as the frequency relations is called quadratic phase coupling [12]. Quadratic phase coupling can arise only among harmonically related components. Three frequencies are harmonically related when one of them is the sum or difference of the other two. Sometimes it is important to find out if peaks at harmonically related positions in the power spectrum are in fact phase coupled. Due to phase suppression, the power spectrum is unable to provide an answer to this problem.

As an example, consider the process [30]

$$X(k) = \sum_{i=1}^{6} \cos\left(\lambda_i k + \phi_i\right) \tag{76.50}$$

where $\lambda_1 > \lambda_2 > 0,\ \lambda_4 + \lambda_5 > 0,\ \lambda_3 = \lambda_1 + \lambda_2,\ \lambda_6 = \lambda_4 + \lambda_5,\ \phi_1, \ldots, \phi_5$ are all independent, uniformly distributed random variables over $(0, 2\pi)$, and $\phi_6 = \phi_4 + \phi_5$. Among the six frequencies, $(\lambda_1, \lambda_2, \lambda_3)$ and $(\lambda_4, \lambda_5, \lambda_6)$ are harmonically related, however, only λ_6 is the result of phase coupling between λ_4 and λ_5. The power spectrum of this process consists of six impulses at $\lambda_i, i = 1, \ldots, 6$ (see Fig. 76.4), offering no indication whether each frequency component is independent or result of frequency coupling. On the other hand, the bispectrum of $X(k)$, $C_3^x(\omega_1, \omega_2)$ (evaluate in its principal region) is zero everywhere, except at point (λ_4, λ_5) of the (ω_1, ω_2) plane, where it exhibits an impulse (Fig. 76.4(b)). The peak indicates that only λ_4, λ_5 are phase coupled.

The bicoherence index, defined as

$$P_3^x(\omega_1, \omega_2) = \frac{C_3^x(\omega_1, \omega_2)}{\sqrt{C_2^x(\omega_1)\, C_2^x(\omega_2)\, C_2^x(\omega_1 + \omega_2)}}\ , \tag{76.51}$$

has been extensively used in practical situations for the detection and quantification of quadratic phase coupling [12]. The value of the bicoherence index at each frequency pair indicates the degree of coupling among the frequencies of that pair. Almost all bispectral estimators can be used in (76.51). However, estimates obtained based on parametric modeling of the bispectrum have been shown to yield superior resolution [30, 31] than the ones obtained with conventional methods.

76.6 Applications/Software Available

Applications of HOS span a wide range of areas [19] such as oceanography (description of wave phenomena), earth sciences (atmospheric pressure, turbulence), crystallography, plasma physics (wave interaction, nonlinear phenomena), mechanical systems (vibration analysis, knock detection), economic time series, biomedical signal analysis (ultrasonic imaging, detection of wave coupling) image processing (texture modeling and characterization, reconstruction, inverse filtering), speech processing (pitch detection, voiced/unvoiced decision), communications (equalization, interference cancellation), array processing (direction of arrival estimation, estimation of number of sources, beamforming, source signal estimation, source classification), harmonic retrieval (frequency estimation), and time delay estimation. Over 500 references can be found in [37]. Additional references can be found in [16, 19, 23].

A software package for signal processing with HOS is the Hi-Spec toolbox, product of Mathworks, Inc. The functions included in Hi-Spec together with a short description are included in Table 76.1.

Acknowledgments

Most of the material presented in this chapter is based on the book *Higher-Order Spectra Analysis: A Non-Linear Signal Processing Framework* [16]. The author wishes to thank Dr. C.L. Nikias for his

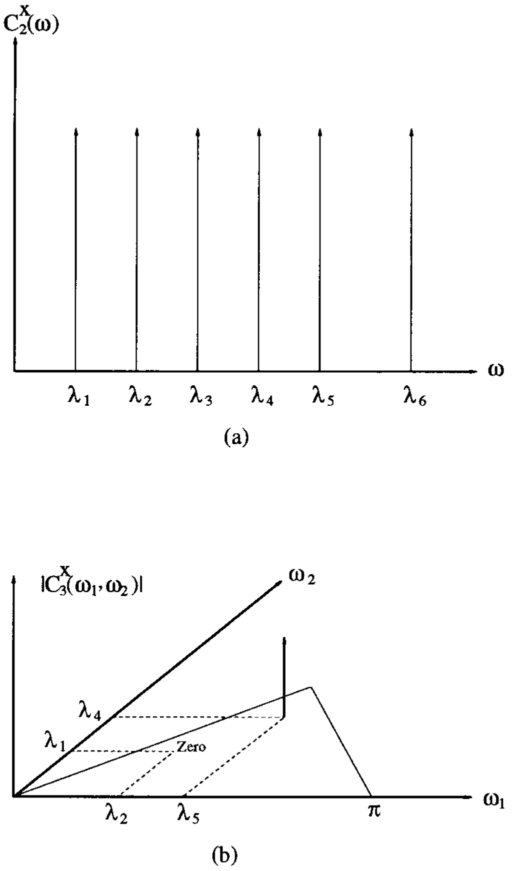


FIGURE 76.4: Quadratic phase coupling. (a) The power spectrum of the process described in Eq. (76.50) cannot determine what frequencies are coupled. (b) The corresponding magnitude bispectrum is zero everywhere in the principle region, except at points corresponding to phase coupled frequencies.

valuable input on the organization of the material. She also thanks U.R. Abeyratne for producing the figures in this chapter. Support for this work came from NSF under grant MIP-9553227 and the Whitaker Foundation.

TABLE 76.1 Functions Included in the Hi-Spec Package

Function name	Description
AR_RCEST	AR parameter estimation based on cumulants
ARMA_QS	ARMA parameter estimation via the Q-slice algorithm
ARMA_RTS	ARMA parameter estimation via the residual time series method
ARMA_SYN	Generates ARMA synthetics
BICEPS	System identification via the bicepstrum approach
BISPEC_D	Bispectrum estimation via the direct method
BISPEC_I	Bispectrum estimation via the indirect method
CUM_EST	Estimates 2nd, 3rd, or 4th order cumulants
CUM_TRUE	Computes the theoretical cumulants of an ARMA model
DOA	Direction-of-arrival estimation
DOA_GEN	Generates synthetics for direction-of-arrival estimation
GL_STAT	Detection statistics for Hinich's Gaussianity and linearity tests
HARM_EST	Estimates frequencies of harmonics in colored noise
HARM_GEN	Generates synthetics for the harmonic retrieval problem
MA_EST	MA parameters estimation
MATUL	System identification via the Matsuoka-Ulrych algorithm
QPC_GEN	Simulation generator for quadratic phase coupling
QPC_TOR	Detects quadratic phase coupling via parametric modeling of bispectrum
RP_IID	Generates samples of an i.i.d. random process
TDE	Estimates time delay between two signals using the parametric cross-cumulant method
TDE_GEN	Synthetics for time delay estimation

References

[1] Abeyratne, U.R. and Petropulu, A.P., α-Weighted cumulant projections: a novel tool for system identification, *29th Annual Asilomar Conference on Signals, Systems and Computers*, California, Oct. 1995.

[2] Brillinger, D.R. and Rosenblatt, M., Computation and interpretation of kth-order spectra, *Spectral Analysis of Time Series*, B. Harris, Ed., John Wiley & Sons, New York, 1967, 189–232.

[3] Brillinger, D.R., The identification of a particular nonlinear time series system, *Biometrika*, 64(3), 509–515, 1977.

[4] Erdem, A.T. and Tekalp, A.M., Linear bispectrum of signals and identification of nonminimum phase FIR systems driven by colored input, *IEEE Trans. on Signal Processing*, 40, 1469–1479, June 1992.

[5] Giannakis, G.B. and Mendel, J.M., Cumulant-based order determination of non-Gaussian ARMA models, *IEEE Trans. on Acoustics, Speech and Signal Processing*, 38, 1411–1423, 1990.

[6] Giannakis, G.B., Cumulants: a powerful tool in signal processing, *Proc. IEEE*, 75, 1987.

[7] Haykin, S., *Nonlinear Methods of Spectral Analysis*, 2nd ed., Berlin, Germany, Springer-Verlag, 1983.

[8] Hinich, M.J., Testing for gaussianity and linearity of a stationary time series, *J. Time Series Analysis*, 3(3), 169–176, 1982.

[9] Hinich, M.J., Identification of the coefficients in a nonlinear time series of the quadratic type, *J. Economics*, 30, 269–288, 1985.

[10] Huber, P.J., Kleiner, B. et.al., Statistical methods for investigating phase relations in stochastic processes, *IEEE Trans. on Audio and Electroacoustics*, Au-19(1), 78–86, 1976.

[11] Kay, S.M., *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[12] Kim, Y.C. and Powers, E.J., Digital bispectral analysis of self-excited fluctuation spectral, *Phys. Fluids*, 21(8), 1452–1453, Aug. 1978.

[13] Marple, Jr., S.L., *Digital Spectral Analysis with Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[14] Matsuoka, T. and Ulrych, T.J., Phase estimation using bispectrum, *Proc. of IEEE*, 72, 1403–1411, Oct., 1984.

[15] Mendel, J.M., Tutorial on higher-order statistics (spectra) in signal processing and system theory: Theoretical results and some applications, *IEEE Proc.*, 79, 278–305, March 1991.

[16] Nikias, C.L. and Petropulu, A.P., *Higher-Order Spectra Analysis: a Nonlinear Signal Processing Framework*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[17] Nikias, C.L. and Raghuveer, M.R., Bispectrum estimation: a digital signal processing framework, *Proc. IEEE*, 75(7), 869–891, July 1987.

[18] Nikias, C.L. and Chiang, H.-H., Higher-order spectrum estimation via noncausal autoregressive modeling and deconvolution, *IEEE Trans. Acoustics, Speech and Signal Processing*, 36(12), 1911–1913, Dec. 1988.

[19] Nikias, C.L. and Mendel, J.M., Signal processing with higher-order spectra, *IEEE Signal Processing Magazine*, 10–37, July 1993.

[20] Oppenheim, A.V. and Schafer, R.W., Discrete-Time Signal Processing, Prentice-Hall, Englewood Cliffs, NJ., 1989.

[21] Pan, R. and Nikias, C.L., The complex cepstrum of higher order cumulants and nonminimum phase system identification, *IEEE Trans. on Acoust., Speech and Signal Processing*, 36(2), 186–205, Feb. 1988.

[22] Papoulis, A., Probability random variables and stochastic processes, McGraw-Hill, New York, 1984.

[23] Petropulu, A.P., Higher-order spectra in biomedical signal processing, *CRC Press Biomedical Engineering Handbook*, CRC Press, Boca Raton, FL, 1995.

[24] Petropulu, A.P., Noncausal nonminimum phase ARMA modeling of non-Gaussian processes, *IEEE Trans. on Signal Processing*, 43(8), 1946–1954, Aug. 1995.

[25] Petropulu, A.P and Nikias, C.L., The complex cepstrum and bicepstrum: analytic performance evaluation in the presence of Gaussian noise, *IEEE Transactions Acoustics, Speech and Signal Processing, special mini-section on Higher-Order Spectral Analysis*, ASSP-38(7), July 1990.

[26] Petropulu, A.P. and Abeyratne U.R., Signal reconstruction for higher-order spectra slices, *IEEE Trans. on Signal Processing*, Sept. 1997.

[27] Powers, E.J., Ritz, C.K. et.al., Applications of digital polyspectral analysis to nonlinear systems modeling and nonlinear wave phenomena, *Workshop on Higher-Order Spectral Analysis*, Vail, CO, 73–77, June 1989.

[28] Pozidis, H. and Petropulu, A.P., System reconstruction from selected bispectrum slices, *IEEE Signal Processing Workshop on Higher-Order Statistics*, Banff, Alberta, Canada, June 1997.

[29] Pozidis, H. and Petropulu, A.P., System reconstruction using selected regions of the discretized HOS, *IEEE Transactions on Signal Processing*, submitted in 1997.

[30] Raghuveer, M.R. and Nikias, C.L., Bispectrum estimation: A parametric approach, *IEEE Trans. on Acoust., Speech and Signal Processing*, ASSP 33(5), 1213–1230, Oct. 1985.

[31] Raghuveer, M.R. and Nikias, C.L., Bispectrum estimation via AR modeling, *Signal Processing*, 10, 35–48, 1986.

[32] Rao, T. Subba and Gabr, M.M., An introduction to bispectral analysis and bilinear time series models, *Lecture Notes in Statistics*, 24, Springer-Verlag, New York, 1984, 24.

[33] Rozario, N. and Papoulis, A., The identification of certain nonlinear systems by only observing the output, *Workshop on Higher-Order Spectral Analysis*, Vail, CO, 73–77, June 1989.

[34] Schetzen, M., *The Volterra and Wiener Theories on Nonlinear System*, updated edition, Krieger Publishing Company, Malabar, FL, 1989.

[35] Swami, A. and Mendel, J.M., ARMA parameter estimation using only output cumulants, *IEEE Trans. Acoust., Speech and Signal Processing*, 38, 1257–1265, July 1990.

[36] Tick, L.J., The estimation of transfer functions of quadratic systems, *Technometrics*, 3(4), 562–567, Nov. 1961.

[37] United Signals & Systems, Inc., Comprehensive bibliography on higher-order statistics (spectra), Culver City, CA, 1992.

Primitive Recursion for Higher-Order Abstract Syntax

Joëlle Despeyroux[1], Frank Pfenning[2], and Carsten Schürmann[2]

[1] INRIA, F-06902 Sophia-Antipolis Cedex, France
joelle.despeyroux@sophia.inria.fr
[2] Carnegie Mellon University, Pittsburgh PA 15213, USA
{fp|carsten}@cs.cmu.edu

1 Introduction

Higher-order abstract syntax is a central representation technique in many logical frameworks, that is, meta-languages designed for the formalization of deductive systems. The basic idea is to represent variables of the object language by variables in the meta-language. Consequently, object language constructs which bind variables must be represented by meta-language constructs which bind the corresponding variables.

This deceptively simple idea, which goes back to Church [1] and Martin-Löf's system of arities [18], has far-reaching consequences for the methodology of logical frameworks. On one hand, encodings of logical systems using this idea are often extremely concise and elegant, since common concepts and operations such as variable binding, variable renaming, capture-avoiding substitution, or parametric and hypothetical judgments are directly supported by the framework and do not need to be encoded separately in each application. On the other hand, higher-order representations are no longer inductive in the usual sense, which means that standard techniques for reasoning by induction do not apply.

Various attempts have been made to preserve the advantages of higher-order abstract syntax in a setting with strong induction principles [5, 4], but none of these is entirely satisfactory from a practical or theoretical point of view.

In this paper we take a first step towards reconciling higher-order abstract syntax with induction by proposing a system of *primitive recursive functionals* that permits iteration over subjects of functional type. In order to avoid the well-known paradoxes which arise in this setting (see Section 3), we decompose the primitive recursive function space $A \Rightarrow B$ into a modal operator and a parametric function space $(\Box A) \rightarrow B$. The inspiration comes from linear logic which arises from a similar decomposition of the intuitionistic function space $A \supset B$ into a modal operator and a linear function space $(!A) \multimap B$.

The resulting system allows, for example, iteration over the structure of expressions from the untyped λ-calculus when represented using higher-order abstract syntax. It is general enough to permit iteration over objects of *any* simple type, constructed over *any* simply typed signature and thereby encompasses Gödel's system T [9]. Moreover, it is conservative over the simply-typed λ-calculus which means that the compositional adequacy of encodings in higher-order abstract syntax is preserved. We view our calculus as an important first step towards a system which allows the methodology of logical frameworks such as LF [10] to be incorporated into systems such as Coq [20] or ALF [12].

The remainder of this paper is organized as follows: Section 2 reviews the idea of higher order abstract syntax and introduces the simply typed λ-calculus ($\lambda^{\rightarrow}$) which we extend to a modal λ-calculus in Section 3. Section 4 then presents the concept of iteration. In Section 5 we sketch the proof of our central result, namely that our extension is conservative over $\lambda^{\rightarrow}$. Finally, Section 6 assesses the results, compares some related work, and outlines future work. A full version of this paper with complete technical developments and detailed proofs is accessible as `http://www.cs.cmu.edu/~carsten/CMU-CS-96-172.ps.gz` [6].

2 Higher-Order Abstract Syntax

Higher-order abstract syntax exploits the full expressive power of a typed λ-calculus for the representation of an object language, where λ-abstraction provides the mechanism to represent binding. In this paper, we restrict ourselves to a simply typed meta-language, although we recognize that an extension allowing dependent types and polymorphism is important future work (see Section 6). Our formulation of the simply-typed meta-language is standard.

$$\begin{array}{lrcl}
\text{Pure types:} & B & ::= & a \mid B_1 \rightarrow B_2 \\
\text{Objects:} & M & ::= & x \mid c \mid \lambda x\!:\!A.\, M \mid M_1\ M_2 \\
\\
\text{Context:} & \Psi & ::= & \cdot \mid \Psi, x : B \\
\text{Signature:} & \Sigma & ::= & \cdot \mid \Sigma, a : \text{type} \mid \Sigma, c : B
\end{array}$$

We use a for type constants, c for object constants and x for variables. We also fix a signature Σ for our typing and evaluation judgments so we do not have to carry it around.

Definition 1 Typing judgment. $\Psi \vdash M : B$ is defined by:

$$\frac{\Psi(x) = B}{\Psi \vdash x : B}\,\mathsf{StpVar} \qquad \frac{\Sigma(c) = B}{\Psi \vdash c : B}\,\mathsf{StpConst}$$

$$\frac{\Psi, x : B_1 \vdash M : B_2}{\Psi \vdash \lambda x\!:\!B_1.\, M : B_1 \rightarrow B_2}\,\mathsf{StpLam} \qquad \frac{\Psi \vdash M_1 : B_2 \rightarrow B_1 \qquad \Psi \vdash M_2 : B_2}{\Psi \vdash M_1\ M_2 : B_1}\,\mathsf{StpApp}$$

As running examples throughout the paper we use the representation of natural numbers and untyped λ-expressions.

Example 1 Natural numbers.

$$\begin{array}{rcll}
 & & \text{nat} & : \text{type} \\
\ulcorner 0 \urcorner & = \text{z} & \text{z} & : \text{nat} \\
\ulcorner n+1 \urcorner & = \text{s}\, \ulcorner n \urcorner & \text{s} & : \text{nat} \rightarrow \text{nat}
\end{array}$$

Untyped λ-expressions illustrate the idea of higher-order abstract syntax: object language variables are represented by meta-language variables.

Example 2 Untyped λ-expressions.

$$\text{Expressions: } e ::= x \mid \mathbf{lam}\ x.e \mid e_1 @ e_2$$

$$\begin{array}{rcl@{\qquad}rcl}
 & & & \text{exp} & : & \text{type} \\
\ulcorner \mathbf{lam}\ x.e \urcorner & = & \text{lam}\ (\lambda x{:}\text{exp}.\ulcorner e \urcorner) & \text{lam} & : & (\text{exp} \to \text{exp}) \to \text{exp} \\
\ulcorner e_1 @ e_2 \urcorner & = & \text{app}\ \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner & \text{app} & : & \text{exp} \to (\text{exp} \to \text{exp}) \\
\ulcorner x \urcorner & = & x & & &
\end{array}$$

Not every well-typed object of the meta-language directly represents an expression of the object language. For example, we can see that $\ulcorner e \urcorner$ will never contain a β-redex. Moreover, the argument to lam which has type exp $\to$ exp will always be a λ-abstraction. Thus the image of the translation in this representation methodology is always a β-normal and η-long form. Following [10], we call these forms *canonical* as defined by the following two judgments.

Definition 2 Atomic and canonical forms.

1. $\Psi \vdash V \downarrow B$ (V is atomic of type B in Ψ)
2. $\Psi \vdash V \Uparrow B$ (V is canonical of type B in Ψ)

are defined by:

$$\frac{\Psi(x) = B}{\Psi \vdash x \downarrow B}\,\mathsf{AtVar} \qquad \frac{\Sigma(c) = B}{\Psi \vdash c \downarrow B}\,\mathsf{AtConst} \qquad \frac{\Psi \vdash V_1 \downarrow B_2 \to B_1 \qquad \Psi \vdash V_2 \Uparrow B_2}{\Psi \vdash V_1\ V_2 \downarrow B_1}\,\mathsf{AtApp}$$

$$\frac{\Psi \vdash V \downarrow a}{\Psi \vdash V \Uparrow a}\,\mathsf{CanAt} \qquad \frac{\Psi, x : B_1 \vdash V \Uparrow B_2}{\Psi \vdash \lambda x{:}B_1.\, V \Uparrow B_1 \to B_2}\,\mathsf{CanLam}$$

Canonical forms play the role of "observable values" in a functional language: they are in one-to-one correspondence with the expressions we are trying to represent. For Example 2 (untyped λ-expressions) this is expressed by the following property, which is proved by simple inductions.

Example 3 Compositional adequacy for untyped λ-expressions.

1. Let e be an expression with free variables among $x_1, \ldots, x_n$.
   Then $x_1 : \text{exp}, \ldots, x_n : \text{exp} \vdash \ulcorner e \urcorner \Uparrow \text{exp}$.
2. Let $x_1 : \text{exp}, \ldots, x_n : \text{exp} \vdash M \Uparrow \text{exp}$.
   Then $M = \ulcorner e \urcorner$ for an expression e with free variables among $x_1, \ldots, x_n$.
3. $\ulcorner \cdot \urcorner$ is a bijection between expressions and canonical forms where $\ulcorner [e'/x]e \urcorner = [\ulcorner e' \urcorner / x]\ulcorner e \urcorner$.

Since every object in $\lambda^{\to}$ has a unique $\beta\eta$-equivalent canonical form, the meaning of every well-typed object is unambiguously given by its canonical form. Our operational semantics (see Definitions 4 and 7) computes this canonical form and therefore the meaning of every well-typed object. That this property is preserved under an extension of the language by primitive recursion for higher-order abstract syntax may be considered the main technical result of this paper.

3 Modal λ-Calculus

The constructors for objects of type exp from Example 2 are lam : $(\mathrm{exp} \rightarrow \mathrm{exp}) \rightarrow \mathrm{exp}$ and app : $\mathrm{exp} \rightarrow (\mathrm{exp} \rightarrow \mathrm{exp})$. These cannot be the constructors of an *inductive* type exp, since we have a negative occurrence of exp in the argument type of lam. This is not just a formal observation, but has practical consequences: we cannot formulate a consistent induction principle for expressions in this representation. Furthermore, if we increase the computational power of the meta-language by adding **case** or an iterator, then not every well-typed object of type exp has a canonical form. For example,

$$\cdot \vdash \mathrm{lam}\ (\lambda E\!:\!\mathrm{exp}.\, \mathrm{case}\ E\ \mathrm{of}\ \mathrm{app}\ E_1\ E_2 \Rightarrow \mathrm{app}\ E_2\ E_1 \mid \mathrm{lam}\ E' \Rightarrow \mathrm{lam}\ E')\ : \mathrm{exp}$$

but the given object does not represent any untyped λ-expression, nor could it be converted to one. The difficulty with a **case** or iteration construct is that there are many new functions of type $\mathrm{exp} \rightarrow \mathrm{exp}$ which cannot be converted to a function in $\lambda^{\rightarrow}$. This becomes a problem when such functions are arguments to constructors, since then the extension is no longer conservative even over expressions of base type (as illustrated in the example above).

Thus we must cleanly separate the *parametric function space* $\mathrm{exp} \rightarrow \mathrm{exp}$ whose elements are convertible to the form $\lambda x : \mathrm{exp}.\, E$ where E is built only from the constructors app, lam, and the variable x, from the *primitive recursive function space* $\mathrm{exp} \Rightarrow \mathrm{exp}$ which is intended to encompass functions defined through case distinction and iteration. This separation can be achieved by using a modal operator: $\mathrm{exp} \rightarrow \mathrm{exp}$ will continue to contain only the parametric functions, while $\mathrm{exp} \Rightarrow \mathrm{exp} = (\Box\mathrm{exp}) \rightarrow \mathrm{exp}$ contains the primitive recursive functions.

Intuitively, we interpret $\Box B$ as the type of *closed* objects of type B. We can iterate or distinguish cases over closed objects, since all constructors are statically known and can be provided for. This is not the case if an object may contain some unknown free variables. The system is non-trivial since we may also abstract over objects of type $\Box A$, but fortunately it is well understood and corresponds (via an extension of the Curry-Howard isomorphism) to the intuitionistic variant of S_4 [3].

In Section 4 we introduce schemas for defining functions by iteration and case distinction which require the subject to be of type $\Box B$. We can recover the ordinary scheme of primitive recursion for type nat if we also add pairs to the language. Pairs (with type $A_1 \times A_2$) are also necessary for the simultaneous definition of mutually recursive functions. Just as the modal type $\Box A$, pairs are lazy and values of these types are not observable—ultimately we are only interested in canonical forms of pure type.

The formulation of the modal λ-calculus below is copied from [3] and goes back to [22]. The language of types includes the pure types from the simply-typed λ-calculus in Section 2.

$$\frac{\Gamma(x) = A}{\Delta;\Gamma \vdash x : A}\,\mathsf{TpVarReg} \qquad \frac{\Delta(x) = A}{\Delta;\Gamma \vdash x : A}\,\mathsf{TpVarMod} \qquad \frac{\Sigma(c) = B}{\Delta;\Gamma \vdash c : B}\,\mathsf{TpConst}$$

$$\frac{\Delta;\Gamma, x : A_1 \vdash M : A_2}{\Delta;\Gamma \vdash \lambda x{:}A_1.\,M : A_1 \to A_2}\,\mathsf{TpLam}$$

$$\frac{\Delta;\Gamma \vdash M_1 : A_2 \to A_1 \qquad \Delta;\Gamma \vdash M_2 : A_2}{\Delta;\Gamma \vdash M_1\ M_2 : A_1}\,\mathsf{TpApp}$$

$$\frac{\Delta;\Gamma \vdash M_1 : A_1 \qquad \Delta;\Gamma \vdash M_2 : A_2}{\Delta;\Gamma \vdash \langle M_1, M_2\rangle : A_1 \times A_2}\,\mathsf{TpPair}$$

$$\frac{\Delta;\Gamma \vdash M : A_1 \times A_2}{\Delta;\Gamma \vdash \mathrm{fst}\ M : A_1}\,\mathsf{TpFst} \qquad \frac{\Delta;\Gamma \vdash M : A_1 \times A_2}{\Delta;\Gamma \vdash \mathrm{snd}\ M : A_2}\,\mathsf{TpSnd}$$

$$\frac{\Delta;\cdot \vdash M : A}{\Delta;\Gamma \vdash \mathrm{box}\ M : \Box A}\,\mathsf{TpBox} \qquad \frac{\Delta;\Gamma \vdash M_1 : \Box A_1 \qquad \Delta, x : A_1;\Gamma \vdash M_2 : A_2}{\Delta;\Gamma \vdash \mathrm{let\ box}\ x = M_1\ \mathrm{in}\ M_2 : A_2}\,\mathsf{TpLet}$$

Fig. 1. Typing judgment $\Delta;\Gamma \vdash M : A$

$$\begin{array}{lll}
\text{Types:} & A & ::= a \mid A_1 \to A_2 \mid \Box A \mid A_1 \times A_2 \\
\text{Objects:} & M & ::= c \mid x \mid \lambda x{:}A.\,M \mid M_1\ M_2 \\
 & & \quad \mid \mathrm{box}\ M \mid \mathrm{let\ box}\ x = M_1\ \mathrm{in}\ M_2 \mid \langle M_1, M_2\rangle \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M \\
\text{Contexts:} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}$$

For the sake of brevity we usually suppress the fixed signature Σ. However, it is important that signatures Σ and contexts denoted by Ψ will continue to contain only pure types, while contexts Γ and Δ may contain arbitrary types. We also continue to use B to range over pure types, while A ranges over arbitrary types. The typing judgment $\Delta;\Gamma \vdash M : A$ uses two contexts: Δ, whose variables range over closed objects, and Γ, whose variables range over arbitrary objects.

Definition 3 Typing judgment. $\Delta;\Gamma \vdash M : A$ is defined in Figure 1.

As examples, we show some basic laws of the (intuitionistic) modal logic S_4.

Example 4 Laws of S_4.

$$\begin{array}{ll}
\mathrm{funlift} & : \Box(A_1 \to A_2) \to \Box A_1 \to \Box A_2 \\
 & = \lambda f{:}\Box(A_1 \to A_2).\,\lambda x{:}\Box A_1. \\
 & \qquad \mathrm{let\ box}\ f' = f\ \mathrm{in\ let\ box}\ x' = x\ \mathrm{in\ box}\ (f'\ x')
\end{array}$$

$$\begin{array}{ll}
\mathrm{unbox} & : \Box A \to A \\
 & = \lambda x{:}\Box A.\,\mathrm{let\ box}\ x' = x\ \mathrm{in}\ x'
\end{array}$$

$$\begin{array}{ll}
\mathrm{boxbox} & : \Box A \to \Box\Box A \\
 & = \lambda x{:}\Box A.\,\mathrm{let\ box}\ x' = x\ \mathrm{in\ box}\ (\mathrm{box}\ x')
\end{array}$$

$$\frac{\Psi \vdash M \hookrightarrow V : a}{\Psi \vdash M \Uparrow V : a}\;\mathsf{EcAtomic} \qquad \frac{\Psi, x : B_1 \vdash M\ x \Uparrow V : B_2}{\Psi \vdash M \Uparrow \lambda x{:}B_1.\, V : B_1 \to B_2}\;\mathsf{EcArrow}$$

$$\frac{\Psi(x) = A}{\Psi \vdash x \hookrightarrow x : A}\;\mathsf{EvVar} \qquad \frac{\Sigma(c) = B}{\Psi \vdash c \hookrightarrow c : B}\;\mathsf{EvConst}$$

$$\frac{\cdot; \Psi, x : A_1 \vdash M : A_2}{\Psi \vdash \lambda x{:}A_1.\, M \hookrightarrow \lambda x{:}A_1.\, M : A_1 \to A_2}\;\mathsf{EvLam}$$

$$\frac{\begin{array}{l}\Psi \vdash M_1 \hookrightarrow \lambda x{:}A_2.\, M_1' : A_2 \to A_1 \\ \Psi \vdash M_2 \hookrightarrow V_2 : A_2 \\ \Psi \vdash [V_2/x](M_1') \hookrightarrow V : A_1\end{array}}{\Psi \vdash M_1\ M_2 \hookrightarrow V : A_1}\;\mathsf{EvApp} \qquad \frac{\begin{array}{l}\Psi \vdash M_1 \hookrightarrow V_1 : B_2 \to B_1 \\ \Psi \vdash V_1 \downarrow B_2 \to B_1 \\ \Psi \vdash M_2 \Uparrow V_2 : B_2\end{array}}{\Psi \vdash M_1\ M_2 \hookrightarrow V_1\ V_2 : B_1}\;\mathsf{EvAtomic}$$

$$\frac{\cdot; \Psi \vdash M_1 : A_1 \qquad \cdot; \Psi \vdash M_2 : A_2}{\Psi \vdash \langle M_1, M_2 \rangle \hookrightarrow \langle M_1, M_2 \rangle : A_1 \times A_2}\;\mathsf{EvPair}$$

$$\frac{\Psi \vdash M \hookrightarrow \langle M_1, M_2 \rangle : A_1 \times A_2 \qquad \Psi \vdash M_1 \hookrightarrow V : A_1}{\Psi \vdash \mathrm{fst}\ M \hookrightarrow V : A_1}\;\mathsf{EvFst}$$

$$\frac{\Psi \vdash M \hookrightarrow \langle M_1, M_2 \rangle : A_1 \times A_2 \qquad \Psi \vdash M_2 \hookrightarrow V : A_2}{\Psi \vdash \mathrm{snd}\ M \hookrightarrow V : A_2}\;\mathsf{EvSnd}$$

$$\frac{\cdot; \cdot \vdash M : A}{\Psi \vdash \mathrm{box}\ M \hookrightarrow \mathrm{box}\ M : \Box A}\;\mathsf{EvBox}$$

$$\frac{\Psi \vdash M_1 \hookrightarrow \mathrm{box}\ M_1' : \Box A \qquad \Psi \vdash [M_1'/x](M_2) \hookrightarrow V : A_2}{\Psi \vdash \mathrm{let\ box}\ x = M_1\ \mathrm{in}\ M_2 \hookrightarrow V : A_2}\;\mathsf{EvLet}$$

Fig. 2. Evaluation judgments $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$

The rules for evaluation must be constructed in such a way that full canonical forms are computed for objects of pure type, that is, we must evaluate under certain λ-abstractions. Objects of type $\Box A$ or $A_1 \times A_2$ on the other hand are not observable and may be computed lazily. We therefore use two mutually recursive judgments for evaluation and conversion to canonical form, written $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$, respectively. The latter is restricted to pure types, since only objects of pure type possess canonical forms. Since we evaluate under some λ-abstractions, free variables of pure type declared in Ψ may occur in M and V during evaluation.

Definition 4 Evaluation judgment. $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$ are defined in Figure 2.

4 Iteration

The modal operator $\Box$ introduced in Section 3 allows us to restrict iteration and case distinction to subjects of type $\Box B$, where B is a pure type. The technical realization of this idea in its full generality is rather complex. We therefore begin by describing the behavior of functions defined by iteration informally, incrementally developing their formal definition within our system. In the informal presentation we elide the box constructor, but we should convince ourselves that the subject of the iteration or case is indeed assumed to be closed.

Example 5 Addition. The usual type of addition is nat → nat → nat. This is no longer a valid type for addition, since it must iterate over either its first or second argument and would therefore not be parametric in that argument. Among the possible types for addition, we will be interested particularly in $\Box\text{nat} \to \text{nat} \to \text{nat}$ and $\Box\text{nat} \to \Box\text{nat} \to \Box\text{nat}$.

$$\begin{aligned}&\text{plus z}\ n &&= n\\ &\text{plus (s}\ m)\ n &&= \text{s (plus}\ m\ n)\end{aligned}$$

Note that this definition cannot be assigned type nat → nat → nat or $\Box\text{nat} \to \text{nat} \to \Box\text{nat}$.

In our system we view iteration as replacing constructors of a canonical term by functions of appropriate type, which is also the idea behind *catamorphisms* [8]. In the case of natural numbers, we replace $\text{z} : \text{nat}$ by a term $M_z : A$ and $\text{s} : \text{nat} \to \text{nat}$ by a function $M_s : A \to A$. Thus iteration over natural numbers replaces type nat by A. We use the notation $a \mapsto A$ for a *type replacement* and $c \mapsto M$ for a *term replacement.* Iteration in its simplest form is written as "it $\langle a \mapsto A\rangle\ M\ \langle \Omega\rangle$" where M is the subject of the iteration, and Ω is a list containing term replacements for all constructors of type a. The formal typing rules for replacements are given later in this section; first some examples.

Example 6 Addition via iteration. Addition from Example 5 can be formulated in a number of ways with an explicit iteration operator. The simplest one:

$$\begin{aligned}\text{plus}' &: \Box\text{nat} \to \text{nat} \to \text{nat}\\ &= \lambda m\!:\!\Box\text{nat}.\ \lambda n\!:\!\text{nat}.\ \text{it}\ \langle \text{nat} \mapsto \text{nat}\rangle\ m\ \langle \text{z} \mapsto n |\ \text{s} \mapsto \text{s}\rangle\end{aligned}$$

Later examples require addition with a result guaranteed to be closed. Its definition is only slightly more complicated.

$$\begin{aligned}\text{plus} &: \Box\text{nat} \to \Box\text{nat} \to \Box\text{nat}\\ &= \lambda m\!:\!\Box\text{nat}.\ \lambda n\!:\!\Box\text{nat}.\ \text{it}\ \langle \text{nat} \mapsto \Box\text{nat}\rangle\ m\\ &\qquad\quad \langle\, \text{z} \mapsto n\\ &\qquad\quad |\ \text{s} \mapsto (\lambda r\!:\!\Box\text{nat}.\ \text{let box}\ r' = r\ \text{in box (s}\ r')))\rangle\end{aligned}$$

If the data type is higher-order, iteration over closed objects must traverse terms with free variables. We model this in the informal presentation by introducing new parameters (written as $\nu x\!:\!B.\,M$) using Odersky's notation [19]. This makes a dynamic extension of the function definition necessary to encompass the new parameters (written as "where $f(x) = M$").

Example 7 Counting variable occurrences. Below is a function which counts the number of occurrences of bound variables in an untyped λ-expression in the representation of Example 2. It can be assigned type $\Box\text{exp} \to \Box\text{nat}$.

$$\begin{array}{ll} \text{cntvar (app } e_1\ e_2) & = \text{plus (cntvar } e_1)\ (\text{cntvar } e_2) \\ \text{cntvar (lam } e) & = \nu x\!:\!\text{exp. cntvar } (e\ x) \text{ where cntvar } x = (\text{s z}) \end{array}$$

It may look like the recursive call in the example above is not well-typed since $(e\ x)$ is not closed as required, but contains a free parameter x. Making sense of this apparent contradiction is the principal difficulty in designing an iteration construct for higher-order abstract syntax. As before, we model iteration via replacements. Here, $\text{exp} \mapsto \Box\text{nat}$ and so $\text{lam} \mapsto M_1$ and $\text{app} \mapsto M_2$ where $M_1 : (\Box\text{nat} \to \Box\text{nat}) \to \Box\text{nat}$ and $M_2 : \Box\text{nat} \to (\Box\text{nat} \to \Box\text{nat})$. The types of replacement terms M_1 and M_2 arise from the types of the constructors $\text{lam} : (\text{exp} \to \text{exp}) \to \text{exp}$ and $\text{app} : \text{exp} \to (\text{exp} \to \text{exp})$ by applying the type replacement $\text{exp} \mapsto \Box\text{nat}$. We write

$$\begin{array}{ll} \text{cntvar} & : \Box\text{exp} \to \Box\text{nat} \\ & = \lambda x\!:\!\Box\text{exp. it } \langle \text{exp} \mapsto \Box\text{nat}\rangle\ x \\ & \qquad \langle\ \text{app} \mapsto \text{plus} \\ & \qquad \mid \text{lam} \mapsto \lambda f\!:\!\Box\text{nat} \to \Box\text{nat}.\ f\ (\text{box (s z)})\rangle \end{array}$$

Informally, the result of $\text{cntvar (lam } (\lambda x\!:\!\text{exp}.\ x))$ can be determined as follows:

$$\begin{array}{l} \text{cntvar (lam } (\lambda x\!:\!\text{exp}.\ x)) \\ = \nu x'\!:\!\text{exp. cntvar } ((\lambda x\!:\!\text{exp}.\ x)\ x') \text{ where cntvar } x' = (\text{s z}) \\ = \nu x'\!:\!\text{exp. cntvar } x' \text{ where cntvar } x' = (\text{s z}) \\ = \nu x'\!:\!\text{exp. } (\text{s z}) \text{ where cntvar } x' = (\text{s z}) \\ = (\text{s z}) \end{array}$$

For the formal operational semantics, see Example 10.

A number of functions can be defined elegantly in this representation. Among them are the conversion from type exp to a representation using de Bruijn indices and one-step parallel reduction. The latter requires mutual iteration and pairs (see [6]).

The following example illustrates two concepts: mutually inductive types and iteration over the form of a (parametric!) function.

Example 8 Substitution in normal forms. Substitution is already directly definable by application, but one may also ask if there is a structural definition in the style of [16]. Normal forms of the untyped λ-calculus are represented by the type nf with an auxiliary definition for atomic forms of type at.

$$\begin{array}{l} \text{Normal forms: } N ::= P \mid \textbf{lam}\ x.N \\ \text{Atomic forms: } P ::= x \mid P@N \end{array}$$

In this example the represention function $\ulcorner . \urcorner$ acts on normal forms, atomic forms are represented by $\ulcorner\!\ulcorner . \urcorner\!\urcorner$.

$$
\begin{array}{rcl@{\qquad}lcl}
 & & & \mathrm{nf} & : & \mathrm{type} \\
 & & & \mathrm{at} & : & \mathrm{type} \\
\ulcorner P \urcorner & = & \mathrm{atnf}\ \ulcorner\!\ulcorner P \urcorner\!\urcorner & \mathrm{atnf} & : & \mathrm{at} \to \mathrm{nf} \\
\ulcorner \mathbf{lam}\ x.N \urcorner & = & \mathrm{lm}\ (\lambda x\!:\!\mathrm{at}.\ \ulcorner N \urcorner) & \mathrm{lm} & : & (\mathrm{at} \to \mathrm{nf}) \to \mathrm{nf} \\
\ulcorner\!\ulcorner P @ N \urcorner\!\urcorner & = & \mathrm{ap}\ \ulcorner\!\ulcorner P \urcorner\!\urcorner\ \ulcorner N \urcorner & \mathrm{ap} & : & \mathrm{at} \to \mathrm{nf} \to \mathrm{at} \\
\ulcorner\!\ulcorner x \urcorner\!\urcorner & = & x & & &
\end{array}
$$

Substitution of atomic objects for variables is defined by two mutually recursive functions, one with type $\mathrm{subnf} : \Box(\mathrm{at} \to \mathrm{nf}) \to \mathrm{at} \to \mathrm{nf}$ and $\mathrm{subat} : \Box(\mathrm{at} \to \mathrm{at}) \to \mathrm{at} \to \mathrm{at}$.

$$
\begin{array}{lcl}
\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\ \mathrm{lm}\ (\lambda y\!:\!\mathrm{at}.\ N\ x\ y))\ Q & = & \mathrm{lm}\ (\lambda y\!:\!\mathrm{at}.\ \mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\ (N\ x\ y))\ Q \\
 & & \qquad \text{where } \mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\ y)\ Q = y) \\
\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\ \mathrm{atnf}\ (P\ x))\ Q & = & \mathrm{atnf}\ (\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\ P\ x)\ Q) \\
\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\ \mathrm{ap}\ (P\ x)\ (N\ x))\ Q & = & \mathrm{ap}\ (\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\ P\ x)\ Q) \\
 & & \quad (\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\ N\ x)\ Q) \\
\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\ x)\ Q & = & Q
\end{array}
$$

The last case arises since the parameter x must be considered as a new constructor in the body of the abstraction. The functions above are realized in our calculus by a simultaneous replacement of objects of type nf and at. In other words, the type replacement must account for all mutually recursive types, and the term replacement for all constructors of those types.

$$
\begin{array}{l}
\mathrm{subnf}\ :\ \Box(\mathrm{at} \to \mathrm{nf}) \to \mathrm{at} \to \mathrm{nf} \\
\qquad = \lambda N\!:\!\Box(\mathrm{at} \to \mathrm{nf}).\ \lambda Q\!:\!\mathrm{at}.\ \mathrm{it}\ \langle \mathrm{nf} \mapsto \mathrm{nf} \mid \mathrm{at} \mapsto \mathrm{at} \rangle\ N \\
\qquad\qquad\qquad \langle\ \mathrm{lm} \mapsto \lambda F\!:\!\mathrm{at} \to \mathrm{nf}.\ \mathrm{lm}\ (\lambda y\!:\!\mathrm{at}.\ (F\ y)) \\
\qquad\qquad\qquad \mid \mathrm{atnf} \mapsto \lambda P\!:\!\mathrm{at}.\ \mathrm{atnf}\ P \\
\qquad\qquad\qquad \mid \mathrm{ap} \mapsto \lambda P\!:\!\mathrm{at}.\ \lambda N\!:\!\mathrm{nf}.\ \mathrm{ap}\ P\ N \rangle \\
\qquad\qquad\qquad Q
\end{array}
$$

Via η-contraction we can see that substitution amounts to a structural identity function.

We begin now with the formal discussion and description of the full language. Due to the possibility of mutual recursion among types, the type replacements must be lists (see Example 8).

$$\text{Type replacement: } \omega ::= \cdot \mid (\omega \mid a \mapsto A)$$

Which types must be replaced by an iteration depends on which types are mutually recursive according to the constructors in the signature Σ and possibly the type of the iteration subject itself. If we iterate over a function, the parameter of a function must be treated like a constructor for its type, since it can appear in that role in the body of a function.

Thus, we define the notion of *type subordination* which summarizes all dependencies between atomic types by separately considering its *static* part $\lhd_{\Sigma}$ which derives from the dependencies induced by the constructor types from Σ and its *dynamic* part $\lhd_{B}$ which accounts for dependencies induced from the argument types of B. The transitive closure $\blacktriangleleft_{\Sigma;B}$ of static and dynamic type subordination relation defines cleanly all dependencies between types which govern the formation of the subject of iteration. We denote the *target type* of a pure type B by $\tau(B)$. All type constants which are mutually dependent with $\tau(B)$, written $\mathcal{I}(\Sigma; B)$, form the domain of the type replacement ω:

$$\mathcal{I}(\Sigma; B) := \{a | \tau(B) \blacktriangleleft_{\Sigma;B} a \text{ and } a \blacktriangleleft_{\Sigma;B} \tau(B)\}$$

In Example 8 of normal and atomic forms we have $\mathcal{I}(\Sigma; \text{at} \to \text{nf}) = \{\text{at}, \text{nf}\}$. Note that type subordination is built into calculi where inductive types are defined explicitly (such as the Calculus of Inductive Constructions [20]); here it must be recovered from the signature since we impose no ordering constraints except that a type must be declared before it is used. Our choice to recover the type subordination relation from the signature allows us to perform iteration over any functional type, without fixing the possibilities in advance.

Let us now address the question of how the type of an iteration is formed: If the subject of iteration has type B, the iterator object has type $\langle\omega\rangle(B)$, where $\langle\omega\rangle(B)$ is defined inductively by replacing each type constant according to ω, leaving types outside the domain fixed.

A similar replacement is applied at the level of terms: the result of an iteration is an object which resembles the (canonical) subject of the iteration in structure, but object constants are replaced by other objects carrying the intended computational meaning of the different cases. Even though the subject of iteration is closed at the beginning of the replacement process, we need to deal with embedded λ-abstractions due to higher-order abstract syntax. But since such functions are parametric we can simply replace variables x of type B by new variables x' of type $\langle\omega\rangle(B)$.

$$\text{Term replacement: } \Omega ::= \cdot \mid (\Omega \mid c \mapsto M) \mid (\Omega \mid x \mapsto x')$$

Initially the domain of a term replacement is a signature containing all constructors whose target type is in $\mathcal{I}(\Sigma; B)$. We refer to this signature as $\mathcal{S}_{\text{it}}(\Sigma; B)$. The form of iteration follows now quite naturally: We extend the notion of objects by

$$M ::= \ldots \mid \text{it } \langle\omega\rangle\ M\ \langle\Omega\rangle$$

and define the following typing rules for iteration and term replacements.

Definition 5 Typing judgment for iteration. (extending Definition 3)

$$\frac{\Delta; \Gamma \vdash M : \Box B \qquad \Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\mathcal{S}_{\text{it}}(\Sigma; B))}{\Delta; \Gamma \vdash \text{it } \langle\omega\rangle\ M\ \langle\Omega\rangle : \langle\omega\rangle(B)}\,\textsf{TpIt}, \qquad \text{dom}(\omega) = \mathcal{I}(\Sigma; B)$$

$$\frac{}{\Delta; \Gamma \vdash \cdot : \langle\omega\rangle(\cdot)}\,\textsf{TrBase} \qquad \frac{\Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\Sigma) \qquad \Delta; \Gamma \vdash M : \langle\omega\rangle(B')}{\Delta; \Gamma \vdash (\Omega \mid c \mapsto M) : \langle\omega\rangle(\Sigma, c : B')}\,\textsf{TrInd}$$

Example 9. In Example 7 we defined $\text{cntvar} = \lambda x{:}\Box\text{exp}.\,\text{it}\ \langle\omega\rangle\ x\ \langle\Omega\rangle$ where

$$\begin{aligned}\omega &= \text{exp} \mapsto \Box\text{nat}\\ \Omega &= \text{app} \mapsto \text{plus}, \text{lam} \mapsto \lambda f{:}\Box\text{nat} \to \Box\text{nat}.\, f\ (\text{box}\ (\text{s}\ \text{z}))\\ \mathcal{S}_{\text{it}}(\Sigma; \text{exp}) &= \text{app} : \text{exp} \to (\text{exp} \to \text{exp}), \text{lam} : (\text{exp} \to \text{exp}) \to \text{exp}\end{aligned}$$

Under the assumption that plus : $\Box\text{nat} \to (\Box\text{nat} \to \Box\text{nat})$ it is easy to see that

(1) $\cdot; x : \Box\text{exp} \vdash \lambda f{:}\Box\text{nat} \to \Box\text{nat}.\, f\ (\text{box}\ (\text{s}\ \text{z})) : (\Box\text{nat} \to \Box\text{nat}) \to \Box\text{nat}$ by TpLam, etc.

(2) $\cdot; x : \Box\text{exp} \vdash \Omega : \langle\omega\rangle(\mathcal{S}_{\text{it}}(\Sigma; \text{exp}))$ by TrBase, Ass., (1)

(3) $\cdot; x : \Box\text{exp} \vdash x : \Box\text{exp}$ by $\mathsf{TpVarReg}$

(4) $\cdot; x : \Box\text{exp} \vdash \text{it}\ \langle\omega\rangle\ x\ \langle\Omega\rangle : \Box\text{nat}$ by TpIt from (3) (2)

(5) $\cdot; \cdot \vdash \text{cntvar} : \Box\text{exp} \to \Box\text{nat}$ by TpLam from (4)

Applying a term replacement must be restricted to canonical forms in order to preserve types. Fortunately, our type system guarantees that the subject of an iteration can be converted to canonical form. Applying a replacement then transforms a canonical form V of type B into a well-typed object $\langle\omega; \Omega\rangle(V)$ of type $\langle\omega\rangle(B)$. We call this operation *elimination.* It is defined along the structure of V.

Definition 6 Elimination.

$$\begin{aligned}\langle\omega; \Omega\rangle(c) &= \begin{cases} M & \text{if } \Omega(c) = M\\ c & \text{otherwise}\end{cases} && (\mathsf{ElConst})\\ \langle\omega; \Omega\rangle(x) &= \Omega(x) && (\mathsf{ElVar})\\ \langle\omega; \Omega\rangle(\lambda x{:}B.\, V) &= \lambda u{:}\langle\omega\rangle(B).\, \langle\omega; \Omega \mid x \mapsto u\rangle(V) && (\mathsf{ElLam})\\ \langle\omega; \Omega\rangle(V_1\ V_2) &= \langle\omega; \Omega\rangle(V_1)\ \langle\omega; \Omega\rangle(V_2) && (\mathsf{ElApp})\end{aligned}$$

The term resulting from elimination might, of course, contain redices and must itself be evaluated to obtain a final value. Thus we obtain the following evaluation rule for iteration.

Definition 7 Evaluation judgment. (extending Definition 4)

$$\frac{\Psi \vdash M \hookrightarrow \text{box}\ M' : \Box B \qquad \cdot \vdash M' \Uparrow V' : B \qquad \Psi \vdash \langle\omega; \Omega\rangle(V') \hookrightarrow V : \langle\omega\rangle(B)}{\Psi \vdash \text{it}\ \langle\omega\rangle\ M\ \langle\Omega\rangle \hookrightarrow V : \langle\omega\rangle(B)}\,\mathsf{EvIt}$$

Example 10. In Example 7, the evaluation of $\text{cntvar}\ (\text{box}\ (\text{lam}\ (\lambda x{:}\text{exp}.\, x)))$ yields $\text{box}\ (\text{s}\ \text{z})$ because

(1) $\cdot \vdash \mathrm{cntvar} \hookrightarrow \mathrm{cntvar} : \Box\mathrm{exp} \to \Box\mathrm{nat}$ by EvLam

(2) $\cdot \vdash \mathrm{box}\ (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x)) \hookrightarrow \mathrm{box}\ (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x)) : \Box\mathrm{exp}$ by EvBox

(3) $\cdot \vdash \mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x) \Uparrow \mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x) : \mathrm{exp}$ by EcAtomic, etc.

(4) $\langle \omega; \Omega \rangle (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x))$
$= (\lambda f\!:\!\Box\mathrm{nat} \to \Box\mathrm{nat}.\, f\ (\mathrm{box}\ (\mathrm{s\ z})))\ (\lambda x'\!:\!\Box\mathrm{nat}.\, x')$ by elimination

(5) $\cdot \vdash \langle \omega; \Omega \rangle (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x)) \hookrightarrow \mathrm{box}\ (\mathrm{s\ z}) : \Box\mathrm{nat}$ by EvApp, etc.

(6) $\cdot \vdash \mathrm{it}\ \langle \omega \rangle\ (\mathrm{box}\ (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x)))\ \langle \Omega \rangle \hookrightarrow \mathrm{box}\ (\mathrm{s\ z}) : \Box\mathrm{nat}$
by EvIt from (2) (3) (5)

(7) $\cdot \vdash \mathrm{cntvar}\ (\mathrm{box}\ (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x))) \hookrightarrow \mathrm{box}\ (\mathrm{s\ z}) : \Box\mathrm{nat}$
by EvApp from (1) (2) (6)

The reader is invited to convince himself that this operational semantics yields the expected results also on the other examples of this section.

Our calculus also contains a **case** construct whose subject may be of type $\Box B$ for arbitrary pure B. It allows us to distinguish cases based on the intensional structure of the subject. For example, we can test if a given (parametric!) function is the identity or not. The typing rules and operational semantics for **case** are similar, but simpler than those for iteration. We therefore elide it here and refer the interested reader to [6].

5 Meta-Theory

The goal of this subsection is to show that the modal λ-calculus obeys the type preservation property and that it is a conservative extension of the simply typed λ-calculus defined in Section 2. We prove this by Tait's method, often called an argument by *logical relations.* After defining the logical relations we then prove the canonical form theorem for the modal λ-calculus which guarantees that every well-typed object eventually evaluates to a canonical form. The type preservation and the conservative extension property follow directly from this theorem.

We now begin the meta-theoretical discussion with the definition of the logical relation. Due to the lazy character of the modal λ-calculus, the interpretation of a type A is twofold: On the one side we would like it to contain all canonical forms of type A, on the other all objects *evaluating* to a canonical form. This is why we introduce two mutual dependent logical relations: In a context Ψ, $[\![A]\!]$ represents the set of objects evaluating to a value being itself an element of $|A|$. For the definition of the logical relation we require the notion of context extension: $\Psi' \geq \Psi$ holds if every declaration in Ψ also occurs in Ψ'.

Definition 8 Logical relation.

$\Psi \vdash M \in [\![A]\!]$ iff $\cdot; \Psi \vdash M : A$ and $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash V \in |A|$

$\Psi \vdash V \in |A|$ iff

Case: $A = a$ and $\Psi \vdash V \Uparrow a$

Case: $A = A_1 \to A_2$ and either

Case: $V = \lambda x : A_1 . M$ and for all $\Psi' \geq \Psi$: $\Psi' \vdash V' \in |A_1|$ implies $\Psi' \vdash [V'/x](M) \in [\![A_2]\!]$

or

Case: $\Psi \vdash V \downarrow A_1 \to A_2$ and for all $\Psi' \geq \Psi$: $\Psi' \vdash V' \Uparrow A_1$ implies $\Psi' \vdash V\ V' \in |A_2|$

Case: $A = A_1 \times A_2$ and $V = \langle M_1, M_2 \rangle$ and $\Psi \vdash M_1 \in [\![A_1]\!]$ and $\Psi \vdash M_2 \in [\![A_2]\!]$

Case: $A = \Box A'$ and $V = \text{box}\ M$ and $\cdot \vdash M \in [\![A']\!]$

Since the operational semantics introduced in Section 4 depends on typing information, we must make sure that only well-typed objects are contained in the logical relation. To do so we require that every object $M \in [\![A]\!]$ has type A. As a side effect of this definition the type preservation property is a direct consequence of the canonical form theorem. The proof of the canonical form theorem is split into two parts. In the first part we prove that every element in $[\![A]\!]$ evaluates to a canonical form, in the second part we show that every well-typed object of type A is contained in the logical relation $[\![A]\!]$.

A direct proof of the first property will fail, hence we must generalize its formulation which we can now prove by mutual induction.

Lemma 9 Logical relations and canonical forms.

1. *If* $\Psi \vdash M \in [\![B]\!]$ *then* $\Psi \vdash M \Uparrow V : B$
2. *If* $\Psi \vdash V \downarrow B$ *then* $\Psi \vdash V \in |B|$

The goal of the second part is to show that every well-typed object is in the logical relation, that is, we want to prove that if $\cdot; \Psi \vdash M : A$ then $\Psi \vdash M \in [\![A]\!]$.

It turns out that we cannot prove this property directly by induction over the structure of the typing derivation, either. The reason is that the context Ψ might grow during the derivation and it may also not remain pure. From the definition of the typing relation it also follows quite directly that Δ need not remain empty. Hence we generalize this property by considering a typing derivation $\Delta; \Gamma \vdash M : A$ and a substitution $(\theta; \varrho)$ which maps the variables defined in $\Delta; \Gamma$ into objects satisfying the logical relation to show that $\Psi \vdash [\theta; \varrho](M) \in [\![A]\!]$. The objects in θ — which are only substituting modal variables — might not yet be evaluated due to the lazy character of unobservable objects. On the other hand it is safe to assume that objects in ϱ — which substitute for variables in Γ — are already evaluated since function application follows a call-by-value discipline. To make this more precise we define a logical relation for contexts $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$ iff $\vdash \theta \in [\![\Delta]\!]$ and $\Psi \vdash \varrho \in |\Gamma|$ which are defined as follows:

Definition 10 Logical relation for contexts.

$\vdash \theta \in [\![\Delta]\!]$ iff $\Delta = \cdot$ implies $\theta = \cdot$

and $\Delta = \Delta', x : A$ implies $\theta = \theta', M/x$ and $\cdot \vdash M \in [\![A]\!]$ and $\vdash \theta' \in [\![\Delta']\!]$

$\Psi \vdash \varrho \in |\Gamma|$ iff $\Gamma = \cdot$ implies $\varrho = \cdot$

and $\Gamma = \Gamma', x : A$ implies $\varrho = \varrho', V/x$ and $\Psi \vdash V \in |A|$ and $\Psi \vdash \varrho' \in |\Gamma'|$

We can now formulate and prove

Lemma 11 Typing and logical relations.

If $\Delta; \Gamma \vdash M : A$ and $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$ then $\Psi \vdash [\theta; \varrho](M) \in [\![A]\!]$

The proof of this lemma is rather difficult. Due to the restrictions in length we are not presenting any details here, but refer the interested reader to [6]. Now, an easy inductive argument using Lemma 9 shows that the identity substitution $(\cdot, \mathrm{id}_\Psi)$, which maps all variables defined in Ψ to themselves, indeed lies within the logical relation $[\cdot; \Psi]$. The soundness of typing is hence an immediate corollary of Lemma 11.

Theorem 12 Soundness of typing.

If $\cdot; \Psi \vdash M : A$ then $\Psi \vdash M \in [\![A]\!]$.

This theorem together with Lemma 9 guarantees that terms of pure type evaluate to a canonical form V.

Theorem 13 Canonical form theorem.

If $\cdot; \Psi \vdash M : B$ then $\Psi \vdash M \Uparrow V : B$ for some V.

Type preservation now follows easily: We need to show that the evaluation result of a well-typed object possesses the same type A. From Lemma 11 we obtain that M lies within the logical relation $[\![A]\!]$, which guarantees that M evaluates to a term V, also in the logical relation. Further, a simple induction over the structure of an evaluation shows the values are unique and have the right type.

Theorem 14 Type preservation.

If $\cdot; \Psi \vdash M : A$ and $\Psi \vdash M \hookrightarrow V : A$ then $\cdot; \Psi \vdash V : A$.

On the basis of the canonical form theorem, we can now prove the main result of the paper: Our calculus is a conservative extension of the simply typed λ-calculus $\lambda^{\rightarrow}$ from Section 2. Let M be an object of pure type B, with free variables from a pure context Ψ. M itself need not be pure but rather some term in the modal λ-calculus including iteration and case. We have seen that M has a canonical form V, and an immediate inductive argument shows that V must be a term in the simply typed λ-calculus.

Theorem 15 Conservative extension.

If $\cdot; \Psi \vdash M : B$ then $\Psi \vdash M \Uparrow V : B$ and $\Psi \vdash V \Uparrow B$.

6 Conclusion and Future Work

We have presented a calculus for primitive recursive functionals over higher-order abstract syntax which guarantees that the adequacy of encodings remains intact. The requisite conservative extension theorem is technically deep and requires a careful system design and analysis of the properties of a modal operator $\Box$ and its interaction with function definition by iteration and cases. To our knowledge, this is the first system in which it is possible to safely program functionally with higher-order abstract syntax representations. It thus complements and refines the logic programming approach to programming with such representations [17, 21].

Our work was inspired by Miller's system [15], which was presented in the context of ML. Due to the presence of unrestricted recursion and the absence of a modal operator, Miller's system is computationally adequate, but has a much weaker meta-theory which would not be sufficient for direct use in a logical framework. The system of Meijer and Hutton [14] and its refinement by Fegaras and Sheard [8] are also related in that they extend primitive recursion to encompass functional objects. However, they treat functional objects extensionally, while our primitives are designed so we can analyze the internal structure of λ-abstractions directly. Fegaras and Sheard also note the problem with adequacy and design more stringent type-checking rules in Section 3.4 of [8] to circumvent this problem. In contrast to our system, their proposal does not appear to have a logical interpretation. Furthermore, they neither claim nor prove type preservation or an appropriate analogue of conservative extension—critical properties which are not obvious in the presence of their internal type tags and **Place** constructor.

Our system is satisfactory from the theoretical point of view and could be the basis for a practical implementation. Such an implementation would allow the definition of functions of arbitrary types, while data constructors are constrained to have pure type. Many natural functions over higher-order representations turn out to be directly definable (e.g., one-step parallel reduction or conversion to de Bruijn indices), others require explicit counters to guarantee termination (e.g., multi-step reduction or full evaluation). On the other hand, it appears that some natural *algorithms* (e.g., a structural equality check which traverses two expressions simultaneously) are not implementable, even though the underlying function is certainly definable (e.g., via a translation to de Bruijn indices). For larger applications, writing programs by iteration becomes tedious and error-prone and a pattern-matching calculus such as employed in ALF [2] or proposed by Jouannaud and Okada [11] seems more practical. Our informal notation in the examples provides some hints what concrete syntax one might envision for an implementation along these lines.

The present paper is a first step towards a system with dependent types in which proofs of meta-logical properties of higher-order encodings can be expressed directly by dependently typed, total functions. The meta-theory of such a system appears to be highly complex, since the modal operators necessitate a *let box* construct which, *prima facie*, requires commutative conversions. Martin Hofmann[3] has proposed a semantical explanation for our iteration operator which has led him to discover an equational formulation of the laws for iteration. This may be the critical insight required for a dependently typed version of our calculus. A similar formulation of these laws is used in [7] for the treatment of recursion. We also plan to reexamine applications in the realm of functional programming [15, 8] and related work on reasoning about higher-order abstract syntax with explicit induction [5, 4] or definitional reflection [13].

Acknowledgments. The work reported here took a long time to come to fruition, largely due to the complex nature of the technical development. During this time we have discussed various aspects of higher-order abstract syntax, iteration, and induction with too many people to acknowledge them individually. Special thanks go to Gérard Huet and Chet Murthy, who provided the original inspiration, and Hao-Chi Wong who helped us understand the nature of modality in this context.

References

1. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
2. Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.
3. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Jr. Guy Steele, editor, *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida, January 1996. ACM Press.
4. Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.
5. Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.
6. Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, September 1996.
7. Thierry Despeyroux and André Hirschowitz. Some theory for abstract syntax and induction. Draft manuscript.
8. Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of 23rd Annual Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, January 1996. ACM Press.
9. Kurt Gödel. On an extension of finitary mathematics which has not yet been used. In Solomon Feferman et al., editors, *Kurt Gödel, Collected Works, Volume II*, pages 271–280. Oxford University Press, 1990.

[3] personal communication

10. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
11. Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In Gilles Kahn, editor, *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
12. Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.
13. Raymond McDowell and Dale Miller. A logic for reasoning about logic specifications. Draft manuscript, July 1996.
14. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995.
15. Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, May 1990.
16. Dale Miller. Unification of simply typed lambda-terms as logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
17. Dale Miller. Abstract syntax and logic programming. In *Proceedings of the First and Second Russian Conferences on Logic Programming*, pages 322–337, Irkutsk and St. Petersburg, Russia, 1992. Springer-Verlag LNAI 592.
18. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.
19. Martin Odersky. A functional theory of local names. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, pages 48–59, Portland, Oregon, January 1994. ACM Press.
20. Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.
21. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
22. Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In S. Brookes and M. Main, editors, *Proceedings of the Eleventh Conference on Mathematical Foundations of Programming Sematics*, New Orleans, Louisiana, March 1995. To appear in *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.

This article was processed using the LaTeX macro package with LLNCS style

Higher Lawvere theories

John D. Berman

March 8, 2019

Abstract

We survey Lawvere theories at the level of ∞-categories, as an alternative framework for higher algebra (rather than ∞-operads). From a pedagogical perspective, they make many key definitions and constructions less technical. They are also better-suited than operads for equivariant homotopy theory and its relatives.

Our main result establishes a universal property for the ∞-category of Lawvere theories, which completely characterizes the relationship between a Lawvere theory and its ∞-category of models. Many familiar properties of Lawvere theories follow directly.

As a consequence, we prove that the Burnside category is a classifying object for additive categories, as promised in an earlier paper, and as part of a more general correspondence between enriched Lawvere theories and module Lawvere theories.

1 Introduction

The primary goal of this paper is to lay the foundations for a Lawvere theoretic approach to higher algebra. As such, we organize the paper as a survey, combining others' work with our own. In this introduction, we summarize the new results for those who are familiar with the subject; however, the casual reader may be better served by reading Section 2, followed by this introduction.

A Lawvere theory $\mathcal{L}$ encodes a particular type of algebraic theory (for example, commutative monoids). A *model* of $\mathcal{L}$ is an instance of that algebraic structure. As we are working with ∞-categories, we take models by default relative to the ∞-category Top of spaces.

Specifically, $\mathcal{L}$ is an ∞-category with finite products, generated by a single object, and models are functors $\mathcal{L} \to \mathrm{Top}$ which preserve finite products.

The author was supported by an NSF Postdoctoral Fellowship under grant 1803089.

The assignment from $\mathcal{L}$ to the ∞-category $\mathrm{Mdl}_{\mathcal{L}}$ of models is functorial

$$\mathrm{Mdl} : \mathrm{Lwv} \to \mathrm{Pr}^L,$$

where an object of Pr^L is a presentable ∞-category, and a morphism is a left adjoint functor. In fact, since any Lawvere theory $\mathcal{L}$ has a distinguished object 1, $\mathrm{Mdl}_{\mathcal{L}}$ also has a distinguished object, the model $\mathrm{Map}_{\mathcal{L}}(1, -)$. Therefore, Mdl may be promoted to a functor $\mathrm{Lwv} \to \mathrm{Pr}^L_*$.

Theorem (Theorems 2.10, 3.3)**.** *The functor $Mdl : Lwv \to Pr^L_*$ is fully faithful and symmetric monoidal, and it has a right adjoint which sends $\mathcal{C} \in Pr^L_*$ to $\mathcal{C}^{op}_{fgf}$, where $\mathcal{C}_{fgf} \subseteq \mathcal{C}$ is the full subcategory of* finitely generated free *objects.*

In this way, Lwv is a *symmetric monoidal colocalization* of the better understood ∞-category Pr^L_*, which allows us to study Lawvere theories using familiar tools like the adjoint functor theorem and Lurie's tensor products of categories ([17] 4.8).

The math here is not new, but the packaging is. That is, the theorem encapsulates the following facts about Lawvere theories:

- (the adjunction) $\mathrm{Mdl}_{\mathcal{L}}$ is the *free cocompletion* of $\mathcal{L}^{\mathrm{op}}$, regarding the latter as an ∞-category with finite coproducts ([18] 5.3.6.10);
- (Mdl is fully faithful) $\mathcal{L}$ can be recovered from $\mathrm{Mdl}_{\mathcal{L}}$ as the full subcategory of finitely generated free objects (Proposition 2.7);

The fact that Mdl is symmetric monoidal may be less familiar, but it is a consequence of Lurie's 'tensor products of categories' machinery in [17] 4.8. In Section 3, we explore two direct corollaries of this monoidality:

- (Corollary 3.4) If a Lawvere theory $\mathcal{L}$ admits a symmetric monoidal structure compatible with finite products, then $\mathrm{Mdl}_{\mathcal{L}}$ inherits a closed symmetric monoidal structure called *Day convolution*;
- (Corollary 3.8) If $\mathcal{L}$ is as above, then $\mathcal{L}$ is a semiring ∞-category. Any module over $\mathcal{L}$ is naturally enriched in $\mathrm{Mdl}_{\mathcal{L}}$ with its Day convolution.

The first of these is well-known, but the second we believe is new. It also admits a partial converse:

Theorem (Theorem 3.11)**.** *If $\mathcal{L}$ admits a symmetric monoidal structure compatible with finite products, and another Lawvere theory $\mathcal{K}$ is enriched in $Mdl_{\mathcal{L}}$, then $\mathcal{K}$ is naturally tensored over $\mathcal{L}$ via a map of Lawvere theories*

$$\mathcal{L} \otimes \mathcal{K} \to \mathcal{K}.$$

This theorem suggests that there may be a strong converse to Corollary 3.8:

Conjecture (Conjecture 3.12). *If $\mathcal{L}$ is as above, the ∞-categories of $Mdl_{\mathcal{L}}$-enriched Lawvere theories and $\mathcal{L}$-module Lawvere theories are equivalent.*

Sections 4 and 5 are devoted to examples and applications:

For any $\mathbb{E}_1$-semiring space R, there is a Lawvere theory whose models are R-modules. In Section 4, we show that these are the only semiadditive Lawvere theories; that is, semiadditive Lawvere theories can be identified with semiring spaces. This is an easy result for classical Lawvere theories. We record it here because it is slightly more subtle for ∞-categories. It also suggests a philosophy that we are fond of: Lawvere theories may be regarded as generalized (non-additive) rings.

In Section 5.1, we use our ideas relating module Lawvere theories to enriched Lawvere theories, proving a result promised in the author's earlier paper [5]:

Theorem (Theorem 5.2). *The Burnside ∞-category is a commutative semiring ∞-category, and there is an equivalence*

$$Mod_{Burn} \cong AddCat_{\infty},$$

where $AddCat_{\infty}$ denotes the ∞-category of additive ∞-categories.

Finally, in Section 5.2, we describe an application to equivariant homotopy theory. There are no results; at this point, the application is just motivational.

1.1 Acknowledgment

This paper is largely drawn from the author's thesis [4], of which it is the second part. It has been in the works for years, and benefitted from conversations with many others, including Ben Antieau, Clark Barwick, Saul Glasman, Rune Haugseng, Mike Hill, Bogdan Krstic, and others.

2 Fundamentals of Lawvere theories

2.1 Lawvere theories and their models

Classical Lawvere theories are one of the earliest formulations of algebraic theory, dating to Lawvere's 1963 thesis [16], and they have been thoroughly studied since then; see [1]. In the setting of ∞-categories, Lawvere theories have been studied by just a few authors, notably by Cranch [6] and Gepner-Groth-Nikolaus [9], and in the prequel to this paper [5].

The literature in this area is sparse in part because Lurie's book Higher Algebra [17] founds the subject on operads, instead. That approach is now well-developed, and it has many benefits, but it suffers from the drawback of being pedagogically unwieldy. Even for those who are already invested in operads, like many homotopy theorists, there is a high barrier of entry: Even the definitions of fundamental objects like ∞-operads, commutative algebras, and symmetric monoidal ∞-categories presuppose a deep understanding of the quasicategory model.

On the other hand, there are surely those who would like to use this machinery without requiring operads at all.

In contrast, the Lawvere theoretic approach can be described in two sentences, and independent of our model of ∞-category:

Definition 2.1. *A* Lawvere theory *is an ∞-category $\mathcal{L}$ which is closed under finite products and generated under finite products by a single distinguished object* 1. *An algebra or* model *of $\mathcal{L}$ in $\mathcal{C}$ is a functor $\mathcal{L} \to \mathcal{C}$ which preserves finite products.*

Typically, we want to take $\mathcal{C}$ to be an ∞-category which is presentable and cartesian closed, such as Set, Top, or Cat_∞, but there is no such requirement. By default, we take $\mathcal{C}$ to be the ∞-category Top of CW complexes, or homotopy types, which is the initial object among ∞-categories which are presentable and closed symmetric monoidal. We write

$$\mathrm{Mdl}_{\mathcal{L}} = \mathrm{Fun}^{\times}(\mathcal{L}, \mathrm{Top}).$$

Example 2.2. *Let Fin denote the category of finite sets. Then Fin^{op} is a Lawvere theory, with product given by disjoint union of sets, and evaluation at the singleton*

$$Mdl_{Fin^{op}} \to Top$$

is an equivalence of ∞-categories. We say Fin^{op} is the trivial Lawvere theory.

Example 2.3. *Let $Burn^{eff}$ denote the effective Burnside 2-category, whose objects are finite sets. For finite sets X, Y, the groupoid of morphisms from X to Y is the groupoid of span diagrams $X \leftarrow T \to Y$, and composition is via pullback. Then $Burn^{eff}$ is a Lawvere theory, with product given by disjoint union of sets.*

If $f : Burn^{\mathit{eff}} \to Top$ is a model, then the spans $0 \overset{=}{\longleftarrow} 0 \to 1$, respectively $2 \overset{=}{\longleftarrow} 2 \to 1$ endow $f(1)$ with a distinguished point, respectively a binary operation $f(1) \times f(1) \to f(1)$. Composition in $Burn^{\mathit{eff}}$ precisely enforces the structure of a commutative (or $\mathbb{E}_\infty$) monoid on $f(1)$, and evaluation at 1

$$Mdl_{Burn^{\mathit{eff}}} \to CMon_\infty$$

is an equivalence of ∞-categories. See [5] Remark 3.6.

We say $Burn^{\mathit{eff}}$ is the commutative (or $\mathbb{E}_\infty$) Lawvere theory.

In fact, for any cartesian monoidal $\mathcal{C}^\times$, commutative monoids in $\mathcal{C}$ are equivalent to models of $\mathrm{Burn}^{\mathrm{eff}}$ in $\mathcal{C}$ ([5] 3.6). Applying this to [17] 2.4.2.4:

Example 2.4. *A symmetric monoidal ∞-category may be regarded as a functor $Burn^{\mathit{eff}} \to Cat_\infty$ which preserves finite products. In contrast to Lurie's definition of a symmetric monoidal ∞-category, this construction is elementary: In order to understand it, we need only understand how to take finite products in an ∞-category (they are the same as products in the homotopy category!), and how to regard a 2-category such as $Burn^{\mathit{eff}}$ as an ∞-category.*

More generally, given any ∞-operad $\mathcal{O}$, there is a Lawvere theory $\mathcal{L}$ such that

$$\mathrm{Alg}_{\mathcal{O}}(\mathcal{C}^\times) \cong \mathrm{Mdl}_{\mathcal{L}}(\mathcal{C}^\times)$$

for any cartesian monoidal $\mathcal{C}^\times$. The Lawvere theory can even be more-or-less explicitly described in terms of $\mathcal{O}$ ([5] 3.16).

It may appear that Lawvere theories are less general than operads because they apply only to cartesian monoidal ∞-categories. This is apparently a significant obstacle: a major application of operads is to understanding multiplicative structure on rings. For example, a ring spectrum is an algebra in spectra under *smash product.* However, this problem can be easily overcome, as long as we restrict attention to *connective* ring spectra:

Example 2.5. *For any connective ring spectrum R, there are Lawvere theories whose models are equivalent to $Mod_R^{\geqslant 0}$, $Alg_R^{\geqslant 0}$, and $CAlg_R^{\geqslant 0}$.*

This example and others like it follow from a result of Gepner, Groth, and Nikolaus [9] which describes *exactly* which ∞-categories are equivalent to $\mathrm{Mdl}_{\mathcal{L}}$ for some Lawvere theory $\mathcal{L}$ (Theorem 2.6 below).

Before stating their result, we recall the theory of presentable ∞-categories. By the adjoint functor theorem ([18] 5.5.2.9), a functor $\mathcal{C} \to \mathcal{D}$ between presentable ∞-categories preserves small colimits if and only if it has a right adjoint. We write Pr^L for the ∞-category of presentable ∞-categories along with these *left adjoint* functors.

If $\mathcal{C} \in \mathrm{Pr}^L$, then since Top is freely generated by one object under colimits, the following data are equivalent:

- an object $X \in \mathcal{C}$ (we say $\mathcal{C}$ is *pointed*);
- a left adjoint functor $L = - \otimes X : \mathrm{Top} \to \mathcal{C}$ (we say $L(S)$ is the *free* object on S);
- a right adjoint functor $R = \mathrm{Map}(X, -) : \mathcal{C} \to \mathrm{Top}$ (we say $R(Y)$ is the *underlying space* of Y).

We will denote by Pr^L_* the ∞-category of these pointed presentable ∞-categories, along with left adjoint basepoint-preserving functors. (Formally, $\mathrm{Pr}^L_* \cong \mathrm{Pr}^L_{\mathrm{Top}/}$.)

If $\mathcal{L}$ is a Lawvere theory, generated by the distinguished object 1, then we regard $\mathrm{Mdl}_{\mathcal{L}}$ as canonically pointed by the right adjoint forgetful functor

$$\text{evaluate at } 1 : \mathrm{Mdl}_{\mathcal{L}} \to \mathrm{Top}.$$

By the Yoneda lemma, the corresponding basepoint is the model

$$\mathrm{Map}(1, -) : \mathcal{L} \to \mathrm{Top}.$$

Theorem 2.6 (Gepner-Groth-Nikolaus [9] Theorem B.7)**.** *A pointed presentable ∞-category $\mathcal{C}$ is equivalent to $Mdl_{\mathcal{L}}$ for some Lawvere theory $\mathcal{L}$ if and only if the forgetful functor $\mathcal{C} \to Top$ is conservative and preserves geometric realizations.*

2.2 Reconstructing Lawvere theories from their models

We have just seen that many ∞-categories $\mathcal{M}$ can be described as models over a Lawvere theory (roughly, those which are presentable and *algebraic* in nature). We may now ask: is that Lawvere theory unique, and to what extent can it be recovered from $\mathcal{M}$?

If $\mathcal{L}$ admits finite products, then $\mathrm{Mdl}_{\mathcal{L}} = \mathrm{Fun}^{\times}(\mathcal{L}, \mathrm{Top})$ is a full subcategory, by definition, of the ∞-category of presheaves, $\mathcal{P}(\mathcal{L}^{\mathrm{op}}) = \mathrm{Fun}(\mathcal{L}, \mathrm{Top})$. Moreover, every representable presheaf $\mathrm{Map}(X, -) : \mathcal{L} \to \mathrm{Top}$ preserves any limits that exist in $\mathcal{L}$, and therefore is in $\mathrm{Mdl}_{\mathcal{L}}$.

By the Yoneda lemma, then $\mathcal{L}^{\mathrm{op}}$ is a full subcategory of $\mathrm{Mdl}_{\mathcal{L}}$. (We called this a *cartesian monoidal Yoneda lemma* in [5] 3.7.)

If $\mathcal{L}$ is a Lawvere theory, we can explicitly identify $\mathcal{L}^{\text{op}}$ as a subcategory of $\text{Mdl}_{\mathcal{L}}$: The embedding $\mathcal{L}^{\text{op}} \subseteq \text{Mdl}_{\mathcal{L}}$ identifies $1^{\amalg n} \in \mathcal{L}^{\text{op}}$ with $\mathbb{I}^{\amalg n} \in \text{Mdl}_{\mathcal{L}}$. Here $\mathbb{I}$ is the distinguished object of $\text{Mdl}_{\mathcal{L}}$, so that $\mathbb{I}^{\amalg n}$ can also be regarded as the free model on n generators. In conclusion:

Proposition 2.7. *Suppose $\mathcal{M}$ is a pointed, presentable ∞-category with distinguished object $\mathbb{I}$. Let $\mathcal{M}_{fgf}$ be the full subcategory of finitely generated free objects; that is, those of the form $\mathbb{I}^{\amalg n}$ for integers $n \geqslant 0$. If $\mathcal{M} \cong Mdl_{\mathcal{L}}$ for some Lawvere theory $\mathcal{L}$, then $\mathcal{L} \cong \mathcal{M}^{op}_{fgf}$.*

Theoretically, this proposition combined with Theorem 2.6 allow us to describe the Lawvere theories associated to any operad, modules over a ring, algebras over a ring, etc. – *provided we already understand the ∞-category of models.*

However, we may instead seek to describe the Lawvere theory first, and use it to construct some new ∞-category of models. (For example, we might want to do this for pedagogical purposes as in Example 2.4.)

Principle 2.8. *Lawvere theories often have combinatorial descriptions, in which their objects are finite sets, morphisms are given by diagrams of finite sets, and products are given by disjoint union.*

Example 2.9. *The commutative Lawvere theory $Burn^{eff}$ is equivalent to the 2-category of spans of finite sets.*

We may revisit this principle in a future paper on *combinatorial Lawvere theories*; for now, we will not emphasize it.

2.3 Main theorem

We have described how to pass back and forth between a Lawvere theory and its ∞-category of models. We will show this relation is exceptionally robust.

Let Lwv denote the ∞-category whose objects are Lawvere theories and morphisms are functors which preserve finite products and the distinguished object[1].

Theorem 2.10. *There is an adjunction*

$$Mdl : Lwv \leftrightarrows Pr^L_* : (-)^{op}_{fgf},$$

and the left adjoint Mdl is fully faithful.

[1] Formally, Lwv is a full subcategory of pointed cartesian monoidal ∞-categories.

In other words, Lwv is a *colocalization* of Pr^L_*. Theorem 2.6 described explicitly *which* colocalization by providing a testable criterion to determine the essential image of Lwv in Pr^L_*.

Proof. Call $L = \mathrm{Mdl}$ and $R = (-)^{\mathrm{op}}_{\mathrm{fgf}}$, suggestive of left and right adjoints. The composition RL is equivalent to the identity by Proposition 2.7. Therefore, applying R induces a map of spaces

$$\mathrm{Fun}^L_*(L(\mathcal{L}),\mathcal{C}) \xrightarrow{R_*} \mathrm{Fun}^\times_*(\mathcal{L},R(\mathcal{C})),$$

natural in both $\mathcal{L} \in \mathrm{Lwv}$ and $\mathcal{C} \in \mathrm{Pr}^L_*$, and this R_* is a natural isomorphism by [18] 5.3.6.10. Therefore, L and R are adjoint.

For $\mathcal{L},\mathcal{K} \in \mathrm{Lwv}$, applying L induces a map of spaces

$$\mathrm{Fun}^\times_*(\mathcal{L},\mathcal{K}) \xrightarrow{L_*} \mathrm{Fun}^L_*(L(\mathcal{L}),L(\mathcal{K})).$$

Since $RL \cong \mathrm{Id}$, R_*L_* is equivalent to the identity, and because R_* is an equivalence (as above), so is L_*. Therefore, L is fully faithful. □

Remark 2.11. *As seen in the proof, Theorem 2.10 combines two facts in one: the adjunction asserts [18] 5.3.6.10, that $Mdl_{\mathcal{L}}$ is the* free cocompletion *of $\mathcal{L}^{op}$ (regarding the latter as an ∞-category which already has finite coproducts).*

That the left adjoint is fully faithful asserts Proposition 2.7, that $\mathcal{L}^{op}$ is a full subcategory of $Mdl_{\mathcal{L}}$.

Remark 2.12. *More generally, if $CartMonCat_\infty$ denotes the ∞-category of ∞-categories which admit finite products (and functors which preserve them), then the same proof shows*

$$Mdl\colon CartMonCat_\infty \to Pr^L$$

is fully faithful. It also comes very close to being a left adjoint to the functor

$$(-)^{op} : Pr^L \to \widehat{CartMon}Cat_\infty,$$

with the following fatal obstacle: The domain of Mdl consists of small *cartesian monoidal ∞-categories, while the codomain of $(-)^{op}$ consists of* large *cartesian monoidal ∞-categories.*

This set-theoretic problem arises because of the definition of presentable ∞-categories: they are required to have a small set of generating objects, but these objects are not remembered as part of the data. Theorem 2.10 solves the problem by introducing a framing*; that is, by remembering these objects (or in this case, a single object).*

3 Algebra of Lawvere theories

In Section 3.1, we show that the functor $\mathrm{Mdl}_{\mathcal{L}} : \mathrm{Lwv} \to \mathrm{Pr}^L_*$ is symmetric monoidal. Then we study its behavior on commutative algebras in 3.2, constructing Day convolution products of models. Finally, we study its behavior on modules in Section 3.3, showing that many Lawvere theories have canonical enrichments. In fact, we provide evidence that $\mathcal{L}$-module Lawvere theories can be identified with $\mathrm{Mdl}_{\mathcal{L}}$-enriched Lawvere theories (Conjecture 3.12).

3.1 Kronecker products of Lawvere theories

The primary technical contribution of this paper is first to cast the relationship between Lawvere theories and their models as a colocalization (Theorem 2.10) and second that this colocalization is compatible with symmetric monoidal structures on Lwv (the Kronecker product) and Pr^L_* (Lurie's tensor product). We review each of these:

Remark 3.1 (Lurie's tensor product of presentable ∞-categories)**.** *There is a closed symmetric monoidal tensor product on Pr^L with the following universal property: left adjoint functors $\mathcal{C} \otimes \mathcal{D} \to \mathcal{E}$ can be identified with functors $\mathcal{C} \times \mathcal{D} \to \mathcal{E}$ which preserve small colimits in each variable separately. This is constructed by Lurie in [17] 4.8.1, and the unit is Top.*

If $\mathcal{V}$ is presentable, to endow $\mathcal{V}$ with the structure of a commutative algebra in $Pr^{L,\otimes}$ is precisely to endow $\mathcal{V}$ with its own closed symmetric monoidal structure[2]*.*

If $\mathcal{C}, \mathcal{D}$ are two *pointed* presentable ∞-categories, $\mathcal{C} \otimes \mathcal{D}$ is also canonically pointed (for example, by the free functor $\mathrm{Top} \cong \mathrm{Top} \otimes \mathrm{Top} \to \mathcal{C} \otimes \mathcal{D}$), so that Pr^L_* is also symmetric monoidal via Lurie's tensor product.

A commutative algebra object in Pr^L_* is a presentable ∞-category with a closed symmetric monoidal structure $\otimes$, pointed by the unit of $\otimes$.

Remark 3.2 (Kronecker tensor product of Lawvere theories)**.** *There is a closed symmetric monoidal tensor product of cartesian monoidal ∞-categories with the following universal property: functors $\mathcal{C} \otimes \mathcal{D} \to \mathcal{E}$ which preserve finite products can be identified with functors $\mathcal{C} \times \mathcal{D} \to \mathcal{E}$ which preserve finite products in each variable separately. This can be made precise in two equivalent ways:*

[2] a consequence of the adjoint functor theorem.

- *by Lurie's general framework of tensor products of categories ([17] 4.8.1);*
- *as in [5]: cartesian monoidal ∞-categories can be identified with modules over the commutative semiring category Fin^{op}, which admit a relative tensor product $\otimes_{Fin^{op}}$.*

If $\mathcal{C}, \mathcal{D}$ are Lawvere theories (that is, generated by a single object under $\times$), then $\mathcal{C} \otimes \mathcal{D}$ is also a Lawvere theory, so that Lwv inherits a symmetric monoidal operation $\otimes$ called Kronecker product*, and the unit is Fin^{op}. See [5] for details.*

For classical Lawvere theories, the Kronecker product goes back to Freyd [8], and it is also compatible with the Boardman-Vogt tensor product ([17] 2.2.5) of operads.

That is, if $\mathcal{L}_{\mathcal{O}}$ is the Lawvere theory associated to an operad $\mathcal{O}$, then $\mathcal{L}_{\mathcal{O}\otimes\mathcal{O}'} \cong \mathcal{L}_{\mathcal{O}} \otimes \mathcal{L}_{\mathcal{O}'}$, because the two sides have equivalent ∞-categories of models.

Theorem 3.3. *The functors*

$$Mdl\colon CartMonCat_{\infty} \to Pr^{L}$$

$$Mdl\colon Lwv \to Pr^{L}_{*}$$

are compatible with the symmetric monoidal structures of the last two remarks.

That is, Lwv is a *symmetric monoidal colocalization* of Pr^{L}_{*}.

In the next two sections, we will explore the consequences of this theorem when applied to (first) commutative algebras and (second) modules with respect to the two symmetric monoidal structures.

Proof. By [17] 4.8.1.8, the functor

$$\mathrm{Fun}^{\times}((-)^{\mathrm{op}}, \mathrm{Top}) : \mathrm{CocartMonCat}_{\infty} \to \mathrm{Pr}^{L}$$

is symmetric monoidal. Passing via the symmetric monoidal equivalence $(-)^{\mathrm{op}} : \mathrm{CartMonCat}_{\infty} \to \mathrm{CocartMonCat}_{\infty}$, we have the first part.

Taking $\mathbb{E}_0$-algebras (that is, pointed objects) on each side, we see that $\mathrm{Mdl} : \mathrm{CartMonCat}_{*} \to \mathrm{Pr}^{L}_{*}$ is symmetric monoidal, and this restricts to the symmetric monoidal full subcategory $\mathrm{Lwv} \subseteq \mathrm{CartMonCat}_{*}$. □

3.2 Algebra Lawvere theories and Day convolution

For suitable Lawvere theories $\mathcal{L}$, we can use Theorem 3.3 to construct tensor products of $\mathcal{L}$-models. A commutative algebra structure on $\mathcal{L} \in \mathrm{Lwv}^{\otimes}$ amounts to a symmetric monoidal structure on $\mathcal{L}$ which preserves finite products independently in each variable, and such that the unit is the distinguished object 1.

Such Lawvere theories are sometimes called *commutative algebraic theories* in the classical literature [15].

Corollary 3.4. *If $\mathcal{L} \in CAlg(Lwv^{\otimes})$, then $Mdl_{\mathcal{L}}$ inherits a closed symmetric monoidal structure called* Day convolution*, with unit $\mathbb{I}$*[3]*.*

Conversely, if $Mdl_{\mathcal{L}}$ has a closed symmetric monoidal structure with unit $\mathbb{I}$, then $\mathcal{L}$ inherits a commutative algebra structure in Lwv.

Proof. Since Mdl is symmetric monoidal, it takes commutative algebras to commutative algebras. Therefore, if $\mathcal{L} \in \mathrm{CAlg}(\mathrm{Lwv}^{\otimes})$, then $\mathrm{Mdl}_{\mathcal{L}}$ has a commutative algebra structure in Pr^L_*, which is to say a closed symmetric monoidal structure with unit $\mathbb{I}$.

Conversely, the right adjoint to a symmetric monoidal functor is lax symmetric monoidal [10]. If $\mathrm{Mdl}_{\mathcal{L}}$ has a closed symmetric monoidal structure with unit $\mathbb{I}$, it is a commutative algebra in Pr^L_*, so $\mathcal{L} \in \mathrm{CAlg}(\mathrm{Lwv}^{\otimes})$. □

Example 3.5. *The effective Burnside 2-category is symmetric monoidal under cartesian product, which makes it a commutative algebraic theory. Therefore, Day convolution provides a closed symmetric monoidal* smash product *for $\mathbb{E}_\infty$-spaces.*

Remark 3.6 (Models in other ∞-categories)**.** *More generally, suppose $\mathcal{L}$ is a Lawvere theory, and $\mathcal{V}$ is a presentable ∞-category. By general theory of presentable ∞-categories,*

$$Fun^{\times}(\mathcal{L}, \mathcal{V}) \cong Fun^{\amalg}(\mathcal{L}^{op}, \mathcal{V}^{op}) \cong Fun^{L}(Mdl_{\mathcal{L}}, \mathcal{V}^{op}) \cong Fun^{R}(\mathcal{V}^{op}, Mdl_{\mathcal{L}}).$$

Lurie proves ([17] 4.8.1.17) for two presentable ∞-categories $\mathcal{C}$ and $\mathcal{D}$, that $\mathcal{C} \otimes \mathcal{D} \cong Fun^R(\mathcal{C}^{op}, \mathcal{D})$. Therefore, models of $\mathcal{L}$ in $\mathcal{V}$ can be identified with the tensor product

$$Mdl_{\mathcal{L}}(\mathcal{V}) \cong Mdl_{\mathcal{L}} \otimes \mathcal{V}.$$

This equivalence is due to [9] Proposition B.3.

[3] This is the Day convolution of Lurie [17] and Glasman [11].

In particular, if $\mathcal{L} \in CAlg(Lwv^{\otimes})$ and $\mathcal{V}$ has a closed symmetric monoidal structure, then $Mdl_{\mathcal{L}}(\mathcal{V})$ also has a closed symmetric monoidal structure[4] *(Day convolution).*

3.3 Module Lawvere theories and enrichment

Let $\mathcal{V}$ be presentable and closed symmetric monoidal; i.e., $\mathcal{V} \in \mathrm{CAlg}(\mathrm{Pr}^{L,\otimes})$. If $\mathcal{M} \in \mathrm{Pr}^L$ is a $\mathcal{V}$-module, and $X \in \mathcal{M}$, then $- \otimes X : \mathcal{V} \to \mathcal{M}$ has a right adjoint $\mathrm{Map}(X, -) : \mathcal{M} \to \mathcal{V}$ which makes $\mathcal{M}$ naturally $\mathcal{V}$-enriched. Gepner and Haugseng have made this precise ([10] 7.4.13).

Conjecture 3.7. *Conversely, we may think of $\mathcal{V}$-modules in Pr^L as precisely those $\mathcal{V}$-enriched categories which are* presentable in an enriched sense. *As far as the author is aware, the notion of 'presentable in an enriched sense' has not yet been made rigorous for ∞-categories, but this is a philosophy already familiar to some.*

We have a second corollary of Theorem 3.3:

Corollary 3.8. *If $\mathcal{L}$ is a commutative semiring ∞-category whose additive structure is cartesian monoidal, and $\mathcal{M}$ is an $\mathcal{L}$-module, then $\mathcal{M}$ is naturally enriched in $Mdl_{\mathcal{L}}$.*

Proof. If $\mathcal{M}$ is an $\mathcal{L}$-module in Lwv, then $\mathrm{Mdl}_{\mathcal{M}}$ is a $\mathrm{Mdl}_{\mathcal{L}}$-model in Pr^L. As above, $\mathrm{Mdl}_{\mathcal{M}}$ inherits a canonical $\mathrm{Mdl}_{\mathcal{L}}$-enrichment, which restricts to a $\mathrm{Mdl}_{\mathcal{L}}$-enrichment on the full subcategory $\mathcal{M} \subseteq \mathrm{Mdl}^{\mathrm{op}}_{\mathcal{M}}$. □

Example 3.9. *If $\mathcal{L} = Burn^{\mathit{eff}}$ is the Lawvere theory for $\mathbb{E}_\infty$-spaces, then $Burn^{\mathit{eff}}$-modules can be identified with semiadditive ∞-categories ([5] Theorem 1.2). By corollary 3.8, any semiadditive ∞-category is naturally enriched in $\mathbb{E}_\infty$-spaces.*

This is the homotopical analogue of a classical fact: semiadditive categories are naturally enriched in commutative monoids.

Conversely, if $\mathcal{L} \in \mathrm{CAlg}(\mathrm{Lwv}^{\otimes})$, then any Lawvere theory enriched in $\mathrm{Mdl}_{\mathcal{L}}$ is naturally tensored over $\mathcal{L}$:

Definition 3.10. *If $\mathcal{V}$ is presentable and closed symmetric monoidal, a $\mathcal{V}$-*enriched Lawvere theory *is a $\mathcal{V}$-enriched category for which the underlying ∞-category is a Lawvere theory.*

[4] because tensor products of commutative algebras are commutative algebras

Theorem 3.11. *Suppose $\mathcal{L} \in CAlg(Lwv^{\otimes})$, and $\mathcal{K}^{enr}$ is a $Mdl_{\mathcal{L}}$-enriched Lawvere theory with underlying ∞-category $\mathcal{K}$. For any $X \in \mathcal{L}$, $K \in \mathcal{K}$, there is some object $X \otimes K \in \mathcal{K}$ with a natural isomorphism*

$$Map^{enr}(-, K)(X) \cong Map(-, X \otimes K),$$

and this $\otimes$ arises from a morphism of Lawvere theories

$$\mathcal{L} \otimes \mathcal{K} \to \mathcal{K}.$$

Proof. Because $\mathcal{K}^{\text{enr}}$ is $\text{Mdl}_{\mathcal{L}}$-enriched, there is a functor

$$\text{Map}^{\text{enr}}_{\mathcal{K}}(-,-) : \mathcal{K}^{\text{op}} \times \mathcal{K} \to \text{Mdl}_{\mathcal{L}},$$

and the composite with the forgetful functor $\text{ev}_1 : \text{Mdl}_{\mathcal{L}} \to \text{Top}$ is the ordinary mapping space $\text{Map}_{\mathcal{K}}$. In particular, that composite preserves finite products in the second variable.

By Theorem 2.6, the forgetful functor ev_1 (evaluation at 1) is conservative. It also preserves finite products because it is a right adjoint. Therefore, the functor $\text{Map}^{\text{enr}}(-,-)$ preserves finite products in the second variable.

It follows that the adjoint $\mathcal{L} \times \mathcal{K} \to \text{Fun}(\mathcal{K}^{\text{op}}, \text{Top})$ preserves finite products independently in each variable, thereby inducing a functor

$$\phi : \mathcal{L} \otimes \mathcal{K} \to \text{Fun}(\mathcal{K}^{\text{op}}, \text{Top})$$

which preserves finite products. By construction, $\phi(X \otimes K) \cong \text{Map}^{\text{enr}}(-, K)(X)$. In particular, if n denotes $1^{\times n}$ in a Lawvere theory,

$$\phi(n) \cong \text{Map}^{\text{enr}}_{\mathcal{K}}(-, n)(1) \cong \text{ev}_1 \circ \text{Map}^{\text{enr}}_{\mathcal{K}}(-, n) \cong \text{Map}_{\mathcal{K}}(-, n),$$

so that ϕ factors through the full subcategory $\mathcal{K} \subseteq \text{Fun}(\mathcal{K}^{\text{op}}, \text{Top})$. This completes the proof. □

Together, Corollary 3.8 and Theorem 3.11 are suggestive of:

Conjecture 3.12. *If $\mathcal{L} \in CAlg(Lwv^{\otimes})$, the ∞-categories of $Mdl_{\mathcal{L}}$-enriched Lawvere theories and $\mathcal{L}$-module Lawvere theories are equivalent.*

4 Additive Lawvere theories

Suppose that $\mathcal{L}$ is semiadditive as an ∞-category; essentially, finite products are also finite coproducts. By Example 3.9, $\mathcal{L}$ is enriched in $\mathbb{E}_\infty$-spaces.

Therefore, End(1) is an $\mathbb{E}_1$-semiring space, where 1 is the distinguished object of $\mathcal{L}$, and there is a functor

$$\text{End}(1) : \text{SemiaddLwv} \to \mathbb{E}_1\text{Semiring}.$$

Proposition 4.1. *This functor is an equivalence of symmetric monoidal ∞-categories, identifying a semiring R with the Lawvere theory modeling Mod_R.*

The proposition should not be surprising; if $\mathcal{L}$ is semiadditive, then we know

$$\mathrm{Map}(1^{\times m}, 1^{\times n}) \cong \mathrm{Map}(1,1)^{\times mn},$$

which depends only on the semiring End(1). Hence, we expect End(1) to encode all the data of the Lawvere theory.

Proof. If R is an $\mathbb{E}_1$-semiring space, write Burn_R for the Lawvere theory whose models are Burn_R, which exists by Theorem 2.6. We claim that any semiadditive Lawvere theory $\mathcal{L}$ is equivalent to $\mathrm{Burn}_{\mathrm{End}(1)}$.

Fix $\mathcal{L}$ and write $R = \mathrm{End}(1)$. Since R acts on $\mathrm{Map}_{\mathrm{Mdl}_{\mathcal{L}}}(1, -)$, we have

$$\mathrm{Map}(1, -) : \mathrm{Mdl}_{\mathcal{L}} \to \mathrm{Mod}_R$$

which maps 1 to 1, preserves finite direct sums, and therefore restricts to a map of Lawvere theories $\alpha : \mathcal{L} \to \mathrm{Burn}_R$.

Certainly α is essentially surjective, so we show it is full and faithful. Given objects $m = 1^{\amalg m}$ and $n = 1^{\amalg n}$ in $\mathcal{L}$, we wish to prove that

$$\mathrm{Map}_{\mathcal{L}}(m, n) \xrightarrow{\alpha_*} \mathrm{Map}_{\mathrm{Burn}_R}(m, n)$$

is an equivalence. When $m = n = 1$, this is true by construction. Otherwise, since $\mathcal{L}$ and Burn_R are semiadditive, we know on both sides that $\mathrm{Map}(m, n) \cong R^{mn}$, so α_* is an equivalence.

Therefore, every semiadditive Lawvere theory is of the form Burn_R. By Theorem 2.10, SemiaddLwv is the symmetric monoidal full subcategory of Pr^L_* spanned by the ∞-categories Mod_R, as R ranges over $\mathbb{E}_1$-semiring spaces. On the other hand, the functor

$$\mathbb{E}_1\mathrm{Semiring} \to \mathrm{Pr}^L_*$$

which sends R to Mod_R is also fully faithful and symmetric monoidal, which completes the proof. □

From the proposition, we deduce an important philosophy: Lawvere theories are more like *algebraic* objects than *categorical* objects. We might even regard them as generalized (non-additive) rings.

Finally, we apply Theorem 2.6 to deduce:

Corollary 4.2. *If $\mathcal{M}$ is presentable and semiadditive, and $\mathcal{M} \to Top$ is a right adjoint functor which is conservative and preserves geometric realizations, then $\mathcal{M} \cong Mod_R$ for some $\mathbb{E}_1$-semiring space R, compatibly with the forgetful functor $Mod_R \to Top$.*

5 Applications of Lawvere theories

We will end with two applications. The first is to the commutative algebra of semiring ∞-categories, and the second to equivariant homotopy theory.

5.1 Commutative algebra of categories

Definition 5.1. *An ∞-category is* additive *if it is semiadditive and each mapping $\mathbb{E}_\infty$-space is grouplike.*

That is, an additive ∞-category is semiadditive and enriched in $\mathrm{Ab}_\infty \cong \mathrm{Sp}_{\geqslant 0}$ (grouplike $\mathbb{E}_\infty$-spaces, or connective spectra). As promised in [5], we prove:

Theorem 5.2. *The Burnside ∞-category Burn (which is the Lawvere theory for connective spectra) is a commutative semiring ∞-category, and Mod_{Burn}, $AddCat_\infty$ are equivalent full subcategories of $SymMonCat_\infty$.*

Remark 5.3. *Note that Theorem 5.2 implies Conjecture 3.12 for the specific Lawvere theory Burn. Indeed, $Mdl_{Burn} \cong Sp_{\geqslant 0}$-enriched Lawvere theories are additive Lawvere theories, so the conjecture asserts that a Lawvere theory is a Burn-module if and only if it is additive.*

Proof. As in [9], the product map $\mathrm{Sp}_{\geqslant 0} \otimes \mathrm{Sp}_{\geqslant 0} \to \mathrm{Sp}_{\geqslant 0}$ is an equivalence. Identifying $\mathrm{Sp}_{\geqslant 0}$ with $\mathrm{Mdl}_{\mathrm{Burn}}$ and noting that Mdl is symmetric monoidal, we find that $\mathrm{Burn} \otimes \mathrm{Burn} \to \mathrm{Burn}$ is also an equivalence.

This means that the forgetful functor $\mathrm{Mod}_{\mathrm{Burn}} \to \mathrm{SymMonCat}_\infty$ is fully faithful. We need to show that a symmetric monoidal ∞-category admits the structure of a Burn-module if and only if it is additive.

First, if $\mathcal{C}$ is a Burn-module, it is a $\mathrm{Burn}^{\mathrm{eff}}$-module and therefore semiadditive (Example 3.9), but it is also $\mathrm{Mdl}_{\mathrm{Burn}}$-enriched by Corollary 3.8. By definition, it is therefore additive.

Conversely, suppose $\mathcal{C}$ is additive, so that $\mathcal{P}(\mathcal{C}) = \mathrm{Fun}(\mathcal{C}^{\mathrm{op}}, \mathrm{Top})$ is additive and presentable. By [9], $\mathcal{P}(\mathcal{C})$ is a $\mathrm{Mdl}_{\mathrm{Burn}}$-module in Pr^L, and therefore also in $\mathrm{SymMonCat}_\infty$, because the functor $\mathrm{Pr}^L \to \mathrm{SymMonCat}_\infty$ is lax symmetric monoidal which forgets everything except the cocartesian monoidal structure.

The embedding $\mathrm{Burn} \cong \mathrm{Burn}^{\mathrm{op}} \subseteq \mathrm{Mdl}_{\mathrm{Burn}}$ respects both symmetric monoidal structures ($\oplus$ and $\otimes$), so $\mathcal{P}(\mathcal{C})$ is a Burn-module, as a cocartesian monoidal ∞-category. Moreover, the full subcategory $\mathcal{C} \in \mathcal{P}(\mathcal{C})$ is closed under direct sum, so it inherits a Burn-module structure. This completes the proof. □

5.2 Equivariant homotopy theory

Throughout this section, G is a finite group. We write Fin_G for the category of finite G-sets, and Burn_G for the associated ∞-category of virtual spans, often referred to as the *Burnside ∞-category* without mention of the particular group. See [3] for more on this.

All group actions will be on the right.

There are two classical model categories of equivariant G-spaces: the 'naive' model structure has weak equivalences those maps which are weak equivalences of the underlying space. The corresponding ∞-category is $\mathrm{Fun}(BG, \mathrm{Top})$, because equivalences in a functor ∞-category are likewise checked objectwise, and BG has only one object (up to equivalence).

On the other hand, the 'genuine' model structure has weak equivalences those maps which have inverses up to homotopy. This model category corresponds to an ∞-category Top_G which is certainly not equivalent to $\mathrm{Fun}(BG, \mathrm{Top})$! For example, the map $EG \to *$ is an equivalence in the former but not in the latter model structure.

For spectra as well, there is a distinction between $\mathrm{Fun}(BG, \mathrm{Sp})$ and the ∞-category Sp_G of genuine equivariant spectra. Consult [12] for a classical survey.

We might ask how to describe Top_G and Sp_G in higher categorical terms. For this, we have the two theorems:

- (Elmendorf's Theorem: [7] Theorem 1) $\mathrm{Top}_G \cong \mathrm{Mdl}(\mathrm{Fin}_G^{\mathrm{op}})$;
- (Guillou-May's Theorem: [13] Theorem 0.1, [3] Example B.6) $\mathrm{Sp}_G^{\geqslant 0} \cong \mathrm{Mdl}(\mathrm{Burn}_G)$.

Recall that we have used the notation $\mathrm{Mdl}(\mathcal{L}) = \mathrm{Fun}^{\times}(\mathcal{L}, \mathrm{Top})$ whenever $\mathcal{L}$ admits finite products, even if it is not a Lawvere theory. However, Fin_G and $\mathrm{Burn}_G^{\mathrm{eff}}$ are not far from being Lawvere theories: although they do not have single generating objects, they are generated freely by the set of orbits G/H, as H ranges over subgroups of G.

We call them *colored Lawvere theories*, with set of colors $\{G/H\}$, or *equivariant Lawvere theories*, because they admit essentially surjective, product-preserving maps from the groupoid of finite G-sets, $\mathrm{Fin}_G^{\mathrm{iso}}$.

Remark 5.4. *The word 'genuine', used to describe equivariant spaces and spectra, can be misleading. Frequently, group actions on spectra arise via abstract homotopy-theoretic means, such as when the spectra themselves are algebraic in nature (as in chromatic homotopy theory). In these cases, we typically do not expect 'genuine' equivariant structures.*

However, when our spaces or spectra arise geometrically out of point-set constructions, group actions will be 'genuine'. This is because we can pass through the model category of genuine equivariant objects, on our way to the abstract ∞-categories Top_G and Sp_G.

It would almost be better to regard the 'genuine' actions as 'geometric', and the 'naive' actions as 'homotopical'.

The theorems of Elmendorf and Guillou-May may be combined with Corollary 3.8 as follows:

Corollary 5.5. *Regarding Fin_G and $Burn_G^{eff}$ as commutative semiring ∞-categories, any Fin_G-module is naturally enriched in genuine G-spaces, and any $Burn_G^{eff}$-module is naturally enriched in (connective) genuine G-spectra.*

More generally, suppose we have some algebraic structure, whose homotopical instances form an ∞-category $\mathcal{C}$. For example, $\mathcal{C} = \mathrm{Sp}_{\geqslant 0}$ corresponds to the structure 'abelian group'. We might ask: what kind of structure does a *genuine* equivariant G-object of $\mathcal{C}$ have?

This is a question which is not entirely idle. Following Remark 5.4, if an object of $\mathcal{C}$ has an action of G at some sufficiently concrete point-set level, we might expect *additional structure* to carry over to the ∞-category $\mathcal{C}$, beyond a naive G-action.

By analogy with the theorems of Elmendorf and Guillou-May, we propose addressing this question via a 3-step procedure:

1. check whether $\mathcal{C}$ is of the form $\mathrm{Mdl}_{\mathcal{L}}$ for some Lawvere theory $\mathcal{L}$ (possibly by means of Theorem 2.6);

2. check whether $\mathcal{L}$ can be described combinatorially, by applying some construction $\mathcal{M}$ to Fin (as in Principle 2.8);

3. $\mathrm{Fun}^{\times}(\mathcal{M}(\mathrm{Fin}_G), \mathcal{C})$ is a candidate for genuine G-objects of $\mathcal{C}$.

Example 5.6. *When $\mathcal{C} = Top$, $\mathcal{L} = Fin^{op}$ and the combinatorial construction $\mathcal{M}$ is the opposite category construction, so that (3) is Elmendorf's Theorem.*

When $\mathcal{C} = Sp_{\geqslant 0}$, $\mathcal{L} = Burn$ and the combinatorial construction $\mathcal{M}$ is the virtual span construction, so that (3) is Guillou-May's Theorem.

One goal is to use this strategy to understand equivariant $\mathbb{E}_\infty$-ring spectra via the Lawvere theory of *bispans* of finite G-sets, by analogy with the construction of Tambara functors [19].

We hope to address these problems in a sequel, in which we will discuss combinatorial constructions of Lawvere theories (as in Principle 2.8).

References

[1] J. Adámek, J. Rosicky, and E.M. Vitale. Algebraic Theories – a Categorical Introduction to General Algebra. Cambridge UP (2010).

[2] C. Barwick, E. Dotto, S. Glasman, D. Nardin, and J. Shah. Equivariant higher category theory and equivariant higher algebra. Preprint.

[3] C. Barwick. Spectral Mackey functors and equivariant algebraic K-theory (I). Adv. in Math., 304(2): 646-727 (2017).

[4] J. Berman. Categorified Algebra and Equivariant Homotopy Theory. Thesis, University of Virginia. arXiv: 1805.08745 (2018).

[5] J. Berman. On the Commutative Algebra of Categories. Algebr. Geom. Topol. 18: 2963-3012 (2018).

[6] J. Cranch. Algebraic theories and $(\infty, 1)$-categories. PhD Thesis. arXiv: 1011.3243 (2010).

[7] A. Elmendorf. Systems of fixed point sets. Trans. Amer. Math. Soc., 277(1): 275-284 (1983).

[8] P. Freyd. Algebra valued functors in general and tensor products in particular. Colloq. Math. 14: 89-106 (1966).

[9] D. Gepner, M. Groth, and T. Nikolaus. Universality of multiplicative infinite loop space machines. Algebr. Geom. Topol. 15: 3107-3153 (2015).

[10] D. Gepner and R. Haugseng. Enriched ∞-categories via non-symmetric ∞-operads. Adv. in Math. 279: 575-716, 2015.

[11] S. Glasman. Day convolution for ∞-categories. arXiv: 1308.4940, 2015. Preprint.

[12] J.P.C. Greenlees and J.P. May. Equivariant stable homotopy theory. Handbook of algebraic topology, North-Holland, Amsterdam: 277-323 (1995).

[13] B. Guillou and J.P. May. Models of G-spectra as presheaves of spectra. arXiv: 1110.3571 (2013). Preprint.

[14] R. Haugseng. Iterated spans and "classical" topological field theories. Mathematische Zeitschrift 289(3): 1427-1488 (2018).

[15] W. Keigher. Symmetric monoidal closed categories generated by commutative adjoint monads. Cahiers de Topologie et Géométrie Différentielle Catégoriques 19(3):269-293 (1978).

[16] W.F. Lawvere. Functorial semantics of algebraic theories. Reports of the Midwest Category Seminar II: 41-61 (1968).

[17] J. Lurie. Higher Algebra. http://www.math.harvard.edu/∼lurie/ (2011). Preprint.

[18] J. Lurie. Higher Topos Theory. Annals of Mathematics Studies, 170, Princeton University Press. Princeton, NJ (2009).

[19] N. Strickland. Tambara functors. arXiv: 1205:2516 (2012). Preprint.

LF: a Foundational Higher-Order Logic

Zachary Goodsell and Juhani Yli-Vakkuri

January 23, 2024

1 Introduction

This paper presents[1] a new system of logic, LF, that is intended to be used as the foundation of the formalization of science. That is, deductive validity according to LF[2] is to be used as the criterion for assessing what follows from the verdicts, hypotheses, or conjectures of any science. In work currently in progress, we argue for the unique suitability of LF for the formalization of logic, mathematics, syntax, and semantics. The present document specifies the language and rules of LF, lays out some key notational conventions, and states some basic technical facts about the system.

LF is a system of higher-order logic based on that of Church (1940), and is closely related to the now-standard system of Henkin (1950). LF improves on these systems by being *intensional*, like Henkin's system, but not *extensional*, unlike Henkin's. An intensional system is one in which provably equivalent formulae may be substituted in any context. Intensionality suffices for the provability of all the equalities we need to be able to prove in mathematics, which we therefore also need to be able to prove throughout the rest of science, such as $1 = 0+1$. In Church's system, $1 = 0+1$ cannot be proved, provided that the numerals are defined, following *Principia Mathematica* (Whitehead and Russell 1910–1913), so as to be suitable for counting.

Henkin's system is an intensional extension of Church's system and is in that respect an improvement on it, but it incorporates the problematic assumption of *extensionality*, which says that the material equivalence of propositions (type t[3]) is sufficient for their identity. Extensionality is problematic for a variety of reasons,

[1] The system has already been discussed in the literature: see Williamson 2023: 220, which cites unpublished work by the authors. However, no explicit formulation of the system, *qua* formal system, has been published to date.

[2] And thus according to any useful conservative extensions thereof, of which we highlight LF_ι and LF_ε in this paper (§4).

[3] Or type o in Church's and Henkin's terms.

most obviously in applications of the theory of probability. Suppose that a fair coin is to be tossed, and let H and T be sentences that respectively formalize 'the coin lands heads' and 'the coin lands tails'. It would be normal to assume each of

$$\Pr(H \wedge T) = 0 \qquad \Pr(H) = 0.5 \qquad \Pr(H \vee T) = 1,$$

but this trio is simply inconsistent in an extensional system. The derivation of a contradiction from the trio in an extensional system requires no extralogical assumptions about probability function Pr or about anything else;[4] the point is that in an extensional system we can prove that there are only two propositions, so no function can give them more than two values.

LF also differs from the systems of Henkin and Church in its assumptions about infinity. LF includes only a rule of *potential* infinity, which says that, for each finite number n, the proposition that there are at least n things is not contradictory. A rule of *actual* infinity would by contrast assert that for each finite number n, there are *in fact* at least n things. Potential Infinity suffices for the provability of the *inequalities* we need to be able to prove in mathematics, which we therefore also need to be able to prove throughout the rest of science, such as $2 + 2 \neq 5$.

Church and Henkin include axioms of (actual) infinity in their systems, but only for individuals (type e[5]). For this reason, they cannot prove either the potential or actual infinity of propositions, and hence cannot prove $2 + 2 \neq 5$ on the interpretation of the numerals suitable for counting propositions, which is required, as we saw above, for elementary reasoning about probability. (Even worse, in Henkin's extensional system can prove $2 + 2 = 5$ when the numerals are so interpreted, but that system, as we saw, is also unsuitable for elementary reasoning about probability for an even more basic reason.)

LF is closely related to earlier intensional systems, most notably the system $\mathsf{GM} = \mathsf{ML_P} + \mathsf{C} + \mathsf{EC} + \mathsf{AC} + \mathsf{PE}$, which is the strongest system that can be assembled out of the components studied by Gallin (1975) and which is gestured at but not axiomatized by Montague (a similar idea in a more baroque form is presented by Church (1951) as "Alternative (2)"); a weakening of LF (with the rule of Potential Infinity restricted to type e) is a conservative extension of GM. Like GM, LF significantly improves on the work by Montague (e.g., 1970) that inspired Gallin's work in its conceptual clarity and economy of notation by eliminating the additional base type of 'indices'. LF is also considerably simpler than GM, in implementing intensionality with the simple rule of Intensionality (attributed, in essence, to Carnap 1942: 92 in Church 1943: 300), which permits one to extend a proof of the material

[4] We assume that, like the numerals, the real number terms are treated as abbreviations of some suitable purely logical terms, so that $0 \neq 0.5$, $0 \neq 1$, and $0.5 \neq 1$ all abbreviate theorems.

[5] Or type ι in Church's and Henkin's terms.

equivalence of propositions to a proof of their identity. Other intensional systems that avoid overcomplication by index types, such as Bacon and Dorr's (forthcoming) *Classicism*, are too weak in important respects that will be described in future work.

2 The formal system

2.1 Terms

We begin by introducing the language of LF. It is made up of *terms* that are classified by their *types*. Both the terms and the types by which they are classified are strings (of typographic characters), but for each type σ there is also the class of terms of that type, to which the string σ will also be used to refer—a tolerable ambiguity that would be eliminated in a fully formalized presentation of the system.

2.1.1 *Types*

There are two *base types*: e (the type of *individual terms*) and t (the type of *sentences* and other *formulae*). The remaining types are *functional types*: for any types σ, τ, $\langle\sigma\tau\rangle$ is a functional type (which may casually be referred to as 'the type of functions from σ to τ').

$$\frac{}{e \in \text{Type}} \qquad \frac{}{t \in \text{Type}} \qquad \frac{\sigma \in \text{Type} \qquad \tau \in \text{Type}}{\langle\sigma\tau\rangle \in \text{Type}} \tag{1}$$

2.1.2 *Variables*

The terms of the language include, for each type σ, an infinite stock of variables of that type, each of which is a string consisting of an (italic Roman) letter with the type σ as a subscript followed by zero or more primes:

$$\frac{\sigma \in \text{Type} \qquad \mathbf{a} \in \text{Letter}}{\mathbf{a}^{\sigma} \in \text{Var}} \quad \frac{\sigma \in \text{Type} \qquad \mathbf{a} \in \text{Letter}}{\mathbf{a}^{\sigma} : \sigma} \quad \frac{\mathbf{a} \in \text{Var}}{\mathbf{a}' \in \text{Var}} \quad \frac{\mathbf{a} \in \text{Var} \qquad \mathbf{a} : \sigma}{\mathbf{a}' : \sigma} \tag{2}$$

2.1.3 *Constants*

The *constants* of the language are the strings consisting of the standard inclusion symbol $\subseteq$ followed by a type σ as a subscript, which indicates the type of the constant according to the rule that the type of $\subseteq_\sigma$ is $\langle\sigma t\rangle\langle\sigma t\rangle t$:

$$\frac{\sigma \in \text{Type}}{\subseteq_\sigma\colon \langle\sigma t\rangle\langle\sigma t\rangle t} \tag{3}$$

$\subseteq_\sigma$ is pronounced "every", or, when written in infix notation (Section 2.2.2), "implies" or "is included in" (and variants).

2.1.4 Complex terms

All *complex terms* of the language are constructed in accordance with one of two formation rules: (function) application and (function) abstraction:

$$\frac{\mathbf{f} : \sigma\tau \qquad \mathbf{a} : \sigma}{(\mathbf{fa}) : \tau}\,\text{App} \qquad \frac{\mathbf{x} \in \text{Var} \qquad \mathbf{x} : \sigma \qquad \mathbf{A} : \tau}{(\lambda\mathbf{x}\,.\,\mathbf{A}) : \sigma\tau}\,\text{Abs} \tag{4}$$

2.1.5 β-equivalence

Terms of the forms $(\lambda\mathbf{x}\,.\,\mathbf{A})\mathbf{B}$ and $[\mathbf{B}/\mathbf{x}]\mathbf{A}$ are said to be *immediately β-equivalent*, and terms are said to be *β-equivalent* when one can be obtained from the other by any number of substitutions of immediately β-equivalent parts (each term counts as part of itself). We write $\mathbf{A} \sim_\beta \mathbf{B}$ when $\mathbf{A}$ and $\mathbf{B}$ are β-equivalent.

2.1.6 Expressions

An *expression* of the language is a term with no free variables.

2.1.7 Formulae and sentences

A *formula* is a term of type t. A *sentence* is an expression of type t.

2.2 Abbreviations and notational conventions

2.2.1 Omission of parentheses and type decorations

Parentheses and angle brackets ($\langle$, $\rangle$) may be omitted in accordance with the following conventions.

1. Variable binders (i.e., λ and other introduced variable binders; which are indicated by a **.** following the bound variable) take widest possible scope.

2. Boolean connectives ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) are written in infix notation and take next widest scope.

3. Other function terms written in infix notation take next greatest scope.

4. Functions written in prefix notation take smallest possible scope.

5. Functions written in infix notation associate to the right, when the conventions above do not say otherwise (we apply this convention only when it is clear that the functions in question provably associate). So, given all of the foregoing,

$$\lambda p \,\mathbf{.}\, f p \rightarrow q \tag{5}$$

omits parentheses which should be filled in as

$$(\lambda p \,\mathbf{.}\, ((f p) \rightarrow q)). \tag{6}$$

6. Angle brackets ($\langle$, $\rangle$) are omitted in accordance with the convention that complex types associate to the right. So, e.g., ttt abbreviates $\langle t\langle tt\rangle\rangle$.

Type decorations may be omitted when writing terms when they can be inferred from context, i.e., when there is a unique term that could result in the written string by deleting type decorations from any variables or constants, subject to the following conventions.

1. Letters that could be bound by the same variable binder must have the same type. E.g., $\lambda x^e \,\mathbf{.}\, x$ and $\lambda x \,\mathbf{.}\, x^e$ must be $\lambda x^e \,\mathbf{.}\, x^e$.

2. p and q always have type t unless otherwise decorated.

For example,

$$\lambda x^e \,\mathbf{.}\, f f x \tag{7}$$

abbreviates

$$\lambda x^e \,\mathbf{.}\, f^{ee} f^{ee} x^e. \tag{8}$$

2.2.2 *Infix Notation*

Introduced constants **f** of types that take at least two arguments, i.e., types of the form $\sigma\tau\rho$, may be written in the form

$$(\mathbf{a}\ \mathbf{f}\ \mathbf{b}) \ := \ ((\mathbf{fa})\mathbf{b}) \tag{9}$$

2.2.3 Metanotation

Bold symbols are used as metavariables. **P** and **Q** always range over formulae. The Greek letters σ, τ, and ρ are also metavariables, but always range over types.

The overset arrow $\vec{\cdot}$ is used atop a metavariable to construct a metavariable ranging over strings of variables or types, of arbitrary length. When this device is used to range over both a string of variables and a string of types, as in

$$\lambda X^{\vec{\sigma}t}\vec{z}\,\mathbf{.}\,\neg X\vec{z}, \tag{10}$$

the intended instances are those in which the variables $\vec{z}$ have the respective types $\vec{\sigma}$.

2.2.4 Basic definitions

$$\begin{align}
\top &:= (\lambda p\,\mathbf{.}\,p) \subseteq (\lambda p\,\mathbf{.}\,p) \tag{11}\\
\forall_\sigma &:= \lambda X^{\sigma t}\,\mathbf{.}\,((\lambda y^\sigma\,\mathbf{.}\,\top) \subseteq X) \tag{12}\\
\forall\mathbf{x}^\sigma\,\mathbf{.}\,\mathbf{P} &:= \forall_\sigma \lambda\mathbf{x}^\sigma\,\mathbf{.}\,\mathbf{P} \tag{13}\\
\bot &:= (\lambda p\,\mathbf{.}\,\top) \subseteq (\lambda p\,\mathbf{.}\,p) \tag{14}\\
\neg &:= \lambda p\,\mathbf{.}\,((\lambda r^t\,\mathbf{.}\,p) \subseteq (\lambda r^t\,\mathbf{.}\,\bot)) \tag{15}\\
\exists_\sigma &:= \lambda X^{\sigma t}\,\mathbf{.}\,\neg\forall y^\sigma\,\mathbf{.}\,\neg Xy \tag{16}\\
\exists\mathbf{x}^\sigma\,\mathbf{.}\,\mathbf{P} &:= \exists_\sigma \lambda\mathbf{x}^\sigma\,\mathbf{.}\,\mathbf{P} \tag{17}\\
\lambda\mathbf{x}\vec{\mathbf{y}}\,\mathbf{.}\,\mathbf{A} &:= \lambda\mathbf{x}\,\mathbf{.}\,\lambda\vec{\mathbf{y}}\,\mathbf{.}\,\mathbf{A} \tag{18}\\
\forall\mathbf{x}\vec{\mathbf{y}}\,\mathbf{.}\,\mathbf{P} &:= \forall\mathbf{x}\,\mathbf{.}\,\forall\vec{\mathbf{y}}\,\mathbf{.}\,\mathbf{P} \tag{19}\\
\exists\mathbf{x}\vec{\mathbf{y}}\,\mathbf{.}\,\mathbf{P} &:= \exists\mathbf{x}\,\mathbf{.}\,\forall\vec{\mathbf{y}}\,\mathbf{.}\,\mathbf{P} \tag{20}\\
\rightarrow &:= \lambda pq\,\mathbf{.}\,((\lambda r^t\,\mathbf{.}\,p) \subseteq (\lambda r^t\,\mathbf{.}\,q)) \tag{21}\\
\vee &:= \lambda pq\,\mathbf{.}\,(\neg p \rightarrow q) \tag{22}\\
\wedge &:= \lambda pq\,\mathbf{.}\,\neg(\neg p \vee \neg q) \tag{23}\\
\leftrightarrow &:= \lambda pq\,\mathbf{.}\,((p \rightarrow q) \wedge (q \rightarrow p)) \tag{24}\\
\equiv_{\vec{\sigma}t} &:= \lambda XY^{\vec{\sigma}t}\vec{z}\,\mathbf{.}\,(X\vec{z} \leftrightarrow Y\vec{z}) \tag{25}\\
=_\sigma &:= \lambda xy^\sigma\,\mathbf{.}\,((\lambda Z\,\mathbf{.}\,Zx) \subseteq (\lambda Z\,\mathbf{.}\,Zy)) \tag{26}\\
\neq_\sigma &:= \lambda xy\,\mathbf{.}\,\neg(x =_\sigma y) \tag{27}\\
\Box &:= =\top \tag{28}\\
\Diamond &:= \neq\bot \tag{29}\\
0_\sigma &:= \lambda X^{\sigma t}\,\mathbf{.}\,\neg\exists X \tag{30}\\
1_\sigma &:= \lambda X^{\sigma t}\,\mathbf{.}\,\exists y\,\mathbf{.}\,(Xy \wedge \forall z\,\mathbf{.}\,(Xz \rightarrow y = z)) \tag{31}
\end{align}$$

$$\setminus_{\vec{\sigma}t} \;:=\; \lambda XY^{\vec{\sigma}t}\vec{z}\,\mathbf{.}\,(X\vec{z} \wedge \neg Y\vec{z}) \tag{32}$$

$$+_{\sigma} \;:=\; \lambda mn^{\langle\sigma t\rangle t}X^{\sigma t}\,\mathbf{.}\,\exists Y\,\mathbf{.}\,(Y \subseteq X \wedge mY \wedge n(X \setminus Y)) \tag{33}$$

$$\mathbb{N}_{\sigma} \;:=\; \lambda n^{\langle\sigma t\rangle t}\forall X\,\mathbf{.}\,(X0 \rightarrow (X \subseteq \lambda y\,\mathbf{.}\,X(y+1)) \rightarrow Xn) \tag{34}$$

2.3 LF

LF is presented as a natural deduction system in sequent calculus notation. A *sequent* is a string consisting of a possibly empty list of formulae (separated by commas), followed by the symbol ⊢, and then a formula. The sequent

$$\Gamma \vdash \mathbf{P} \tag{35}$$

may be read "**P** has been proved (or derived) from the assumptions Γ".

A *rule* is written in the following form, where capital Greek letters range over lists of formulae (i.e., formulae separated by commas),

$$\frac{\Gamma_1 \vdash \mathbf{P_1} \qquad \ldots \qquad \Gamma_n \vdash \mathbf{P_n}}{\Delta \vdash \mathbf{Q}} \tag{36}$$

and it indicates that, when $\mathbf{P_i}$ is provable from Γ_i for each $i \in \{1, \ldots, n\}$, **Q** is provable from Δ.

LF is defined as a class of rules, namely the rules R.1 to R.4 and R.6 to R.9. A formula **P** is *provable* in LF from the assumptions Γ if and only if the sequent

$$\Gamma \vdash \mathbf{P} \tag{37}$$

is derivable by some application of these rules. A *theorem* of LF is a sentence that is provable from no assumptions (i.e., from the empty list of assumptions, in which case we write: ⊢ **P**).

R.1 Structural Rules

$$\frac{}{\Gamma, \mathbf{P} \vdash \mathbf{P}} \qquad \frac{\Gamma, \mathbf{P}, \mathbf{P} \vdash \mathbf{Q}}{\Gamma, \mathbf{P} \vdash \mathbf{Q}} \qquad \frac{\Gamma \vdash \mathbf{Q}}{\Gamma, \mathbf{P} \vdash \mathbf{Q}} \qquad \frac{\Gamma, \mathbf{P}, \mathbf{Q}, \Delta \vdash \mathbf{R}}{\Gamma, \mathbf{Q}, \mathbf{P}, \Delta \vdash \mathbf{R}} \qquad \frac{\Gamma \vdash \mathbf{P} \qquad \Delta, \mathbf{P} \vdash \mathbf{Q}}{\Gamma, \Delta \vdash \mathbf{Q}} \tag{38}$$

R.2 β-Equivalence

$$\frac{\Gamma \vdash \mathbf{P}}{\Gamma \vdash \mathbf{Q}} \tag{39}$$

where $\mathbf{P} \sim_\beta \mathbf{Q}$ (see §2.1.5).

R.3   Universal Instantiation

$$\frac{\Gamma \vdash \mathbf{F} \subseteq_\sigma \mathbf{G} \qquad \Gamma \vdash \mathbf{Fa}}{\Gamma \vdash \mathbf{Fb}} \tag{40}$$

R.4   Universal Generalization

$$\frac{\Gamma, \mathbf{Fx} \vdash \mathbf{Gx}}{\Gamma \vdash \mathbf{F} \subseteq_\sigma \mathbf{G}} \tag{41}$$

where $\mathbf{x}$ is a variable of type σ not free in $\mathbf{F}$, $\mathbf{G}$, or any formula in Γ.

R.5   Negation Elimination

$$\frac{\Gamma, \neg\mathbf{P} \vdash \mathbf{P}}{\Gamma \vdash \mathbf{P}} \tag{42}$$

R.6   Intensionality

$$\frac{\mathbf{P} \vdash \mathbf{Q} \qquad \mathbf{Q} \vdash \mathbf{P}}{\vdash \mathbf{P} = \mathbf{Q}} \tag{43}$$

R.7   Function Extensionality

$$\frac{\Gamma \vdash \mathbf{fx} =_\tau \mathbf{gx}}{\Gamma \vdash \mathbf{f} =_{\sigma\tau} \mathbf{g}} \tag{44}$$

where $\mathbf{x}$ is a variable of type σ not free in $\mathbf{f}$, $\mathbf{g}$, or any formula in Γ.

R.8   Choice

$$\frac{\Gamma \vdash \forall x^\sigma \,.\, \exists y^\tau \,.\, \mathbf{R}xy}{\Gamma \vdash \exists f^{\sigma\tau} \,.\, \forall x \,.\, \mathbf{R}x(fx)} \tag{45}$$

$$\frac{\Gamma \vdash \mathbb{N}_\sigma\, \mathbf{n}}{\Gamma \vdash \bot \neq \exists_{\sigma t}\mathbf{n}} \tag{46}$$

where σ is either e or t.

3 Adding and subtracting assumptions $(+X;\ -\text{R.n})$

By "LF – R.n" we mean the system that has the rules of LF excluding R.n (where n is among 1 through 9).

Where X is a class of formulae, LF + X, is the system whose proofs are proofs in LF from assumptions in X. That is, a formula $\mathbf{P}$ is a theorem of LF + X if and only if there is a proof in LF that concludes with the sequent

$$\Gamma \vdash \mathbf{P}, \tag{47}$$

where Γ is a list of formulae in X. Note that LF + X is not necessarily closed under the rule of Intensionality, i.e., LF + X may prove the extensional equivalence $\mathbf{P} \leftrightarrow \mathbf{Q}$ but not the identity $\mathbf{P} = \mathbf{Q}$.

4 LF$_\iota$; LF$_\varepsilon$

We introduce two new families of constants indexed by simple types σ:

$$\iota_\sigma : \langle \sigma t \rangle \sigma \tag{48}$$

$$\varepsilon_\sigma : \langle \sigma t \rangle \sigma \tag{49}$$

LF$_\iota$ and LF$_\varepsilon$ are used for reasoning with these constants.

4.1 LF$_\iota$

ι is Church's (Church 1940: 61) *description function*: a function which maps every uniquely instantiated property to an instance thereof. D$_\iota$ is a set of axioms to this effect, which also choose a "default" value for ι to take, †, when applied to properties which are not uniquely instantiated:

$$\forall X^{\sigma t} \,.\, (\exists! X \rightarrow X(\iota X)) \tag{50}$$

$$\forall X^{\sigma t} \,.\, (\neg\exists! X \rightarrow \iota X = \dagger) \tag{51}$$

† is defined by the following recursive rewrite rules (although the particular choice of † is immaterial):

$$\dagger_e := \iota\lambda x^e \,.\, \bot \tag{52}$$

$$\dagger_t := \bot \tag{53}$$

$$\dagger_{\sigma\tau} := \lambda x^\sigma \,.\, \dagger_\tau \tag{54}$$

LF_ι is LF + D_ι, i.e., LF plus every sentence of the form (50) or (51).

The utility of LF_ι should be obvious: the language of mathematics is replete with notations (especially, but not only, variable-binding operators[6]) that are definable in LF_ι but not in LF. Perhaps the most common example is class abstraction, which is definable by

$$\{x^\sigma : \mathbf{P}\} := \iota\lambda X^{\sigma t} \,.\, (\forall y \,.\, (Xy = \top \vee Xy = \bot) \wedge X \equiv \lambda x \,.\, \mathbf{P}). \tag{55}$$

4.2 LF_ε

ε is Hilbert's (1922) *epsilon operator*[7] and Church's (1940: 61) *Choice function*. It maps every instantiated property (whether or not uniquely instantiated) to an instance thereof, and everything else to †. This is captured by the class C_ε, which consists of the following sentences, where σ is any type.

$$\forall X^{\sigma t} \,.\, (\exists X \rightarrow X(\varepsilon X)) \tag{58}$$

$$\forall X^{\sigma t} \,.\, (\neg\exists X \rightarrow \varepsilon X = \dagger) \tag{59}$$

[6] See Kalish and Montague 1964: Ch. IX for a representative collection of examples.

[7] Hilbert (1922) does not use the lowercase epsilon (ε) exactly our way. Rather, he uses it as a variable-binding operator (the formation rule for such an operator would be: if **x** is a variable of type σ, then $(\varepsilon\mathbf{x} \,.\, \mathbf{P})$ is a term of type σ). In the present setting, such a variable-binding operator would be redundant: were it to be to be introduced, the ε-function could be defined in terms of it by

$$\varepsilon_\sigma := \lambda X^{\sigma t} \,.\, \varepsilon y^\sigma \,.\, Xy, \tag{56}$$

while, in following Church (1940: 57, 61, where ι is confusingly used for both the description and Choice functions) in adopting the ε-operators as constants, we can (and we do) also follow Church (1940 : 58) in eliminating the corresponding variable-binding operator, as we do others, by

$$(\varepsilon x^\sigma \,.\, \mathbf{P}) := \varepsilon_\sigma \lambda x^\sigma \,.\, \mathbf{P}. \tag{57}$$

LF_ε is $\mathsf{LF} + \mathrm{D}_\iota + \mathrm{C}_\varepsilon$.[8]

The utilty of LF_ε is less obvious than the utility of LF_ι, and, in fact, it is hard to deny that the deductive power LF_ε adds to that of LF_ι is less often useful than the deductive power that LF_ι adds to that of LF. However, the former deductive power is sometimes useful, and that is enough to justify the adoption of LF_ε. As Carnap (1962) points out, εs can be used to extract definitions from non-categorical theories, $\varepsilon\mathbf{F}$ being a term that defines a unique satisfier of $\mathbf{F}$ provided that there is at least one satisfier of $\mathbf{F}$. As a concrete example of where this might be useful, in future work on the foundations of semantics, we show that various useful semantic (or proto-semantic) notions are definable in LF_ε + syntax but not in LF_ι + syntax.

4.3 The slingshot argument

We stress that LF_ι and LF_ε are *not* closed under the rule of Intensionality (if LF is consistent), and consequently there are formulae $\mathbf{P}$ and $\mathbf{Q}$ that are provably equivalent in LF_ι (hence also LF_ε), but for which

$$\mathbf{P} = \mathbf{Q} \tag{62}$$

is not provable, and indeed is refutable,[9] in the same system.

For example, in LF_ι, consider the function @, defined as

$$@ \ := \ \lambda p \,.\, \iota q \,.\, ((p \to q = \top) \wedge (\neg p \to q = \bot)). \tag{64}$$

@ is the function that maps every truth to $\top$ and every falsehood to $\bot$, so the material equivalence of $@p$ with p is intuitively clear (it is also easily provable). However, the provability of the identity

$$p = @p \tag{65}$$

[8] It is redundant to have the primitive description functions ι, since LF_ε categorically reduces to $\mathsf{LF} + \mathrm{C}_\varepsilon$ by the definition schema

$$\dagger_e \ := \ \varepsilon\lambda x^e \,.\, \bot \tag{60}$$

$$\iota_\sigma \ := \ \varepsilon\lambda f^{\langle\sigma t\rangle\sigma} \,.\, \forall X^{\sigma t} \,.\, ((\exists! X \to X(fX)) \wedge (\neg\exists! X \to fX = \dagger_\sigma)) \tag{61}$$

However, the use of ι is convenient to conceptually distinguish between full-blown applications of Choice and mere applications of descriptions.

[9] Let

$$\alpha \ := \ \forall p \,.\, (p \leftrightarrow @p), \tag{63}$$

with @ defined as in (64). Then $\alpha \leftrightarrow \top$ is provable but $\alpha = \top$ is refutable in these systems.

would have a collapsing effect: since @ only ever takes the values $\top$ or $\bot$, the provability of (65) would result in the provability of $\top$ and $\bot$ being the only propositions. This is in contradiction with Potential Infinity (R.9; see Metatheorem 12).

Inferring the identity of $p = @p$ (or some close variant[10]) from the proof of the material equivalence $p \leftrightarrow @p$ is known as the *slingshot* argument. It is deployed, among others, by Church (1943: 299–301) in his review of Carnap's (1942) *Introduction to Semantics*, where Church attributes it to Frege (1892); by Gödel (1944: 450, esp. n. 5) in his "Russell's Mathematical Logic"; again by Church (1956: 24–25) in his *Introduction to Mathematical Logic*, where it is again attributed to Frege (1892); by Quine (1953) in his criticism of quantified modal logic; and by Myhill (1958: 77–78) in relation to Church's (1951) version of the rule of Intensionality (called *Alternative (2)*). The slingshot is not valid in LF_ι or LF_ε, so these systems avoid the collapse.

5 Basic facts

Metatheorem 1 (Modus Ponens and Conditional Proof)**.** LF *is closed under the rules of Modus Ponens and Conditional Proof.*

$$\frac{\Gamma \vdash \mathbf{P} \qquad \Gamma \vdash \mathbf{P} \to \mathbf{Q}}{\Gamma \vdash \mathbf{Q}} \qquad\qquad \frac{\Gamma, \mathbf{P} \vdash \mathbf{Q}}{\Gamma \vdash \mathbf{P} \to \mathbf{Q}} \tag{66}$$

Proof. Assume without loss of generality that p, q, r do not occur in $\Gamma, \mathbf{P}, \mathbf{Q}$ (otherwise use different variables). For Modus Ponens we have

$$\dfrac{\dfrac{\Gamma \vdash \mathbf{P}}{\Gamma \vdash (\lambda r \,.\, \mathbf{P})p}\,\beta \qquad \dfrac{\Gamma \vdash \mathbf{P} \to \mathbf{Q}}{\Gamma \vdash (\lambda r \,.\, \mathbf{P}) \subseteq (\lambda r \,.\, \mathbf{Q})}\,\text{Def. (21)}}{\dfrac{\Gamma \vdash (\lambda r \,.\, \mathbf{Q})p}{\Gamma \vdash \mathbf{Q}}\,\beta}\,\text{Universal Instantiation} \tag{67}$$

[10] Specifically, variants constructed using directly by means of ι-terms (such as $\iota\lambda q \,.\, ((p \to q = \top) \wedge (p \to q = \top)))$ or by means of class abstracts (such as $\{q : (p \to q = \top) \wedge (p \to q = \bot)\}$), which are definable by means of ι (e.g., by (55) or, equivalently, with $\{x : \mathbf{P}\}$ taken to abbreviate $\lambda x \,.\, @\mathbf{P}$). In the literature, these tend to be preferred to $@p$.

for Conditional Proof we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \mathbf{P} \vdash \mathbf{Q} \qquad \dfrac{\overline{\Gamma, (\lambda r \,.\, \mathbf{P})p \vdash (\lambda r \,.\, \mathbf{P})p}}{\Gamma, (\lambda r \,.\, \mathbf{P})p \vdash \mathbf{P}}\,\beta}{\Gamma, (\lambda r \,.\, \mathbf{P})p \vdash \mathbf{Q}}}{\Gamma, (\lambda r \,.\, \mathbf{P})p \vdash (\lambda r \,.\, \mathbf{Q})p}\,\beta}{\Gamma \vdash (\lambda r \,.\, \mathbf{P}) \subseteq (\lambda r \,.\, \mathbf{Q})}\,\text{Universal Generalization}}{\Gamma \vdash \mathbf{P} \to \mathbf{Q}}\,\text{Def. (21)} \tag{68}$$

□

Metatheorem 2 (Intuitionistic Propositional Logic)**.** *Every theorem of intuitionistic propositional logic is provable from rules R.1 to R.4.*

Metatheorem 3 (Classical Quantification Theory)**.** *Classical introduction and elimination rules for $\forall_\sigma$ are derivable from R.1 to R.4.*

Metatheorem 4 (The Logic of Identity)**.** *The reflexivity of identity and Leibniz' law are provable from R.1 to R.4.*

$$\forall x^\sigma \,.\, x = x \tag{69}$$

$$\forall xy^\sigma \,.\, (x = y \to \mathbf{P} \to [y/x]\mathbf{P}) \tag{70}$$

Proof. For Leibniz' law, we first unpack the definition of = in the antecedent as antecedent to get (with R.2)

$$(\lambda Z \,.\, Zx) \subseteq (\lambda Z \,.\, Zy) \tag{71}$$

which, by universal instantiation (R.3) in conjunction with

$$(\lambda x \,.\, \mathbf{P})x, \tag{72}$$

yields

$$(\lambda x \,.\, \mathbf{P})y, \tag{73}$$

from which Leibniz' law follows by conditional proof and Universal Generalization.

□

Metatheorem 5 (Intensionality)**.** LF *is intensional, i.e., when* **P** *and* **Q** *are provable from each other,* **P** *and* **Q** *can be substituted in any context, including cases with variable capture.*

Proof. Let $\vec{\mathbf{x}}$ be the variables that occur free in **P** or **Q**. From Intensionality and Function Extensionality we have

$$\lambda\vec{\mathbf{x}} \,\mathbf{.}\, \mathbf{P} = \lambda\vec{\mathbf{x}} \,\mathbf{.}\, \mathbf{Q} \tag{74}$$

which are closed terms, so can be substituted in any context by Leibniz' law (Metatheorem 4). And by β-Equivalence (R.2), $\lambda\vec{\mathbf{x}} \,\mathbf{.}\, \mathbf{P}$ can be substituted in any context with $(\lambda\vec{\mathbf{x}} \,\mathbf{.}\, \mathbf{P})\vec{\mathbf{x}}$, and similarly for **Q**. □

Metatheorem 6 (Classical Propositional Logic)**.** *Every theorem of classical propositional logic is provable from rules R.1 to R.5.*

Proof. R.5 is sufficient to derive classical propositional logic from intuitionistic propositional logic, which is in turn supplied by Metatheorem 2. □

Metatheorem 7 (The Modal Logic S4)**.** *Every theorem of the propositional modal logic* S4 *is provable from the rules R.1 to R.7. Moreover, the rule of necessitation,*

$$\frac{\vdash \mathbf{P}}{\vdash \Box\mathbf{P}} \tag{75}$$

is derivable from those rules.

Metatheorem 8 (The Modal Logic S5)**.** *The "5" axiom of modal logic,*

$$\forall p \,\mathbf{.}\, (\neg\Box p \rightarrow \Box\neg\Box p), \tag{76}$$

is provable from rules R.1 to R.8. Thus, LF *includes the modal logic* S5*.*

Proof of Metatheorems 7 and 8. $\Box p$ is by definition $p = \top$, so we must show

$$p \neq \top \rightarrow \Box(p \neq \top). \tag{77}$$

We have for any function f,

$$\Box(fp \neq f\top \rightarrow p \neq \top). \tag{78}$$

Hence (by basic modal logic; see Metatheorem 7)

$$\Box(fp \neq f\top) \rightarrow \Box(p \neq \top). \tag{79}$$

So it suffices to find some function f for which

$$p \neq \top \rightarrow \Box(fp \neq f\top), \tag{80}$$

i.e., a function that, for all p distinct from $\top$, maps p and $\top$ to any two necessarily distinct things, such as, 0 and 1 or $\bot$ and $\top$. This is supplied by Choice applied to the relation

$$\lambda pq\,.\,((p \neq \top \to q = \bot) \wedge (p = \top \to q = \top)). \tag{81}$$

□

Metatheorem 9 (Necessity of Identity and Distinctness)**.** *Every instance of the necessity of identity,*

$$\forall xy^{\sigma}\,.\,(x = y \to \Box(x = y)), \tag{82}$$

is provable from rules R.1 to R.4 and R.6, and every instance of the necessity of distinctness,

$$\forall xy^{\sigma}\,.\,(x \neq y \to \Box(x \neq y)), \tag{83}$$

is provable from rules R.1 to R.6 and R.8.

Proof. For necessity of identity, we have $\Box(x = x)$ by intensionality and hence necessity of identity by Leibniz' law (Metatheorem 4). For necessity of distinctness, we use Choice as in Metatheorem 8. □

Metatheorem 10 (The Barcan Formula and its Converse; Bacon 2018)**.** *Every instance of the Barcan formula and its converse is provable from rules R.1 to R.7.*

$$\forall X^{\vec{\sigma}t}\,.\,(\forall\vec{z}\,.\,\Box X\vec{z}) \to \Box\forall\vec{z}\,.\,X\vec{z} \tag{84}$$

$$\forall X^{\vec{\sigma}t}\,.\,(\Box\forall\vec{z}\,.\,X\vec{z}) \to \forall\vec{z}\,.\,\Box X\vec{z} \tag{85}$$

Proof. For (84) (the Barcan formula) in the case where $\vec{\sigma}$ has length one, we observe

$$(\forall z\,.\,\Box Xz) \to \forall z\,.\,(Xz = (\lambda y\,.\,\top)z) \tag{86}$$

hence, by R.7,

$$X = \lambda y\,.\,\top. \tag{87}$$

Moreover, by R.6 we have

$$\Box\forall z\,.\,(\lambda y\,.\,\top)z \tag{88}$$

hence $\Box\forall z\,.\,Xz$ as required. This proof extends to $\vec{\sigma}$ of arbitrary length by induction.

For (85) (the converse Barcan formula), we have by R.6 the identity

$$(\forall \vec{z} \,.\, X\vec{z}) = (X\vec{y} \wedge \forall \vec{z} \,.\, X\vec{z}). \tag{89}$$

we also have $p = (p \wedge \top)$, hence

$$(\Box \forall \vec{z} \,.\, X\vec{z}) \rightarrow \Box(X\vec{y}) \tag{90}$$

which yields (85) by basic quantificational reasoning. □

Metatheorem 11 (Modal Intensionality; Bacon 2018)**.** *All axioms of modal intensionality—asserting that necessarily equivalent propositions and necessarily equivalent properties are identical—are provable from rules R.1 to R.7.*

$$\forall pq \,.\, (\Box(p \leftrightarrow q) \rightarrow p = q) \tag{91}$$

$$\forall FG^{\vec{\sigma}t} \,.\, (\Box(F \equiv G) \rightarrow F = G) \tag{92}$$

Proof. By some basic applications of R.6 we have the equations

$$p = (p \wedge \top) \tag{93}$$

$$(p \wedge q) = (p \wedge (p \rightarrow q)). \tag{94}$$

By (93) we have

$$\Box(p \leftrightarrow q) \rightarrow (p = (p \wedge (p \rightarrow q)) \wedge q = (q \wedge (q \rightarrow p))), \tag{95}$$

hence, by (94)

$$\Box(p \leftrightarrow q) \rightarrow (p = (p \wedge q) \wedge q = (p \wedge q)), \tag{96}$$

which implies (91).

For (92), we prove the case where $\vec{\sigma}$ has length one, i.e.,

$$\forall FG^{\sigma t} \,.\, (\Box(F \equiv G) \rightarrow F = G). \tag{97}$$

The other cases follow straightforwardly. For this, suppose $\Box(F \equiv G)$, i.e.,

$$\Box \forall x^{\sigma} \,.\, (Fx \leftrightarrow Gx). \tag{98}$$

Then by Metatheorem 10 we have

$$\forall x^{\sigma} \,.\, \Box(Fx \leftrightarrow Gx). \tag{99}$$

Hence by (91) we have

$$\forall x^{\sigma} \,\textbf{.}\, (Fx = Gx), \tag{100}$$

which implies $F = G$ by R.7. □

Metatheorem 12 (The Refutation of the Axioms of Extensionality)**.** *The denial of the axiom of propositional extensionality, and hence also of each of the axioms of extensionality for properties and relations, is provable from rules R.1 to R.5 and R.9.*

$$\exists pq \,\textbf{.}\, ((p \leftrightarrow q) \wedge p \neq q) \tag{101}$$

Proof. Suppose otherwise, i.e., $\forall pq \,\textbf{.}\, ((p \leftrightarrow q) \rightarrow p = q)$. Then by R.1 to R.5 we have

$$\forall p \,\textbf{.}\, ((p \rightarrow p = \top) \wedge (\neg p \rightarrow p = \bot)), \tag{102}$$

and hence

$$\neg\exists(1 + 1 + 1_t) \tag{103}$$

by Defs. (30), (31) and (33). Appealing to (102) again, we have

$$\bot = \exists(1 + 1 + 1_t) \tag{104}$$

which is contradictory by R.9. □

Metatheorem 13 (Class Comprehension and Class Extensionality; Church 1940)**.** *Where a class is a* $\{\top, \bot\}$*-valued function, i.e.,*

$$\text{Class}_{\sigma} \;:=\; \lambda X^{\sigma t} \,\textbf{.}\, \forall y \,\textbf{.}\, (Xy = \top \vee Xy = \bot), \tag{105}$$

LF *proves (a) that every property is coextensive with some class:*

$$\forall X^{\sigma t} \,\textbf{.}\, \exists Y \in \text{Class}_{\sigma} \,\textbf{.}\, X \equiv Y, \tag{106}$$

and (b) that classes are extensional:

$$\forall XY \in \text{Class}_{\sigma} \,\textbf{.}\, (X \equiv Y \rightarrow X = Y). \tag{107}$$

Proof. Class comprehension is immediate from Choice; in LF_{ι} the class coextensive with X is

$$\lambda y^{\sigma} \,\textbf{.}\, \iota\lambda p \,\textbf{.}\, ((Xy \rightarrow p = \top) \wedge (\neg Xy \rightarrow p = \bot)). \tag{108}$$

(Cf. Church 1940, p. 61.) The extensionality of classes is immediate from Function Extensionality. □

Metatheorem 14 (The Necessity of Logic)**.** *Where* **P** *is a sentence (i.e., a closed formula),*

$$\Box\mathbf{P} \vee \Box\neg\mathbf{P} \tag{109}$$

is provable from rules R.1 to R.8. Note this no longer holds when **P** *is permitted to contain non-logical constants (including* ι *and* ε*).*

Proof. Omitted. □

Metatheorem 15 (The Complete Atomic Boolean Algebra of Propositions)**.** *It is a theorem of* LF *that the propositions form a complete atomic Boolean algebra under* $\wedge$, $\vee$, $\neg$*. (Hence, by Metatheorem 16,* LF *proves there are at least* $2^{\aleph_0}$ *propositions.)*

Proof. Omitted. See Theorem 11.5 of Gallin (1975). □

Metatheorem 16 (The Axioms of Infinity)**.** *Every instance of*

$$\forall n \in \mathbb{N}_\sigma \,.\, \exists n, \tag{110}$$

is provable in LF*.* (110) *asserts that there is an actual infinity of things.*

Proof. Using Metatheorem 10. □

Metatheorem 17 (Peano Arithmetic)**.** *Every theorem of Peano arithmetic is a theorem of* $\mathsf{LF} - R.8$ *when the constants of arithmetic are defined as above (at any type), and where multiplication is defined in a standard way (omitted here).*

Proof. Omitted. □

Remark 18. Goodsell (2022) shows adding the necessitation of second-order Peano arithmetic for $\mathbb{N}_e$ and $\mathbb{N}_t$ to the rules R.1 to R.7 is equivalent to adding R.9.

Metatheorem 19 (The Closure of Necessitated Extensions Under the Rule of Intensionality)**.** *For any set of formulae X, if every formula in X begins with a* □ *taking widest possible scope, then* $\mathsf{LF} + X$ *is closed under the rule of Intensionality (R.6).*

Proof. Straightforward from Metatheorem 11. □

Metatheorem 20 (LF_ι and LF_ε as Conservative Extensions)**.** LF *is conservatively extended by* LF_ι *and* LF_ε*.*

Proof. Choice (R.8) can be used to show, for any finite set of axioms from D_ι or C_ε, that functions ι or ε satisfying those axioms exist. □

Metatheorem 21 (Consistency Relative to ZFC; Equiconsistency with Henkin's System)**.** LF *is equiconsistent with the system of Henkin (1950), and hence is consistent relative to* ZFC *by Henkin's soundness theorem.*

Proof. Using Benzmüller's (2010) method of embedding. □

6 Relation to other systems

We now sketch some relations of LF to other systems formulated in simply typed languages with abstraction and application, notably omitting various importantly related systems formulated in different languages, such as the aforementioned maximal system GM that can be constructed out of the components studied in Gallin's (1975) Montague-inspired monograph and Church's (1951) Logic of Sense and Denotation, Alternative (2).

The system of Church (1940) is equivalent to LF_ε minus Intensionality (R.6), and with Potential Infinity (R.9) replaced with a rule of actual infinity for $\mathbb{N}_e$ (but not $\mathbb{N}_t$), i.e., with:

$$\frac{\Gamma \vdash \mathbb{N}_e\,\mathbf{n}}{\Gamma \vdash \exists_{et}\mathbf{n}} \tag{111}$$

The system of Henkin (1950) is equivalent to Church's system plus a rule of *extensionality* (notice that, in contrast with R.6, below a Γ occurs on the left-hand side of the turnstile):

$$\frac{\Gamma, \mathbf{P} \vdash \mathbf{Q} \qquad \Gamma, \mathbf{Q} \vdash \mathbf{P}}{\Gamma \vdash \mathbf{P} = \mathbf{Q}} \tag{112}$$

The system HFE of Bacon (2018) is equivalent to LF minus Choice and Potential Infinity (rules R.8 and R.9). In subsequent work, Bacon and Dorr (forthcoming) use a weaker system, *Classicism*, which also omits Function Extensionality (R.7), and strengthens the rule of Intensionality in order to obtain a system that is intensional in the sense of Metatheorem 5. A sufficient such strengthening is the following rule, where **R** and **S** differ by the substitution of an occurrence of **P** for **Q** in any context (including substitutions with variable capture):

$$\frac{\mathbf{P} \vdash \mathbf{Q} \qquad \mathbf{Q} \vdash \mathbf{P}}{\mathbf{R} \vdash \mathbf{S}} \tag{113}$$

Another axiomatization Bacon and Dorr consider, which is equivalent in a language restricted to relational types (i.e., types that, when angle brackets are omitted in accordance with the convention, end with "*t*") leaves the rule of Intensionality as it is, but replaces Function Extensionality with the rule

$$\frac{\Gamma \vdash \Box\forall \mathbf{x} \,.\, \mathbf{fx} = \mathbf{gx}}{\Gamma \vdash \mathbf{f} = \mathbf{g}} \tag{114}$$

where, as in Function Extensionality, **x** must not be among the free variables of **f** or **g**. They show, moreover, that replacing Function Extensionality with this rule in the background of LF's other rules is equivalent with LF.

References

Bacon, Andrew (2018). "The Broadest Necessity". In: *Journal of Philosophical Logic* 47.5, pp. 733–783.

Bacon, Andrew and Cian Dorr (forthcoming). "Classicism". In: *Higher-order Metaphysics*. Ed. by Peter Fritz and Nicholas K. Jones. Oxford University Press.

Benzmueller, Christoph (2010). *Simple Type Theory as Framework for Combining Logics*. arXiv: `1004.5500 [cs.LO]`.

Carnap, Rudolf (1942). *Introduction to Semantics*. Cambridge: Harvard University Press.

— (1962). "On the Use of Hilbert's ε-Operator in Scientific Theories". In: *Essays on the Foundations of Mathematics: Dedicated to A. A. Fraenkel on His 70th Anniversary*. Ed. by Yehoshua Bar-Hillel et al. Amsterdam: North-Holland Publishing Company, pp. 156–164.

Church, Alonzo (1940). "A Formulation of the Simple Theory of Types". In: *Journal of Symbolic Logic* 5.2, pp. 56–68.

— (1943). "Review of Rudolf Carnap, Introduction to Semantics". In: *Philosophical Review* 52.3, pp. 298–304.

— (1951). "A Formulation of the Logic of Sense and Denotation". In: *Structure, Method and Meaning*. Liberal Ars Press.

— (1956). *Introduction to Mathematical Logic*. Princeton: Princeton University Press.

Frege, Gottlob (1892). "Über Sinn und Bedeutung". In: *Zeitschrift für Philosophie und philosophische Kritik* 100, pp. 25–50.

Gallin, Daniel (1975). *Intensional and Higher-Order Modal Logic: With Applications to Montague Semantics*. American Elsevier Pub. Co.

Gödel, Kurt (1944). "Russell's Mathematical Logic". In: *The Philosophy of Bertrand Russell*. Ed. by Paul Arthur Schilpp. Evanston and Chicago: Northwestern University, pp. 123–154.

Goodsell, Zachary (2022). "Arithmetic is Determinate". In: *Journal of Philosophical Logic* 51.1, pp. 127–150.

Henkin, Leon (1950). “Completeness in the Theory of Types”. In: *Journal of Symbolic Logic* 15.2, pp. 81–91.

Hilbert, David (1922). “Neubegründung der Mathematik: Erste Mitteilung”. In: *Abhandlungen aus dem Seminar der Hamburgischen Universität* 1, pp. 157–177.

Kalish, Donald and Richard Montague (1964). *Logic: Techniques of Formal Reasoning*. New York: Harcourt, Brace & World.

Montague, Richard (1970). “Universal Grammar”. In: *Theoria* 36.3, pp. 373–398.

Myhill, John (1958). “Problems Arising in the Formalization of Intensional Logic”. In: *Logique Et Analyse* 1.1, pp. 78–83.

Quine, Willard Van Orman (1953). “Reference and Modality”. In: *From a Logical Point of View*. Cambridge, MA: Harvard University Press, 139—159.

Whitehead, Alfred North and Bertrand Russell (1910–1913). *Principia Mathematica*. Cambridge, England: University Press.

Williamson, Timothy (2023). “Higher-order metaphysics and small differences”. In: *Analysis* 83.1, pp. 213–224.

Higher-Order Model Checking

II: Recursion Schemes and their Algorithmics

Luke Ong

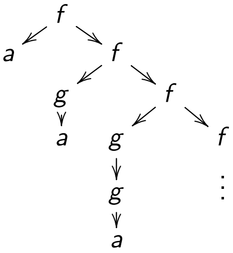
University of Oxford
http://www.cs.ox.ac.uk/people/luke.ong/personal/
http://mjolnir.cs.ox.ac.uk

Estonia Winter School in Computer Science, 3-8 Mar 2013

A challenge problem in higher-order verification

Example: Consider $[\![G]\!]$ on the right

- φ_1 = "Infinitely many f-nodes are reachable".
- φ_2 = "Only finitely many g-nodes are reachable".



Every node on the tree satisfies $\varphi_1 \vee \varphi_2$.

Let **RecSchTree**$_n$ be the class of Σ-labelled trees generated by order-n recursion schemes.

Is the "MSO Model-Checking Problem for **RecSchTree**$_n$" decidable?

- INSTANCE: An order-n recursion scheme G, and an MSO formula φ
- QUESTION: Does the Σ-labelled tree $[\![G]\!]$ satisfy φ?

Why study monadic second-order (MSO) logic?

Because it is the gold standard of logics for describing correctness properties.

- MSO is *very* expressive.
  Over graphs, MSO is more expressive than the modal mu-calculus, into which all standard temporal logics (e.g. LTL, CTL, CTL$*$, etc.) can embed.
- It is hard to extend MSO meaningfully without sacrificing decidability where it holds.

Monadic Second-Order Logic (for Σ-labelled trees)

Fix a vocabulary. Three types of predicate symbols:

1. Parent-child relationship between nodes: $\mathbf{d}_i(x, y) \equiv$ "y is i-child of x"
2. Node labelling: $\mathbf{p}_f(x) \equiv$ "x has label f" where f is a Σ-symbol
3. Set-membership: $x \in X$

First-order variables: x, y, z, etc. (ranging over nodes)
Second-order variables: X, Y, Z, etc. (ranging over sets of nodes)

MSO formulas are generated from three kinds of atomic formulas:

$$\mathbf{d}_i(x, y), \quad \mathbf{p}_f(x), \quad x \in X$$

and closed under boolean connectives, first-order quantification ($\forall x.-, \exists x.-$) and second-order quantifications: ($\forall X.-, \exists X.-$).

A Σ-labelled tree $t : dom(t) \longrightarrow \Sigma$ is represented as a structure

$$\langle\, dom(t),\ \langle\, \mathbf{d}_i : 1 \le i \le m \,\rangle,\ \langle\, \mathbf{p}_f : f \in \Sigma \,\rangle\,\rangle$$

Examples of MSO-definable properties

Our version of MSOL is parsimonious. Several useful predicates are definable:

1. Set inclusion (and hence equality): $X \subseteq Y \ \equiv\ \forall x : x \in X \to x \in Y$.
2. "Is-an-ancestor-of" or prefix ordering $x \leq y$ (and hence $x = y$):

$$\begin{array}{rcl} \mathrm{PrefCl}(X) & \equiv & \forall x, y : y \in X \wedge \bigvee_{i=1}^{m} \mathbf{d}_i(x, y) \ \to\ x \in X \\ x \leq y & \equiv & \forall X : \mathrm{PrefCl}(X) \wedge y \in X \ \to\ x \in X \end{array}$$

3. Reachability property: "X is a path"

$$\begin{array}{rcl} \mathrm{Path}(X) & \equiv & \forall x, y \in X : x \leq y \vee y \leq x \qquad \wedge \\ & & \forall x, y, z : x \in X \wedge z \in X \wedge x \leq y \leq z \ \to\ y \in X \end{array}$$

$$\begin{array}{rcl} \mathrm{MaxPath}(X) & \equiv & \mathrm{Path}(X) \quad \wedge \\ & & \forall Y : \mathrm{Path}(Y) \wedge X \subseteq Y \ \to\ Y \subseteq X. \end{array}$$

Example: "A tree has infinitely many f-labelled nodes"

A set of nodes is a cut if (i) no two nodes in it are $\leq$-compatible, and (ii) it has a non-empty intersection with every maximal path.

$$\begin{aligned}\mathsf{Cut}(X) \;&\equiv\; \forall x, y \in X : \neg(x \leq y \;\vee\; y \leq x) \quad \wedge \\ &\;\; \forall Z : (\mathsf{MaxPath}(Z) \;\rightarrow\; \exists z \in Z : z \in X)\end{aligned}$$

Lemma. A set X of nodes in a finitely-branching tree is finite iff there is a cut C such that every X-node is a prefix of some C-node.

$$\mathsf{Finite}(X) \;\equiv\; \exists Y : (\mathsf{Cut}(Y) \;\wedge\; \forall x \in X : \exists y \in Y : x \leq y)$$

Hence "there are finitely many nodes labelled by f" is expressible in MSOL by

$$\exists X : (\mathsf{Finite}(X) \;\wedge\; \forall x : \mathbf{p}_f(x) \rightarrow x \in X)$$

But "MSOL cannot count": E.g. "X has twice as many elements as Y" is not expressible in MSO.

Lecture 3

Recapitulation

- Two families of generators: HORS and HOPDA
-

Today's lecture

-
-

A (selective) survey of MSO-decidable structures: up to 2002

- Rabin 1969: Infinite binary trees and regular trees. "Mother of all decidability results in algorithmic verification."
- Muller and Schupp 1985: Configuration graphs of PDA.
- Caucal 1996 Prefix-recognisable graphs (ϵ-closures of configuration graphs of pushdown automata, Stirling 2000).
- Knapik, Niwiński and Urzyczyn (TLCA 2001, FOSSACS 2002):
  $\mathbf{PushdownTree}_n\Sigma$ = Trees generated by order-n pushdown automata.
  $\mathbf{SafeRecSchTree}_n\Sigma$ = Trees generated by order-n safe rec. schemes.
- Subsuming all the above:
  Caucal (MFCS 2002). $\mathbf{CaucalTree}_n\Sigma$ and $\mathbf{CaucalGraph}_n\Sigma$.

Theorem (KNU-Caucal 2002)

For $n \geq 0$, $\mathbf{PushdownTree}_n\Sigma = \mathbf{SafeRecSchTree}_n\Sigma = \mathbf{CaucalTree}_n\Sigma$; and they have decidable MSO theories.

What is the safety constraint on recursion schemes?

Safety is a set of constraints on where variables may occur in a term.

Definition (Damm TCS 82, KNU FoSSaCS'02)

An order-2 equation is unsafe if the RHS has a subterm P s.t.

1. P is order 1
2. P occurs in an operand position (i.e. as 2nd argument of application)
3. P contains an order-0 parameter.

Consequence: An order-i subterm of a safe term can only have free variables of order at least i.

Example (unsafe rule).

$$F : (o \to o) \to o \to o \to o,\ f : o^2 \to o,\ x, y : o.$$

$$F\,\varphi\,x\,y \quad = \quad f\,(F\,\underline{(F\,\varphi\,y)}\,y\,(\varphi\,x))\,a$$

The subterm $F\ \varphi\ y$ has order 1, but the free variable y has order 0.

What is the point of safety?

Safety does have an important algorithmic advantage!

Theorem (KNU 02, Blum + O. TLCA 07, LMCS 09)

Substitution (hence β-red.) in safe λ-calculus can be safely implemented without renaming bound variables! Hence no fresh names needed.

Theorem

1. *(Schwichtenberg 76) The numeric functions representable by simply-typed λ-terms are multivariate polynomials with conditional.*
2. *(Blum + O. LMCS 09) The numeric functions representable by simply-typed safe λ-terms are the multivariate polynomials.*

(See (Blum + O. LMCS 09) for a study on the safe lambda calculus.)

Infinite structures generated by recursion schemes: key questions

1. **MSO decidability**: Is safety a genuine constraint for decidability?
   I.e. do trees generated by (arbitrary) recursion schemes have decidable MSO theories?
2. **Machine characterisation**: Find a hierarchy of automata that characterise the expressive power of recursion schemes.
   I.e. how should the power of higher-order pushdown automata be augmented to achieve equi-expressivity with (arbitrary) recursion schemes?
3. **Expressivity**: Is safety a genuine constraint for expressivity?
   I.e. are there inherently unsafe word languages / trees / graphs?

Infinite structures generated by recursion schemes: key questions

4 **Graph families**:

1. **Definition**: What is a good definition of "graphs generated by recursion schemes"?
2. **Model-checking properties**: What are the decidable (modal-) logical theories of the graph families?

Q1. Do trees in $\mathsf{RecSchTree}_n\Sigma$ have decidable MSO theories?

Some progress:

Theorem (Aehlig, de Miranda + O. TLCA 2005)

Σ-labelled trees generated by order-2 recursion schemes (whether safe or not) have decidable MSO theories.

Theorem (Knapik, Niwinski, Urczyczn + Walukiewicz, ICALP 2005)

Modal mu-calculus model checking problem for homogenously-typed order-2 schemes (whether safe or not) is 2-EXPTIME complete.

What about higher orders?

Yes: MSO decidability extends to all orders (O. LICS06).

Q1. Do trees in $\mathbf{RecSchTree}_n\Sigma$ have decidable MSO theories? Yes

Theorem (O. LICS 2006)

For $n \geq 0$, the modal mu-calculus model-checking problem for $\mathbf{RecSchTree}_n\Sigma$ *(i.e. trees generated by order-n recursion schemes) is n-EXPTIME complete. Thus these trees have decidable MSO theories.*

Proof Idea. Two key ingredients:

Generated tree $[\![G]\!]$ satisfies mu-calculus formula φ

$\iff$ { Emerson + Jutla 1991}

APT $\mathcal{B}_\varphi$ has accepting run-tree over generated tree $[\![G]\!]$

$\iff$ { **I. Transference Principle: Traversal-Path Correspondence**}

APT $\mathcal{B}_\varphi$ has accepting traversal-tree over computation tree $\lambda(G)$

$\iff$ { **II. Simulation of traversals by paths** }

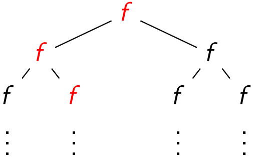
APT $\mathcal{C}_\varphi$ has an accepting run-tree over computation tree $\lambda(G)$

which is decidable because $\lambda(G)$ is regular.

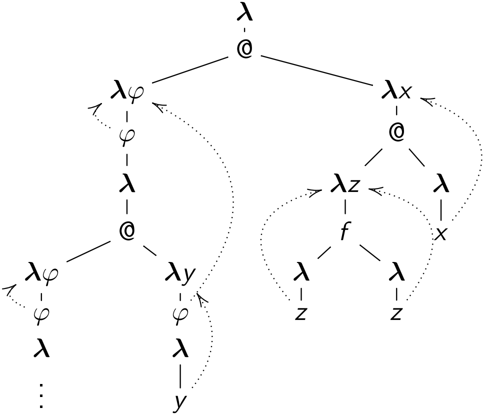
Transference principle, based on a theory of traversals

$$G : \left\{ \begin{array}{rcl} S & = & F\,H \\ F\,\varphi & = & \varphi\,(F\,\varphi) \\ H\,z & = & f z z \end{array} \right. \qquad \mapsto \qquad \overline{G} : \left\{ \begin{array}{rcl} S & = & \boldsymbol{\lambda}.@\,F\,(\boldsymbol{\lambda}x.@\,H\,\boldsymbol{\lambda}.x) \\ F & = & \boldsymbol{\lambda}\varphi.\varphi(\boldsymbol{\lambda}.@\,F\,(\boldsymbol{\lambda}y.\varphi(\boldsymbol{\lambda}.y))) \\ H & = & \boldsymbol{\lambda}z.f(\boldsymbol{\lambda}.z)(\boldsymbol{\lambda}.z) \end{array} \right.$$

$[\![\, G \,]\!]$



$\lambda(G)$



Idea: β-reduction is global (i.e. substitution changes the term being evaluated); game semantics gives an equivalent but local view.
A traversal (over the computation tree $\lambda(G)$) is a trace of the local computation that produces a path (over $[\![G]\!]$).

Theorem (Path-traversal correspondence)

Let G be an order-n recursion scheme.

(i) *There is a 1-1 correspondence between maximal paths p in (Σ-labelled) generated tree $[\![G]\!]$ and maximal traversals t_p over computation tree $\lambda(G)$.*

(ii) *Further for each p, we have $p \upharpoonright \Sigma = t_p \upharpoonright \Sigma$.*

Proof is by game semantics.

Explanation (for game semanticists):

- Term-tree $[\![G]\!]$ is (a representation of) the game semantics of G.
- Paths in $[\![G]\!]$ correspond to plays in the strategy-denotation.
- Traversals t_p over computation tree $\lambda(G)$ are just (representations of) the uncoverings of the plays (= path) p in the game semantics of G.

Four different proofs of the MSO decidability result

1. Game semantics and traversals (O. LICS06)
   - variable profiles. E.g. a profile of $(o \to o) \to o$ is $(\{ (\{ q \}, q), (\{ q, q' \}, q') \}, q)$
2. Collapsible pushdown automata (Hague, Murawski, O. & Serre LICS08)
   - equi-expressivity theorem + rank aware automata
3. type-theoretic characterisation of APT (Kobayashi & O. LICS09)
   - intersection types. E.g. $(q \to q) \wedge (q \wedge q' \to q') \to q$
4. Krivine machine (Salvati & Walukiewicz ICALP11)
   - residuals

A common pattern

1. Decision problem equivalent to solving an infinite parity game.
2. Simulate the infinite parity game by a finite parity game.
3. Key ingredient of the game: variable profiles / automaton control-states / intersection types / residuals.

Q2: Machine characterisation: collapsible pushdown automata

Order-2 collapsible pushdown automata [HOMS, LiCS 08a] are essentially the same as 2PDA with links [AdMO 05] and panic automata [KNUW 05].

Idea: Each stack symbol in 2-stack "remembers" the stack content at the point it was first created (i.e. $push_1$ed onto the stack), by way of a pointer to some 1-stack underneath it (if there is one such).

Two new stack operations: $a \in \Gamma$ (stack alphabet)

- $push_1\ a$: pushes a onto the top of the top 1-stack, together with a pointer to the 1-stack immediately below the top 1-stack.
- *collapse* (= panic) collapses the 2-stack down to the prefix pointed to by the top_1-element of the 2-stack.

Note that the pointer-relation is preserved by $push_2$.

Collapsible pushdown automata: extending to all finite orders

In **order-n CPDA**, there are $n-1$ versions of $push_1$, namely, $push_1^j\ a$, with $1 \leq j \leq n-1$:

> *$push_1^j\ a$: pushes a onto the top of the top 1-stack, together with a pointer to the j-stack immediately below the top j-stack.*

Example: Urzyczyn's Language U over alphabet $\{\,(\,,\,)\,,*\,\}$

Definition (Aehlig, de Miranda + O. FoSSaCS 05) A ***U*-word** has 3 segments:

$$\underbrace{(\cdots(\cdots(}_{A}\;\underbrace{(\cdots)\cdots(\cdots)}_{B}\;\underbrace{*\cdots*}_{C}$$

- Segment A is a prefix of a well-bracketed word that ends in **(**, and the opening **(** is not matched in the entire word.
- Segment B is a well-bracketed word.
- Segment C has length equal to the number of **(** in segment A.

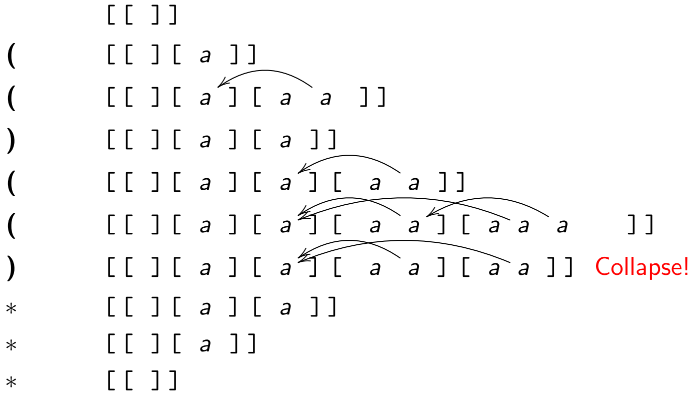
Examples

1. $(\,(\,)\,(\,(\,)\,(\,(\,)\,)\;*\;*\;*$ is a U-word
2. For each $n \geq 0$, we have $(\,(^n\,)^n\,(\;*^n\;**$ is a U-word.
   Hence by "*uvwxy* Lemma", U is not context-free.

Recognising U by a (det.) 2CPDA. E.g. (( ) (( ) $* \; * * \in U$

(Ignoring control states for simplicity)

Upon reading	Do
(	$push_2$; $push_1 a$
)	pop_1
first $*$	*collapse*
subsequent $*$	pop_2



```
      [ [ ] ]
(     [ [ ] [ a ] ]
(     [ [ ] [ a ] [ a a ] ]
)     [ [ ] [ a ] [ a ] ]
(     [ [ ] [ a ] [ a ] [ a a ] ]
(     [ [ ] [ a ] [ a ] [ a a ] [ a a a ] ]
)     [ [ ] [ a ] [ a ] [ a a ] [ a a ] ]   Collapse!
*     [ [ ] [ a ] [ a ] ]
*     [ [ ] [ a ] ]
*     [ [ ] ]
```

What does the depth of the top 1-stack mean?

Is order-n CPDA strictly more expressive than order-n PDA?

Does the *collapse* operation add any expressive power?

Lemma (AdMO FoSSaCS05): Urzyczyn's language U is quite telling!

1. U is not recognised by a 1PDA.
2. U is recognised by a non-deterministic 2PDA.
3. U is recognised by a deterministic 2CPDA.

Question

Is U recognisable by a deterministic 2PDA? or by nPDA for any n?

If true, there is an associated tree that is generated by an order-2 recursion scheme, but not by any order-2 safe recursion scheme.

Q2: Machine characterization: order-n RS = order-n CPDA

Theorem (Equi-expressivity [Hague, Murawski, O. & Serre LICS08])

For each $n \geq 0$, order-n collapsible PDA and order-n recursion schemes are equi-expressive for Σ-labelled trees.

Proof idea

- From recursion scheme to CPDA: Use game semantics.
  Code traversals as n-stacks.
  Invariant: The top 1-stack is the P-view of the encoded traversal.
- From CPDA to recursion scheme:
  Code configuration c as Σ-term M_c, so that $c \to c'$ implies M_c rewrites to $M_{c'}$.

CPDA are a machine characterization of simply-typed lambda calculus with recursions.

A direct proof (without game semantics) [Carayol & Serre LICS12].

Q3: Is safety a genuine constraint on expressivity?

Question (Safety, KNW FoSSaCS02)

Are there inherently unsafe word languages / trees / graphs?

Word languages? Yes

Theorem (Parys STACS11, LICS12)

There is a language (similar to U) recognised by a deterministic 2CPDA but not by any deterministic nPDA for all $n \geq 0$.

Proof uses a powerful pumping lemma for HOPDA.

(Another pumping lemma for nCPDA is used to prove a hierarchy theorem for collapsible graphs and trees [Kartzow & Parys, MFCS12])

Trees? Yes

Corollary (Parys STACS11, LICS12)

There is a tree generated by an order-2 recursion scheme but not by any safe HORS.

Graphs? Yes.

Theorem (Hague, Murawski, O and Serre LICS08)

1. *Solvability of parity games over order-n CPDA graphs is n-EXPTIME complete.*
2. *There is an 2CPDA configuration graph with an undecidable MSO theory.*

Corollary

There is a 2CPDA whose configuration graph (semi-infinite grid) is not that of any nPDA, for any n.

A safety question for non-determinacy

Question (Safety non-determinacy)

Is there a word language recognised by a order-n CPDA which is not recognisable by any non-deterministic HOPDA?

For order 2, the answer is no.

Theorem (Aehlig, de Miranda and O. FoSSaCS 2005)

For every order-2 recursion scheme, there is a safe non-deterministic order-2 recursion scheme that generates the same word language.

Higher Groups in Homotopy Type Theory

Ulrik Buchholtz, Floris van Doorn and Egbert Rijke

February 14, 2018

Abstract

We present a development of the theory of higher groups, including infinity groups and connective spectra, in homotopy type theory. An infinity group is simply the loops in a pointed, connected type, where the group structure comes from the structure inherent in the identity types of Martin-Löf type theory. We investigate ordinary groups from this viewpoint, as well as higher dimensional groups and groups that can be delooped more than once. A major result is the stabilization theorem, which states that if an n-type can be delooped $n+2$ times, then it is an infinite loop type. Most of the results have been formalized in the Lean proof assistant.

Table 1: Periodic table of k-tuply groupal n-groupoids.

$k \setminus n$	0	1	2	$\cdots$	∞
0	pointed set	pointed groupoid	pointed 2-groupoid	$\cdots$	pointed ∞-groupoid
1	group	2-group	3-group	$\cdots$	∞-group
2	abelian group	braided 2-group	braided 3-group	$\cdots$	braided ∞-group
3	— ” —	symmetric 2-group	sylleptic 3-group	$\cdots$	sylleptic ∞-group
4	— ” —	— ” —	symmetric 3-group	$\cdots$	?? ∞-group
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
ω	— ” —	— ” —	— ” —	$\cdots$	connective spectrum

1 Introduction

The homotopy hypothesis is the statement that homotopy n-types (topological spaces with trivial homotopy groups above level n) correspond to n-groupoids for $n \in \mathbb{N} \cup \{\infty\}$ via the fundamental ∞-groupoid construction. In Grothendieck's original version in *Pursuing Stacks* [13] this was a conjecture about a particular model of ∞-groupoids. It is also a theorem for many particular models of ∞-groupoids, for example the Kan simplicial sets, but it is now mostly taken to be a property *defining* ∞-groupoids up to equivalence.

In this paper, we investigate the homotopy hypothesis in the context of homotopy type theory (HoTT). HoTT refers to the homotopical interpretation of Martin-Löf's dependent type theory [2, 26]. In this homotopical interpretation, every type-theoretical construction corresponds to a homotopy-invariant construction on spaces.

In HoTT, every type has a path space given by the identity type. For a pointed type we can construct the loop space, which has the structure of an ∞-group. Moreover, if the type is *truncated*, then we can retreive the usual notion of groups, 2-groups and higher groups. This allows us to define a higher group internally in the language of type theory as a type that is the loop space of a pointed connected type, its delooping.

We also investigate groups that can be delooped more than once, which gives n-groups with additional coherences. The full family of groups we consider is in Table 1, which we will explain in detail in section 3.

Our approach is additionally validated by the corresponding observation in ∞-topos theory, where it is a theorem that the ∞-category of pointed, connected objects in $\mathcal{X}$ is equivalent to the ∞-category of higher group objects in $\mathcal{X}$, for any ∞-topos $\mathcal{X}$ [18, Lemma 7.2.2.11(1)].

We have formalized most of our results in the HoTT library [9] of the Lean Theorem Prover [19]. The formalized results can be found in the file `https://github.com/cmu-phil/Spectral/blob/master/higher_groups.hlean`. We will indicate the major formalized results in this paper by referring to the name in the formalization inside square brackets. For more information about the formalization, see section 8.

We are indebted to Michael Shulman for writing a blog post [22] on classifying spaces from a univalent perspective.

2 Preliminaries

In this paper we will work in the type theory of the HoTT book [25], although all arguments will also hold in a cubical type theory, such as [1, 7]. In this section we briefly introduce the concepts we need for the rest of the paper.

The type theory contains dependent function types $(x : A) \to B(x)$, which are more traditionally denoted as $\Pi_{x:A} B(x)$ and dependent pair types $(x : A) \times B(x)$, which are traditionally denoted as $\Sigma_{x:A} B(x)$. We choose to use this Agda-inspired notation because we often deal with deeply nested dependent sum types.

Within a type A we have the identity type or path type $=_A : A \to A \to \mathrm{Type}$. We have various operations on paths, such as concatenation $p \cdot q$ and inversion p^{-1} of paths. The functorial action of a function $f : A \to B$ on a path $p : a_1 =_A a_2$ is denoted $\mathrm{ap}_f(p) : f(a_1) = f(a_2)$. The constant path is denoted $1_a : a = a$.

A type A can be n-truncated, denoted $\mathrm{istrunc}_n\, A$, which is defined by recursion on $n : \mathbb{N}_{-2} := \mathbb{Z}_{\geq -2}$:

$$\begin{aligned} \mathrm{istrunc}_{-2}\, A &:= \mathrm{iscontr}\, A := (a : A) \times \big((x : A) \to (a = x)\big) \\ \mathrm{istrunc}_{n+1}\, A &:= (x\ y : A) \to \mathrm{istrunc}_n(x = y) \end{aligned}$$

For any type A we write $\|A\|_n$ for its n-truncation, i.e., $\|A\|_n$ is an n-truncated type equipped with a map $|-|_n : A \to \|A\|_n$ such that for any n-truncated type B the precomposition map

$$(\|A\|_n \to B) \to (A \to B)$$

is an equivalence. Then we define being n-connected as $\mathrm{isconn}_n\, A := \mathrm{iscontr}\|A\|_n$. Properties of truncations and connected maps are established in Chapter 7 of [25].

The type of pointed types is $\mathrm{Type}_{\mathrm{pt}} := (A : \mathrm{Type}) \times (\mathrm{pt} : A)$. The type of n-truncated types is $\mathrm{Type}^{\leq n} := (A : \mathrm{Type}) \times \mathrm{istrunc}_n\, A$ and for n-connected types it is $\mathrm{Type}^{>n} := (A : \mathrm{Type}) \times \mathrm{isconn}_n\, A$. We will combine these notations as needed.

Given $A : \mathrm{Type}_{\mathrm{pt}}$ we define the *loop space* $\Omega A := (\mathrm{pt} =_A \mathrm{pt})$, which is pointed with basepoint 1_{pt}. The *homotopy groups* of A are defined to be $\pi_k A := \|\Omega^k A\|_0$. These are group in the usual sense when $k \geq 1$, with neutral element $|1|$ and group operation induced by path concatenation.

Given $A, B : \mathrm{Type}_{\mathrm{pt}}$ the type of *pointed maps* from A to B is $(A \to_{\mathrm{pt}} B) := (f : A \to B) \times (f(\mathrm{pt}) =_B \mathrm{pt})$. Given $f : A \to_{\mathrm{pt}} B$ we write $f(a) : B$ for the first projection and $f_0 : f(\mathrm{pt}) = \mathrm{pt}$ for the second projection. The *fiber* of a pointed map is defined by $\mathrm{fib}(f) := (a : A) \times (f(a) =_B \mathrm{pt})$, which is pointed with basepoint (pt, f_0).

In HoTT we can use *higher inductive types* to construct Eilenberg-MacLane spaces $K(G, n)$ [17]. For a group G we define $K(G, 1)$ as the following HIT.

`HIT` $K(G, 1) :=$

- $\star : K(G, 1)$;
- $p : G \to \star = \star$;
- $q : (g\ h : G) \to p(gh) = p(g) \cdot p(h)$;
- $\epsilon : \mathrm{istrunc}_1\, K(G, 1)$.

(Using the univalent universe Type, other direct definitions are also possible, for instance, $K(G, 1)$ is equivalent to the type of small G-torsors.) Let ΣX denote the suspension of X, i.e., the homotopy pushout of $1 \leftarrow X \to 1$. For an abelian group A can now inductively define $K(A, n+1) := \|\Sigma K(A, n)\|_{n+1}$. Then we have the following result [17].

Theorem 1. *Let G be a group and $n \geq 1$, and assume that G is abelian when $n > 1$. The space $K(G, n)$ is $(n-1)$-connected and n-truncated and there is a group isomorphism $\pi_n K(G, n) \simeq G$.*

In some of our informal arguments we use the descent theorem for pushouts,[1] which states that for a commuting cube of types

$$\begin{tikzcd}
 & & A_{11} \arrow[dll] \arrow[d] \arrow[drr] & & \\
A_{10} \arrow[d] \arrow[drr] & & B_{11} \arrow[dll] \arrow[drr] & & A_{01} \arrow[dll] \arrow[d] \\
B_{10} \arrow[drr] & & A_{00} \arrow[d] & & B_{01} \arrow[dll] \\
 & & B_{00}, & &
\end{tikzcd} \qquad (1)$$

[1] Recall from [18, §6.1.3], following ideas from Charles Rezk, that we can *define* the ∞-toposes among locally cartesian closed ∞-categories as those whose colimits are *van Kampen*, viz., satisfying descent.

if the bottom square is a pushout and the vertical squares are pullbacks, then the top square is also a pushout. We will use the following slight generalization.

Theorem 2. *Consider a commuting cube of types as in* (1)*, and suppose the vertical squares are pullback squares. Then the square*

$$\begin{tikzcd}
A_{10} \sqcup^{A_{11}} A_{01} \arrow[r] \arrow[d] & A_{00} \arrow[d] \\
B_{10} \sqcup^{B_{11}} B_{01} \arrow[r] & B_{00}
\end{tikzcd}$$

is a pullback square.

Proof. It suffices to show that the pullback

$$(B_{10} \sqcup^{B_{11}} B_{01}) \times_{B_{00}} A_{00}$$

has the universal property of the pushout. This follows by the descent theorem, since by the pasting lemma for pullbacks we also have that the vertical squares in the cube

$$\begin{tikzcd}
 & & A_{11} \arrow[dll] \arrow[d] \arrow[drr] & & \\
A_{10} \arrow[d] \arrow[drr] & & B_{11} \arrow[dll, crossing over] \arrow[drr, crossing over] & & A_{01} \arrow[dll] \arrow[d] \\
B_{10} \arrow[drr] & & (B_{10} \sqcup^{B_{11}} B_{01}) \times_{B_{00}} A_{00} \arrow[d] & & B_{01} \arrow[dll] \\
 & & B_{10} \sqcup^{B_{11}} B_{01} & &
\end{tikzcd}$$

are pullback squares. □

In the formalization, arguments using descent are more conveniently done via the equivalent principle captured formally as the flattening lemma [25, §6.12].

3 Higher groups

Recall that types in HoTT may be viewed as ∞-groupoids: elements are objects, paths are morphisms, higher paths are higher morphisms, etc.

It follows that *pointed connected* types A may be viewed as higher groups, with *carrier* $\Omega A :\equiv (\mathrm{pt} =_A \mathrm{pt})$. The neutral element is the identity path, the group operation is given by path composition, and higher paths witness the unit and associativity laws. Of course, these higher paths are themselves subject to further laws, etc., but the beauty of the type-theoretic definition is that we don't have to worry about that: all the (higher) laws follow from the rules of the identity types.

Writing G for the carrier, it is common to write BG for the pointed connected type such that $G = \Omega BG$. We call BG the *delooping* of G. Let us write

$$\begin{aligned}\infty\text{-Group} &:\equiv (G : \mathrm{Type}) \times (BG : \mathrm{Type}^{>0}_{\mathrm{pt}}) \times (G \simeq \Omega BG)\\ &\simeq (G : \mathrm{Type}_{\mathrm{pt}}) \times (BG : \mathrm{Type}^{>0}_{\mathrm{pt}}) \times (G \simeq_{\mathrm{pt}} \Omega BG)\\ &\simeq \mathrm{Type}^{>0}_{\mathrm{pt}}\end{aligned}$$

for the type of higher groups, or ∞*-groups.* Note that for $G : \infty$-Group we also have $G : \mathrm{Type}$ using the first projection as a coercion. Using the last definition, this is the loop space map, and not the usual coercion!

We recover the ordinary set-level groups by requiring that G is a 0-type, or equivalently, that BG is a 1-type. This leads us to introduce

$$\begin{aligned}n\text{-Group} &:\equiv (G : \mathrm{Type}^{<n}_{\mathrm{pt}}) \times (BG : \mathrm{Type}^{>0}_{\mathrm{pt}}) \times (G \simeq_{\mathrm{pt}} \Omega BG)\\ &\simeq \mathrm{Type}^{>0,\leq n}_{\mathrm{pt}}\end{aligned}$$

for the type of *groupal* (group-like) $(n-1)$-groupoids, also known as n*-groups.* For G : 1-Group a set-level group, we have $BG = K(G, 1)$.

For example, the integers $\mathbb{Z}$ as an additive group are from this perspective represented by their delooping $\mathrm{B}\mathbb{Z} = \mathbb{S}^1$, i.e., the circle.

Of course, double loop spaces are even better behaved than mere loop spaces (e.g., they are commutative up to homotopy by the Eckmann-Hilton argument [25, Theorem 2.1.6]). Say a type G is k*-tuply groupal* if we have a k-fold delooping, $B^k G : \mathrm{Type}^{\geq k}_{\mathrm{pt}}$, such that $G = \Omega^k B^k G$.

Mixing the two directions, let us introduce the type

$$\begin{aligned}(n,k)\mathrm{GType} &:= (G : \mathrm{Type}_{\mathrm{pt}}^{\le n}) \times (B^k G : \mathrm{Type}_{\mathrm{pt}}^{\ge k}) \times (G \simeq_{\mathrm{pt}} \Omega^k B^k G) \\ &\simeq \mathrm{Type}_{\mathrm{pt}}^{\ge k, \le n+k} \qquad\qquad\qquad\qquad [\texttt{GType\_equiv}]\end{aligned}$$

for the type of *k-tuply groupal n-groupoids.*[2] (We allow taking $n = \infty$ in which case the truncation requirement is simply dropped. [`InfGType_equiv`])

Note that n-Group $= (n-1,1)$GType. This shift in indexing is slightly annoying, but we keep it to stay consistent with the literature.

Since there are forgetful maps

$$(n,k+1)\mathrm{GType} \to (n,k)\mathrm{GType}$$

given by $B^{k+1}G \mapsto \Omega B^{k+1}G$ we can also allow k to be infinite, $k = \omega$ by setting

$$\begin{aligned}(n,\omega)\mathrm{GType} &:= \lim\nolimits_k\, (n,k)\mathrm{GType} \\ &\simeq \big(B^- G : (k : \mathbb{N}) \to \mathrm{Type}_{\mathrm{pt}}^{\ge k, \le n+k}\big) \\ &\quad\times \big((k : \mathbb{N}) \to B^k G \simeq_{\mathrm{pt}} \Omega B^{k+1} G\big).\end{aligned}$$

In section 6 we prove the stabilization theorem (Theorem 6), from which it follows that (n,ω)GType $= (n,k)$GType for $k \ge n+2$.

When $(n,k) = (\infty,\omega)$, this is the type of stably groupal ∞-groups, also known as *connective spectra*. If we also relax the connectivity requirement, we get the type of all spectra, and we can think of a spectrum as a kind of ∞-groupoid with k-morphisms for all $k \in \mathbb{Z}$.

The class of higher groups is summarized in Table 1. We shall prove the correctness of the $n = 0$ column in section 5.

4 Elementary theory

Given *any* type of objects A, any $a : A$ has an *automorphism group* $\mathrm{Aut}_A\, a := \mathrm{Aut}\, a := (a = a)$ with $\mathrm{BAut}\, a = \mathrm{im}(a : 1 \to A) = (x : A) \times \|a = x\|_{-1}$ (the connected component of A at a). Clearly, if A is $(n+1)$-truncated, then so is $\mathrm{BAut}\, a$ and so $\mathrm{Aut}\, a$ is n-truncated, and hence an $(n+1)$-group.

[2] This is called $n\mathrm{Type}_k$ in [3], but here we give equal billing to n and k, and we add the "G" to indicate group-structure.

Moving across the homotopy hypothesis, for every pointed type (X, x) we have the *fundamental* ∞*-group of* X, $\Pi_\infty(X, x) :\equiv \operatorname{Aut} x$. Its $(n-1)$-truncation (an instance of decategorification, see section 6) is the *fundamental* n*-group of* X, $\Pi_n(X, x)$, with corresponding delooping $\mathrm{B}\Pi_n(X, x) = \|\mathrm{BAut}\, x\|_n$.

If we take $A = \mathrm{Set}$, we get the usual symmetric groups $S_n :\equiv \operatorname{Aut}(\operatorname{Fin} n)$, where $\operatorname{Fin} n$ is a set with n elements. (Note that $BS_n = \mathrm{BAut}(\operatorname{Fin} n)$ is the type of all n-element sets.) We give further constructions related to ordinary groups in section 7.

4.1 Homomorphisms and conjugation

A homomorphism between higher groups is any function that can be suitably delooped. For $G, H : (n, k)\mathrm{GType}$, we define

$$\begin{aligned}\hom_{(n,k)}(G, H) :\equiv (h : G \to_{\mathrm{pt}} H) &\times (B^k h : B^k G \to_{\mathrm{pt}} B^k H)\\ &\times (\Omega^k(B^k h) \sim_{\mathrm{pt}} h)\\ &\simeq (B^k h : B^k G \to_{\mathrm{pt}} B^k H).\end{aligned}$$

For (connective) spectra we need pointed maps between all the deloopings and pointed homotopies showing they cohere.

Note that if $h, k : G \to H$ are homomorphisms between set-level groups, then h and k are *conjugate* if $Bh, Bk : BG \to_{\mathrm{pt}} BH$ are *freely* homotopic (i.e., equal as maps $BG \to BH$).

Also observe that $\pi_j(B^k G \to_{\mathrm{pt}} B^k H) \simeq \|B^k G \to_{\mathrm{pt}} \Omega^k B^k H\|_0 \simeq \|\Sigma^j B^k G \to_{\mathrm{pt}} B^k H\|_0 = 0$ for $j + k - 1 \geq n + k$, that is, for $j > n$, so this suggests that $\hom_{(n,k)}(G, H)$ is n-truncated. (The calculation verifies this for the identity component.) To prove this, we need to use an induction using the definition of n-truncated. If $f : \hom_{(n,k)}(G, H)$, then its self-identity type is equivalent to $\big(\alpha : (z : B^k G) \to (f\, z = f\, z)\big) \times \big(\alpha\, \mathrm{pt} \cdot g_{\mathrm{pt}} = f_{\mathrm{pt}}\big)$. This type is no longer a type of pointed maps, but rather a type of pointed sections of a fibration of pointed types.

Definition 1. If $X : \mathrm{Type}_{\mathrm{pt}}$ and $Y : X \to \mathrm{Type}_{\mathrm{pt}}$, then we introduce the type of *pointed sections*,

$$(x : X) \to_{\mathrm{pt}} Y\, x :\equiv \big(s : (x : X) \to Y\, x\big) \times \big(s\, \mathrm{pt} = \mathrm{pt}\big).$$

This type is itself pointed by the trivial section $\lambda x, \mathrm{pt}$.

Theorem 3. *Let $X : \mathrm{Type}^{\geq k}_{\mathrm{pt}}$ be an $(k-1)$-connected, pointed type for some $k \geq 0$, and let $Y : X \to \mathrm{Type}^{\leq n+k}_{\mathrm{pt}}$ be a fibration of $(n+k)$-truncated, pointed types for some $n \geq -1$. Then the type of pointed sections, $(x : X) \to_{\mathrm{pt}} Y\, x$, is n-truncated.* [`is_trunc_ppi_of_is_conn`]

Proof. The proof is by induction on n.

For the base case $n = -1$ we have to show that the type of pointed sections is a mere proposition. Since it is pointed, it must in fact be contractible. The center of contraction is the trivial section s_0. If s is another section, then we get a pointed homotopy from s to s_0 from the elimination principle for pointed, connected types [25, Lemma 7.5.7], since the types $s\, x = s_0\, x$ are $(k-2)$-truncated.

To show the result for $n+1$, taking the n case as the induction hypothesis, it suffices to show for any pointed section s that its self-identity type is n-truncated. But this type is equivalent to $(x : X) \to_{\mathrm{pt}} \Omega(Y\, x, s\, x)$, which is again a type of pointed sections, and here we can apply the induction hypothesis. □

Corollary 1. *Let $k \geq 0$ and $n \geq -1$. If X is $(k-1)$-connected, and Y is $(n+k)$-truncated, then the type of pointed maps $X \to_{\mathrm{pt}} Y$ is n-truncated. In particular, $\hom_{(n,k)}(G, H)$ is an n-type for $G, H : (n,k)\mathrm{GType}$.*

Corollary 2. *The type $(n,k)\mathrm{GType}$ is $(n+1)$-truncated.* [`is_trunc_GType`]

Proof. This follows immediately from the preceding corollary, as the type of equivalences $G \simeq H$ is a subtype of the homomorphisms from G to H. □

If $k \geq n+2$ (so we're in the stable range), then $\hom_{(n,k)}(G, H)$ becomes a stably groupal n-groupoid. This generalizes the fact that the homomorphisms between abelian groups form an abelian group.

The automorphism group $\mathrm{Aut}\, G$ of a higher group $G : (n,k)\mathrm{GType}$ is in $(n,1)\mathrm{GType}$. This is equivalently the automorphism group of the pointed type $B^k G$. But we can also forget the basepoint and consider the automorphism group $\mathrm{Aut}^c\, G$ of $B^k G : \mathrm{Type}^{\geq k, \leq n+k}$. This now allows for (higher) conjugations. We define the *generalized center* of G to be $ZG :\equiv \Omega^k \mathrm{Aut}^c\, G : (n, k+1)\mathrm{GType}$ (generalizing the center of a set-level group, see below in subsection 4.3).

4.2 Group actions

In this section we consider a fixed group $G : \mathrm{GType}$ with delooping BG. An *action* of G on some object of type A is simply a function $X : BG \to A$. The object of the action is $X(\mathrm{pt}) : A$, and it can be convenient to consider evaluation at $\mathrm{pt} : BG$ to be a coercion from actions of type A to A. To equip $a : A$ with a G-action is to give an action $X : BG \to A$ with $X(\mathrm{pt}) = a$. The *trivial action* is the constant function at a. Clearly, an action of G on $a : A$ is the same as a homomorphism $G \to \mathrm{Aut}_A\, a$.

If A is a universe of types, then we have actions on types $X : BG \to \mathrm{Type}$. These G-types are thus simply types in the context of BG. A map of G-types from X to Y is just a function $\alpha : (z : BG) \to X(z) \to Y(z)$.

If X is a G-type, then we can form the

invariants $X^{hG} := (z : BG) \to X(z)$, also known as the *homotopy fixed points*, and the

coinvariants $X_{hG} := (z : BG) \times X(z)$, which is also known as *homotopy orbit space* or the *homotopy quotient* $X /\!/ G$.

It is easy to see that these constructions are respectively the right and left adjoints of the functor that sends a type X to the trivial G-action on X, $X^{\mathrm{triv}} : BG \to \mathrm{Type}$, which is just the constant family at X. Indeed, the adjunctions are just the usual argument swap and (un)currying equivalences, for $Y : \mathrm{Type}$,

$$\begin{aligned}
\hom(Y, X^{hG}) &= X \to (z : BG) \to Y(z) \simeq (z : BG) \to X \to Y(z) \\
&\simeq \hom(X^{\mathrm{triv}}, Y), \\
\hom(X_{hG}, Y) &= \big((z : BG) \times X(z)\big) \to Y \simeq (z : BG) \to X(z) \to Y \\
&\simeq \hom(X, Y^{\mathrm{triv}}).
\end{aligned}$$

If we think of an action $X : BG \to \mathrm{Type}$ as a type-valued diagram on BG, this means that the homotopy fixed points and the homotopy orbit space form the homotopy limit and homotopy colimit of this diagram, respectively.

Proposition 1. *Let $f : H \to G$ be a homomorphism of higher groups with delooping $Bf : BH \to_{\mathrm{pt}} BG$, and let $\alpha : \hom(X, Y)$ be a map of G-types. By composing with f we can also view X and Y as H-types, in which case we* *get a homotopy pullback square:*

$$\begin{tikzcd} X_{hH} \arrow[r] \arrow[d] & Y_{hH} \arrow[d] \\ X_{hG} \arrow[r] & Y_{hG}. \end{tikzcd}$$

Proof. The vertical maps are induced by Bf, and the horizontal maps are induced by α. The homotopy pullback corner type C is calculated as

$$\begin{aligned} C &\simeq (z : BG) \times (x : X\,z) \times (w : BH) \times (y : Y(Bf\,w)) \\ &\qquad \times (z = Bf\,w) \times (y = \alpha\,z\,x) \\ &\simeq (w : BH) \times (x : X(Bf\,w)) = X_{hH}, \end{aligned}$$

and under this equivalence the top and the left maps are the canonical ones. □

Every group G carries two canonical actions on itself:

the right action $G : BG \to \text{Type}$, $G(x) = (\text{pt} = x)$, and the

the adjoint action $G^{\text{ad}} : BG \to \text{Type}$, $G^{\text{ad}}(x) = (x = x)$ (by conjugation).

We have $1 \mathbin{/\!/} G = BG$, $G \mathbin{/\!/} G = 1$ and $G^{\text{ad}} \mathbin{/\!/} G = LBG := (\mathbb{S}^1 \to BG)$, the free loop space of BG. Recalling that $\mathrm{B}\mathbb{Z} = \mathbb{S}^1$, we see that $G^{\text{ad}} = (\mathrm{B}\mathbb{Z} \to BG)$, i.e., the conjugacy classes of homomorphisms from $\mathbb{Z}$ to G. Since the integers are the free (higher) group on one generator, this is just the conjugacy classes of elements of G. But that is exactly what we should get for the homotopy orbits of G under the conjugation action.

The above proposition has an interesting corollary:

Corollary 3. *If $f : H \to G$ is a homomorphism of higher groups, then $G \mathbin{/\!/} H$ is equivalent to the homotopy fiber of the delooping $Bf : BH \to_{\text{pt}} BG$, where H acts on G via the f-induced right action.*

Proof. We apply Proposition 1 with $\alpha : G \to 1$ being the canonical map from the right action of G to the action of G on the unit type. Then the square becomes:

$$\begin{tikzcd} G \mathbin{/\!/} H \arrow[r] \arrow[d] & BH \arrow[d] \\ 1 \arrow[r] & BG \end{tikzcd}$$

□

By definition, BG classifies *principal G-bundles*: pullbacks of the right action of G. That is, a principal G-bundle over a type A is a family $F : X \to \mathrm{Type}$ represented by a map $\chi : A \to BG$ such that $F(x) \simeq (\mathrm{pt} = \chi(x))$ for all $x : X$.

For example, for every higher group G we have the corresponding Hopf fibration $\Sigma G \to \mathrm{Type}$ represented by the map $\chi_H : \Sigma G \to BG$ corresponding under the loop-suspension adjunction to the identity map on G. (This particular fibration can be defined using only the induced H-space structure on G.)

This perspective underlies the construction of the first and the third named author of the real projective spaces in homotopy type theory [5]. The fiber sequences $\mathbb{S}^0 \to \mathbb{S}^n \to \mathbb{R}\mathrm{P}^n$ are principal bundles for the 2-elements group $\mathbb{S}^0 = S_2$ with delooping $BS_2 \simeq \mathbb{R}\mathrm{P}^\infty$, the type of 2-element types.

4.3 Back to the center

We mentioned the generalized center above and claimed that it generalized the usual notion of the center of a group. Indeed, if G : 1-Group is a set-level group, then an element of ZG corresponds to an element of $\Omega^2 \,\mathrm{BAut}^c\, G$, or equivalently, a map from the 2-sphere $\mathbb{S}^2$ to Type sending the basepoint to BG. By the universal property of $\mathbb{S}^2$ as a HIT, this again corresponds to a homotopy from the identity on BG to itself, $c : (z : BG) \to z = z$. This is precisely a homotopy fixed point of the adjoint action of G on itself, i.e., a central element.

4.4 Equivariant homotopy theory

Fix a group G : GType. Suppose that G is actually the (homotopy) type of a topological group. Consider the type $BG \to \mathrm{Type}$ of (small) *types with a G-action.* Naively, one might think that this represents G-equivariant homotopy types, i.e., sufficiently nice[3] topological spaces with a G-action considered up to G-equivariant homotopy equivalence. But this is not so.

By Elmendorf's theorem [12], this homotopy theory is rather that of presheaves of (ordinary) homotopy types on the *orbit category* $\mathcal{O}_G$ of G.

[3] Sufficiently nice means the G-CW-spaces. The same homotopy category arises by taking all spaces with a G-action, but then the weak equivalences are the G-maps $f : X \to Y$ that induce weak equivalences on H-fixed point spaces $f^H : X^H \to Y^H$ for all closed subgroups H of G.

This is the full subcategory of the category of G-spaces spanned by the homogeneous spaces G/H, where H ranges over the closed subgroups of G.

Inside the orbit category we find a copy of the group G, namely as the endomorphisms of the object $G/1$ corresponding to the trivial subgroup 1. Hence, a G-equivariant homotopy type gives rise to type with a G-action by restriction along the inclusion $BG \hookrightarrow \mathcal{O}_G$. (Here we consider BG as a (pointed and connected) topological groupoid on one object.)

As remarked by Shulman [23], when G is a *compact* Lie group, then $\mathcal{O}_G$ is an inverse EI ∞-category, and hence we know how to model type theory in the presheaf ∞-topos over $\mathcal{O}_G$. And in certain simple cases we can even define this model internally. For instance, if $G = \mathbb{Z}/p\mathbb{Z}$ is a cyclic group of prime order, then a small G-equivariant type consists of a type with a G-action, $X : BG \to \text{Type}$ together with another type family $X^G : X^{hG} \to \text{Type}$, where X^G gives for each homotopy fixed point a type of proofs or "special reasons" why that point should be considered fixed [23, p. 7.6]. Hence the total space of X^G is the type of actual fixed points, and the projection to X^{hG} implements the map from actual fixed points to homotopy fixed points.

Even without going to the orbit category, we can say something about topological groups through their classifying types in type theory. For example [6], if $f : H \to G$ is injective, then the homotopy fiber of Bf is by Corollary 3 is the homotopy orbit space $G /\!/ H$, which in this case is just the coset space G/H, and hence in type theory represents the homotopy type of this coset space. And if

$$1 \to K \to G \to H \to 1$$

is a short exact sequence of topological groups, then $BK \to BG \to BH$ is a fibration sequence, i.e., we can recover the delooping BK of K as the homotopy fiber of the map $BG \to BH$.

4.5 Some elementary constructions

If we are given a homomorphism $\varphi : H \to \text{Aut}(N)$, represented by a pointed map $B\varphi : BH \to_{\text{pt}} \text{BAut}_{\text{pt}}(BN)$ where $\text{BAut}_{\text{pt}}(BN)$ is the type of pointed types merely equivalent to BN, we can build a new group, the *semidirect product*, $G := H \ltimes_\varphi N$ with classifying type $BG := (z : BH) \times (B\varphi\, z)$. The type BG is indeed pointed (by the pair of the basepoint pt in BG and the basepoint in the pointed type $B\varphi(\text{pt})$), and connected, and hence presents a higher group G. An element of g is given by a pair of an element $h : H$ and an identification $g \cdot \mathrm{pt} = \mathrm{pt}$ in $B\varphi(\mathrm{pt}) \simeq_{\mathrm{pt}} BN$. But since the action is via pointed maps, the second component is equivalently an identification $\mathrm{pt} = \mathrm{pt}$ in BN, i.e., an element of N. Under this equivalence, the product of (h, n) and (h', n') is indeed $(h \cdot h', n \cdot \varphi(h)(n'))$.

As a special case we obtain the *direct product* when φ is the trivial action. Here, $B(H \times N) \simeq BH \times BN$.

As another special case we obtain the *wreath products* $N \wr S_n$ of a group N and a symmetric group S_n. Here, S_n acts on the direct power $N^{\mathrm{Fin}\,n}$ by permuting the factors. Indeed, using the representation of BS_n as the type of n-element types, the map $B\varphi$ is simply $A \mapsto (A \to BN)$. Hence the delooping of the wreath product $G := N \wr S_n$ is just $BG := (A : BS_n) \times (A \to BN)$.

5 Set-level groups

In this section we give a proof that the $n = 0$ column of Table 1 is correct. Note that for $n = 0$ the hom-types $\hom_{(0,k)}(G, H)$ are sets, which means that $(0, k)$GType forms a 1-category. Let Group be the category of ordinary set-level groups (a set with multiplication, inverse and unit satisfying the group laws) and AbGroup the category of abelian groups.

Theorem 4. *We have the following equivalences of categories (for $k \geq 2$):*

$$(0,1)\mathrm{GType} \simeq \mathrm{Group}; \qquad [\texttt{cGType\_equivalence\_Grp}]$$
$$(0,k)\mathrm{GType} \simeq \mathrm{AbGroup}. \qquad [\texttt{cGType\_equivalence\_AbGrp}]$$

Since this theorem has been formalized we will not give all details of the proof.

Proof. Let $k \geq 1$ and G be a group which is abelian if $k > 1$ and let $X : \mathrm{Type}_{\mathrm{pt}}^{\geq k, \leq k}$. If we have a group homomorphism $\varphi : G \to \Omega^k X$ we get a map $e^k_\varphi : K(G, k) \to_{\mathrm{pt}} X$. For $k = 1$ this follows directly from the induction principle of $K(G, 1)$. For $k > 1$ we can define the group homomorphism $\widetilde{\varphi}$ as the composite $G \xrightarrow{\varphi} \Omega^k X \simeq \Omega^{k-1}(\Omega X)$, and apply the induction hypothesis to get a map $e^{k-1}_{\widetilde{\varphi}} : K(G, k-1) \to_{\mathrm{pt}} \Omega X$. By the adjunction $\Sigma \dashv \Omega$ we get a pointed map $\Sigma K(G, k-1) \to_{\mathrm{pt}} X$, and by the elimination principle of the truncation we get a map $K(G, k) = \|\Sigma K(G, k-1)\|_k \to_{\mathrm{pt}} X$.

We can now show that $\Omega^k e^k_\varphi$ is the expected map, that is, the following diagram commutes, but we omit this proof here.

$$\begin{tikzcd}
\Omega^n K(G,k) \arrow[rr, "\sim"] \arrow[dr, "\Omega^n e^k_\varphi"'] & & G \\
& \Omega^n X \arrow[ur, "\varphi"'] &
\end{tikzcd}$$

Now if φ is a group isomorphism, by Whitehead's Theorem for truncated types [25, Theorem 8.8.3] we know that e^k_φ is an equivalence, since it induces an equivalence on all homotopy groups (trivially on the levels other than k). We can also show that e^k_φ is natural in φ.

Note that if we have a group homomorphism $\psi : G \to G'$, we also get a group homomorphism $G \to \Omega^k K(G',k)$, and by the above construction we get a pointed map $K(\psi,k) : K(G,k) \to_{\mathrm{pt}} K(G',k)$. This is functorial, which follows from naturality of e^k_φ.

Finally, we can construct the equivalence explicitly. We have a functor $\pi_k : (0,k)\mathrm{GType} \to \mathrm{AbGroup}$ which sends G to $\pi_k BG$. Conversely, we have the functor $K(-,k) : \mathrm{AbGroup} \to (0,k)\mathrm{GType}$. We have natural isomorphisms $\pi_k K(G,k) \simeq G$ by Theorem 1 and $K(\pi_k X,k) \simeq_{\mathrm{pt}} X$ by the application of Whitehead described above. The construction is exactly the same for $k = 1$ after replacing AbGroup by Group. □

6 Stabilization

In this section we discuss some constructions with higher groups [3]. We will give the actions on the carriers and the deloopings, but we omit the third component, the pointed equivalence, for readability. We recommend keeping Table 1 in mind during these constructions.

decategorification $\mathrm{Decat} : (n,k)\mathrm{GType} \to (n-1,k)\mathrm{GType}$
$\langle G, B^k G\rangle \mapsto \langle \|G\|_{n-1}, \|B^k G\|_{n+k-1}\rangle$

discrete categorification $\mathrm{Disc} : (n,k)\mathrm{GType} \to (n+1,k)\mathrm{GType}$
$\langle G, B^k G\rangle \mapsto \langle G, B^k G\rangle$

These functors make $(n,k)\mathrm{GType}$ a reflective sub-$(\infty,1)$-category of $(n+$

$1,k)$GType. That is, there is an adjunction $\mathrm{Decat} \dashv \mathrm{Disc}$ [Decat_adjoint_Disc][4] such that the counit induces an isomorphism $\mathrm{Decat} \circ \mathrm{Disc} = \mathrm{id}$ [Decat_Disc]. These properties are straightforward consequences of the universal property of truncation.

There are also iterated versions of these functors.

∞-decategorification $\infty\text{-Decat} : (\infty,k)\mathrm{GType} \to (n,k)\mathrm{GType}$
$\langle G, B^kG\rangle \mapsto \langle \|G\|_n, \|B^kG\|_{n+k}\rangle$

discrete ∞-categorification $\infty\text{-Disc} : (n,k)\mathrm{GType} \to (\infty,k)\mathrm{GType}$
$\langle G, B^kG\rangle \mapsto \langle G, B^kG\rangle$

These functors satisfy the same properties: $\infty\text{-Decat} \dashv \infty\text{-Disc}$ [InfDecat_adjoint_InfDisc] such that the counit induces an isomorphism $\infty\text{-Decat}\circ\infty\text{-Disc} = \mathrm{id}$ [InfDecat_InfDisc].

For the next constructions, we need the following properties.

Definition 2. For $A : \mathrm{Type}_{\mathrm{pt}}$ we define the *n-connected cover* of A to be $A\langle n\rangle := \mathrm{fib}(A \to \|A\|_n)$. We have the projection $p_1 : A\langle n\rangle \to_{\mathrm{pt}} A$.

Lemma 1. *The universal property of the n-connected cover states the following. For any n-connected pointed type B, the pointed map*

$$(B \to_{\mathrm{pt}} A\langle n\rangle) \to_{\mathrm{pt}} (B \to_{\mathrm{pt}} A),$$

given by postcomposition with p_1, is an equivalence. [connect_intro_pequiv]

Proof. Given a map $f : B \to_{\mathrm{pt}} A$, we can form a map $\widetilde{f} : B \to A\langle n\rangle$. First note that for $b : B$ the type $|fb|_n =_{\|A\|_n} |\mathrm{pt}|_n$ is $(n-1)$-truncated and inhabited for $b = \mathrm{pt}$. Since B is n-connected, the universal property for connected types shows that we can construct a $qb : |fb|_n = |\mathrm{pt}|_n$ for all b such that $q_0 : qb_0 \cdot \mathrm{ap}_{|-|_n}(f_0) = 1$. Then we can define the map $\widetilde{f}(b) := (fb, qb)$. Now $\widetilde{f}$ is pointed, because $(f_0, q_0) : (fb_0, qb_0) = (a_0, 1)$.

Now we show that this is indeed an inverse to the given map. On the one hand, we need to show that if $f : B \to_{\mathrm{pt}} A$, then $p_1 \circ \widetilde{f} = f$. The underlying functions are equal because they both send b to $f(b)$. They respect points in the same way, because $\mathrm{ap}_{p_1}(\widetilde{f_0}) = f_0$. The proof that the other composite is the identity follows from a computation using fibers and connectivity, which we omit here, but can be found in the formalization. □

[4] In the formalization the naturality of the adjunction is a separate statement, [Decat_adjoint_Disc_natural]. This is also true for the other adjunctions.

The next reflective sub-$(\infty,1)$-category is formed by looping and delooping.

looping $\Omega : (n,k)\mathrm{GType} \to (n-1,k+1)\mathrm{GType}$
$\langle G, B^kG\rangle \mapsto \langle \Omega G, B^kG\langle k\rangle\rangle$

delooping $\mathrm{B} : (n,k)\mathrm{GType} \to (n+1,k-1)\mathrm{GType}$
$\langle G, B^kG\rangle \mapsto \langle \Omega^{k-1}B^kG, B^kG\rangle$

We have $\mathrm{B} \dashv \Omega$ [`Deloop_adjoint_Loop`], which follows from Lemma 1 and $\Omega \circ \mathrm{B} = \mathrm{id}$ [`Loop_Deloop`], which follows from the fact that $A\langle n\rangle = A$ if A is n-connected.

The last adjoint pair of functors is given by stabilization and forgetting. This does not form a reflective sub-$(\infty,1)$-category.

forgetting $F : (n,k)\mathrm{GType} \to (n,k-1)\mathrm{GType}$
$\langle G, B^kG\rangle \mapsto \langle G, \Omega B^kG\rangle$

stabilization $S : (n,k)\mathrm{GType} \to (n,k+1)\mathrm{GType}$
$\langle G, B^kG\rangle \mapsto \langle SG, \|\Sigma B^kG\|_{n+k+1}\rangle$,
where $SG = \|\Omega^{k+1}\Sigma B^kG\|_n$

We have the adjunction $S \dashv F$ [`Stabilize_adjoint_Forget`] which follows from the suspension-loop adjunction $\Sigma \dashv \Omega$ on pointed types.

The next main goal in this section is the stabilization theorem, stating that the ditto marks in Table 1 are justified.

The following corollary is almost [25, Lemma 8.6.2], but proving this in Book HoTT is a bit tricky. See the formalization for details.

Lemma 2 (Wedge connectivity). *If $A : \mathrm{Type}_{\mathrm{pt}}$ is n-connected and $B : \mathrm{Type}_{\mathrm{pt}}$ is m-connected, then the map $A \vee B \to A \times B$ is $(n+m)$-connected.* [`is_conn_fun_prod_of_wedge`]

Let us mention that there is an alternative way to prove the wedge connectivity lemma: Recall that if A is n-connected and B is m-connected, then $A * B$ is $(n+m+2)$-connected [20, Theorem 6.8]. Hence the wedge connectivity lemma is also a direct consequence of the following lemma.

Lemma 3. *Let A and B be pointed types. The fiber of the wedge inclusion $A \vee B \to A \times B$ is equivalent to $\Omega A * \Omega B$.*

Proof. Note that the fiber of $A \to A \times B$ is ΩB, the fiber of $B \to A \times B$ is ΩA, and of course the fiber of $1 \to A \times B$ is $\Omega A \times \Omega B$. We get a commuting cube

$$\begin{tikzcd}
 & & \Omega A \times \Omega B \arrow[dll] \arrow[d] \arrow[drr] & & \\
\Omega B \arrow[d] \arrow[drr] & & 1 \arrow[dll] \arrow[drr] & & \Omega A \arrow[dll] \arrow[d] \\
A \arrow[drr] & & 1 \arrow[d] & & B \arrow[dll] \\
 & & A \times B & &
\end{tikzcd}$$

in which the vertical squares are pullback squares.

By the descent theorem for pushouts it now follows that $\Omega A * \Omega B$ is the fiber of the wedge inclusion. □

The second main tool we need for the stabilization theorem is:

Theorem 5 (Freudenthal). *If $A : \mathrm{Type}_{\mathrm{pt}}^{>n}$ with $n \geq 0$, then the map $A \to \Omega\Sigma A$ is $2n$-connected.*

This is [25, Theorem 8.6.4].

The final building block we need is:

Lemma 4. *There is a pullback square*

$$\begin{tikzcd}
\Sigma\Omega A \arrow[r] \arrow[d, "\varepsilon_A"'] & A \vee A \arrow[d] \\
A \arrow[r, "\Delta"'] & A \times A
\end{tikzcd}$$

for any $A : \mathrm{Type}_{\mathrm{pt}}$.

Proof. Note that the pullback of $\Delta : A \to A \times A$ along either inclusion

$A \to A \times A$ is contractible. So we have a cube

$$\begin{tikzcd}
 & \Omega A \arrow[dl] \arrow[d] \arrow[dr] & \\
1 \arrow[d] \arrow[dr] & 1 \arrow[dl] \arrow[dr] & 1 \arrow[dl] \arrow[d] \\
A \arrow[dr] & A \arrow[d, "\Delta"] & A \arrow[dl] \\
 & A \times A &
\end{tikzcd}$$

in which the vertical squares are all pullback squares. Therefore, if we pull back along the wedge inclusion, we obtain by the descent theorem for pushouts that the square in the statement is indeed a pullback square. □

Theorem 6 (Stabilization)**.** *If* $k \geq n+2$, *then* $S : (n,k)\mathrm{GType} \to (n,k+1)\mathrm{GType}$ *is an equivalence, and any* $G : (n,k)\mathrm{GType}$ *is an infinite loop space.* [`stabilization`]

Proof. We show that $F \circ S = \mathrm{id} = S \circ F : (n,k)\mathrm{GType} \to (n,k)\mathrm{GType}$ whenever $k \geq n+2$.

For the first, the unit map of the adjunction factors as

$$B^kG \to \Omega\Sigma B^kG \to \Omega\|\Sigma B^kG\|_{n+k+1}$$

where the first map is $2k-2$-connected by Freudenthal, and the second map is $n+k$-connected. Since the domain is $n+k$-truncated, the composite is an equivalence whenever $2k-2 \geq n+k$.

For the second, the counit map of the adjunction factors as

$$\|\Sigma\Omega B^kG\|_{n+k} \to \|B^kG\|_{n+k} \to B^kG,$$

where the second map is an equivalence. By the two lemmas above, the first map is $2k-2$-connected. □

For example, for $G : (0,2)\mathrm{GType}$ an abelian group, we have $B^nG = K(G,n)$, an Eilenberg-MacLane space.

The adjunction $S \dashv F$ implies that the free group on a pointed set X is $\Omega\|\Sigma X\|_1 = \pi_1(\Sigma X)$. If X has decidable equality, ΣX is already 1-truncated. It is an open problem whether this is true in general.

Also, the abelianization of a set-level group $G : 1\text{-Group}$ is $\pi_2(\Sigma BG)$. If $G : (n,k)\mathrm{GType}$ is in the stable range ($k \geq n+2$), then $SFG = G$.

7 Perspectives on ordinary group theory

In this section we shall indicate how the theory of higher groups can yield a new perspective even on ordinary group theory.

From the symmetric groups S_n, we can get other finite groups using the constructions of subsection 4.5. Other groups can be constructed more directly. For example, BA_n, the classifying type of the alternating group, can be taken to be the type of n-element sets X equipped with a *sign ordering*: this is an equivalence class of an ordering $\mathrm{Fin}\, n \simeq X$ modulo even permutations. Indeed, there are only two possible sign orderings, so this definition corresponds to first considering the short exact sequence

$$1 \to A_n \to S_n \xrightarrow{\mathrm{sgn}} S_2 \to 1$$

where the last map is the sign map, then realizing the sign map as given by the map $\mathrm{Bsgn} : BS_n \to BS_2$ that takes an n-element set to its set of sign orderings, and finally letting BA_n be the homotopy fiber of Bsgn.

Similarly, BC_n, the classifying type of the cyclic group on n elements, can be taken to be the type of n-elements sets X equipped with a *cyclic ordering*: an equivalence class of an ordering $\mathrm{Fin}\, n \simeq X$ modulo cyclic permutations. But unlike the above, where we had the coincidence that $\mathrm{Aut}(S_2) \simeq S_2$, this doesn't corresponds to a short exact sequence. Rather, it corresponds to a sequence

$$1 \to C_n \to S_n \to \mathrm{Aut}(\mathrm{Fin}(n-1)) \simeq S_{(n-1)!}$$

where the delooping of the last map is the map from BS_n to $BS_{(n-1)!}$ that maps an n-element set to the set of cyclic orderings, of which there are $(n-1)!$ many – since once we fix the position in the ordering of a particular element, we are free to permute the rest.

As another example, consider the map $p : BS_4 \to_{\mathrm{pt}} BS_3$ that maps a 4-element set X to its set of 2-by-2 partitions, of which there 3. Using this construction, we can realize some famous semidirect and wreath product identities, such as $A_4 \simeq S_2^2 \rtimes A_3$, $S_4 \simeq S_2^2 \rtimes S_3$, and, for the octahedral group, $O_h \simeq S_2^3 \rtimes S_3 \simeq S_2 \wr S_3$.

Let us turn to a different way of getting new groups from old, namely via covering space theory.

7.1 1-groups and covering spaces

The connections between covering spaces of a pointed connected type X and sets with an action of the fundamental group of X has already been established in homotopy type theory [15]. Let us recall this connection and expand a bit upon it.

For us, a pointed connected type X is equivalently an ∞-group $G : \infty\text{-Group}$ with delooping $BG := X$. A covering space over BG is simply a type family $C : BG \to \mathrm{Set}$ that lands in the universe of sets. Hence by our discussion of actions in subsection 4.2 it is precisely a set with a G-action. Since Set is a 1-type, C extends uniquely to a type family $C' : \|BG\|_1 \to \mathrm{Set}$, but $\|BG\|_1$ is the delooping of the fundamental group of X, and hence C' is the uniquely determined choice of a set with an action of the fundamental group.

The universal covering space is the simply connected cover of BG,

$$\widetilde{BG} : BG \to \mathrm{Set}, \quad z \mapsto \|\mathrm{pt} = z\|_0.$$

Note that the total space of $\widetilde{BG}$ is indeed the 1-connected cover $BG\langle 1\rangle$, since $\|\mathrm{pt} =_{BG} \mathrm{pt}\|_0 \simeq (|\mathrm{pt}| =_{\|BG\|_1} |\mathrm{pt}|)$. Also note that if G is already a 1-group, then this is just the right action of G on itself, and in general, it is the right action of G on the fundamental group (i.e., the decategorification of G) via the truncation homomorphism from G to $\pi_1(BG)$, where we can also view $\pi_1(BG)$ as the 1-Group decategorification of G.

In general, there is a Galois correspondence between connected covers of BG and conjugacy class of subgroups of the fundamental group. Indeed, if $C : BG \to \mathrm{Set}$ has a connected total space, then the space $(g : \|BG\|_1) \times C'(g)$ is itself a connected, 1-truncated type, and the projection to $\|BG\|_1$ induced an inclusion of fundamental groups once a point $\mathrm{pt} : C'(\mathrm{pt})$ has been chosen.

Theorem 7 (Fundamental theorem of Galois theory for covering spaces)**.**

1. *The automorphism group of the universal covering space $\widetilde{BG}$ is isomorphic to the* 1*-group decategorification of* G*,*

$$\mathrm{Aut}(\widetilde{BG}) \simeq \mathrm{Decat}_1(G) \simeq \pi_1(BG).$$

2. *Furthermore, there is an contravariant correspondence between conjugacy classes of subgroups of* $\mathrm{Decat}_1(G)$ *and connected covers of* BG*.*

3. *This lifts to a Galois correspondence between subgroups of* $\mathrm{Decat}_1(G)$ *and pointed, connected covers of* BG*. The normal subgroups correspond to Galois covers.*

Note that the universal covering space and the trivial covering space (constant at the unit type) are canonically pointed, reflecting the fact that the two trivial subgroups are normal.

The first part of the fundamental theorem has a clear generalization to higher groups:

Theorem 8 (Fundamental theorem of Galois theory for n-covers, part one)**.** *The automorphism group of the universal* n*-type cover* $U_n(BG)$*,*

$$U_n(BG) : BG \to \mathrm{Type}^{\leq n}, \quad z \mapsto \|\mathrm{pt} = z\|_n$$

of BG *is isomorphic to the* $(n+1)$*-group decategorification of* G*,*

$$\mathrm{Aut}(U_n(BG)) \simeq \mathrm{Decat}_{n+1}(G) \simeq \Pi_{n+1}(BG).$$

Proof. Note that $\mathrm{BAut}(U_n(BG))$ is the image of the map $1 \to (BG \to \mathrm{Type}^{\leq n})$ that sends the canonical element to $U_n(BG)$. Since BG is connected, this image is exactly $\|BG\|_{n+1}$ by [20, Theorem 7.1]. Then we are done, since $\mathrm{B}\,\Pi_{n+1}(BG) \simeq \|BG\|_{n+1}$, by definition. □

It is possible to use the other parts of Theorem 7 in order to *define* the notions of subgroup and normal subgroup for n-groups, which then become *structure on* rather than a *property of* a homomorphism $f : K \to G$. Explicitly, the structure of a *normal subgroup* on such an f is a delooping $B(G \mathbin{/\!/} K)$ of the type $G \mathbin{/\!/} K$ together with a map $Bq : BG \to_{\mathrm{pt}} B(G \mathbin{/\!/} K)$ giving rise to a fiber sequence

$$G \mathbin{/\!/} K \to BK \xrightarrow{Bf} BG \xrightarrow{Bq} B(G \mathbin{/\!/} K). \tag{2}$$

7.2 Central extensions and group cohomology

The cohomology of a higher group G is simply the cohomology of its delooping BG. Indeed, for any spectrum A, we define

$$H^k_{\mathrm{Grp}}(G, A) :\equiv \|BG \to_{\mathrm{pt}} B^k A\|_0.$$

Of course, to define the k'th cohomology group, we only need the k-fold delooping $B^k A$.

If $A : (\infty, 2)\text{GType}$ is a braided ∞-group, then we have the second cohomology group $H^2_{\text{Grp}}(G, A)$, and an element $c : BG \to_{\text{pt}} B^2 A$ gives rise to a *central extension*

$$BA \to BH \to BG \xrightarrow{c} B^2 A,$$

where BH is the homotopy fiber of c. This lifts to the world of higher groups the usual result that isomorphism classes of central extensions of a 1-group G by an abelian 1-group A are given by cohomology classes in $H^2_{\text{Grp}}(G, A)$.

In the Spectral repository there is full formalization of the Serre spectral sequence for cohomology [8]. If we have any normal subgroup fiber sequence for ∞-groups as in (2), then we get a corresponding spectral sequence with E_2-page

$$H^p_{\text{Grp}}(G /\!/ K, H^q_{\text{Grp}}(K, A))$$

and converging to $H^n_{\text{Grp}}(G, A)$, where A is any truncated, connective spectrum, which could even be a left G-module, in which case we reproduce the *Hochschild-Serre spectral sequence*.

8 Formalization

We have formalized many results of this paper. We use the proof assistant Lean 2[5]. This is an older version of the proof assistant Lean[6] (version 3.3 as of January 2018). We use the old version, since the newer version doesn't officially support HoTT, although there is an experimental library for HoTT[7], but that doesn't have as much theory as the library in Lean 2.

The Lean 2 HoTT library is divided in two parts, the core library[8] and the formalization of spectral sequences[9]. We worked in the latter, so that we could use the results from that repository, such as theorems about Eilenberg-MacLane spaces and pointed maps. All results in this paper are stated in one file[10], although for many results the main parts of the proof is elsewhere (in Emacs, click on a name and press `M-.` to find a definition).

[5] `https://github.com/leanprover/lean2`

[6] `https://leanprover.github.io/`

[7] `https://github.com/gebner/hott3`

[8] `https://github.com/leanprover/lean2/blob/master/hott/hott.md`

[9] `https://github.com/cmu-phil/Spectral`

[10] `https://github.com/cmu-phil/Spectral/blob/master/higher_groups.hlean`

To build the file, install Lean 2 via the instructions from that repository, and then download the Spectral repository and compile it (you can use the command `path/to/lean2/bin/linja` on the command-line to compile the library you're in). The Spectral repository contains some unproven results, marked by **sorry**. You can write `print axioms theoremname` in a file to ensure that **sorry** isn't used in the proof.

9 Conclusion

We have presented a theory and formalization of higher groups in HoTT, and we have proved that for set-level structures we recover the well-known objects: groups and abelian groups. A possible next step would be to do the same for the 1-type objects. The corresponding algebraic objects have a long history. Strict 2-groups predate category theory as they originate in Whitehead's study of *crossed modules* [27]. The theory of weak 2-groups was begun by Grothendieck's student Hoàng Xuân Sính [24] and further developed in [4]. It should be possible to prove within HoTT that weak 2-groups and crossed modules are equivalent to 2-groups in our sense, when we use the respective, correct notions of equivalence.

Symmetric 2-groups are by the stabilization theorem the same as 1-truncated symmetric spectra. These are described more simply than arbitrary crossed modules as *Picard groupoids*. This is part of the stable homotopy hypothesis [14, 16]. It should also be possible to develop the theory of Picard groupoids in HoTT, and thus prove the corresponding stable homotopy hypothesis.

Higher groups have been intensively studied in homotopy theory, in particular after p-completion for p a prime. A *p-compact group* is an $\mathbb{F}_p$-local ∞-group whose carrier is $\mathbb{F}_p$-finite, see [10]. They are good homotopical analogues of Lie groups, and they interact nicely with compact Lie groups, for instance:

Theorem 9 ([11])**.** *Let P be a p-toral group, and let G be a compact Lie group. Then $\|BP \to_{\mathrm{pt}} BG\|_0$ is isomorphic to the conjugacy classes of homomorphisms from P to G.*

Higher groups also play a particularly prominent role in the development of quantum field theory in cohesive homotopy type theory [21]. In cohesive type theory we can actually capture the topological or smooth structure of groups and their classifying types, and hence develop Lie theory properly, including the higher group generalization thereof. All of our results only use the core part of HoTT, and hence they remain valid also in cohesive HoTT.

Note that we have crucially used a trick to study higher groups in HoTT, namely that these can be represented by pointed, connected types. The alternative would have been to define them as group-like algebras for the little k-cubes operad E_k. But this requires exactly the kind of infinitary tower of coherence conditions that we don't yet know how to define in HoTT. (Or whether it is even possible.) Thus, while we have the type of higher groups, we do not have the type of higher monoids (general E_k-algebras). Thus their theory, and the corresponding stabilization theorem, is currently beyond the reach of HoTT.

Acknowledgement

The authors gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

[1] Carlo Angiuli, Robert Harper, and Todd Wilson. "Computational Higher-dimensional Type Theory". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. New York, NY, USA: ACM, 2017, pp. 680–693. DOI: 10.1145/3009837.3009861.

[2] Steve Awodey and Michael A. Warren. "Homotopy theoretic models of identity types". In: *Math. Proc. Cambridge Philos. Soc.* 146.1 (2009), pp. 45–55. DOI: 10.1017/S0305004108001783. arXiv: 0709.0248.

[3] John C. Baez and James Dolan. "Categorification". In: *Higher category theory (Evanston, IL, 1997)*. Vol. 230. Contemp. Math. Providence, RI: Amer. Math. Soc., 1998, pp. 1–36. DOI: 10.1090/conm/230/03336.

[4] John C. Baez and Aaron D. Lauda. "Higher-dimensional algebra. V: 2-Groups". In: *Theory Appl. Categ.* 12 (2004), pp. 423–491. arXiv: math/0307200.

[5] Ulrik Buchholtz and Egbert Rijke. "The real projective spaces in homotopy type theory". In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2017).* New York, NY, USA: IEEE, 2017, pp. 1–8. DOI: `10.1109/LICS.2017.8005146`.

[6] Omar Antolín Camarena. *The homotopy fiber of the map on classifying spaces.* 2017. URL: `http://www.matem.unam.mx/omar/notes/hofib-grphom.html`.

[7] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. "Cubical Type Theory: a constructive interpretation of the univalence axiom". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015).* LIPIcs. Leibniz Int. Proc. Inform. To appear. Wadern: Schloss Dagstuhl. Leibniz-Zent. Inform., 2016. arXiv: `1611.02108`.

[8] Floris van Doorn, Jeremy Avigad, Steve Awodey, Ulrik Buchholtz, Egbert Rijke, and Mike Shulman. "Spectral Sequences in Homotopy Type Theory". Paper forthcoming. 2018. URL: `https://github.com/cmu-phil/Spectral`.

[9] Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. "Homotopy Type Theory in Lean". In: *Interactive Theorem Proving (ITP 2017).* Cham: Springer, 2017, pp. 479–495. ISBN: 978-3-319-66107-0. DOI: `10.1007/978-3-319-66107-0_30`.

[10] Willam G. Dwyer and Clarence W. Wilkerson. "Homotopy fixed-point methods for Lie groups and finite loop spaces". In: *Ann. Math. (2)* 139.2 (1994), pp. 395–442. DOI: `10.2307/2946585`.

[11] William G. Dwyer and Alexander Zabrodsky. "Maps between classifying spaces". In: *Algebraic topology, Barcelona, 1986.* Vol. 1298. Lecture Notes in Math. Berlin: Springer, 1987, pp. 106–119. DOI: `10.1007/BFb0083003`.

[12] Anthony D. Elmendorf. "Systems of fixed point sets." In: *Trans. Am. Math. Soc.* 277 (1983), pp. 275–284. ISSN: 0002-9947; 1088-6850/e. DOI: `10.2307/1999356`.

[13] Alexander Grothendieck. *Pursuing Stacks.* Manuscript. Les Aumettes, Mormoiron, 1983. URL: `http://thescrivener.github.io/PursuingStacks/`.

[14] Nick Gurski. *The Stable Homotopy Hypothesis and Categorified Abelian Groups.* Blog post. The n-Category Café. 2018. URL: `https://golem.ph.utexas.edu/catego`

[15] Kuen-Bang Hou (Favonia) and Robert Harper. “Covering Spaces in Homotopy Type Theory”. In: *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*. Ed. by Herman Geuvers, Silvia Ghilezan, and Jelena Ivetic. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, n.d.

[16] Niles Johnson and Angélica M. Osorno. “Modeling stable one-types”. In: *Theory and Applications of Categories* 26.20 (2012), pp. 520–537. URL: `http://www.tac.mta.ca/tac/volumes/26/20/26-20abs.html`.

[17] Daniel R. Licata and Eric Finster. “Eilenberg-MacLane Spaces in Homotopy Type Theory”. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS ’14. Vienna, Austria: ACM, 2014, 66:1–66:9. ISBN: 978-1-4503-2886-9. DOI: `10.1145/2603088.2603153`.

[18] Jacob Lurie. *Higher topos theory*. Vol. 170. Annals of Mathematics Studies. Princeton, NJ: Princeton University Press, 2009, pp. xviii+925. DOI: `10.1515/9781400830558`.

[19] Leonardo Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction – CADE-25*. Ed. by P. Amy Felty and Aart Middeldorp. Cham: Springer, 2015, pp. 378–388. DOI: `10.1007/978-3-319-21401-6_26`.

[20] Egbert Rijke. *The join construction*. Preprint. 2017. arXiv: `1701.07538`.

[21] Urs Schreiber and Michael Shulman. “Quantum Gauge Field Theory in Cohesive Homotopy Type Theory”. In: Proceedings 9th Workshop on *Quantum Physics and Logic,* Brussels, Belgium, 10-12 October 2012. Ed. by Ross Duncan and Prakash Panangaden. Vol. 158. Electronic Proceedings in Theoretical Computer Science. Waterloo, NSW: Open Publishing Association, 2014, pp. 109–126. DOI: `10.4204/EPTCS.158.8`.

[22] Michael Shulman. *The Univalent Perspective on Classifying Spaces*. Blog post. The n-Category Café. 2015. URL: `https://golem.ph.utexas.edu/category/2015`

[23] Michael Shulman. “Univalence for inverse EI diagrams”. In: *Homology, Homotopy and Applications* 19.2 (2017), pp. 219–249. DOI: `10.4310/HHA.2017.v19.n2.a12`. arXiv: `1508.02410`.

[24] Hoàng Xuân Sính. “Gr-catégories”. PhD thesis. Université Paris Diderot (Paris 7), 1975. URL: `http://www.iaz.uni-stuttgart.de/LstAlg/Kuenzer/Kuenzer/sinh.`

[25] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `http://homotopytypetheory` 2013. arXiv: `1308.0729`.

[26] Vladimir Voevodsky. “A very short note on homotopy λ-calculus”. 2006. URL: `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlamb`

[27] J. H. C. Whitehead. “Combinatorial homotopy. II”. In: *Bull. Am. Math. Soc.* 55 (1949), pp. 453–496. DOI: `10.1090/S0002-9904-1949-09213-3`.

Mechanizing Coinduction and Corecursion in Higher-order Logic

Lawrence C. Paulson
Computer Laboratory, University of Cambridge

Abstract

A theory of recursive and corecursive definitions has been developed in higher-order logic (HOL) and mechanized using Isabelle. Least fixedpoints express inductive data types such as strict lists; greatest fixedpoints express coinductive data types, such as lazy lists. Well-founded recursion expresses recursive functions over inductive data types; corecursion expresses functions that yield elements of coinductive data types. The theory rests on a traditional formalization of infinite trees.

The theory is intended for use in specification and verification. It supports reasoning about a wide range of computable functions, but it does not formalize their operational semantics and can express noncomputable functions also. The theory is illustrated using finite and infinite lists. Corecursion expresses functions over infinite lists; coinduction reasons about such functions.

Key words. Isabelle, higher-order logic, coinduction, corecursion

Contents

1 Introduction

Recursive data structures, and recursive functions over them, are of central interest in Computer Science. The underlying theory is that of inductive definitions [2]. Much recent work has focused on formalizing induction principles in type theories. The type theory of Coq takes inductive definitions as primitive [8]. The second-order λ-calculus (known variously as System F and $\lambda 2$) can express certain inductive definitions as second-order abstractions [13].

Of growing importance is the dual notion: *coinductive* definitions. Infinite data structures, such as streams, can be expressed coinductively. The dual of recursion, called *corecursion*, can express functions involving coinductive types.

Coinduction is well established for reasoning in concurrency theory [20]. Abramsky's Lazy Lambda Calculus [1] has made coinduction equally important in the theory of functional programming. Milner and Tofte motivate coinduction through a simple proof about types in a functional language [21]. Tofte has proved the soundness of a type discipline for polymorphic references by coinduction [32]. Pitts has derived a coinduction rule for proving facts of the form $x \sqsubseteq y$ in domain theory [28].

There are many ways of formalizing coinduction and corecursion. Mendler [19] has proposed extending $\lambda 2$ with inductive and coinductive types, equipped with recursion and corecursion operators. More recently, Geuvers [12] has shown that coinductive types can be constructed from inductive types, and vice versa. Leclerc and Paulin-Mohring [17] investigate various formalizations of streams in the Coq system. Rutten and Turi survey three other approaches [29].

Church's higher-order logic (HOL) is perfectly adequate for formalizing both inductive and coinductive definitions. The constructions are not especially difficult. A key tool is the Knaster-Tarski theorem, which yields least and greatest fixedpoints of monotone functions. Trees, finite and infinite, are represented as sets of nodes. Recursion is derivable in its most general form, for arbitrary well-founded relations. Corecursion and coinduction have straightforward definitions.

Compared with other type theories, HOL has the advantage of being simple, stable and well-understood. But $\lambda 2$ and Coq have an inbuilt operational semantics based on reduction, while a HOL theory can only suggest an operational interpretation. HOL admits non-computable functions, which is sometimes advantageous and sometimes not.

Higher-order logic is extremely successful in verification, mainly hardware verification [7]. The HOL system [14] is particularly popular. Melham has formalized and mechanized a theory of inductive definitions for the HOL system [18]; my work uses different principles to lay the foundation for mechanizing a broader class of definitions. Some of the extensions (mutual recursion, extending the language of type constructors) are also valid in set theory [26]. One unexplored possibility is that trees may have infinite branching.

A *well-founded* (WF) relation $\prec$ admits no infinite descents $\cdots \prec x_n \prec \cdots x_1 \prec x_0$. A data structure is WF provided its substructure relation is well-founded. My work justifies non-WF data structures, which involve infinitely deep nesting.

The paper is chiefly concerned with coinduction and corecursion. §2 briefly introduces Isabelle and its formalization of HOL. §3 describes the least and greatest fixedpoint operators. §4 presents the representation of infinite trees. §5 considers finite lists and other WF data structures. §6 concerns lazy lists, deriving coinduction principles for type-checking and equality. §7 introduces and derives corecursion, while §8 presents examples. Finally, §9 gives conclusions and discusses related work.

2 HOL in Isabelle

The theory described below has been mechanized using Isabelle, a generic theorem prover [25]. Isabelle implements several object-logics: first-order logic, Zermelo-Frænkel set theory, Constructive Type Theory, higher-order logic (HOL), etc.

Isabelle has logic programming features, such as unification and proof search. Every Isabelle object-logic can take advantage of these. Isabelle's rewriter and classical reasoning package can be used with logics having the appropriate properties.

For this paper, we may regard Isabelle simply as an implementation of higher-order logic. There are many others, such as TPS [4], IMPS [10] and the HOL system [14]. I shall describe the theory of recursion in formal detail, to facilitate its mechanization in any suitable system.

2.1 Higher-order logic as an meta-logic

Isabelle exploits the power of higher-order logic on two levels. At the meta-level, Isabelle uses a fragment of intuitionistic HOL to mechanize inference in various object-logics. One of these object-logics is classical HOL.

The meta-logic, as a fragment of HOL, is based upon the typed λ-calculus. It uses λ-abstraction to formalize the object-logic's binding operators, such as $\forall x\ \phi$, $\prod_{x\in A} B$, $\epsilon x.\phi$ and $\bigcup_{x\in A} B$, in the same manner as Church did for higher-order logic [6]. The approach is fully general; each binding operator may involve any fixed pattern of arguments and bound variables, and may denote a formula, term, set, type, etc. In a recent paper [24], I discuss variable binding with examples.

Quantification in the meta-logic expresses axiom and theorem schemes. Binding operators typically involve higher-order definitions. The normalization theorem for natural deduction proofs in HOL can be used to justify the soundness of Isabelle's representation of the object-logic. I have done a detailed proof for the case of intuitionistic first-order logic [23]; the argument applies, with obvious modifications, to any formalization of a similar syntactic form.

2.2 Higher-order logic as an object-logic

Isabelle/HOL, Isabelle's formalization of higher-order logic, follows the HOL system [14] and thus Church [6]. The connectives and quantifiers are defined in terms of three primitives: implication ($\rightarrow$), equality ($=$) and descriptions ($\epsilon x.\phi$). Isabelle emphasizes a natural deduction style; its HOL theory derives natural deduction rules for the connectives and quantifiers from their obscure definitions.

The *types* σ, τ, ..., are simple types. The type of truth values is called `bool`. The type of individuals plays no role in the sequel; instead, we use types of natural numbers, products, etc. Church used subscripting to indicate type, as in x_τ; Isabelle's notation is $x :: \tau$. Church wrote $\tau\sigma$ for the type of functions from σ to τ; Isabelle's notation is $\sigma \Rightarrow \tau$. Nested function types may be abbreviated:

$$(\sigma_1 \Rightarrow \cdots (\sigma_n \Rightarrow \tau) \cdots) \quad \text{as} \quad [\sigma_1, \ldots, \sigma_n] \Rightarrow \tau$$

ML-style polymorphism replaces Church's type-indexed families of constants. *Type schemes* containing type variables α, β, ... , represent families of types. For example, the quantifiers have type $(\alpha \Rightarrow \texttt{bool}) \Rightarrow \texttt{bool}$; the type variable α may be instantiated to any suitable type for the quantification.[1] As in ML, type operators have a postfix notation. For example, $(\sigma)\,\texttt{set}$ is the type of sets over σ, and $(\sigma)\,\texttt{list}$ is the type of lists over σ. The parentheses may be omitted if there is no ambiguity; thus `nat set set` is the type of sets of sets of natural numbers.

The *terms* are those of the typed λ-calculus. The *formulae* are terms of type `bool`. They are constructed from the logical constants $\wedge$, $\vee$, $\rightarrow$ and $\neg$. There is no $\leftrightarrow$ connective; the equality $\phi = \psi$ truth values serves as a biconditional. The *quantifiers* are $\forall$, $\exists$ and $\exists!$ (unique existence).

2.3 Further Isabelle/HOL types

Isabelle's higher-order logic is augmented with Cartesian products, disjoint sums, the natural numbers and arithmetic. Isabelle theories define the appropriate types and constants, and prove a large collection of theorems and derived rules.

The *singleton* type `unit` possesses the single value $() :: \texttt{unit}$.

The *Cartesian product* type $\sigma \times \tau$ possesses as values ordered pairs (x, y), for x of type σ and y of type τ. We have the usual two projections, as well as the eliminator `split`:

$$\begin{aligned} \texttt{fst} &:: (\alpha \times \beta) \Rightarrow \alpha \\ \texttt{snd} &:: (\alpha \times \beta) \Rightarrow \beta \\ \texttt{split} &:: [[\alpha, \beta] \Rightarrow \gamma,\ \alpha \times \beta] \Rightarrow \gamma \end{aligned}$$

Operations over pairs can be couched in terms of `split`, which satisfies the equation $\texttt{split}\ c\ (a, b) = c\ a\ b$.

The *disjoint sum* type $\sigma + \tau$ possesses values of the form $\texttt{Inl}\ x$ for $x :: \sigma$ and $\texttt{Inr}\ y$ for $y :: \tau$. The eliminator, `sum_case`, is similar in spirit to `split`:

$$\texttt{sum\_case} :: [\alpha \Rightarrow \gamma,\ \beta \Rightarrow \gamma,\ \alpha + \beta] \Rightarrow \gamma$$

The eliminator performs case analysis on its first argument:

$$\begin{aligned} \texttt{sum\_case}\ c\ d\ (\texttt{Inl}\ a) &= c\ a \\ \texttt{sum\_case}\ c\ d\ (\texttt{Inr}\ b) &= d\ b \end{aligned}$$

The operators `split` and `sum_case` are easily combined to express pattern matching over more complex arguments. The compound operator

$$\texttt{sum\_case}\,(\texttt{sum\_case}\ c\ d)\,(\texttt{split}\ e)\,z \tag{1}$$

[1] In Isabelle, type variables are classified by sorts in order to control polymorphism, but this need not concern us here.

analyses an argument z of type $(\alpha{+}\beta){+}(\gamma{\times}\delta)$. This is helpful to implementors writing packages to support such data types. On the other hand, expressions involving `split` and `sum_case` are hard to read. Therefore Isabelle allows limited pattern matching and case analysis in definitions. A `case` syntax can be used for `sum_case` and similar operators. In a binding position, a pair of variables stands for a call to `split`. These constructs nest. We can express the operator (1) less concisely but more readably by

$$\begin{array}{lll}\texttt{case}\ z\ \texttt{of}\ \texttt{Inl}\ z' & \Rightarrow (\texttt{case}\ z'\ \texttt{of} & \texttt{Inl}\ x \Rightarrow c\ x \\ & & \mid \texttt{Inr}\ y \Rightarrow d\ y) \\ \quad\mid \texttt{Inr}(v,w) \Rightarrow e\ v\ w & & \end{array}$$

The *natural number* type, `nat`, has the usual arithmetic operations $+$, $-$, $\times$, etc., of type $[\texttt{nat},\texttt{nat}] \Rightarrow \texttt{nat}$. The successor function is $\texttt{Suc} :: \texttt{nat} \Rightarrow \texttt{nat}$. Definitions involving the natural numbers can use the `case` construct, with separate cases for zero and successor numbers. Isabelle also supports a special syntax for definition by primitive recursion. This paper presents definitions using the most readable syntax available in Isabelle, occasionally going beyond this.

2.4 Sets in Isabelle/HOL

Set theory in higher-order logic dates back to *Principia Mathematica*'s theory of classes [33]. Although sets are essentially predicates, Isabelle/HOL defines the type $\alpha\,\texttt{set}$ for sets over type α. Type $\alpha\,\texttt{set}$ possesses values of the form $\{x \mid \phi x\}$ for $\phi :: \alpha \Rightarrow \texttt{bool}$. The eliminator is membership, $\in\, :: [\alpha, \alpha\,\texttt{set}] \Rightarrow \texttt{bool}$. It satisfies the equations

$$\begin{aligned}(a \in \{x \mid \phi x\}) &= \phi a \\ \{x \mid x \in A\} &= A\end{aligned}$$

Types distinguish this set theory from axiomatic set theories such as Zermelo-Frænkel. All elements of a set must have the same type, and formulae such as $x \notin x$ and $x \in y \wedge y \in z \rightarrow x \in z$ are ill-typed. For each type α, there is a universal set, namely $\{x \mid \texttt{True}\}$.

Many set-theoretic operations have obvious definitions:

$$\begin{aligned}\forall_{x\in A}\,\psi &\equiv \forall x\,(x \in A \rightarrow \psi) \\ \exists_{x\in A}\,\psi &\equiv \exists x\,(x \in A \wedge \psi) \\ A \subseteq B &\equiv \forall_{x\in A}\, x \in B \\ A \cup B &\equiv \{x \mid x \in A \vee x \in B\} \\ A \cap B &\equiv \{x \mid x \in A \wedge x \in B\}\end{aligned}$$

We have several forms of large union: the bounded union $\bigcup_{x\in A} B$, the unbounded union $\bigcup_x B$, and the union of a set of sets, $\bigcup S$. Their definitions, and those of the corresponding intersection operators, are straightforward:

$$
\begin{aligned}
\bigcup\nolimits_{x\in A} B &\equiv \{y \mid \exists_{x\in A}\, y \in B\}\\
\bigcap\nolimits_{x\in A} B &\equiv \{y \mid \forall_{x\in A}\, y \in B\}\\
\bigcup\nolimits_{x} B &\equiv \bigcup\nolimits_{x\in\{x\mid\mathtt{True}\}} B\\
\bigcap\nolimits_{x} B &\equiv \bigcap\nolimits_{x\in\{x\mid\mathtt{True}\}} B\\
\bigcup S &\equiv \bigcup\nolimits_{x\in S} x\\
\bigcap S &\equiv \bigcap\nolimits_{x\in S} x
\end{aligned}
$$

Our definitions will frequently refer to the range of a function, and to the image of a set over a function:

$$
\begin{aligned}
\mathtt{range}\ f &\equiv \{y \mid \exists x\, y = f\ x\}\\
f \text{ ``} A &\equiv \{y \mid \exists_{x\in A}\, y = f\ x\}
\end{aligned}
$$

For reasoning about the set operations, I prefer to derive natural deduction rules. For example, there are two introduction rules for $A \cup B$:

$$\frac{x \in A}{x \in A \cup B} \qquad \frac{x \in B}{x \in A \cup B}$$

The corresponding elimination rule resembles disjunction elimination.

The Isabelle theory includes the familiar properties of the set operations. Isabelle's classical reasoner can prove many of these automatically. Examples:

$$
\begin{aligned}
\bigcap(A \cup B) &= \bigcap A \cap \bigcap B\\
\left(\bigcup\nolimits_{x\in C} A\ x \cup B\ x\right) &= \bigcup(A \text{ ``} C) \cup \bigcup(B \text{ ``} C)
\end{aligned}
$$

2.5 Type definitions

In Gordon's HOL system, a new type τ can be defined from an existing type σ and a predicate $\phi :: \sigma \Rightarrow \mathtt{bool}$. Each element of type τ is represented by some element $x :: \sigma$ such that ϕx holds. The type definition is valid only if $\exists x\, \phi x$ is a theorem, since HOL does not admit empty types. Unicity of types demands a distinction between elements of σ and elements of τ, which can be achieved by introducing an *abstraction* function $\mathtt{abs} :: \sigma \Rightarrow \tau$. Each element of τ has the form $\mathtt{abs}\ x$ for some $x :: \sigma$ such that ϕx holds. The function abs has a right inverse, the *representation* function $\mathtt{rep} :: \tau \Rightarrow \sigma$, satisfying

$$\phi(\mathtt{rep}\ y), \qquad \mathtt{abs}(\mathtt{rep}\ y) = y, \qquad \text{and} \qquad \phi x \rightarrow \mathtt{rep}(\mathtt{abs}\ x) = x.$$

As a trivial example, the singleton type unit serves as a nullary Cartesian product. It may be defined from bool by the predicate $\lambda x.\, x = \mathtt{True}$. Calling the abstraction function $\mathtt{abs\_unit} :: \mathtt{bool} \Rightarrow \mathtt{unit}$, we obtain $() :: \mathtt{unit}$ by means of the definition

$$() \equiv \mathtt{abs\_unit}(\mathtt{True})$$

Isabelle and the HOL system have polymorphic type systems; types may contain type variables α, β, ... , which range over types. Type variables may also occur in terms:

$$\lambda f :: [\alpha, \beta] \Rightarrow \texttt{bool}.\, \exists a\, b\, (f = (\lambda x\, y.\, x = a \wedge y = b))$$

This is a predicate over the type scheme $[\alpha, \beta] \Rightarrow \texttt{bool}$. It allows us to define $\alpha \times \beta$, the Cartesian product of two types. We may then write $\sigma \times \tau$ for any two types σ and τ.

Like the HOL system, Isabelle/HOL supports type definitions. A `subtype` declaration specifies the new type name and a set expression. This set determines the representing type and the predicate over it, called σ and ϕ above. Isabelle automatically declares the new type; its abstraction and representation functions receive names of the form $\texttt{Abs\_}X$ and $\texttt{Rep\_}X$.

3 Least and greatest fixedpoints

The Knaster-Tarski Theorem asserts that each monotone function over a complete lattice possesses a fixedpoint.[2] Tarski later proved that the fixedpoints themselves form a complete lattice [31]; we shall be concerned only with the least and greatest fixedpoints. Least fixedpoints yield inductive definitions while greatest fixedpoints yield coinductive definitions.

Our theory of inductive definitions requires only one kind of lattice: the collection, ordered by the relation $\subseteq$, of the subsets of a set. Monotonicity is defined by

$$\texttt{mono}\ f \equiv \forall A\, B (A \subseteq B \rightarrow f\ A \subseteq f\ B).$$

Monotonicity is generally easy to prove. The following results each have one-line proofs in Isabelle:

$$\frac{A \subseteq C \quad B \subseteq D}{A \cup B \subseteq C \cup D} \qquad \frac{A \subseteq B}{f\, ``\, A \subseteq f\, ``\, B} \qquad \frac{A \subseteq C \quad \begin{array}{c}[x \in A]_x \\ \vdots \\ B\, x \subseteq D\, x\end{array}}{(\bigcup_{x \in A} B\, x) \subseteq (\bigcup_{x \in C} D\, x)}$$

Armed with facts such as these, it is trivial to prove that a function composed from such operators is itself monotonic.

The fixedpoint operators are called `lfp` and `gfp`. They both have type $[\alpha\, \texttt{set} \Rightarrow \alpha\, \texttt{set}] \Rightarrow \alpha\, \texttt{set}$; they take a monotone function and yield a set.

3.1 The least fixedpoint

The least fixedpoint operator is defined by $\texttt{lfp}\ f \equiv \bigcap \{X \mid f\ X \subseteq X\}$. Roughly speaking, $\texttt{lfp}\ f$ contains only those objects that must be included: they are common to all fixedpoints of f. The Isabelle theory proves that `lfp` is indeed the least fixedpoint of f:

$$\frac{f\ A \subseteq A}{\texttt{lfp}\ f \subseteq A} \qquad \frac{\texttt{mono}\ f}{\texttt{lfp}\ f = f(\texttt{lfp}\ f)}$$

[2] See Davey and Priestley for a modern discussion [9].

The fixedpoint property justifies both introduction and elimination rules for $\texttt{lfp}\ f$, assuming we already know how to construct and take apart sets of the form $f\ A$. Because $\texttt{lfp}\ f$ is the *least* fixedpoint, it satisfies a better elimination rule, namely induction. The Isabelle theory derives a strong form of induction, which can easily be instantiated to yield structural induction rules:

$$\frac{a \in \texttt{lfp}\ f \quad \texttt{mono}\ f \quad \begin{array}{c}[x \in f(\texttt{lfp}\ f \cap \{x \mid \psi x\})]_x \\ \vdots \\ \psi x\end{array}}{\psi a}$$

The set $\texttt{List}\ A$ of finite lists over A is a typical example of a least fixedpoint. Lists have two introduction rules:

$$\texttt{NIL} \in \texttt{List}\ A \qquad \frac{M \in A \quad N \in \texttt{List}\ A}{\texttt{CONS}\ M\ N \in \texttt{List}\ A}$$

The elimination rule is structural induction:

$$\frac{M \in \texttt{List}\ A \quad \psi\texttt{NIL} \quad \begin{array}{c}[x \in A \quad y \in \texttt{List}\ A \quad \psi y]_{x,y} \\ \vdots \\ \psi(\texttt{CONS}\ x\ y)\end{array}}{\psi M}$$

The related principle of structural recursion expresses recursive functions on finite lists. See §5.1 for details of the definition of lists. Elsewhere [26] I discuss the $\texttt{lfp}$ induction rule and other aspects of the $\texttt{lfp}$ theory.

3.2 The greatest fixedpoint

The greatest fixedpoint operator is defined by $\texttt{gfp}\ f \equiv \bigcup\{X \mid X \subseteq f\ X\}$. The dual of the least fixedpoint, it excludes only those elements that must be excluded. The Isabelle theory proves that $\texttt{gfp}$ is the greatest fixedpoint of f:

$$\frac{A \subseteq f\ A}{A \subseteq \texttt{gfp}\ f} \qquad \frac{\texttt{mono}\ f}{\texttt{gfp}\ f = f(\texttt{gfp}\ f)}$$

As with $\texttt{lfp}\ f$, the fixedpoint property justifies both introduction and elimination rules for $\texttt{gfp}\ f$. But the elimination rule is not induction; instead, a further introduction rule is coinduction.

Typically $\texttt{gfp}\ f$ contains infinite objects. The usual introduction rules, like those for $\texttt{NIL}$ and $\texttt{CONS}$ above, can only justify finite objects — each rule application justifies only one stage of the construction. But a single application of coinduction can prove the existence of an infinite object. Conversely, we should not expect to have a structural induction rule when there are infinite objects.

To show $a \in \texttt{gfp}\ f$ by coinduction, exhibit some set X such that $a \in X$ and $X \subseteq f\ X$:

$$\frac{a \in X \quad X \subseteq f\ X}{a \in \texttt{gfp}\ f}$$

This rule is *weak* coinduction. Its soundness is obvious by the definition of `gfp` and does not even require f to be monotonic. The set X is typically a singleton or the range of some function.

For monotonic f, coinduction can be strengthened in various ways. The following version is called *strong* coinduction below:

$$\frac{a \in X \quad X \subseteq f\ X \cup \mathtt{gfp}\ f \quad \mathtt{mono}\ f}{a \in \mathtt{gfp}\ f}$$

An even stronger version, not required below, is

$$\frac{a \in X \quad X \subseteq f(X \cup \mathtt{gfp}\ f) \quad \mathtt{mono}\ f}{a \in \mathtt{gfp}\ f.}$$

Since `lfp` and `gfp` are dual notions, facts about one can be transformed into facts about the other by reversing the orientation of the $\subseteq$ relation and exchanging $\cap$ for $\cup$, etc.

Milner and Park's work on concurrency is an early use of coinduction. A *bisimulation* is a binary relation r satisfying a property of the form $r \subseteq f\ r$. Two processes are called equivalent if the pair belongs to any bisimulation; thus, process equivalence is defined to be $\mathtt{gfp}\ f$. Two processes can be proved equivalent by exhibiting a suitable bisimulation [20].

The set $\mathtt{LList}\ A$ of lazy lists over A will be our main example of a greatest fixedpoint, starting in §6. The set contains both finite and infinite lists. In fact, $\mathtt{LList}\ A$ and $\mathtt{List}\ A$ are both fixedpoints of the same monotone function. We have $\mathtt{List}\ A \subseteq \mathtt{LList}\ A$ and $\mathtt{LList}\ A$ shares the introduction rules of $\mathtt{List}\ A$, justifying the existence of finite lists.

A new principle, called *corecursion*, defines certain infinite lists; coinduction proves that these lists belong to $\mathtt{LList}\ A$. Finally, the equality relation on $\mathtt{LList}\ A$ happens to coincide with the `gfp` of a certain function. Coinduction can therefore prove equations between infinite lists. The Isabelle theory proves many familiar laws involving the append and map functions.

Coinduction can also prove that the function defined by corecursion is unique. Categorists will note that $\mathtt{LList}\ A$ is a final coalgebra, just as $\mathtt{List}\ A$ is an initial algebra.

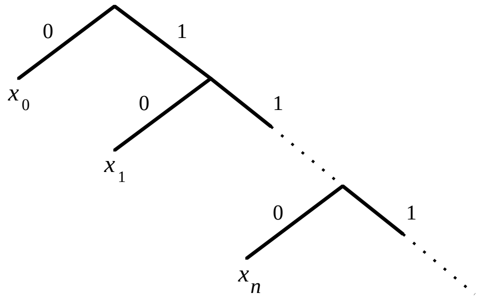
4 Infinite trees in HOL

The `lfp` and `gfp` operators both have type $[\alpha\,\mathtt{set} \Rightarrow \alpha\,\mathtt{set}] \Rightarrow \alpha\,\mathtt{set}$. Applying them to a monotone function $f :: \tau\,\mathtt{set} \Rightarrow \tau\,\mathtt{set}$ automatically instantiates α to τ; the result has type $\tau\,\mathtt{set}$. We should regard τ as a large space, from which `lfp` and `gfp` carve out various subspaces. It matters not if τ contains extraneous or ill-formed elements, since a suitable f will discard them.

My approach is to formalize all recursive data structure definitions using one particular τ, with a rich structure. Constructions — lists, trees, etc. — are sets of nodes. A node is a (*position*, *label*) pair and has type $\alpha\,\mathtt{node}$. Thus τ is $\alpha\,\mathtt{node}\,\mathtt{set}$, where α is the type of atoms that may occur in the construction.

Data structures are defined as sets of type $\alpha\,\mathtt{node}\,\mathtt{set}\,\mathtt{set}$. The binary operators $\otimes$ and $\oplus$, analogues of the Cartesian product and disjoint sum, take two sets of that type

Figure 1: An infinite tree with finite branching



and yield another such set. Similarly, `List` A and `LList` A are unary set operators over type α `node set set`; they can even participate in other recursive data structure definitions.

4.1 A possible coding of lists

To understand the definition of type α `node`, let us examine a simpler case, which only works for lists. The finite or infinite list $[x_0, x_1, \ldots, x_n, \ldots]$ could be represented by the set of pairs

$$\{(0, x_0), (1, x_1), \ldots, (n, x_n), \ldots\}.$$

Each pair consists of the position m, a natural number, paired with the label x_m. To 'cons' an element a to the front of this list, we must add one to all the position numbers. If `succfst` $(m, x) =$ (`Suc` m, x) then `succfst` “ l increases all the position numbers in l. We may define the list constructors by

$$\begin{aligned} \texttt{NIL} &\equiv \{\} \\ \texttt{CONS}\ a\ l &\equiv \{(0, a)\} \cup \texttt{succfst}\ “\ l \end{aligned}$$

The function $\lambda L. \{\texttt{NIL}\} \cup (\bigcup_x \bigcup_{l \in L} \{\texttt{CONS}\ x\ l\})$ is clearly monotone by the properties of unions. Its `lfp` is the set of finite lists; its `gfp` includes the infinite lists too. Each list has type $(\sigma \times \sigma)$ `set`, where σ is the type of the list's elements.

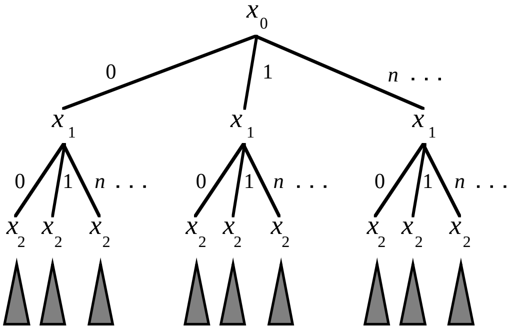
4.2 Non-well-founded trees

In the representation above, the position of a list element is a natural number. To handle trees, let the position of a tree node be a list of natural numbers, giving the path from the root to that node. For example, the infinite tree of Figure 1 could be represented by the set of pairs

$$\{([0], x_0), ([1, 0], x_1), \ldots, ([\underbrace{1, \ldots, 1}_{n}, 0], x_n), \ldots\}.$$

The list $[1, 1, 0]$ gives the position of node x_2 in the tree.

Figure 2: An infinite tree with infinite branching



The labelled nodes determine the tree's structure. A tree that has no labelled nodes is represented by the empty set and is indistinguishable from the empty tree. Each label defines the corresponding position. Moreover, it implies the existence of all its ancestor nodes. With this representation, unlabelled branch nodes exist only because of the labelled leaf nodes underneath them.

We can even represent trees with infinite branching. The tree of Figure 2 is the infinite set of labelled nodes of the form $([k_0, k_1, \ldots, k_{n-1}], x_n)$ for $n \geq 0$ and $k_i :: \texttt{nat}$.

The definitions discussed below do not exploit the representation in its full generality. Branching is finite — binary, in fact — but the depth may be infinite.

4.3 The formal definition of type $\alpha\,\texttt{node}$

A node is essentially a list paired with a label. We could represent lists as described above, but this would immediately be superseded by the resulting representation of trees, a duplication of effort. Luckily, we require only lists of natural numbers. We could encode them by Gödel numbers of the form $2^{k_0}3^{k_1}\cdots$, but they are more simply represented by functions of type $\texttt{nat} \Rightarrow \texttt{nat}$.

Let the list $[k_0, \ldots, k_{n-1}]$ denote some function f such that

$$f\ i = \begin{cases} \texttt{Suc}\ k_i & \text{if } 0 \leq i < n; \\ 0 & \text{if } i = n. \end{cases}$$

Note that $f\ i$ could be anything for $i > n$. The constant function $\lambda i.0$ represents the empty list. To 'cons' an element a to the front of the list f, we use $\texttt{Push}\ a\ f$, which is defined by cases on its third argument:

$$\begin{array}{l} \texttt{Push}\ a\ f\ i \equiv \texttt{case}\ i\ \texttt{of}\ \ 0 \qquad\ \Rightarrow \texttt{Suc}\ a \\ \qquad\qquad\qquad\qquad\qquad \mid \texttt{Suc}\ j \Rightarrow f\ j \end{array}$$

Each node is represented by a pair $\ f\ x$, where $f :: \texttt{nat} \Rightarrow \texttt{nat}$ stands for a list and $x :: \alpha + \texttt{nat}$ is a label. The disjoint sum type allows a label to contain either an element of type α or a natural number; some constructions below require natural numbers in trees.

To prove that equality is a $\texttt{gfp}$, we must impose a finiteness restriction on f. The *take-lemma*, which says that two lazy lists are equal if all their corresponding finite initial segments are equal [5], is a standard reasoning method in lazy functional programming. The take-lemma is valid because a lazy list is nothing more than the set of its finite parts. We may similarly prove that two infinite data structures are equal, if we ensure that a tree cannot contain a node at an infinite depth. The restriction is only necessary because lists of natural numbers are represented by functions.

The set of all nodes is therefore defined as follows:

$$\texttt{Node} \equiv \{p \mid \exists f\, x\, n\, (p = (f,x) \land f\ n = 0)\} \tag{2}$$

The second conjunct ensures that the position list is finite: $f\ n = 0$ for some n. No other conditions need to be imposed upon nodes or node sets; the fixedpoint operators exclude the undesirable elements.

Since `Node` has a complex type, namely

$$((\texttt{nat} \Rightarrow \texttt{nat}) \times (\alpha + \texttt{nat}))\, \texttt{set},$$

let us define $\alpha\,\texttt{node}$ to be the type of nodes taking labels from α. As described in §2.5, the subtype declaration yields abstraction and representation functions:

$$\begin{aligned}
&\texttt{Abs\_Node} :: ((\texttt{nat} \Rightarrow \texttt{nat}) \times (\alpha + \texttt{nat}))\, \texttt{set} \Rightarrow \alpha\, \texttt{node}\\
&\texttt{Rep\_Node} :: \alpha\, \texttt{node} \Rightarrow ((\texttt{nat} \Rightarrow \texttt{nat}) \times (\alpha + \texttt{nat}))\, \texttt{set}
\end{aligned}$$

4.4 The binary tree constructors

Possibly infinite binary trees represent all data structures in this theory. Binary trees are sets of nodes. The simplest binary tree, `Atom` a, consists of a label alone. If M and N are binary trees, then $M \cdot N$ is the tree consisting of the two branches M and N. Thus there are two primitive tree constructors:

$$\begin{aligned}
\texttt{Atom} &:: \alpha + \texttt{nat} \Rightarrow \alpha\, \texttt{node}\, \texttt{set}\\
(\cdot) &:: [\alpha\, \texttt{node}\, \texttt{set}, \alpha\, \texttt{node}\, \texttt{set}] \Rightarrow \alpha\, \texttt{node}\, \texttt{set}
\end{aligned}$$

Writing [] for $\lambda i.0$, we could define these constructors semi-formally as follows:

$$\begin{aligned}
\texttt{Atom}\ a &\equiv \{([], a)\}\\
M \cdot N &\equiv \{(\texttt{Push}\ 0\ f,\ x)\}_{(f,x)\in M} \cup \{(\texttt{Push}\ 1\ f,\ x)\}_{(f,x)\in N}
\end{aligned}$$

These have the expected properties for infinite binary trees. The constructors are injective. We can recover a from `Atom` a; we can recover M and N from $M \cdot N$ by stripping the initial 0 or 1. We always have `Atom` $a \neq M \cdot N$ because `Atom` a contains [] as a position while $M \cdot N$ does not. Trees need not be WF; for example, there are infinite trees such that $M = M \cdot M$.

For the sake of readability, definitions below will analyse their arguments using pattern matching instead `Rep_Node`. Let us also abbreviate (`Suc` 0) as 1. The formal definitions of `Atom` and $(\cdot)$ are cumbersome, but still readable enough:

$$\begin{aligned}
\texttt{Push\_Node}\ k\, (\texttt{Abs\_Node}(f,a)) &\equiv \texttt{Abs\_Node}(\texttt{Push}\ k\ f,\ a)\\
\texttt{Atom}\ a &\equiv \{\texttt{Abs\_Node}\, (\lambda i.0,\ a)\}\\
M \cdot N &\equiv \texttt{Push\_Node}\ 0 \text{ ``} M \cup \texttt{Push\_Node}\ 1 \text{ ``} N
\end{aligned}$$

Now $\texttt{Push\_Node}\ k$ takes the node represented by (f, a) to the one represented by $(\texttt{Push}\ k\ f,\ a)$. Finally, the image $\texttt{Push\_Node}\ 0\ \text{``}\ M$ applies this operation upon every node in M.

4.5 Products and sums for binary trees

One objective of this theory is to justify fixedpoint definitions of the data structures, such as

$$\texttt{List}\ A \equiv \texttt{lfp}(\lambda Z.\, \{\texttt{NIL}\} \oplus (A \otimes Z)),$$

where $\otimes$ is some form of Cartesian product and $\oplus$ is some form of disjoint sum. We cannot found these upon the usual HOL ordered pairs and injections. In both (x, y) and $\texttt{Inl}\ x$, the type of the result is more complex than that of the arguments; the $\texttt{lfp}$ call would be ill-typed. Therefore we must find alternative definitions of $\otimes$ and $\oplus$.

Since $(\cdot)$ is injective, we may use it as an ordered pairing operation and define the product by

$$A \otimes B \equiv \bigcup_{x \in A} \bigcup_{y \in B} \{x \cdot y\}$$

Now A, B and $A \otimes B$ all have the same type, namely $\alpha\,\texttt{node set set}$.

Disjoint sums are typically coded in set theory by

$$A + B \equiv \{(0, x)\}_{x \in A} \cup \{(1, y)\}_{y \in B}.$$

In the present setting, this requires distinct trees to play the roles of 0 and 1. Perhaps we could find two distinct sets of nodes, such as the empty set and the universal set, but this would complicate matters below.[3] Precisely to avoid such complications, natural numbers are always allowed as labels — recall that $\texttt{Atom}$ has type $\alpha + \texttt{nat} \Rightarrow \alpha\,\texttt{node set}$. The derived constructors

$$\begin{aligned}\texttt{Leaf} &:: \alpha \Rightarrow \alpha\,\texttt{node set}\\ \texttt{Numb} &:: \texttt{nat} \Rightarrow \alpha\,\texttt{node set}\end{aligned}$$

are defined by

$$\begin{aligned}\texttt{Leaf}\ a &\equiv \texttt{Atom}(\texttt{Inl}\ a)\\ \texttt{Numb}\ k &\equiv \texttt{Atom}(\texttt{Inr}\ k)\end{aligned}$$

We may now define disjoint sums in the traditional manner:

$$\begin{aligned}\texttt{In0}\ M &\equiv \texttt{Numb}\ 0 \cdot M\\ \texttt{In1}\ N &\equiv \texttt{Numb}\ 1 \cdot N\\ A \oplus B &\equiv (\texttt{In0}\ \text{``}\ A) \cup (\texttt{In1}\ \text{``}\ B)\end{aligned}$$

Both injections have type $\alpha\,\texttt{node set} \Rightarrow \alpha\,\texttt{node set}$, while A, B and $A \oplus B$ all have type $\alpha\,\texttt{node set set}$. Since the latter type is also closed under $\otimes$ and contains copies of

[3] The equation $\{\} \cdot \{\} = \{\}$ would frustrate the development of WF trees.

α and `nat`, it has enough structure to contain virtually every sort of finitely branching tree. Note that $\oplus$ and $\otimes$ are monotonic, by the monotonicity of unions and images.

The eliminators for $\otimes$ and $\oplus$ are analogous to those for the product and sum types, namely `split` and `sum_case`. They are defined in the normal manner, using descriptions, and satisfy the corresponding equations:

$$\begin{aligned}\texttt{Split}\, c\, (M \cdot N) &= c\; M\; N \\ \texttt{Case}\; c\; d\, (\texttt{In0}\; M) &= c\; M \\ \texttt{Case}\; c\; d\, (\texttt{In1}\; N) &= d\; N\end{aligned}$$

5 Well-founded data structures

In other work [26], I have investigated the formalization of recursive data structures in Zermelo-Frænkel (ZF) set theory. It is a general approach, allowing mutual recursion; moreover, recursive set constructions may take part in new recursive definitions, as in `term` $A = A \times$ `list`(`term` A). It has been mechanized within an Isabelle implementation of ZF set theory, just as the present work has been mechanized within an Isabelle implementation of HOL. It even supports non-WF data structures, using a variant form of pairing [27].

The Isabelle/HOL theory handles both WF and non-WF data structures. The WF ones are similar to those investigated in ZF, so let us dispense with them quickly.

5.1 Finite lists

We can now define the operator `List` to be the least solution to the recursion equation `List` $A = \{\texttt{NIL}\} \oplus (A \otimes \texttt{List}\; A)$. The Knaster-Tarski Theorem applies because $\oplus$ and $\otimes$ are monotonic. We can even prove that `List` is a monotonic operator over type $\alpha\,$`node set set`, justifying definitions such as `Term` $A = A \otimes \texttt{List}(\texttt{Term}\; A)$.

The formal definition of `List` A uses `lfp` to get the least fixedpoint:

$$\begin{aligned}\texttt{List\_Fun}\; A &\equiv \lambda Z.\; \{\texttt{Numb}\; 0\} \oplus (A \otimes Z) \\ \texttt{List}\; A &\equiv \texttt{lfp}(\texttt{List\_Fun}\; A) \\ \texttt{NIL} &\equiv \texttt{In0}(\texttt{Numb}\; 0) \\ \texttt{CONS}\; M\; N &\equiv \texttt{In1}(M \cdot N)\end{aligned}$$

From these we can easily derive the list introduction rules (§3.1), and various injectivity properties:

$$\texttt{CONS}\; M\; N \neq \texttt{NIL} \qquad (\texttt{CONS}\; K\; M = \texttt{CONS}\; L\; N) = (K = L \wedge M = N)$$

Now we can define case analysis (for recursion, see the next section). The eliminator for lists is expressed using those for $\oplus$ and $\otimes$:

$$\texttt{List\_case}\; c\; d \equiv \texttt{Case}\, (\lambda x.c)\, (\texttt{Split}\; d)$$

It satisfies the expected equations:

$$\texttt{List\_case}\; c\; d\; \texttt{NIL} = c \tag{3}$$

$$\texttt{List\_case}\; c\; d\, (\texttt{CONS}\; M\; N) = d\; M\; N \tag{4}$$

For readability we can use this operator via Isabelle's `case` syntax.

Later, (§6), we shall define `LList` A, which includes infinite lists, as the greatest fixedpoint of `List_Fun` A. Our definitions of `NIL`, `CONS` and `List_case` will continue to work, even for the infinite lists. If N is an infinite list then `CONS` M N is also infinite.

Since `List` A contains no infinite lists, we may instantiate the `lfp` induction rule to obtain structural induction (§3.1). By induction we may prove the properties expected of a WF data structure, such as

$$\frac{N \in \texttt{List}\ A}{\texttt{CONS}\ M\ N \neq N} \tag{5}$$

The Isabelle theory proceeds to define the type $\alpha\,\texttt{list}$ to contain those values of type $\alpha\,\texttt{node}\,\texttt{set}$ that belong to the set $\texttt{List}(\texttt{range}(\texttt{Leaf}))$. If x_1, x_2, ... , x_n have type α then the list $[x_1, x_2, \ldots, x_n]$ is represented by

$$\texttt{CONS}\,(\texttt{Leaf}\ x_1)(\texttt{CONS}\,(\texttt{Leaf}\ x_2)(\ldots\texttt{CONS}\,(\texttt{Leaf}\ x_n)\ \texttt{NIL}\ldots)).$$

Type $\alpha\,\texttt{list}$ has two constructors, $\texttt{Nil} :: \alpha\,\texttt{list}$ and $\texttt{Cons} :: [\alpha, \alpha\,\texttt{list}] \Rightarrow \alpha\,\texttt{list}$, and the usual induction rule, recursion operator, etc. This type definition (§2.5) is the final stage in making lists convenient to use; the details are routine and omitted.

5.2 A space for well-founded types

Lisp's symbolic expressions are built up from identifiers and numbers by pairing. Formalizing S-expressions in HOL further demonstrates the Isabelle theory. More importantly, it leads to a uniform treatment of recursive functions for virtually all finitely branching WF data structures. An essential feature of S-expressions is that they are finite. Therefore, let us define them using `lfp`:

$$\texttt{Sexp} \equiv \texttt{lfp}(\lambda Z.\ \texttt{range}(\texttt{Leaf}) \cup \texttt{range}(\texttt{Numb}) \cup (Z \otimes Z)) \tag{6}$$

Observe how `range` expresses the sets of all `Leaf` a and `Numb` k constructions. We can develop `Sexp` in the same manner as `List` A, deriving an induction rule, a case analysis operator, etc.

But `Sexp` is a rather special subset of our universe, itself suitable for defining recursive data structures. It contains all constructions of the form `Leaf` a, `Numb` k and $M \cdot N$, and therefore also `In0` M and `In1` N. Thus it is closed under $\otimes$ and $\oplus$. Defined by `lfp`, all the constructions in it are finite.

Now `Sexp` is not necessarily large enough to contain `List` A for arbitrary A, since A might contain infinite constructions. But `Sexp` is closed under `List`: we can easily prove $\texttt{List}(\texttt{Sexp}) \subseteq \texttt{Sexp}$, expressing that lists of finite constructions are themselves finite constructions. Similarly, `Sexp` is closed under many other finite data structures.

Recursion on `Sexp` gives us recursion on all these WF data structures. Isabelle's HOL theory contains a derivation of WF recursion. This justifies defining any function whose recursive calls decrease its argument under some WF relation. The *immediate subexpression* relation on `Sexp` is the set of pairs

$$(\prec) \equiv \bigcup_{M \in \texttt{Sexp}} \bigcup_{N \in \texttt{Sexp}} \{(M, M \cdot N),\ (N, M \cdot N)\}.$$

Structural induction on Sexp proves that this relation is WF. The transitive closure of a relation can be defined using lfp, and can be proved to preserve wellfoundedness [26]. So $M \prec^+ N$ expresses that M is a subexpression of N. WF recursion justifies any function on S-expressions whose recursive calls take subexpressions of the original argument.

Let us apply this to lists. Recall that

$$\texttt{CONS}\ M\ N \equiv \texttt{In1}(M \cdot N) \equiv \texttt{Numb}\ 1 \cdot (M \cdot N).$$

A sublist is therefore a subexpression. Structural recursion on lists is an instance of WF recursion. Suppose f is defined by

$$\begin{aligned} f\ M \equiv \texttt{wfrec}\,(\prec^+)\ M\,(\lambda M\,g.\ \texttt{case}\ M\ \texttt{of}\ &\texttt{NIL} \Rightarrow c \\ &\mid \texttt{CONS}\ x\ y \Rightarrow d\ x\ y\,(g\ y)) \end{aligned}$$

We immediately obtain $f\ \texttt{NIL} = c$. The fact $N \prec^+ \texttt{CONS}\ M\ N$ justifies the recursion equation

$$\frac{M \in \texttt{Sexp} \qquad N \in \texttt{Sexp}}{f(\texttt{CONS}\ M\ N) = d\ M\ N\,(f\ N).}$$

Most familiar list functions — append, reverse, map — have obvious definitions by structural recursion. But the theory does not insist upon structural recursion; it can express functions such as quicksort in their natural form, using WF recursion in its full generality.

Definition (6) is not the only possible way of characterizing the set of finite constructions. We could instead formalize the finite powerset operator using lfp [26]. The set of all finite, non-empty sets of nodes would be larger than Sexp while satisfying the same key closure properties. Defining a suitable WF relation on this set might be tedious.

The Isabelle/HOL theory of WF data structures is quite general, at least for finite branching. Non-WF data structures pose greater challenges.

6 Lazy lists and coinduction

Defining the set of lists as a greatest instead of a least fixedpoint admits infinite as well as finite lists. We can then model computation and equality, realizing to some extent the theory of lists in lazy functional languages [5]. However, our "lazy lists" have no inbuilt operational semantics; after all, HOL can express non-computable functions.

The set of lazy lists is defined by $\texttt{LList}\ A \equiv \texttt{gfp}(\texttt{List\_Fun}\ A)$; recall that List_Fun and the list operations NIL, CONS and List_case were defined in §5.1. The fixedpoint property yields introduction rules for LList A:

$$\texttt{NIL} \in \texttt{LList}\ A \qquad \frac{M \in A \quad N \in \texttt{LList}\ A}{\texttt{CONS}\ M\ N \in \texttt{LList}\ A}$$

These may resemble their finite list counterparts (§3.1), but they differ significantly, for CONS is well-behaved even when applied to an infinite list. In particular, the List_case equation (4) works for everything of the form CONS M N.

Since $\mathtt{LList}\ A$ is a greatest fixedpoint, it does not have a structural induction principle. Well-foundedness properties such as the list theorem (5) have no counterparts for $\mathtt{LList}\ A$. We can construct a counterexample and prove that it belongs to $\mathtt{LList}\ A$ by coinduction.

The weak coinduction rule for $\mathtt{LList}\ A$ performs type checking for infinite lists:

$$\frac{M \in X \quad X \subseteq \mathtt{List\_Fun}\ A\ X}{M \in \mathtt{LList}\ A} \tag{7}$$

The strong coinduction rule, as described in §3.2, implicitly includes $\mathtt{LList}\ A$:

$$\frac{M \in X \quad X \subseteq \mathtt{List\_Fun}\ A\ X \cup \mathtt{LList}\ A}{M \in \mathtt{LList}\ A} \tag{8}$$

6.1 An infinite list

One non-WF list is the infinite list of Ms:

$$\mathtt{Lconst}\ M = \mathtt{CONS}\ M\,(\mathtt{Lconst}\ M) \tag{9}$$

Corecursion, a general method for defining infinite lists, is discussed below. For now, let us construct $\mathtt{Lconst}\ M$ explicitly as a fixedpoint:

$$\mathtt{Lconst}\ M \equiv \mathtt{lfp}(\lambda Z.\, \mathtt{CONS}\ M\ Z)$$

The Knaster-Tarski Theorem applies because lists are sets of nodes and CONS is monotonic in both arguments. (Recall from §4.4 that $(\cdot)$ is defined in terms of the monotonic operations union and image.) I have used lfp but gfp would work just as well — we need only the fixedpoint property (9).

We cannot prove that $\mathtt{Lconst}\ M$ is a lazy list by the introduction rules alone. The coinduction rule (7) proves $\mathtt{Lconst}\ M \in \mathtt{LList}\{M\}$ if we can find a set X containing $\mathtt{Lconst}\ M$ and included in $\mathtt{List\_Fun}\,\{M\}\,X$. A suitable X is $\{\mathtt{Lconst}\ M\}$. Obviously $\mathtt{Lconst}\ M \in \{\mathtt{Lconst}\ M\}$. We must also show

$$\{\mathtt{Lconst}\ M\} \subseteq \mathtt{List\_Fun}\,\{M\}\,\{\mathtt{Lconst}\ M\}.$$

The fixedpoint property (9) transforms this to

$$\{\mathtt{CONS}\ M\,(\mathtt{Lconst}\ M)\} \subseteq \mathtt{List\_Fun}\,\{M\}\,\{\mathtt{Lconst}\ M\},$$

which is obvious by the definitions of CONS and List_Fun.

Deriving introduction rules for List_Fun allows shorter machine proofs:

$$\mathtt{NIL} \in \mathtt{List\_Fun}\ A\ X \tag{10}$$

$$\frac{M \in A \quad N \in X}{\mathtt{CONS}\ M\ N \in \mathtt{List\_Fun}\ A\ X} \tag{11}$$

6.2 Equality of lazy lists; the take-lemma

Because HOL lazy lists are sets of nodes, the equality relation on $\mathtt{LList}\ A$ is an instance of ordinary set equality. By investigating this relation further, we obtain nice, coinductive methods for proving that two lazy lists are equal.

Let $\mathtt{take}\ k\ l$ return l's first k elements as a finite list. Bird and Wadler [5] use the take-lemma to prove equality of lazy lists l_1 and l_2:

$$\frac{\forall k\,\mathtt{take}\ k\ l_1 = \mathtt{take}\ k\ l_2}{l_1 = l_2}$$

It embodies a continuity principle shared by our HOL formalization.

The definition (2) of nodes (in §4.3) requires each node to have a finite depth. A node contains a pair (f, x), where x is the label and f codes the position. Our definition ensures $f\ k = 0$ for some k, which is the depth of the node. We can formalize the depth directly, using the least number principle and pattern matching:

$$\begin{aligned}\mathtt{LEAST}\ k.\,\phi k &\equiv \epsilon k.\,\phi k \wedge (\forall j\,(j < k \to \neg\phi j))\\ \mathtt{ndepth}\,(\mathtt{Abs\_Node}(f, x)) &\equiv \mathtt{LEAST}\ k.\,f\ k = 0\end{aligned}$$

Our generalization of `take`, called `ntrunc`, applies to all data structures, not just lists. It returns the set of all nodes having less than a given depth:

$$\mathtt{ntrunc}\ k\ N \equiv \{nd \mid nd \in N \wedge \mathtt{ndepth}\ nd < k\}$$

Elementary reasoning derives results describing `ntrunc`'s effect upon various constructions:

$$\begin{aligned}\mathtt{ntrunc}\ 0\ M &= \{\}\\ \mathtt{ntrunc}\,(\mathtt{Suc}\ k)\,(\mathtt{Leaf}\ a) &= \mathtt{Leaf}\ a\\ \mathtt{ntrunc}\,(\mathtt{Suc}\ k)\,(\mathtt{Numb}\ a) &= \mathtt{Numb}\ a\\ \mathtt{ntrunc}\,(\mathtt{Suc}\ k)\,(M \cdot N) &= \mathtt{ntrunc}\ k\ M \cdot \mathtt{ntrunc}\ k\ N\end{aligned}$$

Since $\mathtt{In0}\ M \equiv \mathtt{Numb}\ 0 \cdot M$ we obtain

$$\begin{aligned}\mathtt{ntrunc}\ 1\,(\mathtt{In0}\ M) &= \{\}\\ \mathtt{ntrunc}\,(\mathtt{Suc}(\mathtt{Suc}\ k))\,(\mathtt{In0}\ M) &= \mathtt{In0}(\mathtt{ntrunc}\,(\mathtt{Suc}\ k)\ M)\end{aligned}$$

and similarly for `In1`.

Our generalization of `take` enjoys a generalization of the take-lemma:

Lemma 1 If $\mathtt{ntrunc}\ k\ M = \mathtt{ntrunc}\ k\ N$ for all k then $M = N$.

This obvious fact is a key result. It gives us a method for proving the equality of any constructions M and N. We could apply this "ntrunc-lemma" directly, but instead we shall package it into a form suitable for coinduction.

6.3 Diagonal set operators

In order to prove list equations by coinduction, we must demonstrate that the equality relation is the greatest fixedpoint of some monotone operator. To this end, we define diagonal set operators for $\otimes$ and $\oplus$. A *diagonal set* has the form $\{(x,x)\}_{x\in A}$, internalizing the equality relation on A.

A binary relation on sets of nodes has type $(\alpha\,\texttt{node}\,\texttt{set} \times \alpha\,\texttt{node}\,\texttt{set})\,\texttt{set}$. The operators $\otimes_D$ and $\oplus_D$ combine two such relations to yield a third. The operator `diag`, of type $\alpha\,\texttt{set} \Rightarrow (\alpha \times \alpha)\,\texttt{set}$, constructs arbitrary diagonal sets.

$$\begin{aligned}
\texttt{diag}\,A &\equiv \bigcup\nolimits_{x\in A}\{(x,x)\}\\
r \otimes_D s &\equiv \bigcup\nolimits_{(x,x')\in r}\bigcup\nolimits_{(y,y')\in s}\{(x\cdot y,\; x'\cdot y')\}\\
r \oplus_D s &\equiv \bigcup\nolimits_{(x,x')\in r}\{(\texttt{In0}\,x,\; \texttt{In0}\,x')\} \;\cup\; \bigcup\nolimits_{(y,y')\in s}\{(\texttt{In1}\,y,\; \texttt{In1}\,y')\}
\end{aligned}$$

These enjoy readable introduction rules. For $\otimes_D$ we have

$$\frac{(M,M')\in r \qquad (N,N')\in s}{(M\cdot N,\; M'\cdot N')\in r\otimes_D s}$$

while for $\oplus_D$ we have the pair of rules

$$\frac{(M,M')\in r}{(\texttt{In0}\,M,\; \texttt{In0}\,M')\in r\oplus_D s} \qquad \frac{(N,N')\in s}{(\texttt{In1}\,N,\; \texttt{In1}\,N')\in r\oplus_D s}$$

The idea is that $\otimes_D$ and $\oplus_D$ build relations in the same manner as $\otimes$ and $\oplus$ build sets. Since $\texttt{fst}\,``\,r$ is the first projection of the relation r, we can summarize the idea by three obvious equations:

$$\begin{aligned}
\texttt{fst}\,``\,\texttt{diag}\,A &= A\\
\texttt{fst}\,``\,(r\otimes_D s) &= (\texttt{fst}\,``\,r)\otimes(\texttt{fst}\,``\,s)\\
\texttt{fst}\,``\,(r\oplus_D s) &= (\texttt{fst}\,``\,r)\oplus(\texttt{fst}\,``\,s)
\end{aligned}$$

Category theorists may note that $\otimes$ and $\oplus$ are functors on a category of sets where the morphisms are binary relations; in this category, $\otimes_D$ and $\oplus_D$ give the functors' effects on the morphisms. Next, we shall do the same thing to the functor `LList`.

6.4 Equality of lazy lists as a `gfp`

Just as $\otimes_D$ and $\oplus_D$ extend $\otimes$ and $\oplus$ to act upon relations, let `LListD` extend `LList`. Thanks to our new operators, the definition is simple and resembles that of `LList`:

$$\begin{aligned}
\texttt{LListD\_Fun}\,r &\equiv \lambda Z.\,\texttt{diag}\{\texttt{Numb}\;0\}\oplus_D(r\otimes_D Z)\\
\texttt{LListD}\,r &\equiv \texttt{gfp}(\texttt{LListD\_Fun}\,r)
\end{aligned}$$

The theorem that list equality is a `gfp` can now be stated as a succinct equation between relations: $\texttt{LListD}(\texttt{diag}\;A) = \texttt{diag}(\texttt{LList}\;A)$. Here $\texttt{diag}(\texttt{LList}\;A)$ is the equality relation on $\texttt{LList}\;A$, while $\texttt{LListD}(\texttt{diag}\;A)$ is the `gfp` of $\texttt{LListD\_Fun}(\texttt{diag}\,A)$. Proving this requires another lemma about `ntrunc`.

Lemma 2 $\forall M\, N\, [(M,N) \in \texttt{LListD(diag}\ A) \to \texttt{ntrunc}\ k\ M = \texttt{ntrunc}\ k\ N]$.

Proof By complete induction on k, we may assume the formula above after replacing k by any smaller natural number j. By the fixedpoint property

$$\texttt{LListD(diag}\ A) = \texttt{diag}\{\texttt{Numb}\ 0\} \oplus_D (r \otimes_D \texttt{LListD(diag}\ A)),$$

if $(M,N) \in \texttt{LListD(diag}\ A)$ then there are two cases. If

$$M = N = \texttt{In0(Numb}\ 0) = \texttt{NIL}$$

then $\texttt{ntrunc}\ k\ M = \texttt{ntrunc}\ k\ N$ is trivial. Otherwise $M = \texttt{CONS}\ x\ M'$ and $N = \texttt{CONS}\ x\ N'$, where $(M',N') \in \texttt{LListD(diag}\ A)$. Recall the definition $\texttt{CONS}\ x\ y \equiv \texttt{In1}(x \cdot y)$ and the properties of $\texttt{ntrunc}$ (§6.2); we obtain

$$\texttt{ntrunc}\ k\ (\texttt{CONS}\ x\ y) = \begin{cases} \{\} & \text{if } k < 2 \text{, and} \\ \texttt{CONS}\ (\texttt{ntrunc}\ j\ x)\ (\texttt{ntrunc}\ j\ y) & \text{if } k = \texttt{Suc(Suc}\ j). \end{cases}$$

If $k = \texttt{Suc(Suc}\ j)$ then $\texttt{ntrunc}\ k\ M = \texttt{ntrunc}\ k\ N$ reduces to an instance of the induction hypothesis, $\texttt{ntrunc}\ j\ M' = \texttt{ntrunc}\ j\ N'$.

Now we can prove that equation.

Proposition 3 $\texttt{LListD(diag}\ A) = \texttt{diag(LList}\ A)$.

Proof Combining Lemmas 1 and 2 yields half of our desired result, $\texttt{LListD(diag}\ A) \subseteq \texttt{diag(LList}\ A)$. This is the more important half: it lets us show $M = N$ by showing $(M,N) \in \texttt{LListD(diag}\ A)$, which can be done using coinduction.

The opposite inclusion, $\texttt{diag(LList}\ A) \subseteq \texttt{LListD(diag}\ A)$, follows by showing that $\texttt{diag(LList}\ A)$ is a fixedpoint of $\texttt{LListD\_Fun(diag}\,A)$, since $\texttt{LListD(diag}\ A)$ is the greatest fixedpoint. This argument is an example of coinduction.

6.5 Proving lazy list equality by coinduction

The weak coinduction rule for list equality yields $M = N$ provided $(M,N) \in r$ where r is a suitable bisimulation between lazy lists:

$$\frac{(M,N) \in r \qquad r \subseteq \texttt{LListD\_Fun(diag}\,A)r}{M = N} \tag{12}$$

Coinduction has many variant forms (§3.2). Strong coinduction includes the equality relation implicitly in every bisimulation:

$$\frac{(M,N) \in r \qquad r \subseteq \texttt{LListD\_Fun(diag}\,A)r \cup \texttt{diag(LList}\ A)}{M = N} \tag{13}$$

Expanding the definitions of NIL, CONS and LListD_Fun creates unwieldy formulae. The Isabelle theory derives two rules to avoid this, resembling the List_Fun rules (10) and (11):

$$(\texttt{NIL}, \texttt{NIL}) \in \texttt{LListD\_Fun(diag}\,A)r \tag{14}$$

$$\frac{x \in A \quad (M,N) \in r}{(\texttt{CONS}\ x\ M, \texttt{CONS}\ x\ N) \in \texttt{LListD\_Fun(diag}\,A)r} \tag{15}$$

7 Lazy lists and corecursion

We have defined the infinite list $\texttt{Lconst}\ M = [M, M, \ldots]$ using a fixedpoint. The construction clearly generalizes to other repetitive lists such as $[M, N, M, N, \ldots]$. But how can we define infinite lists such as $[1, 2, 3, \ldots]$? And how can we define the usual list operations, like append and map? Structural recursive definitions would work for elements of $\texttt{List}\ A$ but not for the infinite lists in $\texttt{LList}\ A$.

Corecursion is a dual form of structural recursion. Recursion defines functions that consume lists, while corecursion defines functions that create lists. Corecursion originated in the category theoretic notion of final coalgebra; Mendler [19] and Geuvers [12], among others, have investigated it in type theories.

This paper does not attempt to treat corecursion categorically. And instead of describing the general case in all its complexity, it simply treats a key example: lazy lists. Let us begin with motivation and examples.

7.1 Introduction to corecursion

Corecursion defines a lazy list in terms of some seed value $a :: \alpha$ and a function $f :: \alpha \Rightarrow \texttt{unit} + (\beta\, \texttt{node}\, \texttt{set} \times \alpha)$. Recall from §2.3 that $\texttt{unit}$ is the nullary product type, whose sole value is $()$, while $\times$ and $+$ are the product and sum type operators. Thus $\texttt{LList\_corec}$ has type

$$[\alpha, \alpha \Rightarrow \texttt{unit} + (\beta\, \texttt{node}\, \texttt{set} \times \alpha)] \Rightarrow \beta\, \texttt{node}\, \texttt{set}.$$

It must satisfy

$$\texttt{LList\_corec}\ a\ f = \begin{cases} \texttt{NIL} & \text{if } f\ a = \texttt{Inl}\,(); \\ \texttt{CONS}\ x\,(\texttt{LList\_corec}\ b\ f) & \text{if } f\ a = \texttt{Inr}\,(x, b). \end{cases}$$

The idea should be clear: f takes the seed a and either returns $\texttt{Inl}\,()$, to end the list here, or returns $\texttt{Inr}\,(x, b)$, to continue the list with next element x and seed b. By keeping the seed forever M and always returning it as the next element, corecursion can express $\texttt{Lconst}\ M$:

$$\texttt{Lconst}\ M \equiv \texttt{LList\_corec}\ M(\lambda N. \texttt{Inr}\,(N, N))$$

Consider the functional $\texttt{Lmap}$, which applies a function to every element of a list:

$$\texttt{Lmap}\ g\ [x_0, x_1, \ldots, x_n, \ldots] = [g\, x_0, g\, x_1, \ldots, g\, x_n, \ldots]$$

The usual recursion equations are

$$\texttt{Lmap}\ g\ \texttt{NIL} = \texttt{NIL} \tag{16}$$

$$\texttt{Lmap}\ g\,(\texttt{CONS}\ M\ N) = \texttt{CONS}\,(g\ M)\,(\texttt{Lmap}\ g\ N). \tag{17}$$

Corecursion handles these easily. To compute $\texttt{Lmap}\ g\ M$, take M as the seed. If $M = \texttt{NIL}$ then end the result list; if $M = \texttt{CONS}\ x\ M'$ then continue the result list with next element $g\ x$ and seed M'. The formal definition uses $\texttt{List\_case}$ to inspect M:

$$\begin{aligned} \texttt{Lmap}\ g\ M \equiv \texttt{LList\_corec}\ M\,(\lambda M.\ \texttt{case}\ M\ \texttt{of}\ & \texttt{NIL} \Rightarrow \texttt{Inl}\,() \\ \mid\ & \texttt{CONS}\ x\ M' \Rightarrow \texttt{Inr}\,(g\ x,\ M')) \end{aligned}$$

This definition of map has little in common with the standard recursive one. Append comes out stranger still, and other standard functions seem to be lost altogether.

7.2 Harder cases for corecursion

With corecursion, the case analysis is driven by the output list, rather than the input list. The append function highlights this peculiarity. The usual recursion equations for append perform case analysis on the first argument:

$$\begin{aligned}\texttt{Lappend}\,\texttt{NIL}\;N &= N\\ \texttt{Lappend}\,(\texttt{CONS}\;M_1\;M_2)\;N &= \texttt{CONS}\;M_1\;(\texttt{Lappend}\;M_2\;N)\end{aligned}$$

But a NIL input does not guarantee a NIL output, as it did for Lmap; consider

$$\texttt{Lappend}\,\texttt{NIL}\,(\texttt{CONS}\;M\;N) = \texttt{CONS}\;M\;N.$$

The correct equations for corecursion involve both arguments:

$$\begin{aligned}\texttt{Lappend}\,\texttt{NIL}\,\texttt{NIL} &= \texttt{NIL} && (18)\\ \texttt{Lappend}\,\texttt{NIL}\,(\texttt{CONS}\;N_1\;N_2) &= \texttt{CONS}\;N_1\;(\texttt{Lappend}\,\texttt{NIL}\;N_2) && (19)\\ \texttt{Lappend}\,(\texttt{CONS}\;M_1\;M_2)\;N &= \texttt{CONS}\;M_1\;(\texttt{Lappend}\;M_2\;N) && (20)\end{aligned}$$

The second line above forces Lappend NIL N to continue executing until it has made a copy of N. This looks inefficient. In the context of the polymorphic λ-calculus, Geuvers [12] discusses stronger forms of corecursion that allow the seed to return an entire list at once. But my HOL theory has no operational significance; efficiency is meaningless; we may as well keep corecursion simple.[4]

The seed for Lappend M N is the pair (M, N). The corecursive definition performs case analysis on both lists:

- If $M = \texttt{NIL}$ then it looks at N:
  - If $N = \texttt{NIL}$ then end the result list.
  - If $N = \texttt{CONS}\;N_1\;N_2$ then continue the result list with next element N_1 and seed $(\texttt{NIL}, N_2)$.
- If $M = \texttt{CONS}\;M_1\;M_2$ then continue the result list with next element M_1 and seed (M_2, N).

We can formalize this using LList_corec. Define f by case analysis:

$$\begin{aligned}f \equiv \lambda(M,N).\;&\texttt{case}\;M\;\texttt{of}\\ &\quad\texttt{NIL} \Rightarrow (\texttt{case}\;N\;\texttt{of}\;\;\texttt{NIL} \Rightarrow \texttt{Inl}\,()\\ &\qquad\qquad\qquad\qquad\mid \texttt{CONS}\;N_1\;N_2 \Rightarrow \texttt{Inr}\,(N_1,(\texttt{NIL},N_2)))\\ &\quad\mid \texttt{CONS}\;M_1\;M_2 \Rightarrow \texttt{Inr}\,(M_1,(M_2,N))\end{aligned}$$

Then put Lappend M $N \equiv$ LList_corec (M, N) f.

[4] This relates to difficulties with formalizing (co)inductive definitions in type theory. Under some formalizations, the constructors of coinductive types are complicated and inefficient — and dually, so are the destructors of inductive types. This does not occur in my HOL treatment: constructors and destructors are directly available from the fixedpoint equations. Thus we must not push the analogy with type theory too far.

Concatenation. Now let us try to define a function to flatten a list of lists according to the following recursion equations:

$$\texttt{Lflat}\,\texttt{NIL} = \texttt{NIL} \tag{21}$$
$$\texttt{Lflat}(\texttt{CONS}\ M\ N) = \texttt{Lappend}\ M\,(\texttt{Lflat}\ N) \tag{22}$$

Since corecursion is driven by the output, we need to know when a `CONS` is produced. We can refine equation (22) by further case analysis:

$$\begin{aligned}\texttt{Lflat}(\texttt{CONS}\,\texttt{NIL}\ N) &= \texttt{Lflat}\ N\\ \texttt{Lflat}(\texttt{CONS}\,(\texttt{CONS}\ M_1\ M_2)\ N) &= \texttt{CONS}\ M_1\,(\texttt{Lflat}(\texttt{CONS}\ M_2\ N))\end{aligned}$$

Unfortunately, the outcome of `CONS` `NIL` N is still undecided; it could be `NIL` or `CONS`. Given the argument `Lconst` `NIL` — an infinite list of `NIL`s — `Lflat` should run forever. There is no effective way to check whether an infinite list contains a non-`NIL` element.

I do not know of any natural formalization of `Lflat`. Since HOL can formalize non-computable functions, we can force `Lflat` (`Lconst` `NIL`) = `NIL` using a description. Such a definition would be of little relevance to computer science. Really `Lflat` (`Lconst` `NIL`) should be undefined, but HOL does not admit partial functions.

The filter functional, which removes from a list all elements that fail to satisfy a given predicate, poses similar problems. If no list elements satisfy the predicate, then the result is logically `NIL`, but there is no effective way to reach this result. Leclerc and Paulin-Mohring [17] discuss approaches to this problem in the context of the Coq system.

7.3 Deriving corecursion

The characteristic equation for corecursion can be stated using case analysis and pattern matching:

$$\begin{aligned}\texttt{LList\_corec}\ a\ f = \texttt{case}\ &f\ a\ \texttt{of}\\ &\texttt{Inl}\ u \quad\ \Rightarrow \texttt{NIL}\\ \mid\ &\texttt{Inr}(x,b) \Rightarrow \texttt{CONS}\ x\,(\texttt{LList\_corec}\ b\ f)\end{aligned} \tag{23}$$

To realize this equation, recall that an element of `LList` A is a set of nodes, each of finite depth. A primitive recursive function can approximate the corecursion operator for nodes up to some given depth k:

$$\begin{aligned}\texttt{lcorf}\ 0\ a\ f &= \{\}\\ \texttt{lcorf}\,(\texttt{Suc}\ k)\ a\ f &= \texttt{case}\ f\ a\ \texttt{of}\\ &\qquad \texttt{Inl}\ u \quad\ \Rightarrow \texttt{NIL}\\ &\quad\ \mid \texttt{Inr}(x,b) \Rightarrow \texttt{CONS}\ x\,(\texttt{lcorf}\ k\ b\ f)\end{aligned}$$

Now we can easily define corecursion:

$$\texttt{LList\_corec}\ a\ f \equiv \bigcup_k \texttt{lcorf}\ k\ a\ f$$

Equation (23) can then be proved as two separate inclusions, with simple reasoning about unions and (in one direction) induction on k.

The careful reader may have noticed that $\mathtt{lcorf}\ k\ a\ f$ is not strictly primitive recursive, because the parameter a varies in the recursive calls. To be completely formal, we must define a function over a using higher-order primitive recursion. In the Isabelle theory, $\mathtt{lcorf}\ k\ a\ f$ becomes $\mathtt{LList\_corec\_fun}\ k\ f\ a$. It can be defined by primitive recursion:

$$\begin{aligned}
\mathtt{LList\_corec\_fun}\ 0\ f &= \lambda a.\,\{\} \\
\mathtt{LList\_corec\_fun}\ (\mathtt{Suc}\ k)\ f &= \lambda a.\,\mathtt{case}\ f\ a\ \mathtt{of} \\
&\qquad\quad \mathtt{Inl}\, u \qquad \Rightarrow \mathtt{NIL} \\
&\qquad \mid \mathtt{Inr}(x,b) \Rightarrow \mathtt{CONS}\ x\, (\mathtt{LList\_corec\_fun}\ k\ f\ b)
\end{aligned}$$

Obscure as this may look, it trivially implies the equations for $\mathtt{lcorf}$ given above.

8 Examples of coinduction and corecursion

This section demonstrates defining functions by corecursion and verifying their equational and typing properties by coinduction. All these proofs have been checked by machine.

8.1 A type-checking rule for LList_corec

One advantage of $\mathtt{LList\_corec}$ over ad-hoc definitions is that the result is certain to be a lazy list. To express this well-typing property in the simplest possible form, let $U :: \alpha\,\mathtt{node}\,\mathtt{set}\,\mathtt{set}$ abbreviate the universal set of constructions: $U \equiv \{x \mid \mathtt{True}\}$. We can now state a typing result:

Example 4 $\mathtt{LList\_corec}\ a\ f \in \mathtt{LList}\ U$.

Proof Apply the weak coinduction rule (7) to the set

$$V \equiv \mathtt{range}(\lambda x.\ \mathtt{LList\_corec}\ x\ f).$$

We must show $V \subseteq \mathtt{List\_Fun}\ U\ V$, which reduces to

$$\mathtt{LList\_corec}\ a\ f \in \mathtt{List\_Fun}\ U\ V.$$

There are two cases. We simplify each case using equation (23):

- If $f\ a = \mathtt{Inl}\,()$, it reduces to $\mathtt{NIL} \in \mathtt{List\_Fun}\ U\ V$, which holds by Rule (10).
- If $f\ a = \mathtt{Inr}\,(x,b)$, it reduces to

  $$\mathtt{CONS}\ x\,(\mathtt{LList\_corec}\ b\ f) \in \mathtt{List\_Fun}\ U\ V.$$

  This holds by Rule (11) because $x \in U$ and $\mathtt{LList\_corec}\ b\ f \in V$.

This result, referring to the universal set, may seem weak compared with our previous well-typing result, $\mathtt{Lconst}\ M \in \mathtt{LList}\{M\}$, from §6.1. It is difficult to bound the set of possible values for x for the case $f\ a = \mathtt{Inr}\,(x,b)$. Sharper results can be obtained by performing separate coinduction proofs for each function defined by corecursion.

8.2 The uniqueness of corecursive functions

Equation (23) characterizes $\texttt{LList\_corec}$ uniquely; the proof is a typical example of proving an equation by coinduction. Let us define an abbreviation for the corecursion property:

$$\begin{array}{lll} \texttt{is\_corec}\ f\ h \equiv \forall u\, [h\ u = \texttt{case}\ f\ u\ \texttt{of} & \texttt{Inl}\ v & \Rightarrow \texttt{NIL} \\ & \mid \texttt{Inr}(x,b) & \Rightarrow \texttt{CONS}\ x\,(h\ b)] \end{array}$$

Existence, namely $\texttt{is\_corec}\ f\,(\lambda x.\,\texttt{LList\_corec}\ x\ f)$, is immediate by equation (23). Let us now prove uniqueness.

Proposition 5 If $\texttt{is\_corec}\ f\ h_1$ and $\texttt{is\_corec}\ f\ h_2$ then $h_1 = h_2$.

Proof By extensionality it suffices to prove $h_1\ u = h_2\ u$ for all u. Now consider the bisimulation $\{(h_1\ u, h_2\ u)\}_u$, formalized in HOL by

$$r \equiv \texttt{range}(\lambda u.\,(h_1\ u, h_2\ u)).$$

Apply the coinduction rule (12), with r as above and putting $A \equiv U$, the universal set. We must show $r \subseteq \texttt{LListD\_Fun}(\texttt{diag}\,U)r$; since elements of r have the form $(h_1\ u, h_2\ u)$ for arbitrary u, it suffices to show

$$(h_1\ u, h_2\ u) \in \texttt{LListD\_Fun}(\texttt{diag}\,U)r.$$

There are two cases, determined by $f(u)$. In each, we apply the corecursion properties of h_1 and h_2:

- If $f\ u = \texttt{Inl}\,()$, it reduces to $(\texttt{NIL},\texttt{NIL}) \in \texttt{LListD\_Fun}(\texttt{diag}\,U)r$, which holds by Rule (14).
- If $f\ u = \texttt{Inr}\,(x,b)$, it reduces to

  $$(\texttt{CONS}\ x\,(h_1\ b), \texttt{CONS}\ x\,(h_2\ b)) \in \texttt{LListD\_Fun}(\texttt{diag}\,U)r.$$

  This holds by Rule (15) because $x \in U$ and $(h_1\ b, h_2\ b) \in r$.

Note the similarity between this coinduction proof and the previous one, even though the former proves set membership while the latter proves an equation. The role of r in this equality proof is reminiscent of process equivalence proofs in CCS [20], where the bisimulation associates corresponding states of two processes. The equality can also be proved using Lemma 1 directly, by complete induction on k; the proof is considerably more complex.

8.3 A proof about the map functional

To demonstrate that the theory has some relevance to lazy functional programming, this section proves a simple result: that map distributes over composition.

Example 6 If $M \in \texttt{LList}\ A$ then $\texttt{Lmap}\,(f \circ g)\ M = \texttt{Lmap}\ f\,(\texttt{Lmap}\ g\ M)$.

Proof The bisimulation $\{(\texttt{Lmap}\,(f \circ g)\ u,\ \texttt{Lmap}\ f\,(\texttt{Lmap}\ g\ u))\}_{u \in \texttt{LList}\ A}$ may be formalized using the image operator:

$$r \equiv (\lambda u.\,(\texttt{Lmap}\,(f \circ g)\ u,\ \texttt{Lmap}\ f\,(\texttt{Lmap}\ g\ u)))\ ``\,\texttt{LList}\ A$$

As in the previous coinduction proof, apply Rule (12). The key is to show

$$(\texttt{Lmap}\,(f \circ g)\ u,\ \texttt{Lmap}\ f\,(\texttt{Lmap}\ g\ u)) \in \texttt{LListD\_Fun}(\texttt{diag}\,U)r$$

for arbitrary $u \in \texttt{LList}\ A$. There are again two cases, this time depending on the form of u. We simplify each case using the recursion equations for Lmap, which follow from its corecursive definition (§7.1).

- If $u = \texttt{NIL}$, the goal reduces to $(\texttt{NIL}, \texttt{NIL}) \in \texttt{LListD\_Fun}(\texttt{diag}\,U)r$, which holds by Rule (14).
- If $u = \texttt{CONS}\ M'\ N$, it reduces to

  $$\begin{array}{l}(\texttt{CONS}\,(f\,(g\ M'))\,(\texttt{Lmap}\,(f \circ g)\ N),\\ \ \texttt{CONS}\,(f\,(g\ M'))\,(\texttt{Lmap}\ f\,(\texttt{Lmap}\ g\ N))) \ \in\ \texttt{LListD\_Fun}(\texttt{diag}\,U)r.\end{array}$$

  This holds by Rule (15) because $f\,(g\ M') \in U$ and

  $$(\texttt{Lmap}\,(f \circ g)\ N,\ \texttt{Lmap}\ f\,(\texttt{Lmap}\ g\ N)) \in r.$$

The coinductive argument bears little resemblance to the usual proof in the Logic for Computable Functions (LCF) [22, page 283]. On the other hand, all these coinductive proofs have a monotonous regularity. A similar argument proves $\texttt{Lmap}\,(\lambda x.x)\ M = M$.

8.4 Proofs about the list of iterates

Consider the function Iterates defined by

$$\texttt{Iterates}\ f\ M \equiv \texttt{LList\_corec}\ M(\lambda M.\,\texttt{Inr}\,(M, f\ M)).$$

A generalization of Lconst, it constructs the infinite list $[M, f\,M, \ldots, f^n\,M, \ldots]$ by the recursion equation

$$\texttt{Iterates}\ f\ M = \texttt{CONS}\ M\,(\texttt{Iterates}\ f\,(f\ M)).$$

The equation

$$\texttt{Lmap}\ f\,(\texttt{Iterates}\ f\ M) = \texttt{Iterates}\ f\,(f\ M) \tag{24}$$

has a straightforward coinductive proof, using the bisimulation

$$\{(\texttt{Lmap}\ f\,(\texttt{Iterates}\ f\ u),\ \texttt{Iterates}\ f\,(f\ u))\}_{u \in \texttt{LList}\ A}.$$

Combining the previous two equations yields a new recursion equation, involving Lmap:

$$\texttt{Iterates}\ f\ M = \texttt{CONS}\ M\,(\texttt{Lmap}\ f\,(\texttt{Iterates}\ f\ M))$$

Harder is to show that this equation uniquely characterizes Iterates f.

Example 7 If $h\ x = \texttt{CONS}\ x\,(\texttt{Lmap}\ f\ (h\ x))$ for all x then $h = \texttt{Iterates}\ f$.

Proof The bisimulation for this proof is unusually complex. Let f^n stand for the function that applies f to its argument n times. Since Lmap is a curried function, the function $(\texttt{Lmap}\ f)^n$ applies $\texttt{Lmap}\ f$ to a given list n times. The bisimulation has two index variables:

$$r \equiv \{((\texttt{Lmap}\ f)^n\ (h\ u),\ (\texttt{Lmap}\ f)^n\ (\texttt{Iterates}\ f\ u))\}_{u\in\texttt{LList}\,U,\ n\geq 0}$$

Recall that U stands for the universal set of the appropriate type.

Then note two facts, both easily justified by induction on n:

$$(\texttt{Lmap}\ f)^n\ (\texttt{CONS}\ b\ M) = \texttt{CONS}\,(f^n\ b)\,((\texttt{Lmap}\ f)^n\ M) \tag{25}$$

$$f^n\ (f\ x) = f^{\texttt{Suc}\ n}\ x \tag{26}$$

By extensionality it suffices to prove $h\ u = \texttt{Iterates}\ f\ u$ for all u. Again we apply the weak coinduction rule (12), with the bisimulation r shown above. The key step in verifying the bisimulation is to show

$$((\texttt{Lmap}\ f)^n\ (h\ u),\ (\texttt{Lmap}\ f)^n\ (\texttt{Iterates}\ f\ u)) \in \texttt{LListD\_Fun}(\texttt{diag}\,U)r$$

for arbitrary $n \geq 0$ and $u \in \texttt{LList}\ U$. By the recursion equations for h and Iterates, the pair expands to

$$\begin{array}{l}((\texttt{Lmap}\ f)^n\ (\texttt{CONS}\ u\,(\texttt{Lmap}\ f\ (h\ u))),\\ \ (\texttt{Lmap}\ f)^n\ (\texttt{CONS}\ u\,(\texttt{Iterates}\ f\ (f\ u))))\end{array}$$

and now two applications of equation (25) yield

$$\begin{array}{l}(\texttt{CONS}\,(f^n\ u)\,((\texttt{Lmap}\ f)^n\ (\texttt{Lmap}\ f\ (h\ u))),\\ \ \texttt{CONS}\,(f^n\ u)\,((\texttt{Lmap}\ f)^n\ (\texttt{Iterates}\ f\ (f\ u))))\end{array}$$

Applying Rule (15) to show membership in $\texttt{LListD\_Fun}(\texttt{diag}\,U)r$, we are left with the subgoal

$$((\texttt{Lmap}\ f)^n\ (\texttt{Lmap}\ f\ (h\ u)),\ (\texttt{Lmap}\ f)^n\ (\texttt{Iterates}\ f\ (f\ u))) \in r.$$

By equations (24) and (26) we obtain

$$(\texttt{Lmap}\ f^{\texttt{Suc}\ n}\ (h\ u),\ \texttt{Lmap}\ f^{\texttt{Suc}\ n}\ (\texttt{Iterates}\ f\ u)) \in r,$$

which is obviously true, by the definition of r.

Pitts [28] describes a similar coinduction proof in domain theory. The function Iterates does not lend itself to reasoning by structural induction, but is amenable to Scott's fixedpoint induction; I proved equation (24) in the Logic for Computable Functions (LCF) [22, page 286].

8.5 Reasoning about the append function

Many accounts of structural induction start with proofs about append. But append is not an easy example for coinduction. Remember that the corecursive definition does not give us the usual pair of equations, but rather the three equations (18)–(20). Proofs by the weak coinduction rule (12) typically require the same three-way case analysis. Consider proving that map distributes over append:

Example 8 If $M \in \texttt{LList}\ A$ and $N \in \texttt{LList}\ A$ then

$$\texttt{Lmap}\ f\,(\texttt{Lappend}\ M\ N) = \texttt{Lappend}\,(\texttt{Lmap}\ f\ M)\,(\texttt{Lmap}\ f\ N).$$

Proof Applying the weak coinduction rule with the bisimulation

$$\{(\texttt{Lmap}\ f\,(\texttt{Lappend}\ u\ v),\ \texttt{Lappend}\,(\texttt{Lmap}\ f\ u)\,(\texttt{Lmap}\ f\ v))\}_{u\in\texttt{LList}\ A, v\in\texttt{LList}\ A},$$

we must show

$$\begin{aligned}&(\texttt{Lmap}\ f\,(\texttt{Lappend}\ u\ v),\\ &\ \texttt{Lappend}\,(\texttt{Lmap}\ f\ u)\,(\texttt{Lmap}\ f\ v))\ \in\ \texttt{LListD\_Fun}(\texttt{diag}\,U)r.\end{aligned}$$

Considering the form of u and v, there are three cases:

- If $u = v = \texttt{NIL}$, the pair reduces by equations (16) and (18) to $(\texttt{NIL}, \texttt{NIL})$, and Rule (14) solves the goal.
- If $u = \texttt{NIL}$ and $v = \texttt{CONS}\ N_1\ N_2$, the pair reduces by equations (16), (17) and (19) to

  $$\begin{aligned}&(\texttt{CONS}\,(f\ N_1)\,(\texttt{Lmap}\ f\,(\texttt{Lappend}\,\texttt{NIL}\ N_2)),\\ &\ \texttt{CONS}\,(f\ N_1)\,(\texttt{Lappend}\,\texttt{NIL}\,(\texttt{Lmap}\ f\ N_2))).\end{aligned}$$

  Using equation (16) to replace the second NIL by Lmap f NIL restores the form of the bisimulation, so that Rule (15) can conclude this case.
- If $u = \texttt{CONS}\ M_1\ M_2$, the pair reduces by equations (17) and (20) to

  $$\begin{aligned}&(\texttt{CONS}\,(f\ M_1)\,(\texttt{Lmap}\ f\,(\texttt{Lappend}\ M_2\ v)),\\ &\ \texttt{CONS}\,(f\ M_1)\,(\texttt{Lappend}\,(\texttt{Lmap}\ f\ M_2)\,(\texttt{Lmap}\ f\ v))).\end{aligned}$$

  Now Rule (15) solves the goal, proving the distributive law.

Two easy theorems state that NIL is the identity element for Lappend. For $M \in \texttt{LList}\ A$ we have

$$\texttt{Lappend}\,\texttt{NIL}\ M = M \quad \text{and} \quad \texttt{Lappend}\ M\ \texttt{NIL} = M \tag{27}$$

Both proofs have only two cases because M is the only variable.

The strong coinduction rule (13) can prove the distributive law with only two cases. Recall from §6.5 that the latter rule implicitly includes the equality relation in the bisimulation; if we can reduce the pair to the form (a, a), then we are done. The simpler proof applies the strong coinduction rule (13), using the same bisimulation as before. Now we must show

$$\begin{array}{l}(\texttt{Lmap}\ f\ (\texttt{Lappend}\ u\ v),\\ \ \texttt{Lappend}\,(\texttt{Lmap}\ f\ u)\,(\texttt{Lmap}\ f\ v))\ \in\ \texttt{LListD\_Fun}(\texttt{diag}\,U)r\cup\texttt{diag}(\texttt{LList}\ U).\end{array}$$

There are two cases, considering the form of u. If $u = \texttt{CONS}\ M_1\ M_2$ then reason exactly as in the previous proof. If $u = \texttt{NIL}$, the goal reduces by (16) and (27) to

$$(\texttt{Lmap}\ f\ v, \texttt{Lmap}\ f\ v)\ \in\ \texttt{LListD\_Fun}(\texttt{diag}\,U)r\cup\texttt{diag}(\texttt{LList}\ U).$$

The pair belongs to $\texttt{diag}(\texttt{LList}\ U)$ because $\texttt{Lmap}\ f\ v \in \texttt{LList}\ U$.

Typing rules. Most of our coinduction examples prove equations. But coinduction can also prove typing facts such as $\texttt{Lappend}\ M\ N \in \texttt{LList}\ A$. Again, strong coinduction works better for `Lappend` than weak coinduction. The weak rule (7), requires case analysis on both arguments of `Lappend`; three cases must be considered. The strong rule (8) requires only case analysis on the first argument. The two proofs are closely analogous to those of the distributive law.

Associativity. A classic example of structural induction is the associativity of append:

$$\texttt{Lappend}\,(\texttt{Lappend}\ M_1\ M_2)\ M_3 = \texttt{Lappend}\ M_1\ (\texttt{Lappend}\ M_2\ M_3)$$

With weak coinduction, the proof is no longer trivial; it involves a bisimulation in three variables and the proof consists of four cases: `NIL-NIL-NIL`, `NIL-NIL-CONS`, `NIL-CONS`, and `CONS`. Strong coinduction with the bisimulation

$$\{(\texttt{Lappend}\,(\texttt{Lappend}\ u\ M_2)\ M_3,\ \texttt{Lappend}\ u\,(\texttt{Lappend}\ M_2\ M_3))\}_{u\in\texttt{LList}\ A}$$

accomplishes the proof easily. There are only two cases. If $u = \texttt{CONS}\ M\ M'$ then the pair reduces to

$$\begin{array}{l}(\texttt{CONS}\ M\ (\texttt{Lappend}\,(\texttt{Lappend}\ M'\ M_2)\ M_3),\\ \ \texttt{CONS}\ M\ (\texttt{Lappend}\ M'\ (\texttt{Lappend}\ M_2\ M_3)))\end{array}$$

and Rule (15) applies as usual. If $u = \texttt{NIL}$ then both components collapse to $\texttt{Lappend}\ M_2\ M_3$, and the pair belongs to the diagonal set.

8.6 A comparison with LCF

Scott's Logic for Computable Functions (LCF) is ideal for reasoning about lazy data structures.[5] Types denote domains and function symbols denote continuous functions. Strict and lazy recursive data types can be defined. Domains contain the bottom

[5] Scott's 1969 paper, which laid the foundations of domain theory, has finally been published [30]. Edinburgh LCF [15], a highly influential system, implemented Scott's logic. Still in print is my account of a successor LCF system [22]. Isabelle provides two versions of LCF: one built upon first-order logic, the other built upon Isabelle/HOL.

element $\bot$, which is the denotation of a divergent computation. Objects can be partial, with components that equal $\bot$. A function f can be partial, with $f\ x = \bot$ for some x. A lazy list can be partial, with its head, tail or some elements equal to $\bot$. The relation $x \sqsubseteq y$, meaning "x is less defined or equal to y," compares partial objects.

LCF's fixedpoint operator expresses recursive objects, including partial functions and recursive lists. The fixedpoint induction rule reasons about the unwinding of recursive objects. It subsumes the familiar structural induction rules and even generalizes them to reason about infinite objects, when the induction formula is chain-complete. Since all formulae of the form $x \sqsubseteq y$ and $x = y$ are chain-complete, LCF can prove equations about lazy lists.

Can our HOL treatment of lazy lists compare with LCF's? The Isabelle theory defines the new type $\alpha\,$`llist` to contain the elements of `LList(range(Leaf))`, replacing the clumsy set reasoning by automatic type checking. We can still define lists by corecursion and prove equations by coinduction.

The resulting theory of lazy lists is superficially similar to LCF's. But it lacks general recursion and cannot handle divergent computations. Leclerc and Paulin-Mohring's construction of the prime numbers in Coq [17] reflects these problems. Corecursion cannot express the filter functional, needed for the Sieve of Eratosthenes.

A direct comparison between fixedpoint induction and coinduction is difficult. The rules differ greatly in form and their area of overlap is small. Fixedpoint induction can reason about recursive programs at an abstract level, for instance to prove equivalence of partial functions. Coinduction can prove the equivalence of infinite processes.

I prefer LCF for problems that can be stated entirely in domain theory. But LCF is a restrictive framework: all types must denote domains; all functions must be continuous. HOL is a general logic, free of such restrictions, and yet capable of handling a substantial part of the theory of lazy lists.

9 Conclusions

This mechanized theory is a comprehensive treatment of recursive data structures in higher-order logic, generalizing Melham's theory [18]. Melham's approach is concrete: one particular tree structure represents all recursive types. My fixedpoint approach is more abstract, which facilitates extensions such as mutual recursion [26] and infinite branching trees. Users may also add new monotone operators to the type definition language. Types need not be free (such that each constructor function is injective); for example, we may define the set `Fin` A of all finite subsets of A as a least fixedpoint:

$$\texttt{Fin}\ A \equiv \texttt{lfp}\left(\lambda Z.\{\{\}\} \ \cup\ \left(\bigcup_{y\in Z}\bigcup_{x\in A}\{\{x\}\cup y\}\right)\right)$$

Most importantly, the theory justifies non-WF data structures.

Elsa Gunter [16] has independently developed a theory of trees, using ideas similar to those of §4. Her aim is to extend Melham's package with infinite branching, rather than coinduction.

What about set theory? The ordered pair (a, b) is traditionally defined to be $\{\{a\}, \{a, b\}\}$. Non-WF data structures presuppose non-WF sets, with infinite descents along the $\in$ relation. Such sets are normally forbidden by the Foundation Axiom, but the Anti-Foundation Axiom (AFA) asserts their existence. Aczel [3] has analysed and advocated this axiom; some authors, such as Milner and Tofte [21], have suggested formalizing coinductive arguments using it.

But non-WF lists and trees are easily expressed in set theory without new axioms. Simply use the ideas presented above for Isabelle/HOL. A new definition of ordered pair, based upon the old one, allows infinite descents. Define the variant ordered pair $(a; b)$ to be $\{(0, x)\}_{x\in a} \cup \{(1, y)\}_{y\in b}$. This is equivalent to the disjoint sum $a + b$ as usually defined, but note also its similarity to $M \cdot N$ (see §4.4). Elsewhere [27] I have developed this approach; it handles recursive data structures in full generality, but not the models of concurrency that motivated Aczel. The proof of the main theorem is inspired by that of Lemma 2. Isabelle/ZF mechanizes this theory.

The theory of coinduction requires further development. Its treatment of lazy lists is clumsy compared with LCF's. Stronger principles of coinduction and corecursion might help. Its connection with similar work in stronger type theories [12, 17] deserves investigation. Leclerc and Paulin-Mohring [17] remark that Coq can express finite and infinite data structures in a manner strongly reminiscent of the Knaster-Tarski Theorem. They proceed to consider a representation of streams that specifies the possible values of the stream member at position i, where i is a natural number. Generalizing this approach to other infinite data structures requires generalizing the notion of position, perhaps as in §4.2.

Jacob Frost [11] has performed Milner and Tofte's coinduction example [21] using Isabelle/HOL and Isabelle/ZF. The most difficult task is not proving the theorem but formalizing the paper's non-WF definitions. As of this writing, Isabelle/HOL provides no automatic means of constructing the necessary definitions and proofs.

Acknowledgements. Jacob Frost, Sara Kalvala and the referees commented on the paper. Martin Coen helped to set up the environment for coinduction proofs. Andrew Pitts gave much advice, for example on proving that equality is a `gfp`. Tobias Nipkow and his colleagues have made substantial contributions to Isabelle. The research was funded by the SERC (grants GR/G53279, GR/H40570) and by the ESPRIT Basic Research Actions 3245 'Logical Frameworks' and 6453 'Types'.

References

[1] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Resesarch Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1977.

[2] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[3] P. Aczel. *Non-Well-Founded Sets.* CSLI, 1988.

[4] P. B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem proving system for classical type theory. *J. Auto. Reas.*, 16(1), 1996. In press.

[5] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[6] A. Church. A formulation of the simple theory of types. *J. Symb. Logic*, 5:56–68, 1940.

[7] L. J. M. Claesen and M. J. C. Gordon, editors. *Higher Order Logic Theorem Proving and Its Applications.* North-Holland, 1993.

[8] T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic*, LNCS 417, pages 50–66, Tallinn, Published 1990. Estonian Academy of Sciences, Springer.

[9] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Univ. Press, 1990.

[10] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *J. Auto. Reas.*, 11(2):213–248, 1993.

[11] J. Frost. A case study of co-induction in Isabelle. Technical Report 359, Comp. Lab., Univ. Cambridge, Feb. 1995.

[12] H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Types for Proofs and Programs*, pages 193–217, Båstad, June 1992. informal proceedings.

[13] J.-Y. Girard. *Proofs and Types*. Cambridge Univ. Press, 1989. Translated by Yves LaFont and Paul Taylor.

[14] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge Univ. Press, 1993.

[15] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer, 1979. LNCS 78.

[16] E. L. Gunter. A broader class of trees for recursive type definitions for hol. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: HUG '93*, LNCS 780, pages 141–154. Springer, Published 1994.

[17] F. Leclerc and C. Paulin-Mohring. Programming with streams in Coq. a case study: the sieve of Eratosthenes. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES '93*, LNCS 806, pages 191–212. Springer, 1993.

[18] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer, 1989.

[19] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Second Annual Symposium on Logic in Computer Science*, pages 30–36. IEEE Comp. Soc. Press, 1987.

[20] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[21] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Comput. Sci.*, 87:209–220, 1991.

[22] L. C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge Univ. Press, 1987.

[23] L. C. Paulson. The foundation of a generic theorem prover. *J. Auto. Reas.*, 5(3):363–397, 1989.

[24] L. C. Paulson. Set theory for verification: I. From foundations to functions. *J. Auto. Reas.*, 11(3):353–389, 1993.

[25] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[26] L. C. Paulson. Set theory for verification: II. Induction and recursion. *J. Auto. Reas.*, 15(2):167–215, 1995.

[27] L. C. Paulson. A concrete final coalgebra theorem for ZF set theory. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs: International Workshop TYPES '94*, LNCS 996, pages 120–139. Springer, published 1995.

[28] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Comput. Sci.*, 124:195–219, 1994.

[29] J. J. M. M. Rutten and D. Turi. On the foundations of final semantics: Non-standard sets, metric spaces, partial orders. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, pages 477–530. Springer, 1993. LNCS 666.

[30] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Comput. Sci.*, 121:411–440, 1993. Annotated version of the 1969 manuscript.

[31] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[32] M. Tofte. Type inference for polymorphic references. *Info. and Comput.*, 89:1–34, 1990.

[33] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge Univ. Press, 1962. Paperback edition to *56, abridged from the 2nd edition (1927).

Modal Fracture of Higher Groups

David Jaz Myers

June 30, 2021

Abstract

In this paper, we examine the modal aspects of higher groups in Shulman's Cohesive Homotopy Type Theory. We show that every higher group sits within a modal fracture hexagon which renders it into its discrete, infinitesimal, and contractible components. This gives an unstable and synthetic construction of Schreiber's differential cohomology hexagon. As an example of this modal fracture hexagon, we recover the character diagram characterizing ordinary differential cohomology by its relation to its underlying integral cohomology and differential form data, although there is a subtle obstruction to generalizing the usual hexagon to higher types. Assuming the existence of a long exact sequence of differential form classifiers, we construct the classifiers for circle k-gerbes with connection and describe their modal fracture hexagon.

Contents

Acknowledgements. I would like to thank Emily Riehl for her careful reading and helpful comments during the drafting of this paper. The author is grateful for financial support from the ARO under MURI Grant W911NF-20-1-0082.

1 Introduction

There are many situations where cohomology is useful but we need more than just the information of cohomology classes and their relations in cohomology — we need the information of specific cocycles which give rise to those classes and cochains which witness these relations. A striking example of this situation is *ordinary differential cohomology*. To give a home for calculations done in [4], Cheeger and Simons [3] gave a series of lectures in 1973 defining and studying *differential characters*, which equip classes in ordinary integral cohomology with explicit differential form representatives. Slightly earlier, Deligne [6] had put forward a cohomology theory in the complex analytic setting which would go on to be called Deligne cohomology. It was later realized that when put in the differential geometric setting, Deligne cohomology gave a presentation of the theory of differential characters. This combined theory has become known as ordinary differential cohomology.

The ordinary differential cohomology $D_k(X)$ of a manifold X is characterized by its relationship to the ordinary cohomology of X and the differential forms on X by a diagram known as the *differential cohomology hexagon* or the *character diagram* [23]:

$$\begin{tikzcd}
& \Lambda^k(X)/\mathrm{im}(d) \arrow[rr, "d"] \arrow[dr] & & \Lambda^{k+1}_{\mathrm{cl}}(X) \arrow[dr] & \\
H^k(X;\mathbb{R}) \arrow[ur] \arrow[dr] & & D_k(X) \arrow[ur] \arrow[dr] & & H^{k+1}(X;\mathbb{R}) \\
& H^k(X;U(1)) \arrow[rr, "\beta"'] \arrow[ur] & & H^{k+1}(X;\mathbb{Z}) \arrow[ur] &
\end{tikzcd} \tag{1.0.1}$$

In this diagram, the top and bottom sequences are long exact, and the diagonal sequences are exact in the middle. The bottom sequence is the Bockstein sequence associated to the universal cover short exact sequence

$$0 \to \mathbb{Z} \to \mathbb{R} \to U(1) \to 0$$

while the top sequence is given by de Rham's theorem representing real cohomology classes by differential forms.

This sort of diagram is characteristic of differential cohomology theories in general. Bunke, Nikolaus, and Vokel [2] interpret differential cohomology theories as sheaves on the site of smooth manifolds and construct differential cohomology hexagons very generally in this setting:

$$\begin{tikzcd}
& \mathcal{A}(\hat{E})^k(X) \arrow[rr] \arrow[dr] & & \mathcal{Z}(\hat{E})^k(X) \arrow[dr] & \\
H^{k-1}(X;Z(\hat{E})) \arrow[ur] \arrow[dr] & & \hat{E}^k(X) \arrow[ur] \arrow[dr] & & H^k(X;Z(\hat{E})) \\
& H^k(X;S(\hat{E})) \arrow[rr] \arrow[ur] & & H^k(X;U(E)) \arrow[ur] &
\end{tikzcd}$$

Here, $\hat{E}$ is the differential cohomology theory, $U(\hat{E})$ is its underlying topological cohomology, $\mathcal{Z}(\hat{E})$ are the differential cycles, S is the secondary cohomology theory given by flat classes, and $\mathcal{A}$ classifies differential deformations (this summary discussion is lifted from [2]). Here, as with ordinary differential cohomology, the top and bottom sequences are exact, and the diagonal sequences are exact in the middle.

The arguments of Bunke, Nikolaus, and Vokel are abstract and *modal* in character. This is emphasized by Schreiber in his book [20], where he constructs similar diagrams in the setting of an adjoint triple

$$\begin{tikzcd}
\mathcal{E} \arrow[d, bend right=60, "\Pi_\infty"'] \arrow[d, bend left=60, "\Gamma"] \\
\mathcal{H} \arrow[u, "\dashv\ \Delta\ \dashv" description]
\end{tikzcd} \tag{1.0.2}$$

in which the middle functor is fully faithful and the leftmost adjoint Π_∞ preserves products. In the case that Γ is the global sections functor of an ∞-topos $\mathcal{E}$ landing in the ∞-topos of homotopy types $\mathcal{H}$, this structure makes $\mathcal{E}$ into a *strongly* ∞*-connected* ∞-topos. In the case that $\mathcal{E}$ is the ∞-topos of sheaves of homotopy types on manifolds, the leftmost adjoint Π_∞ is given by localizing at the sheaf of real-valued functions; for a representable, this recovers the homotopy type or fundamental ∞-groupoid of the manifold. Schreiber shows in Proposition 4.1.17 of [20] that any such adjoint triple gives rise to differential cohomology hexagons, specializing to those of Bunke, Nikolaus, and Vokel in the case that $\mathcal{E}$ is the ∞-topos of sheaves of homotopy types on smooth manifolds.

This abstract re-reading of the differential cohomology hexagons shows that there is nothing specifically "differential" about them, and that they arise in situations where there is no differential calculus to be found. Schreiber emphasizes this point in an nLab article [21] where he refigures these hexagons as *modal fracture squares*. To recover more traditional fracture theorems, Schreiber considers the case where $\mathcal{E} = A$-Mod is the ∞-category of modules over an $\mathbb{E}^2$-ring A; $\Gamma = \Gamma_I$ is the reflection in to I-nilpotent modules (with $I \subseteq \pi_0 A$ a finitely generated ideal) constructed by Lurie in [14, Notation 4.1.13], and $\Pi_\infty M = M_I^\wedge$ is the I-completion constructed in [14, Notation 4.2.3]. The subcategories of I-nilpotent and I-complete modules are distinct but equivalent, allowing us to see them as a single ∞-category $\mathcal{H}$.

In this paper, we will construct the *modal fracture hexagon* associated to a higher group — a homotopy type which may be delooped — synthetically, working in an appropriately modal homotopy type theory. We will work in Shulman's *flat homotopy type theory* [22], a variant of homotopy type theory that adds a *comodality* $\flat$ which may be thought of as the comonad $\Delta\Gamma$ from Diagram 1.0.2.We will equip this type theory equipped with a modality $\int$ (which may be thought of as the monad $\Delta\Pi_\infty$) which satisfies a *unity of opposites* axiom (Axiom 1) implying that $\int$ is left adjoint to $\flat$ in the sense that

$$\flat(\textstyle\int X \to Y) = \flat(X \to \flat Y).$$

We can think of this axiom as the internalization of the adjunction $\Delta\Pi_\infty \dashv \Delta\Gamma$ induced by Diagram 1.0.2.

This type theory should have models in all *strongly ∞-connected geometric morphisms* between ∞-toposes. A geometric morphism $f : \mathcal{E} \to \mathcal{S}$ is strongly ∞-connected when its inverse image $f^* : \mathcal{S} \to \mathcal{E}$ is fully faithful and has a left adjoint $f_! : \mathcal{E} \to \mathcal{S}$ which preserves finite products. We then have $\int = f^* f_!$ and $\flat = f^* f_*$. An ∞-topos $\mathcal{E}$ is strongly ∞-connected when its terminal geometric morphism $\Gamma : \mathcal{E} \to \infty\mathbf{Grpd}$ is strongly ∞-connected.

Our main theorem (an unstable and synthetic version of Proposition 4.1.17 in [20]) is as follows:

Theorem 2.4.2. *For a crisp ∞-group G, there is a* modal fracture hexagon*:*

$$\begin{tikzcd}
 & \overset{\infty}{G} \arrow[rr] \arrow[dr, "\pi"] & & \mathfrak{g} \arrow[dr, "(-)^{\int}"] & \\
\flat\overset{\infty}{G} \arrow[ur, "(-)_\flat"] \arrow[dr, "\flat\pi"'] & & G \arrow[ur, "\theta"] \arrow[dr, "(-)^{\int}"] & & \int\mathfrak{g} \\
 & \flat G \arrow[rr] \arrow[ur, "(-)_\flat"] & & \int G \arrow[ur, "\int\theta"'] &
\end{tikzcd}$$

where

- $\theta : G \to \mathfrak{g}$ *is the* infinitesimal remainder *of G, the quotient $G \mathbin{/\!\!/} \flat G$, and*
- $\pi : \overset{\infty}{G} \to G$ *is the* universal (contractible) ∞-cover *of G.*

Moreover,

1. *The middle diagonal sequences are fiber sequences.*
2. *The top and bottom sequences are fiber sequences.*
3. *Both squares are pullbacks.*

Furthermore, the homotopy type of $\mathfrak{g}$ is a delooping of $\flat\overset{\infty}{G}$:

$$\textstyle\int\mathfrak{g} = \flat\boldsymbol{B}\overset{\infty}{G}.$$

Therefore, if G is k-commutative for $k \geq 1$ (that is, admits futher deloopings $\boldsymbol{B}^{k+1}G$), then we may continue the modal fracture hexagon on to $\boldsymbol{B}^k G$.

We will define the notions of *universal* ∞*-cover* and of *infinitesimal remainder* in Sections 2.2 and 2.3 respectively. Our proof of this theorem will make extensive use of the theory of modalities in homotopy type theory developed by Rijke, Shulman, and Spitters [19], as well as the theory of modal étale maps developed in [5] and the theory of modal fibrations developed in [17].

Having proven this theorem, we will turn our attention to providing interesting examples of it. To that end, in Section 3 we will construct ordinary differential cohomology (in the guise of the classifying bundles $\mathbf{B}^k_\nabla U(1)$ of connections on k-gerbes with band $U(1)$, see Definition 3.2.1) in smooth real cohesive homotopy type theory. For this, we assume the existence of a long exact sequence

$$0 \to \flat\mathbb{R} \to \mathbb{R} \xrightarrow{d} \Lambda^1 \xrightarrow{d} \Lambda^2 \to \cdots$$

where the Λ^k classify *differential k-forms*. It should be possible to construct the Λ^k from the axioms of synthetic differential geometry with tiny infinitesimals, but we do not do so here for reasons of space and self-containment. See Remark 3.1.2 for a full discussion.

Our construction of ordinary differential cohomology is clean, conceptual, and modal. We do not, however, recover exactly the character diagram 1.0.1, because de Rham's theorem does not hold for all types (see Proposition 3.3.6). In Section 3.3, we do recover a very similar diagram and find that the obstruction to these two diagrams being the same lies in a shifted version of ordinary differential cohomology.

Because the arguments given in Section 3 are abstract and modal in character, they are applicable in other settings. In Section 3.4, we express our construction of ordinary differential cohomology in the abstract setting of a *contractible and infinitesimal resolution* of a crisp abelian group. In Section 3.5, we briefly describe how to construct combinatorial analogues of ordinary differential cohomology in symmetric simplicial homotopy types, making use of an observation of Lawvere that cocycle classifiers may be constructed using the tinyness of the simplices.

2 The Modal Fracture Hexagon

In this section, we will construct the modal fracture hexagon of a higher group.

A higher group G is a type equipped with a 0-connected delooping $\mathbf{B}G$. An ordinary group G may be considered as a higher group by taking $\mathbf{B}G$ to be the type of G-torsors and equating G with the group of automorhpisms of G considered as a G-torsor.

The theory of higher groups is expressed in terms of their deloopings: for example a homomorphism $G \to H$ is equivalently a pointed map $\mathbf{B}G \cdot\!\!\to \mathbf{B}H$. See [1] for a development of the elementary theory of higher groups in homotopy type theory.

The modal fracture hexagon associated to a (crisp) higher group G will factor G into its *universal* ∞*-cover* $\overset{\infty}{G}$ and its *infinitesimal remainder* $\mathfrak{g}$. We will therefore introduce $\overset{\infty}{G}$ and $\mathfrak{g}$ and prove some lemmas about them which will set the stage for the modal fracture hexagon.

Notation 1. We will use the Agda-inspired notation for dependent pair types (also known as dependent sum types) and dependent function types (also known as dependent product types):

$$(a : A) \times B(a) \equiv \sum_{a:A} B(a)$$
$$(a : A) \to B(a) \equiv \prod_{a:A} B(a).$$

If X is a pointed type, we refer to its base point as $\mathsf{pt}_X : X$. If X and Y are pointed types, then we define $X \cdot\!\!\to Y$ to be the type of pointed functions between them:

$$(X \cdot\!\!\to Y) \equiv (f : X \to Y) \times (f(\mathsf{pt}_X) = \mathsf{pt}_Y).$$

2.1 Preliminaries

In this section, we will review Shulman's flat type theory [22] and the necessary lemmas.

In constructive mathematics, the proposition that all functions $\mathbb{R} \to \mathbb{R}$ are continuous is undecided — there are models of constructive set theory (and homotopy type theory) in which every function $\mathbb{R} \to \mathbb{R}$ is continuous (and, of course, familiar models where there are discontinuous functions $\mathbb{R} \to \mathbb{R}$). Since, in type theory, such a function $f : \mathbb{R} \to \mathbb{R}$ is defined by giving the image $f(x)$ of a free variable $x : \mathbb{R}$, we see that in a pure constructive setting, the dependence of terms on their free variables confers a liminal sort of continuity. This is a very powerful observation which, in its various guises, lets us avoid the menial checking of continuity, smoothness, regularity, and so on for various sorts of functions in various models of homotopy type theory. It extends far beyond real valued functions; for example, the assignment of a vector space V_p to a point p in a manifold M, constructively, gives a vector bundle $(p : M) \times V_p \to M$ over M with all its requirements of continuity or smoothness (depending on the model). Since all sorts of continuity (continuity, smoothness, regularity, analyticity) can be captured in various models, Lawvere named the general notion "cohesion" in his paper [10], whose generalization to ∞-categories in [20] inspired the type theory of [22].

However, not every dependency is cohesive (continuous, smooth, etc.). To enable discontinuous dependencies, then, we must mark our free variables as varying cohesively or not. For this reason, Shulman introduces *crisp* variables, which are free variables in which terms depend discontinuously:

$$a :: A.$$

Any variable appearing in the type of a crisp variable must also be crisp, and a crisp variable may only be substituted by expressions that *only* involve crisp variables. When all the variables in an expression are crisp, we say that that expression is crisp; so, we may only substitute crisp expressions in for crisp variables. Constants — like $0 : \mathbb{N}$ or $\mathbb{N}$: **Type** — appearing in an empty context are therefore always crisp.[1] This means that one cannot give a closed form example of a term which is *not* crisp; all terms with no free variables are crisp. For emphasis, we will say that a term which is not crisp is *cohesive.* The rules for crisp type theory can be found in Section 2 of [22].

Given this notion of discontinuous dependence of terms on their free variables, we can now define an operation on types which removes the cohesion amongst their points. Given a *crisp* type X, we have a type $\flat X$ whose points are, in a sense, the *crisp* points of X. Since it is free variables that may be crisp, we express this idea by allowing ourselves to assume that a (cohesive) variable $x : \flat X$ is of the form $u^\flat$ for a *crisp* $u :: X$. More precisely, whenever we have type family $C : \flat X \to \textbf{Type}$, an $x : \flat X$, and an element $f(u) : C(u^\flat)$ depending on a *crisp* $u :: X$, we get an element

$$(\text{let } u^\flat := x \text{ in } f(u)) : C(x)$$

and if $x \equiv v^\flat$, then $(\text{let } u^\flat := x \text{ in } f(u)) \equiv f(v)$. We refer to this method of proof as "$\flat$-induction". The full rules for $\flat$ can be found in Section 4 of [22].

We have an inclusion $(-)_\flat : \flat X \to X$ given by $x_\flat :\equiv \text{let } u^\flat := x \text{ in } u$. Since we are thinking of a dependence on a crisp variable as a *discontinuous* dependence, if this map $(-)_\flat : \flat X \to X$ is an equivalence then every *discontinuous* dependence on $x :: X$ underlies a *continuous* dependence on x. This leads us to the following defintion:

Definition 2.1.1. A crisp type X :: **Type** is *crisply discrete* if the counit $(-)_\flat : \flat X \to X$ is an equivalence.[2]

Note that this definition is only sensible for crisp types, since we may only form $\flat X$ for crisp X :: **Type**. We would also like a notion of discreteness which applies to any type, and a reflection $X \to \int X$ of a type into a discrete type. For that reason, we will also presume that there is a modality $\int$ called the *shape* (and which we think of as sending a type X to its homotopy type or shape $\int X$). We refer to the $\int$-modal types as discrete. To make sure that these two notions of discreteness coincide, we assume the following axiom:

Axiom 1 (Unity of Opposites)**.** For any *crisp* type X, the counit $(-)^\flat : \flat X \to X$ is an equivalence if and only if the unit $(-)^{\int} : X \to \int X$ is an equivalence.

This axiom implies that $\int$ is left adjoint to $\flat$, at least for crisp maps. In [22], Shulman assumes an axiom C0 which lets him define the $\int$ modality as a localization and prove our Unity of Opposites axiom. The two

[1] Note that as these are terms and not free variables, we don't need to use the special syntax $a :: A$. The double colon introduces a crisp *free* variable.

[2] See Remark 6.13 of [22] for a discussion on some of the subtleties in the notion of crisp discreteness.

axioms have roughly the same strength, though C0 is slightly stronger since it assumes that $\int$ is an *accessible* modality.

Theorem 2.1.2. *Let X and Y be crisp types. Then*

$$\flat(X \to \flat Y) = \flat(\textstyle\int X \to Y).$$

Proof. This is Theorem 9.15 of [22]. Note that Axiom C0 is only used via our Unity of Opposites axiom. □

Remark 2.1.3. Theorem 2.1.2 justifies the use of the symbol "$\flat$" in flat type theory. If we think of $\int X$ as the homotopy type of X, then the adjointness of $\int$ with $\flat$ tells us that $\flat\mathbf{B}G$ modulates principal G-bundles with a *homotopy invariant parallel transport* — that is, bundles with *flat* connection. This terminology is due to Schreiber in [20].

We may also define the *truncated shape modalities* $\int_n$ to have as modal types the types which are both n-truncated and $\int$-modal. It is not known whether $\int_n X = \|\int X\|_n$ for general X, but it is true for *crisp* X (see Proposition 4.5 of [17]).

We will now prepare ourselves by proving a few preservation properties of the $\flat$ comodality and the $\int$ modality. A first time reader may content themselves with the the statements of the lemmas, as the proofs are mere technicalities.

Lemma 2.1.4. The comodality $\flat$ preserves fiber sequences. Let $f :: X \to Y$ be a crisp map and $y :: Y$ a crisp point. Then we have and equivalence $\flat\,\mathsf{fib}_f(y) = \mathsf{fib}_{\flat f}(y^\flat)$ such that

$$\begin{tikzcd}
\flat\,\mathsf{fib}_f(y) \arrow[dd, equal] \arrow[dr, "(-)_\flat"] & \\
& \mathsf{fib}_f(y) \\
\mathsf{fib}_{\flat f}(y^\flat) \arrow[ur, "\delta"'] &
\end{tikzcd}$$

commutes. In particular, $\flat\,\mathsf{fib}_f(y) \to \flat X \to \flat Y$ is a fiber sequence and that the naturality squares give a map of fiber sequences:

$$\begin{tikzcd}
\flat\,\mathsf{fib}_f(y) \arrow[r, "\delta"] \arrow[d] & \mathsf{fib}_f(y) \arrow[d] \\
\flat X \arrow[r] \arrow[d] & X \arrow[d] \\
\flat Y \arrow[r] & Y
\end{tikzcd}$$

Proof. We begin by constructing the equivalence:

$$\begin{aligned}
\flat\,\mathsf{fib}_f(y) &\equiv \flat\left((x : X) \times (f(x) = y)\right) \\
&= (u : \flat X) \times (\text{let } x^\flat := u \text{ in } \flat(f(x) = y)) && \text{[22, Lemma. 6.8]} \\
&= (u : \flat X) \times (\text{let } x^\flat := u \text{ in } f(x)^\flat = y^\flat) && \text{[22, Theorem. 6.1]} \\
&\equiv (u : \flat X) \times (\text{let } x^\flat := u \text{ in } \flat f(x^\flat) = y^\flat) \\
&= (u : \flat X) \times (\flat f(u) = y^\flat) && \text{[22, Lemma. 4.4]} \\
&\equiv \mathsf{fib}_{\flat f}(y^\flat).
\end{aligned}$$

We will need to understand what this equivalence does on elements $(x,p)^\flat$ for $(x,p) :: \mathsf{fib}_f(y)$. The first equivalence in the composite sends $(x,p)^\flat$ to $(x^\flat, p^\flat)$, and no other equivalence affects the first component, so the first component of the result will be $x^\flat$. The second equivalence will send $p^\flat$ to $\mathsf{ap}_\flat\,(-)^\flat\,p$, where $\mathsf{ap}_\flat$ is the crisp application function. The next equivalence is given by reflexivity, since $\flat f(x^\flat) \equiv f(x)^\flat$. In total, then, this equivalence acts as

$$(x,p)^\flat \mapsto (x^\flat, \mathsf{ap}_\flat(-)^\flat p).$$

Now, to show the triangle commutes, it will suffice to show that it commutes for $(x, p)^\flat$ where $(x, p) :: \mathsf{fib}_f(y)$. This is to say, we need to show that sending $(x, p)^\flat$ through the above equivalence and then into $\mathsf{fib}_f(y)$ yields (x, p). The map δ from $\mathsf{fib}_{\flat f}(y^\flat)$ to $\mathsf{fib}_f(y)$ sends (u, q) to $(u_\flat, \Box_\flat(u) \bullet \mathsf{ap}\,(-)_\flat\, q)$, where $\Box_\flat(u) : f(u_\flat) = \flat f(u)_\flat$ is the naturality square. So, the round trip $\flat\,\mathsf{fib}_f(y) \to \mathsf{fib}_{\flat f}(y^\flat) \to \mathsf{fib}_f(y)$ acts as

$$(x, p)^\flat \mapsto (x^\flat,\ \mathsf{ap}_\flat\,(-)^\flat\, p) \mapsto ({x^\flat}_\flat,\ \Box_\flat(x^\flat) \bullet \mathsf{ap}\,(-)_\flat\,(\mathsf{ap}_\flat\,(-)^\flat\, p)).$$

Now, ${x^\flat}_\flat \equiv x$, so it remains to show that $\Box_\flat(x^\flat) \bullet \mathsf{ap}\,(-)_\flat\,(\mathsf{ap}_\flat\,(-)^\flat\, p) = p$. However, the naturality square is defined by $\Box_\flat(x^\flat) \equiv \mathsf{refl}_{f(x)} : f({x^\flat}_\flat) = \flat f(x^\flat)_\flat$, so it only remains to show that the two applications cancel. This can easily be shown by a crisp path induction. □

Lemma 2.1.5. Let $f :: X \to Y$ be a crisp map between crisp types. The following are equivalent:

1. For every crisp $y :: Y$, $\mathsf{fib}_f(y)$ is discrete.
2. The naturality square

$$\begin{tikzcd}
\flat X \arrow[r, "(-)_\flat"] \arrow[d, "\flat f"'] & X \arrow[d, "\pi"] \\
\flat Y \arrow[r, "(-)_\flat"'] & Y
\end{tikzcd}$$

   is a pullback.

Proof. We note that the naturality square being a pullback is equivalent to the induced map

$$\mathsf{fib}_{\flat f}(u) \to \mathsf{fib}_f(u_\flat)$$

begin an equivalence for all $u : \flat Y$. By the universal property of $\flat Y$, we may assume that u is of the form $y^\flat$ for a crisp $y :: Y$. By Lemma 2.1.4, we have that

$$\begin{tikzcd}
\flat\,\mathsf{fib}_f(y) \arrow[dd, equal] \arrow[dr, "(-)_\flat"] & \\
& \mathsf{fib}_f(y) \\
\mathsf{fib}_{\flat f}(y^\flat) \arrow[ur] &
\end{tikzcd}$$

commutes. Therefore, the naturality square is a pullback if and only if for all crisp $y :: Y$, we have that $(-)_\flat : \flat\,\mathsf{fib}_f(y) \to \mathsf{fib}_f(y)$ is an equivalence; but this is precisely what it means for $\mathsf{fib}_f(y)$ to be discrete. □

Lemma 2.1.6. Let X be a crisp type, and let a, $b :: X$ be crisp elements. Then there is an equivalence $e : (a^\flat = b^\flat) \simeq \flat(a = b)$ together with a commutation of the following triangle:

$$\begin{tikzcd}
(a^\flat = b^\flat) \arrow[dd, equal, "e"'] \arrow[dr, "\mathsf{ap}\,(-)_\flat"] & \\
& (a = b) \\
\flat(a = b) \arrow[ur, "(-)_\flat"'] &
\end{tikzcd}$$

Proof. For the construction of the equivalence e we refer to [22, Theorem. 6.1]. For the commutativity, we use function extensionality to work from $u : \flat(a = b)$ seeking $e^{-1}(u) = u_\flat$ and proceed by $\flat$-induction and then identity induction in which case both sides reduce to refl. □

Lemma 2.1.7. Let G be a crisp higher group; that is, suppose that $\mathbf{B}G$ is a crisp, 0-connected type and its base point $\mathsf{pt} :: \mathbf{B}G$ is also crisp. Then $\flat G$ is also a higher group and we may take

$$\mathbf{B}\flat G \equiv \flat\mathbf{B}G$$

pointed at $\mathsf{pt}^\flat$. Furthermore, the counit $(-)_\flat : \flat G \to G$ is a homomorphism delooped by the counit $(-)_\flat : \flat\mathbf{B}G \to \mathbf{B}G$.

Proof. We need to show that $\flat\mathbf{B}G$ deloops $\flat G$ via an equivalence $e : \Omega\flat\mathbf{B}G = \flat G$, that it is 0-connected, and that looping the counit $(-)_\flat : \flat\mathbf{B}G \to \mathbf{B}G$ corresponds to the counit $(-)_\flat : \flat G \to G$ along the equivalence e.

For the equivalence $e : \Omega\flat\mathbf{B}G = \flat G$, we may take the equivalence $(\mathsf{pt}^\flat = \mathsf{pt}^\flat) = \flat(\mathsf{pt} = \mathsf{pt})$ of Lemma 2.1.6. The commutation of the triangle

$$\begin{tikzcd}
(\mathsf{pt}^\flat = \mathsf{pt}^\flat) \arrow[dd, equal, "e"'] \arrow[dr, "\mathsf{ap}\,(-)_\flat"] & \\
& (\mathsf{pt} = \mathsf{pt}) \\
\flat(\mathsf{pt} = \mathsf{pt}) \arrow[ur, "(-)_\flat"'] &
\end{tikzcd}$$

shows that $(-)_\flat : \flat\mathbf{B}G \to \mathbf{B}G$ deloops $(-)_\flat : \flat G \to G$.

To show that $\flat\mathbf{B}G$ is connected, we rely on [22, Corollary. 6.7] which says that $\|\flat\mathbf{B}G\|_0 = \flat\,\|\mathbf{B}G\|_0$, which is $*$ by the hypothesis that $\mathbf{B}G$ is 0-connected. □

We end with a useful lemma: $\flat$ preserves long exact sequences of groups.

Lemma 2.1.8. The comodality $\flat$ preserves crisp short and long exact sequences of groups.

Proof. A sequence

$$0 \to K \to G \to H \to 0$$

of groups is short exact if and only if its delooping

$$\mathbf{B}K \;\cdot\!\!\to \mathbf{B}G \;\cdot\!\!\to \mathbf{B}H$$

is a fiber sequence. But $\flat$ preserves crisp fiber sequences by Lemma 2.1.4 and by **??** we have that the fiber sequence

$$\flat\mathbf{B}K \;\cdot\!\!\to \flat\mathbf{B}G \;\cdot\!\!\to \flat\mathbf{B}H$$

deloops the sequence $\flat K \to \flat G \to \flat H$, so this sequence is also short exact.

Now, a complex of groups

$$\cdots \to A_{n-1} \xrightarrow{d} A_n \xrightarrow{d} A_{n+1} \to \cdots$$

satisfying $d \circ d = 0$ is long exact if and only if the sequences

$$0 \to K_n \to A_n \to K_{n+1} \to 0$$

are short exact, where $K_n :\equiv \mathsf{ker}(A_n \to A_{n+1})$. Now, we have a complex

$$\cdots \to \flat A_{n-1} \xrightarrow{\flat d} \flat A_n \xrightarrow{\flat d} A_{n+1} \to \cdots$$

by the functoriality of $\flat$. Since $\flat$ preserves short exact sequences, the sequences

$$0 \to \flat K_n \to \flat A_n \to \flat A_{n+1} \to 0$$

are short exact. Now, since $\flat$ preserves fibers we have that

$$\flat K_n = \mathsf{ker}(\flat A_n \to \flat A_{n+1}),$$

so that the $\flat$-ed complex is long exact. □

2.2 The Universal ∞-Cover of a Higher Group

An ∞-cover of a type X is a generalization of the notion of cover from a theory concerning 1-types (the fundamental groupoid of X, with the universal cover being simply connected) to arbitrary types (the homotopy type of X, with the universal ∞-cover being contractible).

Recall that, classically, a covering map $\pi : \tilde{X} \to X$ satisfies a *unique path lifting property*; that is, every square of the following form admits a unique filler:

$$\begin{tikzcd}
* \arrow[r, "\tilde{x}"] \arrow[d, "0"'] & \tilde{X} \arrow[d, "\pi"] \\
\mathbb{R} \arrow[r, "\gamma"'] \arrow[ur, dashed, "\exists!"] & X
\end{tikzcd}$$

This property can be extended to a unique lifting property against any map which induces an equivalence fundamental groupoids. That is, whenever $f : A \to B$ induces an equivalence $\int_1 f : \int_1 A \to \int_1 B$, every square of the following form admits a unique filler:

$$\begin{tikzcd}
A \arrow[r, "\tilde{\gamma}"] \arrow[d, "f"'] & \tilde{X} \arrow[d, "\pi"] \\
B \arrow[r, "\gamma"'] \arrow[ur, dashed, "\exists!"] & X
\end{tikzcd}$$

For any modality !, there is an orthogonal factorization system where !-equivalences (those maps f such that $!f$ is an equivalence) lift uniquely against !-*étale maps* ([18, 5]).

Definition 2.2.1. A map $f : A \to B$ is !-étale for a modality ! if the naturality square

$$\begin{tikzcd}
A \arrow[r, "(-)^!"] \arrow[d, "f"'] & !A \arrow[d, "!f"] \\
B \arrow[r, "(-)^!"'] & !B
\end{tikzcd}$$

is a pullback.

We may single out the covering maps as the $\int_1$-étale maps whose fibers are sets. For more on this point of view, see the last section of [17]. Here, however, we will be more concerned with $\int$-étale maps, which we will call ∞-covers. This notion was called a "modal covering" in [24], and was referred to as an ∞-cover in the setting of ∞-categories by Schreiber in [20].

Definition 2.2.2. A map $\pi : E \to B$ is an ∞-*cover* if the naturality square

$$\begin{tikzcd}
E \arrow[r, "(-)^{\int}"] \arrow[d, "\pi"'] & \int E \arrow[d, "\int \pi"] \\
B \arrow[r, "(-)^{\int}"] & \int B
\end{tikzcd}$$

is a pullback. That is, an ∞-cover is precisely a $\int$-étale map.

A map $\pi : E \to B$ is an n-cover if it is $\int_{n+1}$-étale and its fibers are n-types. We call a 1-cover just a cover, or a covering map.

Theorem 6.1 of [17] gives a useful way for proving that a map is an ∞-cover.

Proposition 2.2.3 (Theorem 6.1 of [17])**.** Let $\pi : E \to B$ and suppose that there is a crisp, discrete type F so that for all $b : B$, $\|\mathsf{fib}_\pi(b) = F\|$. Then π is an ∞-cover.

Example 2.2.4. As an example of an ∞-cover, consider the exponential map $\mathbb{R} \to \mathbb{S}^1$ from the real line to the circle. The fibers of this map are all merely $\mathbb{Z}$, so by Theorem 2.2.3, this map is an ∞-cover. Since $\mathbb{R}$ is contractible, it is in fact the *universal* ∞-cover of the circle.

Just as the universal cover of a space X is any simply connected cover $\tilde{X}$, the universal ∞-cover $\overset{\infty}{X}$ of a type X is any *contractible* cover — contractible in the sense of being $\int$-connected, meaning $\int \overset{\infty}{X} = *$. Since units of a modality are modally connected, we may always construct a universal ∞-cover by taking the fiber of the $\int$-unit $(-)^{\int} : X \to \int X$.

Definition 2.2.5. The *universal* ∞-cover of a pointed type X is defined to be the fiber of the $\int$-unit:

$$\overset{\infty}{X} :\equiv \mathsf{fib}((-)^{\int} : X \to \int X).$$

Since the units of modalities are modally connected, $\overset{\infty}{X}$ is homotopically contractible:

$$\int \overset{\infty}{X} = *.$$

Let's take a bit to get an image of the universal ∞-cover of a type. The universal ∞-cover of a type only differs from its universal cover in the identifications between its points; in other words, it is a "stacky" version of the universal cover.

Proposition 2.2.6. Let X be a crisp type. Then the map $\overset{\infty}{X} \to \tilde{X}$ from the universal ∞-cover of X to its universal cover induced by the square

$$\begin{tikzcd}
\overset{\infty}{X} \arrow[r, "\pi"] \arrow[d, dashed] & X \arrow[r, "(-)^{\int}"] \arrow[d, equal] & \int X \arrow[d] \\
\tilde{X} \arrow[r, "\pi"'] & X \arrow[r, "(-)^{\int_1}"'] & \int_1 X
\end{tikzcd}$$

is $\|-\|_0$-connected and $\int$-modal. In particular, if X is a set then $\left\| \overset{\infty}{X} \right\|_0 = \tilde{X}$. Furthermore, its fibers may be identified with the loop space $\Omega((\int X)\langle 1 \rangle)$ of the first stage of the Whitehead tower of the shape of X.

Proof. We begin by noting that the unique factorization $f : \int X \to \int_1 X$ of the $\int_1$ unit $(-)^{\int_1} : X \to \int_1 X$ through the $\int$ unit is a $\|-\|_1$ unit. We note that $(-)^{\int_1} : X \to \int_1 X$ and $|(-)^{\int}|_1 : X \to \|\int X\|_1$ have the same universal property: any map from X to a discrete 1-type factors uniquely through them. However, unless X is crisp, we do not know that $\|\int X\|_1$ is itself discrete; in general, we can only conclude that there is a map $\|\int X\|_1 \to \int_1 X$. This is why we must assume that X is crisp. By Proposition 4.5 of [17], $\|\int X\|_1$ is discrete and therefore the map $\|\int X\|_1 \to \int_1 X$ is an equivalence. Since f factors uniquely through this map (since $\int_1$ is a 1-type), we see that f is equal to the $\|-\|_1$ unit of $\int X$ and is therefore a $\|-\|_1$ unit. In particular, $f : \int X \to \int_1 X$ is 1-connected.

/ow we will show that the fibers of the induced map $\overset{\infty}{X} \to \tilde{X}$ are 0-connected and discrete. Consider the following diagram:

$$\begin{tikzcd}
\mathsf{fib}(p) \arrow[r] \arrow[d] & * \arrow[r] \arrow[d] & \mathsf{fib}((\pi p)^{\int_1}) \arrow[d] \\
\overset{\infty}{X} \arrow[r, "\pi"] \arrow[d] & X \arrow[r, "(-)^{\int}"] \arrow[d, equal] & \int X \arrow[d] \\
\tilde{X} \arrow[r, "\pi"'] & X \arrow[r, "(-)^{\int_1}"'] & \int_1 X
\end{tikzcd}$$

All vertical sequences are fiber sequences, and the bottom two sequences are fiber sequences; therefore, the top sequence is a fiber sequence, which tells us that the fiber over any point $p : \tilde{X}$ is equivalent to $\Omega\, \mathsf{fib}_f((\pi p)^{\int_1})$. As we have shown that the fibers of f are 1-connected, their loop spaces are 0-connected. And as f is a map between discrete types, its fibers are discrete and so their loop spaces are discrete. Finally, we note that the fiber of the 1-truncation $|\cdot|_1 : \int X \to \|\int X\|_1$ is the first stage $(\int X)\langle 1 \rangle$ of the Whitehead tower of $\int X$. □

We are now ready to prove a simple sort of fracture theorem for any crisp, pointed type. This square will make up the left square of our modal fracture hexagon.

Proposition 2.2.7. Let X be a crisp, pointed type. Then the $\flat$ naturality square of the universal ∞-cover $\pi : \overset{\infty}{X} \to X$ is a pullback:

$$\begin{tikzcd}
\flat\overset{\infty}{X} \arrow[r, "(-)_\flat"] \arrow[d, "\flat\pi"'] & \overset{\infty}{X} \arrow[d, "\pi"] \\
\flat X \arrow[r, "(-)_\flat"'] & X
\end{tikzcd}$$

Proof. By Lemma 2.1.5, it will suffice to show that over a crisp point $x :: X$, $\mathsf{fib}_\pi(x)$ is discrete. But since π is, by definition, the fiber of $(-)^{\int}$, we have that

$$\mathsf{fib}_\pi(x) = \Omega(\textstyle\int X, x^{\int}).$$

Since $\int X$ is discrete by assumption, so is $\Omega(\int X, x^{\int})$. □

Although $\int$ is *not* a left exact modality — it does not preserve all pullbacks — it does preserve pullbacks and fibers of *$\int$-fibrations.* The theory of modal fibrations was developed in [17]. Included amongst the $\int$-fibrations are the $\int$-étale maps, and so $\int$ preserves fiber sequences of $\int$-étale maps.

Lemma 2.2.8. Let $f : X \to Y$ be an ∞-cover. Then for any $y : Y$, the sequence

$$\textstyle\int \mathsf{fib}_f(y) \to \int X \xrightarrow{\int f} \int Y$$

is a fiber sequence.

Proof. Since a $\int$-étale map f is modal, its étale and modal factors agree (they are equivalently f), so by Theorem 1.2 of [17], f is a $\int$-fibration. The result then follows since $\int$ preserves all fibers of $\int$-fibrations (see also Theorem 1.2 of [17]). □

Importantly, it is also true that the shape of a *crisp* n-connected type is also n-connected by Theorem 8.6 of [17]. It follows that $\int \mathbf{B}G$ is a delooping of $\int G$ for crisp higher groups G, and that this can continue for higher deloopings.

Proposition 2.2.9. Let G be a crisp higher group. Then its universal ∞-cover $\overset{\infty}{G}$ is a higher group and $\pi : \overset{\infty}{G} \to G$ is a homomorphism. Futhermore, if G is k-commutative, then so is $\overset{\infty}{G}$.

Proof. We may define

$$\mathbf{B}^i\overset{\infty}{G} :\equiv \mathsf{fib}((-)^{\int} : \mathbf{B}^i G \to \textstyle\int \mathbf{B}^i G).$$

This lets us extend the fiber sequence:

$$\begin{tikzcd}
\overset{\infty}{G} \arrow[r, "\pi"] & G \arrow[r, "(-)^{\int}"] & \int G \arrow[dll, rounded corners] \\
\mathbf{B}\overset{\infty}{G} \arrow[r, "\mathbf{B}\pi"] & \mathbf{B}G \arrow[r, "(-)^{\int}"] & \int \mathbf{B}G \arrow[dll, rounded corners] \\
\mathbf{B}^2\overset{\infty}{G} \arrow[r, "\mathbf{B}^2\pi"] & \mathbf{B}^2G \arrow[r, "(-)^{\int}"] & \int \mathbf{B}^2G \arrow[dll, rounded corners] \\
\cdots & &
\end{tikzcd}$$

□

2.3 The Infinitesimal Remainder of a Higher Group

In this section, we will investigate the infinitesimal remainder $\theta : G \to \mathbb{g}$ of higher group G. The infinitesimal remainder is what is left of a higher group when all of its crisp points have been made equal. Having trivialized all substanstial difference between points, we are left with the infinitesimal differences that remain.

Definition 2.3.1. Let G be a higher group. Define its *infinitesimal remainder* to be

$$\mathbb{g} :\equiv \mathsf{fib}((-)_\flat : \flat\mathbf{B}G \to \mathbf{B}G).$$

Then, continuing the fiber sequence, we have

$$\begin{tikzcd}
\flat G \arrow[r, "(-)_\flat"] & G \arrow[r, "\theta"] & \mathbb{g} \arrow[dll, rounded corners] \\
\flat\mathbf{B}G \arrow[r] & \mathbf{B}G &
\end{tikzcd}$$

which defines the quotient map $\theta : G \to \mathbb{g}$.

Remark 2.3.2. By its construction, we can see that $\mathbb{g}$ modulates flat connections on trivial principal G-bundles, with respect to the interpretation of $\flat\mathbf{B}G$ given in Remark 2.1.3. In the setting of differential geometry, such flat connections on trivial principal G-bundles are given by closed $\mathfrak{g}$-valued 1-forms, where here $\mathfrak{g}$ is the Lie algebra of the Lie group G. In this setting, θ is the Mauer-Cartan form on G. This is why we adopt the name $\theta : G \to \mathbb{g}$ for the infinitesimal remainder in general. This can in fact be *proven* in the setting of synthetic differential geometry with tiny infinitesimals satisfying a principle of constancy using a purely modal argument. See Remark 3.1.2 for a further discussion.

Remark 2.3.3. The infinitesimal remainder $\mathbb{g}$ is defined as the *de Rham coefficient object* of $\mathbf{B}G$ in Definition 5.2.59 of [20]. Schreiber defined $\flat_{dR}X$ for any (crisp) pointed type X as the fiber of $(-)^\flat : \flat X \to X$, so that $\mathbb{g} \equiv \flat_{dR}\mathbf{B}G$. We focus on the case that X is 0-connected — of the form $\mathbf{B}G$ — and so only consider the infinitesimal remainder of a higher group G.

While the infinitesimal remainder exists for any (crisp) higher group, it is not necessarily itself a higher group. However, if G is braided, then $\mathbb{g}$ will be a higher group.

Proposition 2.3.4. If G is a crisp k-commutative higher group, then $\mathbb{g}$ is a $(k-1)$-commutative higher group. In particular, if G is a braided higher group, then $\mathbb{g}$ is a higher group and the remainder map $\theta : G \to \mathbb{g}$ is a homomorphism.

Proof. We may define

$$\mathbf{B}^i\mathbb{g} :\equiv \mathsf{fib}((-)_\flat : \flat\mathbf{B}^{i+1}G \to \mathbf{B}^{i+1}G),$$

which lets us continue the fiber sequence:

$$\begin{tikzcd}
\flat G \arrow[r, "(-)_\flat"] & G \arrow[r, "\theta"] & \mathbb{g} \arrow[dll, rounded corners] \\
\flat\mathbf{B}G \arrow[r] & \mathbf{B}G \arrow[r, "\mathbf{B}\theta"] & \mathbf{B}\mathbb{g} \arrow[dll, rounded corners] \\
\flat\mathbf{B}^2G \arrow[r] & \mathbf{B}^2G \arrow[r, "\mathbf{B}^2\theta"] & \mathbf{B}^2\mathbb{g} \arrow[dll, rounded corners] \\
\dots & &
\end{tikzcd}$$

□

Remark 2.3.5. We can see the delooping $\mathbf{B}\theta : \mathbf{B}G \to \mathbf{B}\mathbb{g}$ of the infinitesimal remainder $\theta : G \to \mathbb{g}$ as taking the *curvature* of a principal G-bundle, in that $\mathbf{B}\theta$ is an obstruction to the flatness of that bundle since

$$\flat\mathbf{B}G \to \mathbf{B}G \to \mathbf{B}\mathbb{g}$$

is a fiber sequence.

As with any good construction, the infinitesimal remainder is functorial in its higher group. This is defined easily since the infinitesimal remainder is constructed as a fiber.

Definition 2.3.6. Let $f :: G \to H$ be a crisp homomorphism of higher groups with delooping $\mathbf{B}f : \mathbf{B}G \cdot\!\!\to \mathbf{B}H$. Then we have a pushforward $f_* : \mathbb{g} \cdot\!\!\to \mathbb{h}$ given by $(t, p) \mapsto (\flat\mathbf{B}f(t), (\mathsf{ap}\ \mathbf{B}f\, p) \bullet \mathsf{pt}_{\mathbf{B}f})$. This is the unique map fitting into the following diagram:

$$\begin{tikzcd}
\mathbb{g} \arrow[r, "f_*"] \arrow[d] & \mathbb{h} \arrow[d] \\
\flat\mathbf{B}G \arrow[r, "\flat\mathbf{B}f"] \arrow[d] & \flat\mathbf{B}H \arrow[d] \\
\mathbf{B}G \arrow[r, "\mathbf{B}f"'] & \mathbf{B}H
\end{tikzcd}$$

If G and H are k-commutative and f is a k-commutative homomorphism, then f_* admits a unique structure of a $(k-1)$-commutative homomorphism by defining $\mathbf{B}^{k-1}f_*$ to be the map induced by $\flat\mathbf{B}^k f$ on the fiber.

We record a useful lemma: the fibers of the quotient map $\theta : G \to \mathbb{g}$ are all are identifiable with $\flat G$.

Lemma 2.3.7. Let G be a crisp higher group. For $t : \mathbb{g}$, we have $\|\mathsf{fib}_\theta(t) = \flat G\|$.

Proof. By definition, $t : \mathbb{g}$ is of the form (T, p) for $T : \flat\mathbf{B}G$ and $p : T_\flat = \mathsf{pt}_{\mathbf{B}G}$. Since $\flat\mathbf{B}G$ is 0-connected and we are trying to prove a proposition, we may suppose that $q : T = \mathsf{pt}_{\flat\mathbf{B}G}$. We then have that $t = (\mathsf{pt}_{\flat BG}, (-)_{\flat *}(q) \bullet p)$, and therefore:

$$\begin{aligned}
\mathsf{fib}_\theta(t) &\equiv (g : G) \times ((\mathsf{pt}_{\flat\mathbf{B}G}, g) = (\mathsf{pt}_{\flat\mathbf{B}G}, (-)_{\flat *}(q) \bullet p)) \\
&= (g : G) \times (a : \flat G) \times (a_\flat \bullet g = (-)_{\flat *}(q) \bullet p) \\
&= (g : G) \times (a : \flat G) \times \left(g = {a_\flat}^{-1} \bullet (-)_{\flat *}(q) \bullet p\right) \\
&= \flat G.
\end{aligned}$$

□

The infinitesimal remainder is *infinitesimal* in the sense that it has a single crisp point.

Proposition 2.3.8. Let G be a higher group. Then its infinitesimal remainder $\mathbb{g}$ is infinitesimal in the sense that

$$\flat\mathbb{g} = *.$$

Proof. By Lemma 2.1.4, $\flat$ preserves the fiber sequence

$$\mathbb{g} \to \flat\mathbf{B}G \to \mathbf{B}G.$$

But $\flat(-)_\flat : \flat\flat\mathbf{B}G \to \flat\mathbf{B}G$ is an equivalence by Theorem 6.18 of [22], so $\flat\mathbb{g}$ is contractible. □

Despite being infinitesimal, we will see that $\mathbb{g}$ has (in general) a highly non-trivial homotopy type.

Remark 2.3.9. The infinitesimal remainder $\mathbb{g}$ is of special interest when G is a Lie group, since in this case the vanishing of the cohomology groups $H^*(\mathbb{g}; \mathbb{Z}/p)$ for all primes p is equivalent to the Friedlander-Milnor conjecture. The fact that this conjecture remains unproven is a testament to the intricacy of the homotopy type of the infinitesimal space $\mathbb{g}$.

We note that $\mathbb{g}$ itself represents an obstruction to the discreteness of G.

Proposition 2.3.10. A crisp higher group G is discrete if and only if its infinitesimal remainder $\mathbb{g}$ is contractible.

Proof. If G is discrete, then $(-)_\flat : \flat G \to G$ is an equivalence and so $(-)_\flat : \flat\mathbf{B}G \to \mathbf{B}G$ is an equivalence: this implies that $\mathbb{g} = *$. On the other hand, if $\mathbb{g} = *$ then $(-)_\flat : \flat\mathbf{B}G \to \mathbf{B}G$ is an equivalence and so its action on loops is an equivalence. □

Using Proposition 2.2.3, we can quickly show that $\theta : G \to \mathbb{g}$ is an ∞-cover. This gives us the right hand pullback square in our modal fracture hexagon.

Proposition 2.3.11. Let G be a crisp ∞-group. Then the infinitesimal remainder $\theta : G \to \mathbb{g}$ is an ∞-cover. In particular, the $\int$-naturality square:

$$\begin{tikzcd}
G \arrow[r, "\theta"] \arrow[d, "(-)^{\int}"'] & \mathbb{g} \arrow[d, "(-)^{\int}"] \\
\int G \arrow[r, "\int \theta"'] & \int \mathbb{g}
\end{tikzcd}$$

is a pullback. If, furthermore, G is (crisply) an n-type, then θ is an $(n+1)$-cover.

Proof. By Proposition 2.2.3, to show that $\theta : G \to \mathbb{g}$ is an ∞-cover (resp. an $(n+1)$-cover) it suffices to show that the fibers are merely equivalent to a crisply discrete type (resp. a crisply discrete n-type). But by Lemma 2.3.7, the fibers of $\theta : G \to \mathbb{g}$ are all merely equivalent to $\flat G$ which is crisply discrete (and, by Theorem 6.6 of [22], if G is (crisply) an n-type then so is $\flat G$). □

There is a sense in which the infinitesimal remainder of a higher group behaves like its Lie algebra. Just as the Lie algebra of a Lie group is the same as the Lie algebra of its universal cover, we can show that the infinitesimal remainder of a higher group is the same as that of its universal ∞-cover.

Proposition 2.3.12. Let $G \xrightarrow{\phi} H \xrightarrow{\psi} K$ be a crisp exact sequence of higher groups. Then

1. K is discrete if and only if $\phi_* : \mathbb{g} \to \mathbb{h}$ is an equivalence.
2. G is discrete if and only if $\psi_* : \mathbb{h} \to \mathbb{k}$ is an equivalence.

Proof. We consider the following diagram in which each horizontal and vertical sequence is a fiber sequence:

$$\begin{tikzcd}
\mathbb{g} \arrow[r] \arrow[d] & \mathbb{h} \arrow[r] \arrow[d] & \mathbb{k} \arrow[d] \\
\flat \mathbf{B}G \arrow[r] \arrow[d] & \flat \mathbf{B}H \arrow[r] \arrow[d] & \flat \mathbf{B}K \arrow[d] \\
\mathbf{B}G \arrow[r] & \mathbf{B}H \arrow[r] & \mathbf{B}K
\end{tikzcd}$$

If K is discrete, then $\mathbb{k} = *$ and so $\mathbb{g} = \mathbb{h}$. On the other hand, if $\mathbb{g} = \mathbb{h}$, then the bottom left square of the above diagram is a pullback. Therefore, the it induces an equivalence on the fibers of the horizontal maps:

$$\begin{tikzcd}
\flat K \arrow[r] \arrow[d, "\sim"] & \flat \mathbf{B}G \arrow[r] \arrow[d] & \flat \mathbf{B}H \arrow[d] \\
K \arrow[r] & \mathbf{B}G \arrow[r] & \mathbf{B}H
\end{tikzcd}$$

This shows that K is discrete.

If $\psi_* : \mathbb{h} \to \mathbb{k}$ is an equivalence, then its fiber $\mathbb{g}$ is contractible. Therefore, G is discrete. On the other hand, if G is discrete, then the bottom right square is a pullback, and therefore the induced map on vertical fibers is an equivalence. This map is $\psi_* : \mathbb{h} \to \mathbb{k}$. □

Corollary 2.3.13. The universal ∞-cover $\pi : \overset{\infty}{G} \to G$ induces an equivalence $\overset{\infty}{\mathbb{g}} = \mathbb{g}$ fitting into the following commutative diagram:

$$\begin{tikzcd}
\overset{\infty}{\mathbb{g}} \arrow[r, equal] \arrow[d] & \mathbb{g} \arrow[r] \arrow[d] & * \arrow[d] \\
\flat \mathbf{B}\overset{\infty}{G} \arrow[r] \arrow[d] & \flat \mathbf{B}G \arrow[r] \arrow[d] & \flat \int \mathbf{B}G \arrow[d, equal] \\
\mathbf{B}\overset{\infty}{G} \arrow[r] & \mathbf{B}G \arrow[r] & \int \mathbf{B}G
\end{tikzcd}$$

In particular, this gives us a long fiber sequence

$$\begin{tikzcd}
\flat\overset{\infty}{G} \arrow[r, "(-)_\flat"] & \overset{\infty}{G} \arrow[r, "\theta"] & \mathfrak{g} \arrow[dll, out=-30, in=150] \\
\flat\mathbf{B}\overset{\infty}{G} \arrow[r, "\mathbf{B}\pi"] & \mathbf{B}\overset{\infty}{G} &
\end{tikzcd}$$

which forms the top fiber sequence of the modal fracture hexagon.

2.4 The Modal Fracture Hexagon

We have seen the two main fiber sequences

$$\begin{tikzcd}
\overset{\infty}{G} \arrow[r, "\pi"] & G \arrow[r, "(-)^{\int}"] & \int G \arrow[dll, out=-30, in=150] \\
\mathbf{B}\overset{\infty}{G} \arrow[r, "\mathbf{B}\pi"] & \mathbf{B}G \arrow[r, "(-)^{\int}"] & \int \mathbf{B}G
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\flat G \arrow[r, "(-)_\flat"] & G \arrow[r, "\theta"] & \mathfrak{g} \arrow[dll, out=-30, in=150] \\
\flat\mathbf{B}G \arrow[r] & \mathbf{B}G &
\end{tikzcd}$$

associated to a higher group G. Now, when we apply $\flat$ to the left sequence and $\int$ to the right sequence, we find the sequences

$$\begin{tikzcd}
\flat\overset{\infty}{G} \arrow[r, "\flat\pi"] & \flat G \arrow[r, "(-)^{\int}\circ(-)_\flat"] & \int G \arrow[dll, out=-30, in=150] \\
\flat\mathbf{B}\overset{\infty}{G} \arrow[r, "\flat\mathbf{B}\pi"] & \flat\mathbf{B}G \arrow[r, "(-)^{\int}\circ(-)_\flat"] & \int \mathbf{B}G
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\flat G \arrow[r, "(-)^{\int}\circ(-)_\flat"] & \int G \arrow[r, "\int\theta"] & \int\mathfrak{g} \arrow[dll, out=-30, in=150] \\
\flat\mathbf{B}G \arrow[r] & \int\mathbf{B}G &
\end{tikzcd}$$

which are *the same sequence, just shifted over*. This gives us the bottom exact sequence of our modal fracture hexagon, reading the sequence on the left. But it also proves that $\int\mathfrak{g} = \flat\mathbf{B}\overset{\infty}{G}$, which gives us the top exact sequence of our modal fracture hexagon by Corollary 2.3.13.

Of course, we need to be able to apply $\int$ freely to fiber sequences to fulfil this argument. But $\int$ is not left exact, and so does not preserve fiber sequences in general. Luckily, Theorem 6.1 of [17] gives us a trick for showing that a map is a $\int$-fibration which allows us to prove this general lemma.

Lemma 2.4.1. Let G be a crisp higher group (that is, $\mathbf{B}G$ is a crisply pointed 0-connected type). Then any crisp map $f :: X \to \mathbf{B}G$ is a $\int$-fibration.

Proof. Since $\mathbf{B}G$ is 0-connected, all the fibers of f are merely equivalent to the fiber $\mathsf{fib}_f(\mathsf{pt})$ over the basepoint. Therefore, their homotopy types are merely equivalent to $\int\mathsf{fib}_f(\mathsf{pt})$, which is a crisp, discrete type. It follows by Theorem 6.1 of [17] that f is a $\int$-fibration. □

This means that we can freely apply $\int$ to crisp fiber sequences of 0-connected types. This concludes our proof of the main theorem.

Theorem 2.4.2. *For a crisp ∞-group G, there is a* modal fracture hexagon*:*

$$\begin{tikzcd}
 & \overset{\infty}{G} \arrow[rr] \arrow[dr, "\pi"] & & \mathfrak{g} \arrow[dr, "(-)^{\int}"] & \\
\flat\overset{\infty}{G} \arrow[ur, "(-)_\flat"] \arrow[dr, "\flat\pi"'] & & G \arrow[ur, "\theta"] \arrow[dr, "(-)^{\int}"] & & \int\mathfrak{g} \\
 & \flat G \arrow[rr] \arrow[ur, "(-)_\flat"] & & \int G \arrow[ur, "\int\theta"'] &
\end{tikzcd}$$

where

- *$\theta : G \to \mathfrak{g}$ is the* infinitesimal remainder *of G, the quotient $G \mathbin{/\!/} \flat G$, and*
- *$\pi : \overset{\infty}{G} \to G$ is the* universal (contractible) ∞-cover *of G.*

Moreover,

1. *The middle diagonal sequences are fiber sequences.*
2. *The top and bottom sequences are fiber sequences.*
3. *Both squares are pullbacks.*

Furthermore, the homotopy type of $\mathfrak{g}$ is a delooping of $\flat\overset{\infty}{G}$:

$$\int \mathfrak{g} = \flat \boldsymbol{B}\overset{\infty}{G}.$$

Therefore, if G is k-commutative for $k \geq 1$ (that is, admits futher deloopings $\boldsymbol{B}^{k+1}G$), then we may continue the modal fracture hexagon on to $\boldsymbol{B}^k G$.

Proof. We assemble the various components of the proof here.

1. The middle diagonal sequences are fiber sequences by definition (see Definition 2.2.5 and Definition 2.3.6).
2. The top sequence was shown to be a fiber sequence in Corollary 2.3.13. We showed that the bottom sequence is a fiber sequence at the beginning of this subsection.
3. The left square was shown to be a pullback in Proposition 2.2.7, and the right sequence in Proposition 2.3.11.

Finally, we calculated the homotopy type of $\mathfrak{g}$ at the beginning of this subsection. □

3 Ordinary Differential Cohomology

In this section, we will use modal fracture to construct ordinary differential cohomology in cohesive homotopy type theory. We will recover a differential hexagon for ordinary differential cohomology which very closely resembles the classical hexagon; however, as de Rham's theorem does not hold for all types, we will not recover the classical hexagon exactly. For more discussion of these subtleties, see Section 3.3.

In [2], Bunke, Nikolaus, and Vokel show that differential cohomology theories can be understood as spectra in the ∞-topos of sheaves on a site of manifolds. Schreiber notes in Proposition 4.4.9 of [20] that the simpler site consisting of Euclidean spaces and smooth maps between them yields the same topos of sheaves, and proves in Proposition 4.4.8 that this ∞-topos is cohesive. This topos, and the similar ∞-Dubuc topos (called the ∞-Cahiers topos in Remark 4.5.6 and **SynthDiff∞Grpd** in Definition 4.5.7 of *ibid.*), will be our intended model for cohesive homotopy type theory in this section.

The theme of this paper is that the main feature of differential cohomology — the differential cohomology hexagon — is not of a particularly differential character, but arises from the more basic opposition between an adjoint modality $\int$ and comodality $\flat$. As we saw in the previous section, in the presence of these (co)modalities, any higher group may be fractured in a manner resembling the differential cohomology hexagon.

We will take a similarly general view in constructing ordinary differential cohomology. The key idea in ordinary differential cohomology is the equipping of differential form data to integral cohomology. We will therefore focus on cohomology theories (in particular, ∞-commutative higher groups or connective spectra) which arise by equipping an existing cohomology theory with extra data representing the cocycles. Our exposition will focus on ordinary differential cohomology, but this extra generality will enable us to define combinatorial analogues of ordinary differential cohomology as well (see Section 3.5).

3.1 Assumptions and Preliminaries

For this section, we make the following assumption.

Assumption 1. In cohesive homotopy type theory with the axioms of synthetic differential geometry, *tiny infinitesimal varieties*, and a *principle of constancy*, we have a contractible and infinitesimal resolution of $U(1)$

$$0 \to \flat U(1) \to U(1) \xrightarrow{d} \Lambda^1 \xrightarrow{d} \Lambda^2 \xrightarrow{d} \cdots \tag{3.1.1}$$

given by the *differential k-form classifiers* Λ^k. That is:

- The Λ^k are crisp $\mathbb{R}$-vector spaces.
- The maps $d :: \Lambda^k \to \Lambda^{k+1}$ are crisply $\flat\,\mathbb{R}$-linear (not $\mathbb{R}$-linear!), and the sequence Eq. (3.1.1) is crisply long exact.
- The Λ^k are infinitesimal: $\flat\Lambda^k = *$. Therefore also the closed k-form classifiers $\Lambda^k_{\mathrm{cl}} :\equiv \mathsf{ker}(d : \Lambda^k \to \Lambda^{k+1})$ are infinitesimal.

Here, $U(1) \equiv \{z : \mathbb{C} \mid z\bar{z} = 1\}$ is the abelian group of units in the *smooth* complex numbers, which are defined as $\mathbb{C} \equiv \mathbb{R}[i]/(i^2+1)$ where $\mathbb{R}$ are the *smooth real numbers* presumed by synthetic differential geometry.

Remark 3.1.2. For reasons of space, we will not justify this assumption in this paper. In forthcoming work, we will show how one can construct the form classifiers and their long exact sequence

$$0 \to \flat\,\mathbb{R} \to \mathbb{R} \xrightarrow{d} \Lambda^1 \xrightarrow{d} \Lambda^2 \to \cdots \tag{3.1.3}$$

from the axioms of synthetic diffential geometry with tiny infinitesimals and a principle of constancy. Synthetic differential geometry is an axiomatic system for working with nilpotent infinitesimals put forward first by Lawvere [11] and developed by Bunge, Dubuc, Kock, Wraith, and others. It admits a model in sheaves on infinitesimally extended Euclidean spaces, known as the Dubuc topos (or Cahiers topos) [7]; for a review of models see [16]. The Dubuc topos is cohesive, and is our intended model for this section.

It was noted by Lawvere [13] that the exceptional projectivity enjoyed by the infinitesimal interval $\mathbb{D} = \{\epsilon : \mathbb{R} \mid \epsilon^2 = 0\}$ was equivalent to the existence of an (external) right adjoint to the exponential functor $X \mapsto X^{\mathbb{D}}$. We will follow Yetter [25] in calling objects T for which the functor $X \mapsto X^T$ admits a right adjoint *tiny objects*. Lawvere and Kock showed how one could use this "amazing" right adjoint to construct the form classifiers Λ^k (see Section I.20 of [9] for a construction of Λ^1).

However, working with the form classifiers was difficult in synthetic differential geometry since the adjoint which defines them only exists externally. This may be remedied by using Shulman's Cohesive HoTT, where the $\sharp$ modality allows for an internalization of the external. This allows us to give a fully internal theory of the form classifiers. We will, however, post-pone a discussion of this internal theory of tiny objects to future work.

The principle of constancy says that if the differential of a function $f : \mathbb{R} \to \mathbb{R}$ vanishes uniformly, then f is constant. This extra principle has been long considered in synthetic differential geometry (see, for example, the second chapter of [15]), but when combined with real cohesion it implies the exactness of the sequence

$$0 \to \flat\,\mathbb{R} \to \mathbb{R} \xrightarrow{d} \Lambda^1$$

and so begins the theory of differential cohomology we will see shortly. The interaction with cohesion is non-trivial in many ways for synthetic differential geomtry. For example, the principle of constancy in the presence of real cohesion implies the existence of primitives, and the exponential functions $\exp(-) : \mathbb{R} \to \mathbb{R}^+$ and $\exp(2\pi i -) : \mathbb{R} \to U(1)$ (where $U(1) := \{z : \mathbb{C} \mid z\overline{z} = 1\}$).

Remark 3.1.4. Externally, the smooth reals $\mathbb{R}$ correspond to the sheaf of smooth real valued functions, $U(1)$ corresponds to the sheaf of smooth $U(1)$-valued functions, and Λ^k is the sheaf sending a manifold to its set of differential k-forms.

We have assumed the existence of a crisp long exact sequence of abelian groups in which each of the Λ^i are real vector spaces (but $d : \Lambda^i \to \Lambda^{i+1}$ are not $\mathbb{R}$-linear). These are to be the *differential form classifiers*, which externally are the sheaves of $\mathbb{R}$-valued n-forms on manifolds (or infinitesimally extended manifolds).

If we define

$$\Lambda^n_{\text{cl}} :\equiv \mathsf{ker}(d : \Lambda^n \to \Lambda^{n+1})$$

to be the closed n-form classifier, then we can reorganize the long exact sequence (3.1.3) into a series of short exact sequences of abelian groups

$$0 \to \Lambda^n_{\text{cl}} \to \Lambda^n \overset{d}{\to} \Lambda^{n+1}_{\text{cl}} \to 0.$$

The first of these short exact sequences is

$$0 \to \flat\,\mathbb{R} \to \mathbb{R} \overset{d}{\to} \Lambda^1_{\text{cl}} \to 0$$

which we may extend into a long fiber sequence

$$\begin{tikzcd}
\flat\,\mathbb{R} \arrow[r, "(-)_\flat"] & \mathbb{R} \arrow[r, "d"] & \Lambda^1_{\text{cl}} \arrow[dll, out=-30, in=150] \\
\flat\mathbf{B}\,\mathbb{R} \arrow[r] & \mathbf{B}\,\mathbb{R} \arrow[r] & \cdots
\end{tikzcd}$$

This shows that Λ^1_{cl} is the infinitesimal remainder of the additive Lie group $\mathbb{R}$. Since $\mathbb{R}$ has contractible shape by definition, we see that $\mathbb{R} \overset{d}{\to} \Lambda^1_{\text{cl}}$ is the universal ∞-cover of Λ^1_{cl}. This gives us the following theorem, a form of de Rham's theorem in smooth cohesion.

Lemma 3.1.5. The n-form classifiers Λ^n are contractible.

Proof. This follows immediately from the assumption that they are real vector spaces. By Lemma 6.9 of [17], to show that Λ^n is contractible it suffices to give for every $\omega : \Lambda^n$ a path $\gamma : \mathbb{R} \to \Lambda^n$ from ω to 0. We can of course define

$$\gamma(t) :\equiv t\omega$$

which gives our desired contraction. □

Theorem 3.1.6. *Let $\Lambda^n_{cl} :\equiv \mathsf{ker}(d : \Lambda^n \to \Lambda^{n+1})$ be the closed n-form classifier. Then*

$$\int \Lambda^n_{cl} = \flat \boldsymbol{B}^n\,\mathbb{R}\,.$$

Proof. Since Λ^1_{cl} is the infinitesimal remainder of $\mathbb{R}$, this follows from Theorem 2.4.2:

$$\int \Lambda^1_{\text{cl}} = \flat\mathbf{B}\,\mathbb{R}\,.$$

We then proceed by induction. We have a short exact sequence of abelian groups

$$0 \to \Lambda^n_{\text{cl}} \to \Lambda^n \overset{d}{\to} \Lambda^{n+1}_{\text{cl}} \to 0.$$

We note that since $d : \Lambda^n \to \Lambda^{n+1}_{\text{cl}}$ is an abelian group homomorphism, all of its fibers are identifiable with the crisp type Λ^n_{cl} and therefore, by the "good fibrations" trick (Theorem 6.1 of [17]), it is a $\int$-fibration. Therefore, we get a fiber sequence

$$\int \Lambda^n_{\text{cl}} \to \int \Lambda^n \to \int \Lambda^{n+1}_{\text{cl}}.$$

Now, since Λ^n is contractible by Lemma 3.1.5, we see that

$$\Omega \int \Lambda^{n+1}_{\text{cl}} = \int \Lambda^n_{\text{cl}}$$

By inductive hypothesis, $\int \Lambda^n_{\text{cl}} = \flat B^n\,\mathbb{R}$, so all that remains is to show that $\int \Lambda^{n+1}_{\text{cl}}$ is n-connected. We will do this by showing that for any $u : \int \Lambda^{n+1}_{\text{cl}}$, the loop space $\Omega(\int \lambda^{n+1}_{\text{cl}}, u)$ is $(n-1)$-connected. By Corollary 9.12 of [22], the $\int$-unit $(-)^{\int} : \Lambda^{n+1}_{\text{cl}} \to \int \Lambda^{n+1}_{\text{cl}}$ is surjective, so there exists an $\omega : \Lambda^{n+1}_{\text{cl}}$ with $u = \omega^{\int}$. We then have a fiber sequence

$$\mathsf{fib}_d(\omega) \to \Lambda^n \xrightarrow{d} \Lambda^{n+1}_{\mathrm{cl}}$$

which, since $\Lambda^n \xrightarrow{d} \Lambda^{n+1}$ is a $\int$-fibration descends to a fiber sequence

$$\int \mathsf{fib}_d(\omega) \to \int \Lambda^n \xrightarrow{\int} \int \Lambda^{n+1}_{\mathrm{cl}}. \tag{3.1.7}$$

Since d is surjective, there is a $\alpha : \Lambda^n$ with $d\alpha = \omega$, and we may therefore contract $\int \Lambda^n$onto $\alpha^{\int}$. This lets us equate the sequence (3.1.7) with the sequence

$$\Omega(\int \Lambda^{n+1}_{\mathrm{cl}}, \omega^{\int}) \to * \xrightarrow{\omega^{\int}} \int \Lambda^{n+1}_{\mathrm{cl}}.$$

But as $d : \Lambda^n \to \Lambda^{n+1}_{\mathrm{cl}}$ is an abelian group homomophism, its fibers are all identifiable with its kernel Λ^n_{cl}; this means that $\Omega(\int \Lambda^{n+1}_{\mathrm{cl}}, u)$ is identifiable with $\int \Lambda^n_{\mathrm{cl}}$, which by inductive hypothesis is $(n-1)$-connected. □

We may understand this theorem as a form of de Rham theorem in smooth cohesive homotopy type theory. We may think of the unit $(-)^{\int} : \Lambda^n_{\mathrm{cl}} \to \flat B^n\,\mathbb{R}$ as giving the de Rham class of a closed n-form. That this map is the $\int$-unit says that this is the universal discrete cohomological invariant of closed n-forms. Explicitly, if $E_\bullet$ is a loop spectrum, then $H^k(\Lambda^n_{\mathrm{cl}}; E_\bullet) :\equiv \|\Lambda^n_{\mathrm{cl}} \to E_k\|_0$. Therefore, if the E_k are discrete, then any cohomology class $c : \Lambda^n_{\mathrm{cl}} \to E_k$ factors through the de Rham class $(-)^{\int} : \Lambda^n_{\mathrm{cl}} \to \flat\mathbf{B}^n\,\mathbb{R}$. In this sense, every discrete cohomological invariant of closed n-forms is in fact an invariant of their de Rham class in discrete real cohomology.

3.2 Circle k-Gerbes with Connection

We can now go about defining ordinary differential cohomology. We understand ordinary differential cohomology as equipping integral cohomology with differential form data. Hopkins and Singer define (Definition 2.4 of [8]) a *differential cocycle* of degree $k+1$ on X to be a triple (c, h, ω) consisting of an underlying cocycle $c \in Z^{k+1}(X, \mathbb{Z})$ in integral cohomology, a *curvature* form $\omega \in \Lambda^{k+1}_{\mathrm{cl}}(X)$, and a *monodromy* term $h \in C^k(X, \mathbb{R})$ satisfying the equation $dh = \omega - c$.

We will follow their lead, at least in spirit. In true homotopy type theoretic fashion, we will define the classifying types first and then derive the cohomology theory by truncation.

Definition 3.2.1. We define the classifier $\mathbf{B}^k_\nabla U(1)$ of degree $(k+1)$ classes in ordinary differential cohomology to be the pullback:

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r, "F_-"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d, "(-)^{\int}"] \\
\mathbf{B}^{k+1}\,\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+1}\,\mathbb{R}
\end{tikzcd}$$

Therefore, a cocycle $\tilde{c} : X \to \mathbf{B}^k_\nabla U(1)$ in differential cohomology will consist of an underlying cocycle $c : X \to \mathbf{B}^{k+1}\,\mathbb{Z}$, a curvature form $\omega : X \to \Lambda^{k+1}_{\mathrm{cl}}$, together with an identification $h : c = \omega$ in $X \to \flat\mathbf{B}^{k+1}\,\mathbb{R}$. Since h lands in types identifiable with $\Omega\flat\mathbf{B}^{k+1}\,\mathbb{R}$, which equals $\flat\mathbf{B}^k\,\mathbb{R}$, we may consider it as the monodromy term in discrete real cohomology. We will now set about justifying this terminology.

We may note immediately from this definition that the map $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^{k+1}\,\mathbb{Z}$ (which we may think of as taking the underlying class in ordinary cohomology) is the $\int$-unit. This means that the underlying cocycle is the universal discrete cohomological invariant of a differential cocycle.

Lemma 3.2.2. The pullback square

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d, "(-)^{\int}"] \\
\mathbf{B}^{k+1}\,\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+1}\,\mathbb{R}
\end{tikzcd}$$

is a $\int$-naturality square. That is, $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^{k+1}\,\mathbb{Z}$ is a $\int$-unit.

Proof. Since $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^{k+1}\mathbb{Z}$ is a map into a $\int$-modal type, to show that it is a $\int$ unit it suffices to show that it is $\int$-connected. Since we have a pullback square, the fibers of $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^{k+1}\mathbb{Z}$ are the same as those of $(-)^{\int} : \Lambda^{k+1}_{\mathrm{cl}} \to \flat\mathbf{B}^{k+1}\mathbb{R}$. But as this map is a $\int$-unit, its fibers are $\int$-connected. □

The reason for our change of index — defining $\mathbf{B}^k_\nabla U(1)$ to represent degree $(k+1)$ classes — is because we would like to think of $\mathbf{B}^k_\nabla U(1)$ as more directly classifying connections on k-gerbes with band $U(1)$. To reify this idea, let's give the map $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^k U(1)$ which we think of as taking the underlying k-gerbe.

Construction 3.2.3. We construct a map $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^k U(1)$ which makes the following triangle commute:

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[rr] \arrow[dr, "(-)^{\int}"'] & & \mathbf{B}^k U(1) \arrow[dl, "(-)^{\int}"] \\
& \mathbf{B}^{k+1}\mathbb{Z} &
\end{tikzcd}$$

Construction. Since $\mathbb{R} \to U(1)$ is the universal ∞-cover of $U(1)$, by Corollary 2.3.13, $U(1)$ has the same infinitesimal remainder as $\mathbb{R}$, which is Λ^1_{cl}. Therefore, by modal fracture Theorem 2.4.2, we have a pullback square

$$\begin{tikzcd}
\mathbf{B}^k U(1) \arrow[r] \arrow[d, "(-)^{\int}"'] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^k \Lambda^1_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^{k+1}\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+1}\mathbb{R}
\end{tikzcd}$$

Now, since we have a series of short exact sequences

$$0 \to \Lambda^n_{\mathrm{cl}} \to \Lambda^n \xrightarrow{d} \Lambda^{n+1}_{\mathrm{cl}} \to 0$$

we have long fiber sequences

$$\begin{tikzcd}
\Lambda^n_{\mathrm{cl}} \arrow[r] & \Lambda^n \arrow[r, "d"] & \Lambda^{n+1}_{\mathrm{cl}} \arrow[dll, rounded corners] \\
\mathbf{B}\Lambda^n_{\mathrm{cl}} \arrow[r] & \mathbf{B}\Lambda^n \arrow[r] & \cdots
\end{tikzcd}$$

for each n. In particular, we have maps $\mathbf{B}^n\Lambda^{m+1}_{\mathrm{cl}} \to \mathbf{B}^{n+1}\Lambda^m_{\mathrm{cl}}$ for all n and m. Taking repeated pullbacks along these maps gives us a diagram

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[dddd, dashed, bend right] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\bullet \arrow[r] \arrow[d] & \mathbf{B}\Lambda^k_{\mathrm{cl}} \arrow[d] \\
\vdots \arrow[d] & \vdots \arrow[d] \\
\bullet \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^{k-1}\Lambda^2_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^k U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^k\Lambda^1_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^{k+1}\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+1}\mathbb{R}
\end{tikzcd} \tag{3.2.4}$$

The dashed composite in this diagram is what we were seeking to construct. □

Remark 3.2.5. Diagram 3.2.4 shows us that the following square is a pullback:

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d, "(-)^\int"] \\
\mathbf{B}^k U(1) \arrow[r] & \mathbf{B}^k \Lambda^1_{\mathrm{cl}}
\end{tikzcd}$$

If we note that $\mathbf{B}^k \Lambda^1_{\mathrm{cl}}$ is $\mathbf{B}^k \mathbb{u}(1)$, we get an alternate definition of $\mathbf{B}^k_\nabla U(1)$ by this pullback. This shows that our definition agrees with Schreiber's Definition 4.4.93 in [20].

We can now see that the map $\mathbf{B}^k_\nabla U(1) \to \Lambda^{k+1}_{\mathrm{cl}}$ takes the the *curvature* $(k+1)$*-form.* We can justify this by showing that the fiber of this map is $\flat \mathbf{B}^k U(1)$; in other words, a circle k-gerbe with connection is flat if and only if its curvature vanishes.

Lemma 3.2.6. The map $F_{(-)} : \mathbf{B}^k_\nabla U(1) \to \Lambda^{k+1}_{\mathrm{cl}}$ has fiber $\flat \mathbf{B}^k U(1)$. Since this map gives an obstruction to flatness, we refer to it as the *curvature* $(k+1)$-form.

Proof. By considering the top part from the diagram (3.2.4), we find a pullback square

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^k U(1) \arrow[r] & \mathbf{B}^k \Lambda^1_{\mathrm{cl}}
\end{tikzcd}$$

For this reason, we get an equivalence on fibers:

$$\begin{tikzcd}
\bullet \arrow[r] \arrow[d, "\wr"'] & \mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\flat \mathbf{B}^k U(1) \arrow[r] & \mathbf{B}^k U(1) \arrow[r] & \mathbf{B}^k \Lambda^1_{\mathrm{cl}}
\end{tikzcd}$$

□

As a corollary, we may characterize the curvature $F_{(-)} : \mathbf{B}^k_\nabla U(1) \to \Lambda^{k+1}_{\mathrm{cl}}$ modally.

Corollary 3.2.7. The curvature $F_{(-)} : \mathbf{B}^k_\nabla U(1) \to \Lambda^{k+1}_{\mathrm{cl}}$ is a unit for the $(k-1)$-truncation modality. In particular,

$$\left\| \mathbf{B}^k_\nabla U(1) \right\|_j = \Lambda^{k+1}_{\mathrm{cl}}$$

for any $0 \leq j < k$.

Proof. As $\Lambda^{k+1}_{\mathrm{cl}}$ is 0-truncated and so $(k-1)$ truncated, it will suffice to show that $F_{(-)}$ is $(k-1)$-connected. But by Lemma 3.2.6, the fiber of $F_{(-)}$ over any point $\omega : \Lambda^{k+1}_{\mathrm{cl}}$ is identifiable with $\flat B^k U(1)$, which is $(k-1)$-connected. □

Though our notation may have suggested that the $\mathbf{B}^k_\nabla U(1)$ form a loop spectrum, they do not. Indeed, $\Omega \mathbf{B}^k_\nabla U(1) = \flat \mathbf{B}^{k-1} U(1)$, as can be seen by taking loops of the pullback square defining $\mathbf{B}^k_\nabla U(1)$ and noting that $\Lambda^{n+1}_{\mathrm{cl}}$ is a set (0-type). In total,

$$\pi_* \mathbf{B}^k_\nabla U(1) = \begin{cases} \Lambda^{k+1}_{\mathrm{cl}} & \text{if } * = 0 \\ \flat U(1) & \text{if } * = k \\ 0 & \text{otherwise.} \end{cases}$$

Nevertheless, each $\mathbf{B}^k_\nabla U(1)$ is an infinite loop space in its own right.

Definition 3.2.8. For n, $k \geq 0$, define $\mathbf{B}^n\mathbf{B}^k_\nabla U(1)$ to be the following pullback:

$$\begin{tikzcd}
\mathbf{B}^n\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^{n+k+1}\,\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{n+k+1}\,\mathbb{R}
\end{tikzcd}$$

It is immediate from this definition and the commutation of taking loops with taking pullbacks that $\Omega\mathbf{B}^{n+1}\mathbf{B}^k_\nabla U(1) = \mathbf{B}^n\mathbf{B}^k_\nabla U(1)$. We have already seen these deloopings before in Diagram (3.2.4):

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}\mathbf{B}^{k-1}_\nabla U(1) \arrow[r] \arrow[d] & \mathbf{B}\Lambda^{k}_{\mathrm{cl}} \arrow[d] \\
\vdots \arrow[d] & \vdots \arrow[d] \\
\mathbf{B}^{k-1}\mathbf{B}_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^{k-1}\Lambda^{2}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^k U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^k\Lambda^{1}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^{k+1}\,\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+1}\,\mathbb{R}
\end{tikzcd}$$

These maps along the left hand side give us maps of loop spectra

$$\mathbf{B}^\bullet\mathbf{B}^k_\nabla U(1) \to \Sigma\mathbf{B}^\bullet\mathbf{B}^{k-1}_\nabla U(1)$$

We will see in Section 3.3 that for $\bullet = 0$, these maps give obstructions to de Rham's theorem for general types.

Each $\mathbf{B}^k_\nabla U(1)$ is a higher group itself. We may therefore ask: what is it's infinitesimal remainder?

Lemma 3.2.9. The infinitesimal remainder of $\mathbf{B}^k_\nabla U(1)$ is the curvature $F_{(-)} : \mathbf{B}^k_\nabla U(1) \to \Lambda^{k+1}_{\mathrm{cl}}$.

Proof. Consider the following diagram:

$$\begin{tikzcd}
\flat_{\mathrm{dR}}\mathbf{B}\mathbf{B}^k_\nabla U(1) \arrow[r, "\sim"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[drr] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \arrow[drr] & & & & \\
* \arrow[r] \arrow[drr] & * \arrow[drr] & \flat\mathbf{B}\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[drr] & \flat\mathbf{B}\Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \arrow[drr] & & \\
& & \mathbf{B}^{k+2}\,\mathbb{Z} \arrow[r] \arrow[drr, equal] & \flat\mathbf{B}^{k+2}\,\mathbb{R} \arrow[drr, equal] & \mathbf{B}\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}\Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
& & & & \mathbf{B}^{k+2}\,\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+2}\,\mathbb{R}
\end{tikzcd}$$

The diagonal sequences in this diagram are fiber sequences of the $\flat$-counits which define the infinitesimal remainders. Now, since $\flat\mathbf{B}\Lambda^{k+1}_{\mathrm{cl}} = *$ since the form classifiers are infinitesimal, we find that the fiber of the $\flat$-counit $(-)_\flat : \flat\mathbf{B}\Lambda^{k+1}_{\mathrm{cl}} \to \mathbf{B}\Lambda^{k+1}_{\mathrm{cl}}$ is $\Omega\mathbf{B}\Lambda^{k+1}_{\mathrm{cl}}$, which is $\Lambda^{k+1}_{\mathrm{cl}}$.

Now, the frontmost square is a crisp pullback and $\flat$ is left exact, so the middle square is also a pullback. Then, since the diagonal sequences are fiber sequences, the back square is also a pullback. But this shows that the infinitesimal remainder of $\mathbf{B}^k_\nabla U(1)$ is $\Lambda^{k+1}_{\mathrm{cl}}$.

If we take one more fiber, we can continue the diagram to give us the following diagram:

$$\begin{tikzcd}
\mathbf{B}^k_\nabla U(1) \arrow[r, "F_{(-)}"] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] \arrow[drr, "\theta"] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \arrow[drr, equal] & & & & \\
\mathbf{B}^{k+1}\mathbb{Z} \arrow[r] \arrow[drr] & \flat\mathbf{B}^{k+1}\mathbb{R} \arrow[drr] & \flat_{\mathrm{dR}}\mathbf{B}\mathbf{B}^k_\nabla U(1) \arrow[r, "\sim"] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] \arrow[drr] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \arrow[drr] & & \\
 & & * \arrow[r] \arrow[drr] & * \arrow[drr] & \flat\mathbf{B}\mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \flat\mathbf{B}\Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
 & & & & \mathbf{B}^{k+2}\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+2}\mathbb{R}
\end{tikzcd}$$

This shows that the infinitesimal remainder θ is equal, modulo our constructed equivalence, to the curvature $F_{(-)}$. □

Now that we know the infinitesimal remainder of $\mathbf{B}^k_\nabla U(1)$, we are almost ready to understand its modal fracture hexagon. But first, we must understand its universal ∞-cover. We will show that the universal ∞-cover of $\mathbf{B}^k_\nabla U(1)$ is an analogous type $\mathbf{B}^k_\nabla \mathbb{R}$.

Definition 3.2.10. For $n, k \geq 0$, define $\mathbf{B}^n\mathbf{B}^k_\nabla \mathbb{R}$ to be the universal ∞-cover of $\mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}}$:

$$\begin{tikzcd}
\mathbf{B}^n\mathbf{B}^k_\nabla \mathbb{R} \arrow[r, "F_{(-)}"] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
* \arrow[r] & \flat\mathbf{B}^{n+k+1}\mathbb{R}
\end{tikzcd}$$

We refer to the cohomology theories $\mathbf{B}^k_\nabla \mathbb{R}$ as *pure differential cohomology.*

Just as we may think of $\mathbf{B}^k_\nabla U(1)$ as classifying circle k-gerbes with connection, we may think of $\mathbf{B}^k_\nabla \mathbb{R}$ as classifying affine k-gerbes with connection. We can now show that $\mathbf{B}^k_\nabla \mathbb{R}$ is the universal ∞-cover of $\mathbf{B}^k_\nabla U(1)$.

Proposition 3.2.11. The map $(\omega, p) \mapsto (\mathsf{pt}_{\mathbf{B}^{k+1}\mathbb{Z}}, \omega, \lambda\_.\, p) : \mathbf{B}^k_\nabla \mathbb{R} \to \mathbf{B}^k_\nabla U(1)$ is the universal ∞-cover of $\mathbf{B}^k_\nabla U(1)$.

Proof. Consider the following cube:

$$\begin{tikzcd}
\mathbf{B}^k_\nabla \mathbb{R} \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] \arrow[drr, dashed] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \arrow[drr, equal] & & \\
* \arrow[r] \arrow[drr] & \flat\mathbf{B}^{k+1}\mathbb{R} \arrow[drr, equal] & \mathbf{B}^k_\nabla U(1) \arrow[r] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
 & & \mathbf{B}^{k+1}\mathbb{Z} \arrow[r] & \flat\mathbf{B}^{k+1}\mathbb{R}
\end{tikzcd}$$

In this cube, the front and back spaces are pullbacks by definition, and the right face is a pullback because its top and bottom sides are identities. Therefore, the left face is a pullback. Since $\mathbf{B}^k_\nabla U(1) \to \mathbf{B}^{k+1}\mathbb{Z}$ is a $\int$-unit by Lemma 3.2.2, this shows that the dashed map is the fiber of a $\int$-unit, and therefore the universal ∞-cover. □

Remark 3.2.12. The fiber sequence

$$\mathbf{B}^k_\nabla \mathbb{R} \to \mathbf{B}^k_\nabla U(1) \to \mathbf{B}^{k+1}\mathbb{Z}$$

expresses the informal identity

ordinary differential cohomology = pure differential cohomology + ordinary cohomology.

We are now ready to assemble what we have learned into the modal facture hexagon of $\mathbf{B}^k_\nabla U(1)$:

$$\begin{tikzcd}
& \mathbf{B}^k_\nabla \mathbb{R} \arrow[rr] \arrow[dr, "\pi"] & & \Lambda^{k+1}_{\mathrm{cl}} \arrow[dr, "(-)^{\int}"] & \\
\flat\mathbf{B}^k\mathbb{R} \arrow[ur, "(-)_\flat"] \arrow[dr] & & \mathbf{B}^k_\nabla U(1) \arrow[ur, "F_{(-)}"] \arrow[dr, "(-)^{\int}"'] & & \flat\mathbf{B}^{k+1}\mathbb{R} \\
& \flat\mathbf{B}^k U(1) \arrow[ur, "(-)_\flat"'] \arrow[rr, "\beta"'] & & \mathbf{B}^{k+1}\mathbb{Z} \arrow[ur] &
\end{tikzcd} \tag{3.2.13}$$

3.3 Descending to Cohomology and the Character Diagram

In this section, we will discuss how the modal fracture hexagon (3.2.13) descends to cohomology. In general, if $E_\bullet$ is a loop spectrum, then the we may define the cohomology groups of a type valued in $E_\bullet$ to be the 0-truncated types of maps:

$$H^k(X; E_\bullet) :\equiv \|X \to E_k\|_0 .$$

However, these abelian groups are not discrete — externally, they are (possible non-constant) sheaves of abelian groups. We will want the discrete (externally, constant) invariants. With this in mind, we make the following definitions.

Definition 3.3.1. Let X be a crisp type. We then make the following definitions:

$$\begin{aligned}
H^n(X;\mathbb{Z}) &:\equiv \|\flat(X \to \mathbf{B}^n\mathbb{Z})\|_0 \\
H^n(X;\flat\,\mathbb{R}) &:\equiv \|\flat(X \to \flat\mathbf{B}^n\mathbb{R})\|_0 \\
H^n(X;\flat U(1)) &:\equiv \|\flat(X \to \flat\mathbf{B}^n U(1))\|_0 \\
H^{n,k}_\nabla(X;U(1)) &:\equiv \left\|\flat(X \to \mathbf{B}^n\mathbf{B}^k_\nabla U(1))\right\|_0 \\
H^{n,k}_\nabla(X;\mathbb{R}) &:\equiv \left\|\flat(X \to \mathbf{B}^n\mathbf{B}^k\mathbb{R})\right\|_0 \\
\Lambda^k(X) &:\equiv \flat(X \to \Lambda^k) \\
\Lambda^k_{\mathrm{cl}}(X) &:\equiv \flat(X \to \Lambda^k_{\mathrm{cl}}).
\end{aligned}$$

Remark 3.3.2. In full cohesion, it would be better to work with codiscrete cohomology groups, rather than discrete cohomology groups. This way the definition could be given for all types and not just crisp ones. But we will continue to use discrete groups so that we do not need to work with the codiscrete modality $\sharp$ in this paper.

We note that with these definitions we may reduce the calculation of ordinary differential cohomology for discrete and homotopically contractible types.

Proposition 3.3.3. Let X be a crisp type and let $k \geq 1$.

1. If X is discrete (that is, $X = \int X$), then $H^{n,k}_\nabla(X;U(1)) = H^{n+k}(X;\flat U(1))$.

2. If X is homotopically contractible (that is, $\int X = *$), then $H^{n,k}_\nabla(X;U(1)) = H^n(X;\Lambda^{k+1}_{\mathrm{cl}})$.

We may make similar calculations for pure differential cohomology:

1. If X is discrete, then $H^{n,k}(X;\mathbb{R}) = H^{n+k}_{\nabla}(X;\flat\,\mathbb{R})$.
2. If X is homotopically contractible, then $H^{n,k}_{\nabla}(X;\mathbb{R}) = H^{n}(X;\Lambda^{k+1}_{\mathrm{cl}})$.

Proof. We will only prove the identities for ordinary differential cohomology; the proofs for pure differential cohomology are identical. We take advantage of the adjointness between $\int$ and $\flat$.

1. Suppose that X is discrete. Then

$$\begin{aligned}
H^{n,k}_{\nabla}(X;U(1)) &= \left\|\flat(X \to \mathbf{B}^n\mathbf{B}^k_{\nabla}U(1))\right\|_0 \\
&= \left\|\flat(\int X \to \mathbf{B}^n\mathbf{B}^k_{\nabla}U(1))\right\|_0 \\
&= \left\|\flat(X \to \flat\mathbf{B}^n\mathbf{B}^k_{\nabla}U(1))\right\|_0 \\
&= \left\|\flat(X \to \flat\mathbf{B}^{n+k}U(1))\right\|_0 \\
&= H^{n+k}(X;\flat U(1))
\end{aligned}$$

2. Suppose that X is homotopically contractible, and let $i : \mathbf{B}^{n+k+1}\,\mathbb{Z} \to \flat\mathbf{B}^{n+k+1}\,\mathbb{R}$ denote the (delooping of) the inclusion. Then

$$\begin{aligned}
H^{n,k}_{\nabla}(X;U(1)) &= \left\|\flat(X \to \mathbf{B}^n\mathbf{B}^k_{\nabla}U(1))\right\| \\
&= \flat\left\|\begin{aligned}&(\omega : X \to \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}}) \times (c : X \to \mathbf{B}^{n+k+1}\,\mathbb{Z}) \\ &\qquad\qquad \times (h : ic = \omega^{\int})\end{aligned}\right\|_0 \\
&= \flat\left\|\begin{aligned}&(\omega : X \to \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}}) \times (c : \mathbf{B}^{n+k+1}\,\mathbb{Z}) \\ &\qquad\qquad \times (h : (x : X) \to ic = \omega(x)^{\int})\end{aligned}\right\|_0
\end{aligned}$$

Since X is homotopically contractible, we have an equivalence $e : (X \to \flat\mathbf{B}^{n+k+1}\,\mathbb{R}) \simeq \flat\mathbf{B}^{n+k+1}$. We therefore have $e((-)^{\int}\circ\omega) : \flat\mathbf{B}^{n+k+1}$ and for all $x : X$ a witness $\omega(x)^{\int} = e((-)^{\int}\circ\omega)$. We may therefore continue:

$$\begin{aligned}
&= \flat\left\|\begin{aligned}&(\omega : X \to \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}}) \times (c : \mathbf{B}^{n+k+1}\,\mathbb{Z}) \\ &\qquad\qquad \times (h : (x : X) \to ic = e((-)^{\int}\circ\omega))\end{aligned}\right\|_0 \\
&= \flat\left\|\begin{aligned}&(\omega : X \to \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}}) \times (c : \mathbf{B}^{n+k+1}\,\mathbb{Z}) \\ &\qquad\qquad \times (ic = e((-)^{\int}\circ\omega))\end{aligned}\right\|_0
\end{aligned}$$

Now, both $\mathbf{B}^{n+k+1}\,\mathbb{Z}$ and $(ic = e((-)^{\int}\circ\omega))$ are 0-connected. The latter because it is identifiable with $\Omega\flat\mathbf{B}^{n+k+1}\,\mathbb{R}$, which is $\flat\mathbf{B}^{n+k}\,\mathbb{R}$ and so 0-connected for $k \geq 1$ and any n. We may therefore continue:

$$\begin{aligned}
&= \flat\left\|X \to \mathbf{B}^n\Lambda^{k+1}_{\mathrm{cl}}\right\| \\
&= H^n(X;\Lambda^{k+1}_{\mathrm{cl}}). \qquad \square
\end{aligned}$$

Remark 3.3.4. We note here that since every type X lives in the center of a fiber sequence

$$\overset{\infty}{X} \to X \to \int X$$

between a homotopically contractible type and a discrete type, we get a Serre spectral sequence converging to the k^{th} ordinary differential cohomology of X with E_2 page depending on it's $\flat U(1)$ cohomology and the $\Lambda^{k+1}_{\mathrm{cl}}$ valued cohomology of it's universal ∞-cover.

Now, since the top, bottom, and diagonal sequences in the modal fracture hexagon (3.2.13) of $\mathbf{B}^k_\nabla U(1)$ are fiber sequences, when we take $\flat$ and 0-truncations we will get long exact sequences. With the above definitions, we get the following diagram:

$$\begin{tikzcd}
 & H^{0,k}_\nabla(X;\mathbb{R}) \arrow[rr] \arrow[dr] & & \Lambda^{k+1}_{\mathrm{cl}}(X) \arrow[dr] & \\
H^k(X;\flat\,\mathbb{R}) \arrow[ur] \arrow[dr] & & H^{0,k}_\nabla(X;U(1)) \arrow[ur] \arrow[dr] & & H^{k+1}(X;\flat\,\mathbb{R}) \\
 & H^k(X;\flat U(1)) \arrow[rr, "\beta"'] \arrow[ur] & & H^{k+1}(X;\mathbb{Z}) \arrow[ur] &
\end{tikzcd} \tag{3.3.5}$$

in which the top and bottom sequences are long exact, and the diagonal sequences are exact in the middle. This looks very much like the character diagram for ordinary differential cohomology [23] except for two differences:

1. Where we have the pure cohomology $H^{0,k}_\nabla(X;\mathbb{R})$, one would normally find $\Lambda^k(X)/\operatorname{\mathsf{im}}(d)$, the abelian group which fits into an exact sequence
$$\Lambda^{k-1}(X) \xrightarrow{d} \Lambda^k(X) \to \Lambda^k(X)/\operatorname{\mathsf{im}}(d) \to 0.$$

2. Where we have $H^{k+1}(X;\flat\,\mathbb{R})$, which is ordinary (discrete) cohomology with real coefficients, one would normally find the de Rham cohomology $H^{k+1}_{\mathrm{dR}}(X)$. The de Rham cohomology is defined as closed forms mod exact forms, and so $H^{k+1}_{\mathrm{dR}}(X)$ is the abelian group fitting the following exact sequence:
$$\Lambda^k(X) \xrightarrow{d} \Lambda^{k+1}_{\mathrm{cl}}(X) \to H^{k+1}_{\mathrm{dR}}(X) \to 0.$$

Both of these discrepancies are instances of de Rham's theorem that the de Rham cohomology of forms is the (discrete) ordinary cohomology with real coefficients. Classically and externally, this holds for smooth manifolds. We note that de Rham's theorem cannot hold for all types for rather trivial reasons: the form classifiers are sets, and so $\Lambda^k(X)$ depends only on the set tranctation of X whereas $H^k(X;\flat\,\mathbb{R})$ can depend on the k-truncation of X.

Proposition 3.3.6. The de Rham theorem does not hold for the delooping $\flat\mathbf{B}\,\mathbb{R}$ of the discrete additive group of real numbers. Explicitly,

$$H^1_{\mathrm{dR}}(\flat\mathbf{B}\,\mathbb{R}) = 0$$
$$H^1(\flat\mathbf{B}\,\mathbb{R};\flat\,\mathbb{R}) = \flat\mathrm{Hom}(\flat\,\mathbb{R},\flat\,\mathbb{R}) \neq 0$$

Proof. Since $\flat\mathbf{B}\,\mathbb{R}$ is 0-connected and the form classifiers are sets, every map $\flat\mathbf{B}\,\mathbb{R} \to \Lambda^k$ is constant for all k. Therefore,
$$H^1_{\mathrm{dR}}(\flat\mathbf{B}\,\mathbb{R}) = \Lambda^1_{\mathrm{cl}}(\flat\mathbf{B}\,\mathbb{R})/\Lambda^0(\flat\mathbf{B}\,\mathbb{R}) = 0$$
On the other hand, $H^1(\flat\mathbf{B}\,\mathbb{R};\flat\,\mathbb{R}) = \|\flat(\flat\mathbf{B}\,\mathbb{R} \to \flat\mathbf{B}\,\mathbb{R})\|_0$ is the set of group homomorphisms from $\flat\,\mathbb{R}$ to itself (modulo conjugacy, which makes no difference). The identity is not conjugate to 0, and so this group is not trivial. □

We can, however, make explicit the obstruction to de Rham's theorem lying in the first (cohomological) degree pure differential cohomology groups $H^{1,k}_\nabla(X;\mathbb{R})$. We begin first by trying to construct an exact sequence
$$\Lambda^k(X) \xrightarrow{d} \Lambda^{k+1}(X) \to H^{0,k}_\nabla(X;\mathbb{R}) \to 0.$$

Recall Diagram 3.2.4. There is a similar diagram for pure differential cohomology:

$$\begin{tikzcd}
\mathbf{B}^k_\nabla \mathbb{R} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}\mathbf{B}^{k-1}_\nabla \mathbb{R} \arrow[r] \arrow[d] & \mathbf{B}\Lambda^k_{\mathrm{cl}} \arrow[d] \\
\vdots \arrow[d] & \vdots \arrow[d] \\
\mathbf{B}^{k-1}\mathbf{B}_\nabla \mathbb{R} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^{k-1}\Lambda^2_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}^k \mathbb{R} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^k\Lambda^1_{\mathrm{cl}} \arrow[d] \\
* \arrow[r] & \flat \mathbf{B}^{k+1} \mathbb{R}
\end{tikzcd}$$

If we focus at the top, we see that we have a pullback square which induces an equivalence on fibers:

$$\begin{tikzcd}
\bullet \arrow[r, "\sim"] \arrow[d] & \Lambda^k \arrow[d, "d"] \\
\mathbf{B}^k_\nabla \mathbb{R} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \Lambda^{k+1}_{\mathrm{cl}} \arrow[d] \\
\mathbf{B}\mathbf{B}^{k-1}_\nabla \mathbb{R} \arrow[r] & \mathbf{B}\Lambda^k_{\mathrm{cl}}
\end{tikzcd} \tag{3.3.7}$$

This gives us a fiber sequence

$$\Lambda^k \to \mathbf{B}^k_\nabla \mathbb{R} \to \mathbf{B}\mathbf{B}^{k-1}_\nabla \mathbb{R},$$

which we may deloop as much as we like. Noting that $\Omega \mathbf{B}^k_\nabla \mathbb{R} = \flat \mathbf{B}^{k-1} \mathbb{R}$, we therefore have a long exact sequence:

$$0 \to H^{k-1}(X; \flat \mathbb{R}) \to H^{0,k-1}_\nabla(X; \mathbb{R}) \to \Lambda^k(X) \to H^{0,k}_\nabla(X; \mathbb{R}) \to H^{1,k-1}(X; \mathbb{R}) \to \cdots$$

From this, we see that the surjectivity of the map $\Lambda^k(X) \to H^{0,k}(X; \mathbb{R})$ is determined by the vanishing of the map $H^{0,k}_\nabla(X; \mathbb{R}) \to H^{1,k-1}(X; \mathbb{R})$. Furthermore, the version of Diagram 3.3.7 for $k-1$ shows us that $d : \Lambda^{k-1}(X) \to \Lambda^k(X)$ factors through $H^{0,k}_\nabla(X; \mathbb{R})$. This means that for the kernel of $\Lambda^k(X) \to H^{0,k}_\nabla(X; \mathbb{R})$ to be the image of $d : \Lambda^{k-1}(X) \to \Lambda^k(X)$, we need for $\Lambda^{k-1}(X) \to H^{0,k-1}_\nabla(X; \mathbb{R})$ to be surjective; this is controlled by the vanishing of $H^{0,k-1}_\nabla(X; \mathbb{R}) \to H^{1,k-2}_\nabla(X; \mathbb{R})$. In general, we see the obstructions to having exact sequences

$$\Lambda^{k-1}(X) \to \Lambda^k(X) \to H^{0,k}_\nabla(X; \mathbb{R})$$

lie in $H^{1,k-1}_\nabla(X; \mathbb{R})$ and $H^{1,k-2}_\nabla(X; \mathbb{R})$.

First cohomological degree pure differential cohomology groups also control obstructions to de Rham's theorem for general types X. By definition we have a fiber sequence $\mathbf{B}^k_\nabla \mathbb{R} \to \Lambda^{k+1}_{\mathrm{cl}} \to \flat \mathbf{B}^{k+1} \mathbb{R}$ which may be delooped arbitrarily. We therefore get exact sequences

$$0 \to H^k(X; \flat \mathbb{R}) \to H^{0,k}_\nabla(X; \mathbb{R}) \to \Lambda^{k+1}_{\mathrm{cl}}(X) \to H^{k+1}(X; \flat \mathbb{R}) \to H^{1,k}_\nabla(X; \mathbb{R}) \cdots$$

This exact sequence shows us that the surjectivity of the map $\Lambda^{k+1}_{\mathrm{cl}}(X) \to H^{k+1}(X; \flat \mathbb{R})$ is controlled by the vanishing of the map $H^{k+1}(X; \flat \mathbb{R}) \to H^{1,k}_\nabla(X; \mathbb{R})$. Furthermore, in order for the kernel of $\Lambda^{k+1}_{\mathrm{cl}}(X) \to H^{k+1}(X; \flat \mathbb{R})$ to be $d : \Lambda^k(X) \to \Lambda^{k+1}_{\mathrm{cl}}$, we need for $\Lambda^k(X) \to H^{0,k}_\nabla(X; \mathbb{R})$ to be surjective. As we saw above, for the map $\Lambda^k(X) \to H^{0,k}_\nabla(X; \mathbb{R})$ to be surjective, we must have that $H^{0,k}_\nabla(X; \mathbb{R}) \to H^{1,k-1}_\nabla(X; \mathbb{R})$ vanishes.

Remembering the classical, external differential cohomology hexagon, we are led to the following conjecture:

Conjecture 3.3.8. Let X be a crisp smooth manifold. Then $H_{\nabla}^{1,k}(X;\mathbb{R})$ vanishes for all k.

3.4 Abstract Ordinary Differential Cohomology

In the above sections, we constructed ordinary differential cohomology from the assumption of a long exact sequence of form classifiers. Apart from the concrete differential geometric input of the form classifiers, the construction was entirely abstract. In this section, we will describe the abstract ordinary differential cohomology theory from an axiomatic perspective.

The role of the form classifiers will be played by a *contractible and infinitesimal resolution* of a crisp abelian group C.

Definition 3.4.1. Let C be a crisp abelian group. A *contractible and infinitesimal resolution* (CIR) of C is a crisp long exact sequence

$$0 \to \flat C \to C \xrightarrow{d} C_1 \xrightarrow{d} C_2 \xrightarrow{d} \cdots$$

where the C_n are homotopically contractible — $\int C_n = *$ — and where the kernels $Z_n :\equiv \mathsf{ker}(d : C_n \to C_{n+1})$ are infinitesimal — $\flat Z_n = *$. We may think of C_n as the abelian group of n-cochains, and Z_n as the abelian group of n-cocycles.

Remark 3.4.2. In an ∞-topos of sheaves of homotopy types, an abelian group C in the empty context (which would therefore be crisp) is a sheaf of abelian groups. In this setting, we can understand a contractible and infinitesimal resolution of C as presenting a cohomology theory on the site. The C_n are the sheaves of n-cochains, and the Z_n the sheaves of n-cocycles. To suppose that the chain complex $d : C_n \to C_{n+1}$ is exact is to say that representables have vanishing cohomology. To say that Z_n is infinitesimal for $n > 0$ is to say that there is a unique n-cocycle on the terminal sheaf, namely 0. To say that the C_n are contractible may be understood as saying that for any two objects of the site, there is a homotopically unique *concordance* between any n-cochains on them.

Remark 3.4.3. It's likely that the generality could be pushed even further by taking C to be a spectrum and giving the following definition of a contractible and infinitesimal resolution of C:

- Two sequences C_n and Z_n of spectra, $n \geq 0$, with $C_0 = C$ and $Z_0 = \flat C$. We may think of C_n as the spectrum of n-cochains, and Z_n as the spectrum of n-cocyles.
- Fiber sequences $Z_n \xrightarrow{i} C_n \xrightarrow{d} Z_{n+1}$ in which all maps d are $\int$-fibrations, and where $i_0 : Z_0 \to C_0$ is $(-)_\flat : \flat C \to C$.
- The C_n are contractible, and the Z_n are infinitesimal.

This definition re-expresses the long exact sequence $C_{n-1} \xrightarrow{d} C_n \xrightarrow{d} C_{n+1}$ in terms of the short exact sequences

$$0 \to Z_n \to C_n \xrightarrow{d} Z_{n+1} \to 0$$

where $Z_n \equiv \mathsf{ker}(d : C_n \to C_{n+1})$. As we have no concrete examples at this level of generality in mind, we leave the details of this generalization to future work.

For the rest of this section, we fix a crisp abelian group C and an contractible and infinitesimal resolution of it. We can then prove analogues of the lemmas in the above sections. We begin by an analogue of Theorem 3.1.6.

Lemma 3.4.4. Let C be a crisp abelian group and $C_\bullet$ a contractible and infinitesimal resolution of C. Then $d : C \to Z_1$ is the infinitesimal remainder of C.

Proof. By hypothesis, we have a short exact sequence

$$0 \to \flat C \to C \xrightarrow{d} Z_1 \to 0.$$

We therefore have a long fiber sequence

$$C \xrightarrow{d} Z_1 \to \flat \mathbf{B} C \to \mathbf{B} C$$

which exhibits $d : C \to Z_1$ as the infinitesimal remainder of C. □

Theorem 3.4.5. *For C a crisp abelian group and $C_\bullet$ a contractible and infinitesimal resolution of C, we have*

$$\int Z_n = \flat \boldsymbol{B}^n \overset{\infty}{C}.$$

Proof. The same as the proof of Theorem 3.1.6. □

We now define analogues of the ordinary differential geometry classifiers $\mathbf{B}^n \mathbf{B}^k_\nabla U(1)$.

Definition 3.4.6. For $n, k \geq 0$, define $\mathbf{B}^n D_k$ to be the following pullback:

$$\begin{tikzcd}
\mathbf{B}^n D_k \arrow[r, "\mathbf{B}^n F_-"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^n Z_{k+1} \arrow[d, "(-)^{\int}"] \\
\int \mathbf{B}^{n+k} C \arrow[r] & \flat \mathbf{B}^{n+k+1} \overset{\infty}{C}
\end{tikzcd}$$

We refer to $F_{(-)} : D_k \to Z_{k+1}$ as the *curvature.*

We begin by noting that D_0 is simply C. This went without saying before; we refrained from defining $\mathbf{B}^0_\nabla U(1)$, but if we had it would have been $U(1)$.

Lemma 3.4.7. As abelian groups, $D_0 = C$.

Proof. The defining pullback of D_0 is

$$\begin{tikzcd}
D_0 \arrow[r, "F_-"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & Z_1 \arrow[d, "(-)^{\int}"] \\
\int C \arrow[r] & \flat \mathbf{B}^1 \overset{\infty}{C}
\end{tikzcd}$$

But Z_1 is the infinitesimal remainder of C, so the right square in the modal fracture hexagon of C shows that C is the pullback of the same diagram. □

We can prove an analogue of Lemma 3.2.2.

Lemma 3.4.8. The defining diagram

$$\begin{tikzcd}
\mathbf{B}^n D_k \arrow[r, "\mathbf{B}^n F_-"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^n Z_{k+1} \arrow[d, "(-)^{\int}"] \\
\int \mathbf{B}^{n+k} C \arrow[r] & \flat \mathbf{B}^{n+k+1} \overset{\infty}{C}
\end{tikzcd}$$

is a $\int$-naturality square. In particular, $\int D_k = \int \mathbf{B}^k C$.

Proof. Since $\int \mathbf{B}^{n+k} C$ is discrete, it suffices to show that the fibers of $\mathbf{B}^n D_k \to \int B^{n+k} C$ are $\int$-connected. But they are equivalent to the fibers of $(-)^{\int} : \mathbf{B}^n Z_{k+1} \to \flat \mathbf{B}^{n+k+1} \overset{\infty}{C}$, which are $\int$-contractible. □

As a corollary, we can deduce an analogue of Proposition 3.2.11.

Corollary 3.4.9. The curvature $F_{(-)} : D_k \to Z_{k+1}$ induces an equivalence on universal ∞-covers: $\overset{\infty}{D}_k = \overset{\infty}{Z}_{k+1}$.

Proof. The defining pullback

$$\begin{tikzcd}
D_k \arrow[r, "F_-"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & Z_{k+1} \arrow[d, "(-)^{\int}"] \\
\int \mathbf{B}^k C \arrow[r] & \flat \mathbf{B}^{k+1} \overset{\infty}{C}
\end{tikzcd}$$

induces an equivalence on the fibers of the vertical maps. Since these maps are $\int$-units, the fibers are by definition the respective universal ∞-covers. □

As with $\mathbf{B}^k_\nabla U(1)$, we may see D_k as equipping k-gerbes with band C with cocycle data coming from Z_{k+1}. We have the analogue of Diagram 3.2.4:

$$\begin{tikzcd}
D_k \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & Z_{k+1} \arrow[d] \\
\mathbf{B} D_k \arrow[r] \arrow[d] & \mathbf{B} Z_k \arrow[d] \\
\vdots \arrow[d] & \vdots \arrow[d] \\
\mathbf{B}^{k-1} D_1 \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^{k-1} Z_2 \arrow[d] \\
\mathbf{B}^k C \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbf{B}^k Z_1 \arrow[d] \\
\int \mathbf{B}^k C \arrow[r] & \flat \mathbf{B}^{k+1} \mathbb{R}
\end{tikzcd} \tag{3.4.10}$$

By applying $\flat$ to this diagram and recalling that the Z_i (and therefore their deloopings) are infinitesimal, we see that $\flat D_k = \flat \mathbf{B}^k C$. We can use a composite square from this diagram to prove an analogue of Lemma 3.2.6.

Lemma 3.4.11. The fiber of the curvature $F_{(-)} : D_k \to Z_{k+1}$ is $\flat \mathbf{B}^k C$. We are therefore justified in seeing the curvature as an obstruction to the flatness of the underlying gerbe.

Proof. The defining pullback

$$\begin{tikzcd}
D_k \arrow[r, "F_-"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & Z_{k+1} \arrow[d, "(-)^{\int}"] \\
\int \mathbf{B}^k C \arrow[r] & \flat \mathbf{B}^{k+1} \overset{\infty}{C}
\end{tikzcd}$$

induces an equivalence on the fibers of the horizontal maps. But the fiber of $\int \mathbf{B}^k C \to \flat \mathbf{B}^{k+1} \overset{\infty}{C}$ is $\flat \mathbf{B}^k C$. □

Corollary 3.4.12. The curvature $F_{(-)} : D_k \to Z_{k+1}$ is a unit for the $(k-1)$-truncation modality.

Finally, we record an analogue to Lemma 3.2.9.

Lemma 3.4.13. The infinitesimal remainder of D_k is the curvature $F_{(-)} : D_k \to Z_{k+1}$.

Proof. Exactly as Lemma 3.2.9. □

We may put these results together to find the modal fracture hexagon of D_k.

Theorem 3.4.14. *The modal fracture hexagon of D_k (Definition 3.4.6) is:*

$$\begin{tikzcd}
 & \overset{\infty}{Z}_{k+1} \arrow[rr] \arrow[dr, "\pi"] & & Z_{k+1} \arrow[dr, "(-)^{\int}"] & \\
\flat \boldsymbol{B}^k \overset{\infty}{C} \arrow[ur, "(-)_\flat"] \arrow[dr] & & D_k \arrow[ur, "F_{(-)}"] \arrow[dr, "(-)^{\int}"] & & \flat \boldsymbol{B}^{k+1} \overset{\infty}{C} \\
 & \flat \boldsymbol{B}^k C \arrow[rr] \arrow[ur, "(-)_\flat"] & & \textstyle\int \boldsymbol{B}^k C \arrow[ur] &
\end{tikzcd} \tag{3.4.15}$$

3.5 A combinatorial analogue of differential cohomology

Our arguments in the preceeding sections have been abstract and modal in character. This abstraction means that we can apply these arguments in settings other than differential geometry. In this subsection, we will sketch a combinatorial analogue of differential cohomology taking place in the cohesive ∞-topos of symmetric simplicial homotopy types. We will mix internal and external reasoning in sketching the setup. We will give a fuller — and properly internal — exploration of symmetric simplicial cohesion in future work.

A symmetric simplicial homotopy type S is an ∞-functor $X : \mathbf{Fin}_{>0} \to \mathbf{H}$ from the category of non-empty finite sets into the ∞-category of homotopy types. These are the unordered analogue of simplicial homotopy types.

The ∞-topos of symmetric simplicial homotopy types is cohesive. The modalities operate on a symmetric simplicial homotopy type X in the following ways:

- $\flat X$ is the discrete (0-skeletal) inclusion of $X_0 \equiv X([0])$ the homotopy type of 0-simplices in X.
- $\int X$ is the discrete inclusion of the *geometric realization* (or colimit) of X.
- $\sharp X$ is the codiscrete (0-coskeletal) inclusion of X_0.

We have thus far avoided using the codiscrete modality $\sharp$ in this paper, but it plays a crucial role in this section. This is because the n-simplex $\Delta[n]$ may be defined to be the codiscrete reflection of the $(n+1)$-element set $[n] \equiv \{0, \ldots, n\}$.

$$\Delta[n] :\equiv \sharp[n].$$

We may therefore axiomatize symmetric simplical cohesion internally with the following axiom:

Axiom 2 (Symmetric Simplicial Cohesion)**.** A crisp type X is crisply discrete if and only if it is $\sharp[n]$-local for all n.

We may therefore define $\int = \mathrm{Loc}_{\{\sharp[n] \mid n:\mathbb{N}\}}$ to be the localization at the simplices, and the Symmetric Simplicial Cohesion axiom will ensure that $\int$ is adjoint to $\flat$ as required by the Unity of Opposites axiom.

In his paper [12], Lawvere points out that the simplices $\Delta[n]$ are *tiny*, much like the infinitesimal disks in synthetic differential geometry. $\Delta[n]$ being tiny means the functor $(-)^{\Delta[n]}$ admits an *external right* adjoint. We may refer to this adjoint as $(-)^{\frac{1}{\Delta[n]}}$, following Lawvere. If C is a crisp codiscrete abelian group, then the external adjointness shows that maps $X \to C^{\frac{1}{\Delta[n]}}$ correspond to maps $X^{\Delta[n]} \to C$, which, since C is codiscrete, correspond to maps $\flat X^{\Delta_n} \to C$; but $\flat X^{\Delta_n} = X^{\Delta_n}([0]) = X([n])$, so such maps ultimate correspond to maps $X([n]) \to C$ — that is, to C-valued n-cochains on the symmetric simplicial homotopy type X! In total, $C^{\frac{1}{\Delta[n]}}$ classifies n-cochains, much in the way that Λ^n classifies differential n-forms. We note that $C^{\frac{1}{\Delta[n]}}$ inherits the (crisp) algebraic structure of C since $(-)^{\frac{1}{\Delta[n]}}$ is a right adjoint.

If furthermore C is a ring, then the $C^{\frac{1}{\Delta[n]}}$ will be modules and since codiscretes are contractible in this topos (by Theorem 10.2 of [22], noting that it satisfies Shulman's Axiom C2), we see that the $C^{\frac{1}{\Delta[n]}}$ are contractible. We may use the face inclusions $\Delta[n] \to \Delta[n+1]$ to give maps $C^{\frac{1}{\Delta[n]}} \to C^{\frac{1}{\Delta[n+1]}}$, and taking their alternating sum gives us a chain complex

$$C \xrightarrow{d} C^{\frac{1}{\Delta[1]}} \xrightarrow{d} C^{\frac{1}{\Delta[2]}} \xrightarrow{d} \cdots$$

Reasoning externally, we can see that this sequence will be exact since the C-valued cohomology of the n-simplices is trivial. Furthermore, since $\flat C_k = \flat C$ by adjointness ($\flat(* \to C_k) = \flat(*^{\Delta_k} \to C)$), we see that the Z_k are infinitesimal:

$$\flat Z_k = \mathsf{ker}(\flat d : \flat C_k \to \flat C_{k+1}) = *.$$

For this reason, we may make the following assumption in the setting of symetric simplicial cohesion, mirroring Assumption 1 of the existence of form classifiers in synthetic differential cohesion.

Assumption 2. Let C be a codiscrete ring, and define $C_n :\equiv C^{\frac{1}{\Delta[n]}}$. Then the sequence

$$0 \to \flat C \to C \xrightarrow{d} C_1 \xrightarrow{d} C_2 \xrightarrow{d} \cdots$$

forms a contractible and infinitesimal resolution of C.

We can now interpret the abstract language of Section 3.4 into the more concrete language of Assumption 2:

- We have begun with a codiscrete abelian group C. We note that since C is codiscrete, it is homotopically contractible: $\int C = *$. Therefore, $\overset{\infty}{C} = C$.
- The abelian groups C_k are the k-cochain classifiers.
- The kernels $Z_k :\equiv \mathsf{ker}(d : C_k \to C_{k+1})$ classify k-cocycles. Applying Theorem 3.4.5 here shows us that

  $$\int Z_k = \flat B^k C.$$

  From this, we see that cohomology valued in the discrete group $\flat C$ is the universal discrete cohomological invariant of k-cocycles value in C. This justifies a remark of Lawvere in [12] that the Z_k have the homotopy type of the Eilenberg-MacLane space $K(\flat C, k)$.
- Since C is contractible, we have that D_k as defined in Definition 3.4.6 is the universal ∞-cover $\overset{\infty}{Z}_{k+1}$ of Z_{k+1}. We see that D_k classifies $(k+1)$-cocycles together with witnesses that their induced cohomology class vanishes in $\flat \mathbf{B}^{k+1} C$.

The D_k in this setting have more in common with pure differential cohomology $B^k_\nabla \mathbb{R}$ than with ordinary differential cohomology $\mathbf{B}^k_\nabla U(1)$ on account of being contractible. We can remedy this by introducing some new data. Suppose that we have an exact sequence

$$0 \to K \to C \to G \to 0$$

of crisp codiscrete abelian groups. We may then redefine D_k to instead be the following pullback:

$$\begin{tikzcd}
D_k \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & Z_{k+1} \arrow[d] \\
\flat \mathbf{B}^{k+1} K \arrow[r] & \flat \mathbf{B}^{k+1} C
\end{tikzcd}$$

We will then have $\int D_k = \flat B^{k+1} K$, $\overset{\infty}{D}_k = \overset{\infty}{Z}_{k+1}$, and $\flat D_k = \flat \mathbf{B}^k G$, giving us a modal fracture hexagon:

$$\begin{tikzcd}
 & \overset{\infty}{Z}_{k+1} \arrow[rr] \arrow[dr, "\pi"] & & Z_{k+1} \arrow[dr, "(-)^{\int}"] & \\
\flat \mathbf{B}^k C \arrow[ur, "(-)_\flat"] \arrow[dr] & & D_k \arrow[ur, "F_{(-)}"] \arrow[dr, "(-)^{\int}"'] & & \flat \mathbf{B}^{k+1} C \\
 & \flat \mathbf{B}^k G \arrow[rr, "\beta"'] \arrow[ur, "(-)_\flat"'] & & \flat \mathbf{B}^{k+1} K \arrow[ur] &
\end{tikzcd} \tag{3.5.1}$$

Taking the short exact sequence $0 \to K \to C \to G \to 0$ to be

$$0 \to \sharp\mathbb{Z} \to \sharp\mathbb{R} \to \sharp U(1) \to 0$$

gives us a bona-fide combinatorial analogue of ordinary differential cohomology, fitting within a similar hexagon. However, instead of equipping the integral cohomology of manifolds with differential form data, we are equipping the integral cohomology of symmetric simplicial sets with real cocycle data.

We intend to give this combinatorial analogue of ordinary differential cohomology a fully internal treatment in future work.

References

[1] Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. "Higher Groups in Homotopy Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, 205–214. ISBN: 9781450355834. DOI: 10.1145/3209108.3209150. URL: https://doi.org/10.1145/3209108.3209150.

[2] Ulrich Bunke, Thomas Nikolaus, and Michael Völkl. "Differential cohomology theories as sheaves of spectra". In: *Journal of Homotopy and Related Structures* 11.1 (Oct. 2014), pp. 1–66. DOI: 10.1007/s40062-014-0092-5. URL: https://doi.org/10.1007/s40062-014-0092-5.

[3] Jeff Cheeger and James Simons. "Differential characters and geometric invariants". In: *Geometry and Topology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985. ISBN: 978-3-540-39738-0.

[4] Shiing-Shen Chern and James Simons. "Characteristic Forms and Geometric Invariants". In: *Annals of Mathematics* 99.1 (1974), pp. 48–69. ISSN: 0003486X. URL: http://www.jstor.org/stable/1971013.

[5] Felix Cherubini and Egbert Rijke. "Modal descent". In: *Mathematical Structures in Computer Science* (2020), 1–29. DOI: 10.1017/S0960129520000201.

[6] Pierre Deligne. "Théorie de Hodge : II". fr. In: *Publications Mathématiques de l'IHÉS* 40 (1971), pp. 5–57. URL: http://www.numdam.org/item/PMIHES_1971__40__5_0/.

[7] Eduardo J. Dubuc. "Sur les modèles de la géométrie différentielle synthétique". fr. In: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 20.3 (1979), pp. 231–279. URL: http://www.numdam.org/item/CTGDC_1979__20_3_231_0/.

[8] M.J. Hopkins and I.M. Singer. "Quadratic functions in geometry, topology, and M-theory". In: *Journal of Differential Geometry* 70.3 (2005), pp. 329 –452. DOI: 10.4310/jdg/1143642908. URL: https://doi.org/10.4310/jdg/1143642908.

[9] Anders Kock. *Synthetic Differential Geometry*. Cambridge University Press, June 2006. ISBN: 9780521687386.

[10] F Lawvere. "Axiomatic Cohesion". In: *Theory and Applications of Categories* 19 (June 2007), pp. 41–49.

[11] *Categorical Dynamics*. Vol. 30. Topos Theoretic Methods in Geometry. Aarhus University, 1979, pp. 1–28.

[12] F. William Lawvere. "Toposes generated by codiscrete objects in combinatorial topology and functional analysis". In: *Reprints in Theory and Applications of Categories* 27 (2021), pp. 1–11. URL: http://www.tac.mta.ca/tac/reprints/articles/27/tr27abs.html.

[13] F. William Lawvere. "Toward the description in a smooth topos of the dynamically possible motions and deformations of a continuous body". en. In: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 21.4 (1980), pp. 377–392. URL: http://www.numdam.org/item/CTGDC_1980__21_4_377_0/.

[14] Jacob Lurie. *Derived Algebraic Geometry XII: Proper Morphisms, Completions, and the Grothendieck Existence Theorem*. http://people.math.harvard.edu/~lurie/papers/DAG-XII.pdf. 2011.

[15] Felipe Gago Marta Bunge and Ana María San Luis. *Synthetic Differential Topology*. Vol. 448. Mar. 2018. ISBN: 9781108447232.

[16] Ieke Moerdijk and Gonzalo E. Reyes. *Models for Smooth Infinitesimal Analysis*. Springer New York, 1991. DOI: 10.1007/978-1-4757-4143-8. URL: https://doi.org/10.1007/978-1-4757-4143-8.

[17] David Jaz Myers. *Good Fibrations through the Modal Prism*. 2020. arXiv: 1908.08034 [math.CT].

[18] Egbert Rijke. *Classifying Types*. 2019. arXiv: 1906.09435 [math.LO].

[19] Egbert Rijke, Michael Shulman, and Bas Spitters. "Modalities in homotopy type theory". In: *Logical Methods in Computer Science* Volume 16, Issue 1 (Jan. 2020). DOI: 10.23638/LMCS-16(1:2)2020. URL: https://lmcs.episciences.org/6015.

[20] Urs Schreiber. *Differential cohomology in a cohesive infinity-topos*. 2013. arXiv: 1310.7930 [math-ph].

[21] Schreiber, Urs. *Differential cohesion and idelic structure*. http://ncatlab.org/nlab/show/differential\%20cohesion\%20and\%20idelic\%20structure. May 2021.

[22] Michael Shulman. "Brouwer's fixed-point theorem in real-cohesive homotopy type theory". In: *Mathematical Structures in Computer Science* 28.6 (2018), 856–941. DOI: 10.1017/S0960129517000147.

[23] James Simons and Dennis Sullivan. "Axiomatic characterization of ordinary differential cohomology". In: *Journal of Topology* 1.1 (2008), pp. 45–56. DOI: https://doi.org/10.1112/jtopol/jtm006. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/jtopol/jtm006. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/jtopol/jtm006.

[24] Felix Wellen. "Cohesive Covering Theory". In: *Proceedings of HoTT/UF 2018*. Ed. by Huber Ahrens and Mörtberg. Aug. 2018. URL: https://hott-uf.github.io/2018/abstracts/HoTTUF18_paper_15.pdf.

[25] David Yetter. "On right adjoints to exponential functors". In: *Journal of Pure and Applied Algebra* 45.3 (1987), pp. 287–304. ISSN: 0022-4049. DOI: https://doi.org/10.1016/0022-4049(87)90077-6. URL: https://www.sciencedirect.com/science/article/pii/0022404987900776.

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Self-Formalisation of Higher-Order Logic

Semantics, Soundness, and a Verified Implementation

Ramana Kumar · Rob Arthan ·
Magnus O. Myreen · Scott Owens

Received: date / Accepted: date

Abstract We present a mechanised semantics for higher-order logic (HOL), and a proof of soundness for the inference system, including the rules for making definitions, implemented by the kernel of the HOL Light theorem prover. Our work extends Harrison's verification of the inference system without definitions. Soundness of the logic extends to soundness of a theorem prover, because we also show that a synthesised implementation of the kernel in CakeML refines the inference system. Apart from adding support for definitions and synthesising an implementation, we improve on Harrison's work by making our model of HOL parametric on the universe of sets, and we prove soundness for an improved principle of constant specification in the hope of encouraging its adoption. Our semantics supports defined constants directly via a context, and we find this approach cleaner than our previous work formalising Wiedijk's Stateless HOL.

1 Introduction

In this paper, we present a mechanised proof of the soundness of higher-order logic (HOL), including its principles for defining new types and new polymorphic constants, and describe production of a verified implementation of its inference rules. This work is part of a larger

The first author was supported by the Gates Cambridge Trust. The second author was funded in part by the EPSRC (grant number EP/K503769/1). The third author was partially supported by the Royal Society UK and the Swedish Research Council.

R. Kumar
Computer Laboratory, University of Cambridge
E-mail: Ramana.Kumar@cl.cam.ac.uk

R. D. Arthan
Department of Computer Science, University of Oxford
E-mail: rda@lemma-one.com

M. O. Myreen
CSE Department, Chalmers University of Technology
E-mail: Myreen@chalmers.se

S. Owens
School of Computing, University of Kent
E-mail: S.A.Owens@kent.ac.uk

project (see Myreen et. al. [26] and Kumar et. al. [16]), to produce a verified machine-code implementation of a HOL theorem prover. This paper continues the top half of the project: soundness of the logic, and a verified implementation of the logical kernel in CakeML [17].

What is the point of verifying a theorem prover and formalising the semantics of the logic it implements? One answer is that it raises our confidence in its correctness. A theorem prover implementation usually sits at the centre of the trusted code base for verification work, so effort spent verifying the theorem prover multiplies outwards. Secondly, it helps us understand our systems (logical and software), to the level of precision possible only via formalisation. Finally, a theorem prover is a non-trivial piece of software that admits a high-level specification and whose correctness is important: we see it as a catalyst for tools and methods aimed at developing complete verified systems, readying them for larger systems with less obvious specifications.

We build on Harrison's proof of the consistency of HOL without definitions [10], which shares our larger goal of verifying concrete HOL theorem prover implementations, and advance this project by verifying an implementation of the HOL Light [11] kernel in CakeML, an ML designed to support fully verified applications. We discuss the merits of Harrison's model of set theory defined within HOL, and provide an alternative not requiring axiomatic extensions to the theorem prover's logic.

Definition by conservative extension is one of the hallmarks of using HOL, and makes the logic expressive with a small number of primitives. When considering implementations of the logic, the definitional rules are important because defined constants are not merely abbreviations, but are distinguished variants in the datatypes representing the syntax of the logic. For this reason, we think it is important to formalise the rules of definition and to use a semantic framework that supports that goal.

The rule for defining term constants that we formalise, which is actually a rule for constant specification, generalises the one found in the various HOL systems by adding support for implicit definitions with fewer constraints and no new primitives. A full account of its history and design can be found in Arthan [4] (and the extended version in this issue). As reported there, our proof of its soundness has cleared the way for adoption of the improved rule.

The main result of the work described in this paper is a verified CakeML implementation of the logical kernel of HOL Light. We intend to use this kernel implementation as the foundation for a verified machine-code implementation of a complete LCF-style [22] theorem prover with the kernel as a module. The specific contributions of this paper are:

- a formal semantics for HOL that supports definitions (§4), against a new specification of set theory (§3),
- proofs of soundness and consistency (§5) for the HOL inference system, including type definitions, the new rule for constant specification, and the three axioms used in HOL Light,
- a verified implementation of a theorem-prover kernel (based on HOL Light's) in CakeML (§6) that should be a suitable basis for a verified implementation of a theorem prover in machine code.

All our definitions and proofs have been formalised in the HOL4 theorem prover [29] and are available from `https://cakeml.org`.[1] We briefly discuss the natural question of trusting an unverified theorem prover for this work, and how we might skate along the barriers around true self-verification, in the conclusion.

[1] Specifically, `https://code.cakeml.org/tree/version1/hol-light`.

A version of this paper [16] was originally published in the conference proceedings of ITP 2014, and describes a semantics for Stateless HOL [34]. By contrast, the semantics presented in this paper works directly on standard HOL and uses a theory context to handle definitions (in the style of Arthan [3]). We found this new style of semantics to be both easier to understand and simpler to work with. It also enables us to better characterise the abstraction barrier provided by the generalised rule of constant specification. As well as describing an improved formalisation, we use the space available in this Special Issue to give a more complete description than was possible in the conference paper.

2 Approach

Higher-order logic, or HOL, as we use the term in this paper, is a logic that was first proposed and implemented as an alternative object logic for LCF by Mike Gordon [9]. It adds polymorphism *à la* Milner [21] to Church's simple type theory ([7], [2, Chapter 5]) resulting in a tool that has proved remarkably powerful over the last 30 years in a wide range of mechanised theorem-proving applications. The logic is well-understood: a rigorous informal account of its simple and natural set-theoretic semantics has been given by Pitts and is included in the HOL4 theorem prover's documentation [27]. Our aim here is to build a formal model of this semantics, against which we can verify a formalisation of the HOL inference system. We will then take the inference system as a specification, against which we can verify an implementation of a theorem-prover kernel based on HOL Light's. This task can be broken down into the following steps:

1. Specify the set-theoretic notions needed. (§3)
2. Define the syntax of HOL types, terms, and sequents. (§4.1)
3. Define semantic functions assigning appropriate sets to HOL types and terms, and use these to specify validity of a sequent. (§4.2)
4. Define the inference system: how to construct sequents-in-context (the rules of inference), and how to extend a context (the rules of definition). (§4.3)
5. Verify the inference system: prove that every derivable sequent is valid. Deduce that the inference system is consistent: it does not derive a contradiction. (§5)
6. Write an implementation of the inference system as recursive functions in HOL, and verify it against the relational specification of the inference rules. (§6.1)
7. Synthesise a verified implementation of the inference rules in CakeML. (§6.2)

We construct the specifications, definitions, and proofs above in HOL itself (using the HOL4 theorem prover), in the style of Harrison's work [10] towards self-verification of HOL Light. Compared to Harrison's work, items 6–7 are new, items 2–5 are extended and reworked to support a context of definitions, and item 1 uses an improved specification. Our implementation language, CakeML [16], is an ML-like language (syntactically a subset of Standard ML) with a formal semantics specified in HOL and for which there are automated techniques (Myreen and Owens [25]) for accomplishing item 7.

The bulk of our programme is concerned with soundness: we say an inference rule is *sound* if whenever its antecedents hold in all models[2] then so does its succedent. Observe that soundness does not require the existence of models. Given a deductive system defined

[2] Throughout the paper, we are concerned only with standard models of HOL, that is, where the Boolean and function types and the equality constant are interpreted in the standard way, and function spaces are full (unlike in Henkin semantics [12]). See the paragraph on standard interpretations in Section 4.2.

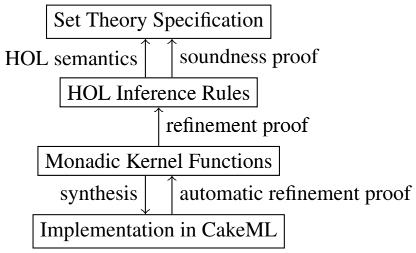


Fig. 1 Producing a Sound Implementation by Refinement

by axioms and inference rules, the soundness of the inference rules does not imply the consistency of the axioms, but it does imply that deductive closure of any set of axioms that admits a model is consistent. It is useful to distinguish between soundness of the inference rules and consistency of the deductive system, as we are able to give a formal proof of the soundness of the inference system within HOL, while we cannot expect to prove consistency: Gödel's second incompleteness theorem [8] prevents us from proving HOL's consistency in HOL, assuming HOL is consistent. Nonetheless, to validate our formalisation of the semantics we have carried out some proofs that give evidence of consistency. In particular, we prove in HOL the consistency of HOL with its axiom of infinity omitted. This gives fairly convincing evidence that we have correctly captured the semantics.

Our approach avoids making any axiomatic extensions to HOL. We also isolate results that are dependent on the set-theoretic axiom of infinity, so that as much as possible is proved without any undischarged assumptions. We are able to use assumptions on our theorems rather than asserting new axioms in the logic because we formalise a specification of set theory rather than defining a particular instance of a set theory as Harrison [10] did.

The results of following the plan above fit together as shown in Figure 1. The overall theorems we obtain are about evaluating the CakeML implementations of the HOL Light kernel functions in CakeML's operational semantics. For each kernel function, we prove that if the function is run in a good state on good arguments, it terminates in a good state and produces good results. Here "good" refers to our invariants. In particular, a good value of type *thm* must be a HOL sequent that is valid according to the set-theoretic semantics.

We prove these results by composing the three proof layers in the diagram. The top layer is the result of steps 1–5. The HOL semantics gives meaning to HOL sequents, from which we obtain definitions of validity and consistency. Validity concerns the truth of a sequent within a fixed context of definitions, whereas consistency is about whether the context itself has a model. The soundness proof says that each of the HOL inference rules preserves validity of sequents, and each of the HOL principles of definition preserves consistency of the context.

The middle layer corresponds to step 6. As described in our previous work [26,16], we define shallowly-embedded HOL functions, using a state-exception monad, for each of the HOL Light kernel functions. These "monadic kernel functions" are a hand-crafted implementation, but are written following the original OCaml code for HOL Light closely, and we prove that they implement the inference rules. Specifically, if one of these functions is applied to good arguments, it terminates with a good result; any theorem result must refine a sequent that is provable in the inference system.

Finally, for step 7 we use the method developed by Myreen and Owens [25] to synthesise CakeML implementations of the monadic kernel functions. The automatic translation from shallowly- to deeply-embedded code is proof-producing. We use the generated certificate theorems to complete the refinement proof that links theorem values produced by the implementation to sequents that are semantically valid.

In the context of our larger project, the next steps include: a) proving, against CakeML's semantics, that our implementation of the kernel can be wrapped in a module to protect the key property, provability, of values of type *thm*; and b) using CakeML's verified compiler to generate a machine-code implementation of the kernel embedded in an interactive read-eval-print loop that is verified to never print a false theorem.

The focus for most of the rest of the paper is the top layer of Figure 1, wherein we describe how we formalise the syntax, semantics, and soundness of HOL with full support for the definition of new types and constants as well as support for non-definitional context-extension. We return to the verified implementation part of the story, corresponding to the lower layers of the figure, in Section 6. The paper ends with a discussion about self-verification and related work.

Terminology and Notation As our programme involves both refinement proofs (linking implementations to specifications) and soundness proofs (linking an inference system to the semantics of a logic), and for the latter both our meta-logic and object-logic are HOL, we often need to refer to similar but different things and some care must be paid for clarity. By "HOL" we refer, as in the section above, to higher-order logic itself. Particular inference systems for HOL are implemented by interactive theorem-proving systems. The two theorem provers of interest to us are HOL Light, because we formalise its inference system and use its implementation as inspiration for our implementation in CakeML; and HOL4, because we use it to mechanise our proofs. We use "HOL4" and "HOL Light" unqualified to refer to the theorem provers, and clarify explicitly when we mean the inference system instead. HOL4 implements a different inference system from HOL Light's, but the two are inter-translatable.

We include extracts, generated by HOL4, from our formalisation in the paper. These include definitions, for example, here is the standard library function for checking a predicate holds for all elements of a list:

$$\begin{array}{l} \mathsf{every}\ P\ []\ \iff\ \mathsf{T} \\ \mathsf{every}\ P\ (h{::}t)\ \iff\ P\ h \wedge \mathsf{every}\ P\ t \end{array}$$

Since the result of $\mathsf{every}\ P\ ls$ is Boolean, we use $(\iff)$ in the defining equations; at other types we simply use $(=)$. As well as definitions we have theorems, which are shown with a turnstile, for example:

$$\vdash\ \neg\mathsf{every}\ P\ ls\ \iff\ \mathsf{exists}\ ((\neg) \circ P)\ ls$$

Free variables may appear in theorems; semantically, they behave as if universally quantified. Datatype definitions are shown as in the following example of the polymorphic option type with two constructors:

$$\alpha\ \mathit{option}\ =\ \mathsf{None}\ \mid\ \mathsf{Some}\ \alpha$$

Terms are sometimes annotated with their types, for example: $(\mathsf{Some} : \mathit{bool} \to \mathit{bool\,option})\ \mathsf{F}$. Quantifiers are printed as binders, as in $\forall x.\ \exists y.\ x \neq \mathsf{Some}\ y$, although in HOL the quantifiers are ordinary constants (that operate on predicates, that is, functions with codomain *bool*). The existential quantifier in the previous sentence might more pedantically be printed as an application of $(\exists)$ to $\lambda y.\ x \neq \mathsf{Some}\ y$. Finally, we show the rules of inductive relations using a horizontal line to separate premises from the conclusion. Thus the rule,

$\vdash\ R\,x\,y \wedge R^*\,y\,z \Rightarrow R^*\,x\,z$, about the reflexive transitive closure of a relation can also be written as follows:

$$\frac{\begin{array}{c}R\,x\,y\\ R^*\,y\,z\end{array}}{R^*\,x\,z}$$

3 Set Theory Specification

We now begin the technical part of the paper, starting with a specification of set theory over which the semantics of HOL will be specified in the next section. Axiomatic set theory can be specified in terms of a single binary relation, the membership relation. In HOL, we can give a quite straightforward development of the basic concepts of set theory as may be found in any standard text (e.g., Vaught [30]) thus achieving clarity through familiarity and making it easy to compare our formalisation with Pitts' informal account [27]. Since our specification is in HOL, we can write the membership relation and its axioms within the logic without resorting to the metavariables and schemata required in the first-order setting[3].

The most common set theory in textbook accounts is Zermelo-Fraenkel set theory ZF. However, ZF's axiom of replacement plays no rôle in giving semantics to HOL, so all we need are the axioms of Zermelo's original system: extensionality, separation (a.k.a. comprehension or specification), power set, union, (unordered) pairing, and infinity. It will be convenient to deal with the axiom of infinity separately. So we begin by defining a predicate on membership relations, $\mathsf{is\_set\_theory}\ (mem : \mathscr{U} \to \mathscr{U} \to \mathit{bool})$, that asserts that the membership relation satisfies each of the Zermelo axioms apart from the axiom of infinity. By formalising the set-theoretic universe as a type variable, $\mathscr{U}$, we can specify what it means to be a model of Zermelo set theory, while deferring the problem of whether such a model can be constructed.

The specification of the set-theoretic axioms is as follows:

$$\begin{array}{l}\mathsf{is\_set\_theory}\ mem \iff \\ \ \ \mathsf{extensional}\ mem \wedge (\exists sub.\ \mathsf{is\_separation}\ mem\ sub) \wedge \\ \ \ (\exists power.\ \mathsf{is\_power}\ mem\ power) \wedge (\exists union.\ \mathsf{is\_union}\ mem\ union) \wedge \\ \ \ \exists upair.\ \mathsf{is\_upair}\ mem\ upair\end{array}$$

[3] In our statement of the separation axiom, if the set x is infinite then P ranges over an uncountable set corresponding to all subsets of x. Technically, this is a significant strengthening of the axiom of separation, since it is not restricted to the countably many subsets of x that can be specified in the language of first-order set theory. However, this is irrelevant to our purposes: it would simply complicate the description of the semantics to impose this restriction (although our proofs in fact do not need instances of the axiom that could not be expressed in first-order set theory). Similarly, we find it convenient to use the metalanguage choice function and the metalanguage notion of finiteness rather than trying to give a first-order description of these notions in a model.

where

$$\begin{aligned}&\mathsf{extensional}\ \mathit{mem} \iff \\ &\quad \forall x\, y.\ x = y \iff \forall a.\ \mathit{mem}\ a\ x \iff \mathit{mem}\ a\ y\end{aligned}$$

$$\begin{aligned}&\mathsf{is\_separation}\ \mathit{mem}\ \mathit{sub} \iff \\ &\quad \forall x\, P\, a.\ \mathit{mem}\ a\ (\mathit{sub}\ x\ P) \iff \mathit{mem}\ a\ x \wedge P\ a\end{aligned}$$

$$\begin{aligned}&\mathsf{is\_power}\ \mathit{mem}\ \mathit{power} \iff \\ &\quad \forall x\, a.\ \mathit{mem}\ a\ (\mathit{power}\ x) \iff \forall b.\ \mathit{mem}\ b\ a \Rightarrow \mathit{mem}\ b\ x\end{aligned}$$

$$\begin{aligned}&\mathsf{is\_union}\ \mathit{mem}\ \mathit{union} \iff \\ &\quad \forall x\, a.\ \mathit{mem}\ a\ (\mathit{union}\ x) \iff \exists b.\ \mathit{mem}\ a\ b \wedge \mathit{mem}\ b\ x\end{aligned}$$

$$\begin{aligned}&\mathsf{is\_upair}\ \mathit{mem}\ \mathit{upair} \iff \\ &\quad \forall x\, y\, a.\ \mathit{mem}\ a\ (\mathit{upair}\ x\ y) \iff a = x \vee a = y\end{aligned}$$

To state the (set-theoretic) axiom of infinity, we define what it means for an element of $\mathscr{U}$ to be infinite: $\mathsf{is\_infinite}\ \mathit{mem}\ s \iff \neg\mathsf{FINITE}\ \{\, a \mid \mathit{mem}\ a\ s \,\}$. Here FINITE is inductively defined (in HOL) for sets-as-predicates, so we are saying a set is infinite if it does not have finitely many members. The (set-theoretic) axiom of infinity asserts that such a set exists.

3.1 Derived Operations

Using the axioms above, it is straightforward to define standard set-theoretic constructions that will support our specification of the semantics of HOL. In this subsection, we introduce some of our notation for such derived operations. All our definitions are parametrised by the membership relation, $(\mathit{mem} : \mathscr{U} \to \mathscr{U} \to \mathsf{bool})$; we often elide this argument with a pretty-printing abbreviation, for example writing $\mathsf{Funspace}\ x\ y$ instead of $\mathsf{funspace}\ \mathit{mem}\ x\ y$.[4] When *mem* is used as a binary operator, we will from now on write it infix as $x \lessdot y$ instead of $\mathit{mem}\ x\ y$. Also, most of our theorems are under the assumption $\mathsf{is\_set\_theory}\ \mathit{mem}$; we often elide this assumption.

Using the axiom of separation we define the empty set and prove it has no elements, then using pairing we define sets containing exactly one and exactly two elements. The latter serves as our representation of the set of Booleans. We have the following, shown with and without abbreviations for clarity:

(full notation)

$$\begin{aligned}&\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\ &\quad \forall x.\ \mathit{mem}\ x\ (\mathsf{two}\ \mathit{mem}) \iff x = \mathsf{true}\ \mathit{mem} \vee x = \mathsf{false}\ \mathit{mem}\end{aligned}$$

(abbreviated notation)

$$\vdash\ x \lessdot \mathsf{Boolset} \iff x = \mathsf{True} \vee x = \mathsf{False}$$

We define Kuratowski pairs (defn. V in [18]) as well as the cross product of two sets so that the following properties hold.

$$\begin{aligned}&\vdash\ (a,b) = (c,d) \iff a = c \wedge b = d \\ &\vdash\ a \lessdot x \times y \iff \exists b\, c.\ a = (b,c) \wedge b \lessdot x \wedge c \lessdot y\end{aligned}$$

[4] We follow a loose convention that capitalised functions have the hidden *mem* argument. Be aware that datatype constructors, which are also capitalised, are amongst the exceptions.

From cross products, we can define relations, and then functions (graphs) as functional relations. $\mathsf{Abstract}\ s\ t\ f$ is our notation for the subset of $s \times t$ that is the graph of $(f : \mathscr{U} \rightarrow \mathscr{U})$, and $a \,\prime\, x$ denotes application of such a set-theoretic function a to an argument x. The main theorem about application in set theory is that it acts like application in HOL:

$$\vdash\ x \lessdot s \wedge f\ x \lessdot t \Rightarrow \mathsf{Abstract}\ s\ t\ f \,\prime\, x = f\ x$$

Furthermore, we know functions obey extensional equality:

$$\begin{aligned}\vdash\ &(\forall x.\ x \lessdot s \Rightarrow f_1\ x \lessdot t_1 \wedge f_2\ x \lessdot t_2 \wedge f_1\ x = f_2\ x) \Rightarrow\\ &\mathsf{Abstract}\ s\ t_1\ f_1 = \mathsf{Abstract}\ s\ t_2\ f_2\end{aligned}$$

We define the set of functions between two sets, and prove that its elements are precisely those made using $\mathsf{Abstract}$:

$$\begin{aligned}&\vdash\ (\forall x.\ x \lessdot s \Rightarrow f\ x \lessdot t) \Rightarrow \mathsf{Abstract}\ s\ t\ f \lessdot \mathsf{Funspace}\ s\ t\\ &\vdash\ a \lessdot \mathsf{Funspace}\ s\ t \Rightarrow\\ &\qquad \exists f.\ a = \mathsf{Abstract}\ s\ t\ f \wedge \forall x.\ x \lessdot s \Rightarrow f\ x \lessdot t\end{aligned}$$

The derived operations in our formalisation (a selection of which were shown in this subsection) may be considered as an alternative description (compared to the Zermelo axioms) of the interface required for giving semantics to HOL. In other words, any structure supporting such constructions as pairs and functions is suitable.

It is worth noting that a relation $(\mathit{mem} : \mathscr{U} \rightarrow \mathscr{U} \rightarrow \mathit{bool})$ satisfying $\mathsf{is\_set\_theory}\ \mathit{mem}$ will automatically satisfy the set-theoretic axiom of choice (AC), that is, we can prove the following (where $\mathsf{inhabited}\ a$ stands for $\exists s.\ s \lessdot a$):

$$\vdash\ \forall x.\ (\forall a.\ a \lessdot x \Rightarrow \mathsf{inhabited}\ a) \Rightarrow \exists f.\ \forall a.\ a \lessdot x \Rightarrow f \,\prime\, a \lessdot a$$

We prove this by using the axiom of choice in HOL (i.e., the language we are using to formalise set theory) to provide a HOL function $(g : \mathscr{U} \rightarrow \mathscr{U})$ such that for every nonempty $(a : \mathscr{U})$, we have $g\ a \lessdot a$. Then, given a set $(x : \mathscr{U})$ whose members are all nonempty, we use $\mathsf{Abstract}$ (ultimately depending on our strong form of the axiom of separation) to define the graph of g restricted to the members of x (as a member of the set-theoretic universe $\mathscr{U}$) and hence conclude that AC holds in our set theory. This is a consequence of our decision to give a standard semantics to HOL.

3.2 Consistency of the Specification

So far we have specified a predicate on a membership relation asserting that it represents a set theory with our desired structure (without the axiom of infinity). As a sanity check to convince us that this part of the specification is consistent, we can construct a membership relation that satisfies the predicate.

The hereditarily finite sets provide a simple model of set theory without the axiom of infinity. This model can be represented concretely by taking $\mathscr{U}$ to be the type num of natural numbers and defining $\mathit{mem}\ n\ m$ to hold whenever the nth bit in the binary representation of m is not zero. Of course, the axiom of infinity fails in this model since every set in the model is finite. Nevertheless, we can prove that it satisfies the other axioms ($\mathsf{V\_mem}$ is essentially the membership relation just described):

$$\vdash\ \mathsf{is\_set\_theory}\ \mathsf{V\_mem}$$

This shows that the notion of a set theory that we have formalised is not vacuous. One might argue that we could just use the monomorphic type *num* in place of the type variable $\mathscr{U}$. Or a little more abstractly, we could introduce a new type witnessed by the above construction on *num*. However, we wish to state some properties that are conditional on the set-theoretic axiom of infinity. Unfortunately, the axiom of infinity is provably false in a model obtained from the countable set *num* and so results that assumed the axiom of infinity would be trivially true.

Instead, if we identify the universe of the hereditarily finite sets construction with the right-hand summand of the polymorphic type $\alpha + \mathit{num}$, we can define a subtype $\alpha\ V$ of $\alpha + \mathit{num}$ whose defining property is the existence of a membership relation satisfying the Zermelo axioms other than infinity. Hence we can introduce a constant V_mem of type $\alpha\ V \rightarrow \alpha\ V \rightarrow \mathit{bool}$ with $\vdash$ is_set_theory V_mem as its defining property. Thus if we work with V_mem the axiom of infinity is not provably false and we can meaningfully take it is as an assumption when necessary.

In the remainder of our development, we leave *mem* as a free variable and add one or both of the assumptions, is_set_theory *mem* and $\exists\mathit{inf}.$ is_infinite *mem inf*, whenever they are required. We provide V_mem in this section as a possible non-contradictory instantiation for the free variable *mem* in our theorems. Any instantiation that satisfies both assumptions would do, but we know we cannot exhibit one within HOL itself, so we prefer to leave the theorems uninstantiated. Our decision to leave *mem* loosely specified (i.e., as a free variable) throughout the development is made easier by HOL4's parsing/printing support for hiding the free variable. In a theorem prover without such syntactic abbreviations, the notational clutter might have lent some encouragement to picking an instantiation up front.

Comparison to Harrison's Approach Rather than formalising a specification of axiomatic set theory (which can then be instantiated), Harrison [10] constructs his model of HOL in HOL at the same time as proving its requisite properties. In fact, his proof scripts allow one to choose (by rearranging comments) between an axiomatic formalisation of the set theory with an axiom of infinity or a conservative definition of the set theory without that axiom. More specifically, with the first option, he declares a new type, intended to be the universe of sets and asserts the axiom of infinity and a certain closure property[5] hold in that type, while in the second option he defines the type to be countably infinite. He then (in both options) uses what amounts to a type system (Harrison calls the types "levels") to define a coherent notion of membership in terms of injections into the universe. As a result, his sets are not extensional since there are empty sets at every level; because of this technicality his construction does not satisfy our is_set_theory.

Under the conservative option, Harrison still achieves a model without axiomatic extensions for HOL without the axiom of infinity, since he can prove his closure property using a construction similar to our V_mem above and that is all that is required to construct the type system. The disadvantage is that the set-theoretic axiom of infinity is provably false when one chooses the conservative option. In our approach, the polymorphism means that the set-theoretic axiom of infinity is unprovable rather than false, and so it is meaningful to prove theorems with that axiom as an assumption.

Harrison's use of levels is motivated by the desire to assert just one axiom: the cardinality property of the universe. With only this property, levels are required to distinguish

[5] The closure property asserts that, if X is any set of cardinality strictly less than that of the universe, then the cardinality of the power set of X is also strictly less than that of the universe. This property holds but the axiom of infinity fails in a countably infinite universe.

different embeddings into the universe (e.g., to distinguish powersets from cross products). Our approach with an explicit membership relation gives us a specification where these distinctions are explicit. We do not need to appeal to a theory of cardinalities in the meta-logic, since the assumptions we make ($\mathsf{is\_set\_theory}\ \mathit{mem} \land \exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}$) mirror the standard axioms of Zermelo set theory.

4 HOL Specification

With our specification of set theory in place, we now turn to the task of specifying the syntax and semantics of HOL. At the level of terms and types, our specification of the syntax is almost the same as Harrison's [10]. Our terms are simplified by not needing to bake in any primitive constants since we support a general mechanism for introducing new constants, and our approach to substitution and instantiation of bound variables is improved. Also, our abstraction terms match the implementation better by using a term (rather than just a name and type) for the bound variable (see §4.1.1 below).

At the level of sequents, we introduce the notion of a *theory* describing the defined constants, which we implement in the inference system as a *context* of theory-extending updates. Compared to our previous work [16] on Stateless HOL [34], where information about defined constants is carried on the terms and types themselves, this separate context of definitions makes the inference system clearer and allows us to easily quantify over all interpretations of the constants.

We present the HOL specification concisely, but give the important definitions in full so that it might serve as a reference.

4.1 Sequents: the judgements of the logic

Formally, derivations in HOL produce judgements of the following form[6]:

$$(\mathit{thy},h)\ \texttt{|-}\ c$$

This judgement is known as a *sequent*. It has a conclusion, c, a set of hypotheses (represented by a list of terms), h, and is interpreted in a theory, *thy* consisting of axioms and a signature. The meaning of a sequent is that the conclusion is true whenever all the hypotheses are true, all the axioms are true, and all the terms are well-formed with respect to the signature. We begin our specification at the bottom of this structure, starting with terms and types.

4.1.1 Terms and Types

The syntax of HOL is the syntax of the (polymorphic) simply-typed lambda-calculus. Types are either variables or applied type operators.

$$\mathit{type}\ =\ \mathsf{Tyvar}\ \mathit{string}\ \mid\ \mathsf{Tyapp}\ \mathit{string}\ (\mathit{type\ list})$$

Primitive type operators include Booleans and function spaces. We abbreviate $\mathsf{Tyapp}\ \texttt{"bool"}\ [\,]$ by Bool and $\mathsf{Tyapp}\ \texttt{"fun"}\ [x;\ y]$ by $\mathsf{Fun}\ x\ y$.

[6] We use the symbol ($\texttt{|-}$) for the sequents defined in our specification of HOL, reserving ($\vdash$) for theorems proved in the meta-logic (that of HOL4).

A term is either a variable, a constant, a combination (application), or an abstraction.

$$\begin{array}{l} \mathit{term} = \\ \quad \mathsf{Var}\ \mathit{string}\ \mathit{type} \\ \ \mid \mathsf{Const}\ \mathit{string}\ \mathit{type} \\ \ \mid \mathsf{Comb}\ \mathit{term}\ \mathit{term} \\ \ \mid \mathsf{Abs}\ \mathit{term}\ \mathit{term} \end{array}$$

Variables carry their types: two variables with the same name but different types are distinct. We expect the first argument to Abs to be a variable, but use a *term* so the implementation can avoid destructing and reconstructing variables whenever it manipulates an abstraction.

Constants also carry a type, but are identified by their name: the type is there only to indicate the instantiation of polymorphic constants. (Different constants with the same name are disallowed, as we will see when we describe the signature of a theory and how it is updated.) The sole primitive constant is equality; we abbreviate $\mathsf{Const}\ \texttt{"="}\ (\mathsf{Fun}\ ty\ (\mathsf{Fun}\ ty\ \mathsf{Bool}))$ by $\mathsf{Equal}\ ty$.

Well-Typed Terms The datatype above might better be called “pre-terms”, because the only terms of interest are those that are well-typed. Every well-typed term has a unique type, which is specified by the following relation.

$$\frac{}{(\mathsf{Var}\ n\ ty)\ \mathsf{has\_type}\ ty} \qquad \frac{}{(\mathsf{Const}\ n\ ty)\ \mathsf{has\_type}\ ty}$$

$$\frac{s\ \mathsf{has\_type}\ (\mathsf{Fun}\ dty\ rty) \quad t\ \mathsf{has\_type}\ dty}{(\mathsf{Comb}\ s\ t)\ \mathsf{has\_type}\ rty} \qquad \frac{t\ \mathsf{has\_type}\ rty}{(\mathsf{Abs}\ (\mathsf{Var}\ n\ dty)\ t)\ \mathsf{has\_type}\ (\mathsf{Fun}\ dty\ rty)}$$

Well-typed terms differ from pre-terms only in that the first argument of every combination has a function type, where the domain matches the second argument’s type, and the first argument of every abstraction is a variable. We say $\mathsf{welltyped}\ tm \iff \exists ty.\ tm\ \mathsf{has\_type}\ ty$ and it is straightforward to define a function, typeof, to calculate the type of a term if it exists so that $\mathsf{welltyped}\ tm \iff tm\ \mathsf{has\_type}\ (\mathsf{typeof}\ tm)$ holds.

Two operations over terms and types remain for us to describe, namely alpha-equivalence and substitution. Both are complicated by the need to correctly implement the concept of variable binding.

4.1.2 Alpha-Equivalence

Terms are alpha-equivalent when they are equal up to a renaming of bound variables. The key idea of Harrison’s original approach is to formalise when two variables are equivalent in a context of pairs of equivalent bound variables.

$$\begin{array}{l} \mathsf{avars}\ [\,]\ (v_1,v_2) \iff v_1 = v_2 \\ \mathsf{avars}\ ((b_1,b_2)::bvs)\ (v_1,v_2) \iff \\ \quad v_1 = b_1 \land v_2 = b_2 \lor v_1 \neq b_1 \land v_2 \neq b_2 \land \mathsf{avars}\ bvs\ (v_1,v_2) \end{array}$$

The variables must be equal to some pair of bound variables (or to themselves) without either of them being equal to (captured by) any of the bound variables above. We lift this relation up to terms, for example:

$$\frac{\mathsf{avars}\ bvs\ (\mathsf{Var}\ x_1\ ty_1,\mathsf{Var}\ x_2\ ty_2)}{\mathsf{aterms}\ bvs\ (\mathsf{Var}\ x_1\ ty_1,\mathsf{Var}\ x_2\ ty_2)}$$

$$\frac{\begin{array}{c}\mathsf{typeof}\ v_1 = \mathsf{typeof}\ v_2\\ \mathsf{aterms}\ ((v_1,v_2)::bvs)\ (t_1,t_2)\end{array}}{\mathsf{aterms}\ bvs\ (\mathsf{Abs}\ v_1\ t_1,\mathsf{Abs}\ v_2\ t_2)}$$

Finally, we define $\mathsf{aconv}\ t_1\ t_2 \iff \mathsf{aterms}\ []\ (t_1,t_2)$ It is straightforward to show that this is an equivalence relation.

4.1.3 Substitution and Instantiation

Now on substitution, let us first deal with types. Since there are no type variable binders, type variables can simply be replaced uniformly throughout a type, given a type substitution mapping variable names to types. We define $\mathsf{tysubst}\ i\ ty$ as the type obtained by instantiating ty according to the type substitution i. We say $\mathsf{is\_instance}\ ty_0\ ty$ if $\exists i.\ ty = \mathsf{tysubst}\ i\ ty_0$.

Substitution of terms for variables and of types for type variables in terms are the most complex operations we need to deal with. Naïve substitution for variables in a term may introduce unwanted binding, for example when substituting $\mathsf{Comb}\ v_1\ t_1$ for v_2 in $\mathsf{Abs}\ v_1\ v_2$ the variable v_1 ought to remain free. The algorithm for term substitution (subst) therefore renames bound variables as required to avoid unintended capture.

The algorithm for type instantiation (inst) in terms is also complicated by this kind of problem. Consider, with $x_1 = \mathsf{Var}\ \texttt{"x"}\ (\mathsf{Tyvar}\ \texttt{"A"})$ and $x_2 = \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool}$, substitution of Bool for $\mathsf{Tyvar}\ \texttt{"A"}$ in $\mathsf{Abs}\ x_1\ (\mathsf{Abs}\ x_2\ x_1)$. The inner x_1 refers to the outer binder, but after a naïve substitution (which makes $x_1 = x_2$) it would incorrectly refer to the inner binder. The solution is for the type instantiation algorithm to keep track of potential shadowing as it traverses the term, and if any occurs to backtrack and rename the shadowing bound variable.

In Harrison's original formulation of HOL in HOL, the main lemma about type instantiation takes 377 lines of proof script and mixes reasoning about name clashes with the semantics of instantiation itself. To clarify our formalisation[7], we develop a small theory of nameless terms using de Bruijn indices, where substitution and instantiation are relatively straightforward, and shift the required effort to the task of translating to and from de Bruijn terms, which is somewhat easier than tackling capture-avoiding substitution directly. The analogous lemma about type instantiation in our formalisation is only 47 lines: the bulk of the work about name clashes appears in two lemmas totalling 166 lines about how instantiation can just as well be done on de Bruijn terms.[8]

We prove that two terms are alpha-equivalent if and only if their de Bruijn representations are equal. Using this fact, the main theorems we obtain about substitution and instantiation are that they both respect alpha-equivalence:

$$\begin{array}{l}\vdash\ \mathsf{welltyped}\ t_1 \wedge \mathsf{welltyped}\ t_2 \wedge \mathsf{subst\_ok}\ ilist \wedge \mathsf{aconv}\ t_1\ t_2 \Rightarrow\\ \qquad \mathsf{aconv}\ (\mathsf{subst}\ ilist\ t_1)\ (\mathsf{subst}\ ilist\ t_2)\\ \vdash\ \mathsf{welltyped}\ t_1 \wedge \mathsf{welltyped}\ t_2 \wedge \mathsf{aconv}\ t_1\ t_2 \Rightarrow\\ \qquad \mathsf{aconv}\ (\mathsf{inst}\ tyin\ t_1)\ (\mathsf{inst}\ tyin\ t_2)\end{array}$$

[7] Porting Harrison's proof directly would have been another option, but a less rewarding one involving reconciling all the minor differences between our definitions and between HOL Light and HOL4.

[8] Harrison's lemma is called SEMANTICS_INST_CORE, and ours are INST_CORE_dbINST, INST_CORE_simple_inst, and termsem_simple_inst.

Here $\mathsf{subst\_ok}\ \mathit{ilist}$ means *ilist* is a substitution mapping variables to well-typed terms of the same type. Since we also prove that alpha-equivalent terms have the same semantics, these theorems allow us to prove soundness of the inference rules that do substitution and instantiation.

To be clear, the inference rules do not use de Bruijn terms and our reasoning about them is simply a proof technique. Verifying substitution and instantiation for an inference system that used de Bruijn terms natively may have been easier. But additional work would then have been required to verify a user-friendly interface, with named variables, for the kernel.

4.1.4 Theories

In our specification of HOL, every sequent carries a *theory*, which embodies information about constants and type operators and thereby allows us to support principles of definition. Formally, a theory (*thy*) consists of a signature ($\mathsf{sigof}\ \mathit{thy}$) together with a set of axioms ($\mathsf{axsof}\ \mathit{thy}$). The signature restricts the constants and type operators that may appear in a sequent, and the axioms provide sequents that may be derived immediately. The principles of definition introduce axioms to characterise the things that are defined.

We specify a *signature* as a pair of maps, $(\mathsf{tysof}\ \mathit{sig},\mathsf{tmsof}\ \mathit{sig})$, assigning the defined type operator names to their arities and the defined term constant names to their types. Well-formed types obey the type signature:

$$\frac{}{\mathsf{type\_ok}\ \mathit{tysig}\ (\mathsf{Tyvar}\ x)} \qquad \frac{\begin{array}{c}\mathsf{lookup}\ \mathit{tysig}\ \mathit{name} = \mathsf{Some}\ (\mathsf{length}\ \mathit{args})\\ \mathsf{every}\ (\mathsf{type\_ok}\ \mathit{tysig})\ \mathit{args}\end{array}}{\mathsf{type\_ok}\ \mathit{tysig}\ (\mathsf{Tyapp}\ \mathit{name}\ \mathit{args})}$$

And well-formed terms obey both signatures:

$$\frac{\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{sig})\ \mathit{ty}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Var}\ x\ \mathit{ty})} \qquad \frac{\begin{array}{c}\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{sig})\ \mathit{ty}\\ \mathsf{lookup}\ (\mathsf{tmsof}\ \mathit{sig})\ \mathit{name} = \mathsf{Some}\ \mathit{ty}_0\\ \mathsf{is\_instance}\ \mathit{ty}_0\ \mathit{ty}\end{array}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Const}\ \mathit{name}\ \mathit{ty})}$$

$$\frac{\begin{array}{c}\mathsf{term\_ok}\ \mathit{sig}\ \mathit{tm}_1\\ \mathsf{term\_ok}\ \mathit{sig}\ \mathit{tm}_2\\ \mathsf{welltyped}\ (\mathsf{Comb}\ \mathit{tm}_1\ \mathit{tm}_2)\end{array}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Comb}\ \mathit{tm}_1\ \mathit{tm}_2)} \qquad \frac{\begin{array}{c}\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{sig})\ \mathit{ty}\\ \mathsf{term\_ok}\ \mathit{sig}\ \mathit{tm}\end{array}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ \mathit{tm})}$$

We include the condition that the term be well-typed in the combination case, and only allow abstractions of variables, hence we have $\vdash\ \mathsf{term\_ok}\ \mathit{sig}\ t \Rightarrow \mathsf{welltyped}\ t$.

We say a signature is standard if it maps the primitive type operators—function spaces and Booleans—and the primitive constant—equality—in the way we would expect:

$$\begin{array}{l}\mathsf{is\_std\_sig}\ \mathit{sig} \iff\\ \ \ \mathsf{lookup}\ (\mathsf{tysof}\ \mathit{sig})\ \texttt{"fun"} = \mathsf{Some}\ 2\ \land\\ \ \ \mathsf{lookup}\ (\mathsf{tysof}\ \mathit{sig})\ \texttt{"bool"} = \mathsf{Some}\ 0\ \land\\ \ \ \mathsf{lookup}\ (\mathsf{tmsof}\ \mathit{sig})\ \texttt{"="} = \mathsf{Some}\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ \mathsf{Bool}))\end{array}$$

We have a straightforward condition for a theory to be well-formed: all its components are well-formed and the axioms are Boolean terms.

$$\begin{array}{l}\mathsf{theory\_ok}\ \mathit{thy} \iff \\ \quad (\forall ty.\ ty \in \mathsf{range}\ (\mathsf{tmsof}\ \mathit{thy}) \Rightarrow \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy})\ ty) \land \\ \quad (\forall p.\ p \in \mathsf{axsof}\ \mathit{thy} \Rightarrow \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{thy})\ p \land p\ \mathsf{has\_type}\ \mathsf{Bool}) \land \\ \quad \mathsf{is\_std\_sig}\ (\mathsf{sigof}\ \mathit{thy})\end{array}$$

(Here $\mathsf{tmsof}\ \mathit{thy}$ is shorthand for $\mathsf{tmsof}\ (\mathsf{sigof}\ \mathit{thy})$, and similarly for the types.)

4.2 Semantics

The idea behind the standard (e.g., Pitts [27]) semantics for HOL is to interpret types as non-empty sets and terms as their elements. Equality and function application and abstraction are interpreted as in set theory, and a sequent is considered true if its interpretation is the true element of the set of Booleans. Semantics for HOL in this style are a mostly straightforward example of model theory.

The most fiddly parts of the semantics arise when dealing with polymorphic constants and type operators with arguments, followed closely by issues arising from substitution and type instantiation (which we covered in Section 4.1.3). Polymorphism is especially relevant to being able to support defined constants. The approach we have taken here is to keep the treatment of constants and type operators separate from the semantics of the lambda-calculus terms, by parametrising the semantics by an interpretation, so that both pieces remain simple.

Our goal is to show how we give semantics to sequents (and their component parts) in a theory. The ultimate notion we are aiming for is validity, $(\mathit{thy},h) \models c$, which says that the semantics of c is true whenever the semantics of all the h are true and the axioms of *thy* are satisfied. Validity quantifies over, and hence does not need to mention the semantic parameters that give meaning to constants and variables. But these parameters, called interpretations and valuations, are required for building the definition of validity out of semantics for the component parts of a sequent.

The details of our semantic apparatus are new, compared to Harrison's work [10] on HOL semantics in a fixed context without definitions, and are inspired by the second author's specification [3] of ProofPower HOL's logic.

Semantics of Types The meaning of a HOL type is a non-empty set. Thus, we require type valuations (τ) to assign type variables to non-empty sets.

$$\mathsf{is\_type\_valuation}\ \tau \iff \forall x.\ \mathsf{inhabited}\ (\tau\ x)$$

The type signature (*tysig* below) says what the type operators are and how many arguments they each expect. A type assignment (δ) gives semantics to type operators; we require it to assign correct applications of type operators to non-empty sets.

$$\begin{array}{l}\mathsf{is\_type\_assignment}\ \mathit{tysig}\ \delta \iff \\ \quad \mathsf{every} \\ \quad\ (\lambda\ (\mathit{name},\mathit{arity}). \\ \quad\quad \forall ls.\ \mathsf{length}\ ls = \mathit{arity} \land \mathsf{every\ inhabited}\ ls \Rightarrow \mathsf{inhabited}\ (\delta\ \mathit{name}\ ls)) \\ \quad\ \mathit{tysig}\end{array}$$

The semantics of types simply maps the type valuation and type assignment through the type, as follows:

$$\begin{aligned}&\mathsf{typesem}\ \delta\ \tau\ (\mathsf{Tyvar}\ s) = \tau\ s\\ &\mathsf{typesem}\ \delta\ \tau\ (\mathsf{Tyapp}\ \mathit{name}\ \mathit{args}) = \delta\ \mathit{name}\ (\mathsf{map}\ (\mathsf{typesem}\ \delta\ \tau)\ \mathit{args})\end{aligned}$$

Observe that since the type assignment (δ) is a function in HOL, there are not necessarily any set-theoretic functions involved in the semantics of type operators.

Semantics of Terms The meaning of a HOL term is an element of the meaning of its type. Thus, a term valuation (σ) must assign each variable to an element of the meaning of its type. To speak of valid types and their meanings requires a type signature and type assignment, so the notion of a term valuation depends on them.

$$\begin{aligned}&\mathsf{is\_term\_valuation}\ \mathit{tysig}\ \delta\ \tau\ \sigma \iff\\ &\quad\forall v\ ty.\ \mathsf{type\_ok}\ \mathit{tysig}\ ty \Rightarrow \sigma\ (v,ty) \lessdot \mathsf{typesem}\ \delta\ \tau\ ty\end{aligned}$$

The constant signature (*tmsig* below) gives the names and types of the constants, and a term assignment (γ) must assign each constant to an element of the meaning of the appropriate type. This picture is complicated by the fact that constants may be polymorphic (that is, their types may contain type variables), so a term assignment takes not only the name of the constant but a list of meanings for the type variables, and the condition it must satisfy quantifies over type valuations. For any type valuation, the term assignment must assign the constant under that type valuation to an element of the meaning of the constant's type.

$$\begin{aligned}&\mathsf{is\_term\_assignment}\ \mathit{tmsig}\ \delta\ \gamma \iff\\ &\ \mathsf{every}\\ &\quad(\lambda\ (\mathit{name},\mathit{ty}).\\ &\qquad\forall \tau.\\ &\qquad\ \mathsf{is\_type\_valuation}\ \tau \Rightarrow\\ &\qquad\quad \gamma\ \mathit{name}\ (\mathsf{map}\ \tau\ (\mathsf{sorted\_tyvars}\ \mathit{ty})) \lessdot \mathsf{typesem}\ \delta\ \tau\ \mathit{ty})\ \mathit{tmsig}\end{aligned}$$

The semantics of terms is defined recursively as follows. For variables, we simply apply the valuation.

$$\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Var}\ x\ \mathit{ty}) = \sigma\ (x,\mathit{ty})$$

For constants, we apply the interpretation but need to match the instantiated type of the constant against its generic type, that is, the type given for the constant in the signature. This is done using the instance function, explained in the next paragraph.

$$\begin{aligned}&\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Const}\ \mathit{name}\ \mathit{ty}) =\\ &\quad\mathsf{instance}\ \mathit{tmsig}\ (\delta,\gamma)\ \mathit{name}\ \mathit{ty}\ \tau\end{aligned}$$

Assuming[9] the given type is an instance of the generic type under some type substitution *i*, instance applies the term assignment for the constant passing the meanings of the types to which the type variables are bound under *i*.

$$\begin{aligned}&\mathsf{lookup}\ \mathit{tmsig}\ \mathit{name} = \mathsf{Some}\ \mathit{ty}_0 \Rightarrow\\ &\ \mathsf{instance}\ \mathit{tmsig}\ (\delta,\gamma)\ \mathit{name}\ (\mathsf{tysubst}\ i\ \mathit{ty}_0)\ \tau =\\ &\quad\gamma\ \mathit{name}\ (\mathsf{map}\ (\mathsf{typesem}\ \delta\ \tau \circ \mathsf{tysubst}\ i \circ \mathsf{Tyvar})\ (\mathsf{sorted\_tyvars}\ \mathit{ty}_0))\end{aligned}$$

[9] We leave unspecified the semantics of constants that are not in the signature or whose types do not match the type in the signature.

For applications, we simply use function application at the set-theoretic level.

$$\begin{array}{l}\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Comb}\ t_1\ t_2) = \\ \ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ t_1 \ {}'\ (\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ t_2)\end{array}$$

Similarly, for abstractions we create a set-theoretic function that takes an element, *m*, of the meaning of the type of the abstracted variable to the meaning of the body under the appropriately extended valuation.

$$\begin{array}{l}\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ b) = \\ \ \mathsf{Abstract}\ (\mathsf{typesem}\ \delta\ \tau\ \mathit{ty})\ (\mathsf{typesem}\ \delta\ \tau\ (\mathsf{typeof}\ b)) \\ \ \ (\lambda\, m.\ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,((x,\mathit{ty}) \mapsto m)\ \sigma)\ b)\end{array}$$

Above, $((x,ty) \mapsto m)\ \sigma$ means the valuation that returns m when applied to (x,ty) but otherwise acts like σ.

Standard Interpretations Our semantics so far makes no special treatment of HOL's primitive types and constants; indeed, we can neatly factor out the special treatment as a condition on interpretations. First, we collect the parameters for terms and types together. A pair of a type valuation and a term valuation is called a *valuation*. Similarly, a pair of a type assignment and a term assignment is called an *interpretation*.

$$\begin{array}{l}\mathsf{is\_valuation}\ \mathit{tysig}\ \delta\ (\tau,\sigma) \iff \\ \ \mathsf{is\_type\_valuation}\ \tau \wedge \mathsf{is\_term\_valuation}\ \mathit{tysig}\ \delta\ \tau\ \sigma \\ \mathsf{is\_interpretation}\ (\mathit{tysig},\mathit{tmsig})\ (\delta,\gamma) \iff \\ \ \mathsf{is\_type\_assignment}\ \mathit{tysig}\ \delta \wedge \mathsf{is\_term\_assignment}\ \mathit{tmsig}\ \delta\ \gamma\end{array}$$

We say an interpretation is *standard* if it interprets the primitive constants in the standard way; namely, functions types as set-theoretic function spaces, Booleans as the set of Booleans, and equality as set-theoretic equality (which is inherited from the meta-logic).

$$\begin{array}{l}\mathsf{is\_std\_type\_assignment}\ \delta \iff \\ \ (\forall\, \mathit{dom}\ \mathit{rng}.\ \delta\ \texttt{"fun"}\ [\mathit{dom};\ \mathit{rng}] = \mathsf{Funspace}\ \mathit{dom}\ \mathit{rng}) \wedge \\ \ \delta\ \texttt{"bool"}\ [\,] = \mathsf{Boolset}\end{array}$$

$$\begin{array}{l}\mathsf{is\_std\_interpretation}\ (\delta,\gamma) \iff \\ \ \mathsf{is\_std\_type\_assignment}\ \delta \wedge \\ \ \gamma\ \mathsf{interprets}\ \texttt{"="}\ \mathsf{on}\ [\texttt{"A"}]\ \mathsf{as} \\ \ \ (\lambda\, l. \\ \ \ \ \ \mathsf{Abstract}\ (\mathsf{head}\ l)\ (\mathsf{Funspace}\ (\mathsf{head}\ l)\ \mathsf{Boolset}) \\ \ \ \ \ \ \ (\lambda\, x.\ \mathsf{Abstract}\ (\mathsf{head}\ l)\ \mathsf{Boolset}\ (\lambda\, y.\ \mathsf{Boolean}\ (x = y))))\end{array}$$

The notation used above is defined as follows:

$$\begin{array}{l}\gamma\ \mathsf{interprets}\ \mathit{name}\ \mathsf{on}\ \mathit{args}\ \mathsf{as}\ f \iff \\ \ \forall\, \tau.\ \mathsf{is\_type\_valuation}\ \tau \Rightarrow \gamma\ \mathit{name}\ (\mathsf{map}\ \tau\ \mathit{args}) = f\ (\mathsf{map}\ \tau\ \mathit{args})\end{array}$$

We will only be concerned with standard interpretations.

Satisfaction We now turn to packaging the basic semantics of types and terms up and lifting it to the level of sequents. A sequent, containing both hypotheses and a conclusion, represents an implication. We say that an interpretation *satisfies* a sequent if the conclusion of the sequent is true whenever the hypotheses are (for all valuations). Precisely,

$$\begin{array}{l}(\delta,\gamma)\ \mathsf{satisfies}\ ((\mathit{tysig},\mathit{tmsig}),h,c) \iff \\ \ \forall v. \\ \quad \mathsf{is\_valuation}\ \mathit{tysig}\ \delta\ v \wedge \mathsf{every}\ (\lambda\, t.\ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ v\ t = \mathsf{True})\ h \Rightarrow \\ \qquad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ v\ c = \mathsf{True}\end{array}$$

We defer checking syntactic well-formedness of the sequent (for example, that c has type Bool) until the definition of validity below.

Modelling An interpretation *models* a theory if it is standard and satisfies the theory's axioms.

$$\begin{array}{l}i\ \mathsf{models}\ \mathit{thy} \iff \\ \ \mathsf{is\_interpretation}\ (\mathsf{sigof}\ \mathit{thy})\ i \wedge \mathsf{is\_std\_interpretation}\ i \wedge \\ \ \forall p.\ p \in \mathsf{axsof}\ \mathit{thy} \Rightarrow i\ \mathsf{satisfies}\ (\mathsf{sigof}\ \mathit{thy},[\,],p)\end{array}$$

Validity Finally, a sequent is *valid* if every model of the sequent's theory also satisfies the sequent itself.

$$\begin{array}{l}(\mathit{thy},h) \models c \iff \\ \ \mathsf{theory\_ok}\ \mathit{thy} \wedge \mathsf{every}\ (\mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{thy}))\ (c::h) \wedge \\ \ \mathsf{every}\ (\lambda\, p.\ p\ \mathsf{has\_type}\ \mathsf{Bool})\ (c::h) \wedge \mathsf{hypset\_ok}\ h \wedge \\ \ \forall i.\ i\ \mathsf{models}\ \mathit{thy} \Rightarrow i\ \mathsf{satisfies}\ (\mathsf{sigof}\ \mathit{thy},h,c)\end{array}$$

(hypset_ok is a syntactic check on the list of hypotheses, ensuring that it is sorted; see comments at the start of §4.3.1.)

4.3 Inference System

We have seen what HOL sequents look like and how they are to be interpreted in set theory. Now we turn to the inference system used to construct derivations of sequents.

Whereas the notion of a derivable sequent in a particular theory depends only on the abstract formulation (signature plus axioms) of theories, when it comes to extending the theory with new definitions (and other extensions) we introduce the more concrete notion of a *context*. A context is a linear sequence of theory-extending[10] updates. This formulation corresponds nicely to the actual behaviour of an implementation of the inference system (that is, a theorem prover).

We first look at the (within-theory) inference rules, then turn to the rules for theory extension (definitions and non-definitional updates). At the end of the subsection, we look at how we use theory extension to provide the initial axioms for the system.

[10] Our updates have a finer granularity than HOL4 theory segments or Isabelle/HOL theories, which usually include multiple updates in our sense.

4.3.1 Inference Rules

Recall that a sequent has the form $(thy,h)\ \mathsf{|-}\ c$ where thy is a theory, h is a set of hypotheses (Boolean terms) and c is the conclusion (another Boolean term). We represent the hypothesis set by a list whose elements are sorted (and hence distinct) according to a term ordering that equates only alpha-equivalent terms; we write the union of two such lists as $h_1 \cup h_2$, removal of an element c from h as $h \setminus c$, and the image of h under f as $\mathsf{map\_set}\ f\ h$.

In the HOL Light kernel, there are ten inference rules. Like Harrison, we define an abbreviation for equations since they appear frequently:

$$s == t = \mathsf{Comb}\ (\mathsf{Comb}\ (\mathsf{Equal}\ (\mathsf{typeof}\ s))\ s)\ t$$

We also use the following helper functions: $\mathsf{vfree\_in}\ v\ tm$ means v occurs free in tm, and $\mathsf{subst\_ok}\ sig\ ilist$ ensures only well-formed terms are substituted and only for variables of the same type. The rules are as follows:

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy\\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ t\end{array}}{(thy,[\,])\ \mathsf{|-}\ t == t}\ \mathsf{REFL}
\qquad
\frac{\begin{array}{c}(thy,h_1)\ \mathsf{|-}\ l == m_1\\ (thy,h_2)\ \mathsf{|-}\ m_2 == r\\ \mathsf{aconv}\ m_1\ m_2\end{array}}{(thy,h_1 \cup h_2)\ \mathsf{|-}\ l == r}\ \mathsf{TRANS}$$

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy\\ p\ \mathsf{has\_type}\ \mathsf{Bool}\\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ p\end{array}}{(thy,[p])\ \mathsf{|-}\ p}\ \mathsf{ASSUME}
\qquad
\frac{\begin{array}{c}(thy,h_1)\ \mathsf{|-}\ p == q\\ (thy,h_2)\ \mathsf{|-}\ p'\\ \mathsf{aconv}\ p\ p'\end{array}}{(thy,h_1 \cup h_2)\ \mathsf{|-}\ q}\ \mathsf{EQ\_MP}$$

$$\frac{\begin{array}{c}(thy,h_1)\ \mathsf{|-}\ c_1\\ (thy,h_2)\ \mathsf{|-}\ c_2\end{array}}{(thy,h_1 \setminus c_2 \cup h_2 \setminus c_1)\ \mathsf{|-}\ c_1 == c_2}\ \mathsf{DEDUCT\_ANTISYM}$$

$$\frac{\begin{array}{c}(thy,h_1)\ \mathsf{|-}\ l_1 == r_1\\ (thy,h_2)\ \mathsf{|-}\ l_2 == r_2\\ \mathsf{welltyped}\ (\mathsf{Comb}\ l_1\ l_2)\end{array}}{(thy,h_1 \cup h_2)\ \mathsf{|-}\ \mathsf{Comb}\ l_1\ l_2 == \mathsf{Comb}\ r_1\ r_2}\ \mathsf{MK\_COMB}$$

$$\frac{\begin{array}{c}\neg\mathsf{exists}\ (\mathsf{vfree\_in}\ (\mathsf{Var}\ x\ ty))\ h\\ \mathsf{type\_ok}\ (\mathsf{tysof}\ thy)\ ty\\ (thy,h)\ \mathsf{|-}\ l == r\end{array}}{(thy,h)\ \mathsf{|-}\ \mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ l == \mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ r}\ \mathsf{ABS}$$

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy\\ \mathsf{type\_ok}\ (\mathsf{tysof}\ thy)\ ty\\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ t\end{array}}{(thy,[\,])\ \mathsf{|-}\ \mathsf{Comb}\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ ty)\ t)\ (\mathsf{Var}\ x\ ty) == t}\ \mathsf{BETA}$$

$$\frac{\begin{array}{c}\mathsf{subst\_ok}\ (\mathsf{sigof}\ thy)\ ilist \\ (thy,h)\ \mathsf{|\text{-}}\ c\end{array}}{(thy,\mathsf{map\_set}\ (\mathsf{subst}\ ilist)\ h)\ \mathsf{|\text{-}}\ \mathsf{subst}\ ilist\ c}\ \mathsf{INST}$$

$$\frac{\begin{array}{c}\mathsf{every}\ (\mathsf{type\_ok}\ (\mathsf{tysof}\ thy))\ (\mathsf{map\ fst}\ tyin) \\ (thy,h)\ \mathsf{|\text{-}}\ c\end{array}}{(thy,\mathsf{map\_set}\ (\mathsf{inst}\ tyin)\ h)\ \mathsf{|\text{-}}\ \mathsf{inst}\ tyin\ c}\ \mathsf{INST\_TYPE}$$

There is one additional way for a sequent to be provable, namely, if it is an axiom of the theory:

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy \\ c \in \mathsf{axsof}\ thy\end{array}}{(thy,[])\ \mathsf{|\text{-}}\ c}$$

Thus the new piece of the sequent syntax, the theory, interacts with the inference system (which remains essentially as formalised by Harrison) only via the axioms of the theory and the checks that all types and terms respect the signature of the theory.

4.3.2 Theory Extension

In the previous section, we defined provability within a fixed theory. To complete the inference system, we also need mechanisms for changing the theory. At this point, we take a more concrete view of theories, which we call contexts, by considering the specific changes that can be made. For simplicity, we restrict ourselves to a linear sequence of extensions and do not allow redefinition or branching or merging of theories. This linear view is sufficient for HOL Light; a more complicated model might be necessary for theorem provers like HOL4, which supports redefinition, or Isabelle/HOL [33] which supports both redefinition and context merging.

In our linear view, each change is an update and updates come in two kinds: definitional extensions (the first two) and postulates (a.k.a. axiomatic extensions, the last three).

```
update =
  ConstSpec ((string × term) list) term
 | TypeDefn string term string string
 | NewType string num
 | NewConst string type
 | NewAxiom term
```

We call a list of such updates a *context*. From a context (*ctxt*) we can recover a theory (thyof *ctxt*) by calculating the constants and axioms introduced by each kind of update. Postulates simply add new constants or axioms to the theory. We will specify exactly how the definitional updates extend a theory shortly.

Some basic well-formedness conditions are required. The relation *upd* updates *ctxt* specifies when *upd* is a valid extension of *ctxt*. It can be considered as specifying the conditions under which an update is allowed to be made. For example, the conditions for the postulates, which simply ensure names remain distinct and that each piece of the postulate is well-formed, are shown below.

$$\frac{\mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt})}{\mathsf{NewType}\ \mathit{name}\ \mathit{arity}\ \mathsf{updates}\ \mathit{ctxt}} \qquad \frac{\begin{array}{c}\mathit{name} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \\ \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty}\end{array}}{\mathsf{NewConst}\ \mathit{name}\ \mathit{ty}\ \mathsf{updates}\ \mathit{ctxt}}$$

$$\frac{\begin{array}{c}\mathit{prop}\ \mathsf{has\_type}\ \mathsf{Bool} \\ \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{ctxt})\ \mathit{prop}\end{array}}{\mathsf{NewAxiom}\ \mathit{prop}\ \mathsf{updates}\ \mathit{ctxt}}$$

A context *extends* another if it is a series of valid updates applied to the other. The initial context, supporting only equality, can be specified as an extension of the empty context:

```
init_ctxt =
 [NewConst "=" (Fun (Tyvar "A") (Fun (Tyvar "A") Bool));
  NewType "bool" 0; NewType "fun" 2]
```

We turn now to the changes introduced by definitional extensions and the conditions on making them. Let us start with the definition of new types, represented by the update TypeDefn *name pred abs rep*. Here *name* is the name of the new type and *pred* is a predicate on an existing type called the *representing type*. The intuition behind the principle of type definition is to make the new type isomorphic to the subset of the representing type carved out by *pred*, which is required to be inhabited. Type definition introduces the new type and also two constants between the new type and the representing type, asserting a bijection via the following two axioms.

```
axioms_of_upd (TypeDefn name pred abs_name rep_name) =
 (let abs_type = Tyapp name (sorted_tyvars pred) in
  let rep_type = domain (typeof pred) in
  let abs = Const abs_name (Fun rep_type abs_type) in
  let rep = Const rep_name (Fun abs_type rep_type) in
  let a = Var "a" abs_type in
  let r = Var "r" rep_type
  in
   [Comb abs (Comb rep a) == a;
    Comb pred r == (Comb rep (Comb abs r) == r)])
```

As can be seen in the construction of *abs_type* above, the new type has a type argument for each of the type variables appearing in *pred*. The type variables are sorted (according to their name) to ensure a canonical order for the new type's arguments. The two introduced axioms assert that the introduced constants, *abs* and *rep*, are inverses (when restricted to elements of the representing type that satisfy *pred*).

The main condition on making a type definition is that the new type is non-empty. This is ensured by requiring a sequent asserting that the predicate holds of some witness. Additionally the predicate itself must not contain free variables, and the new names must not already appear in the context.

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt},[\,])\ \mathsf{|\text{-}}\ \mathsf{Comb}\ \mathit{pred}\ \mathit{witness}\\ \mathsf{closed}\ \mathit{pred}\\ \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt})\\ \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathit{abs} \neq \mathit{rep}\end{array}}{\mathsf{TypeDefn}\ \mathit{name}\ \mathit{pred}\ \mathit{abs}\ \mathit{rep}\ \mathsf{updates}\ \mathit{ctxt}}$$

We will prove that context extension by type definition is sound, that is, the axioms it introduces are not contradictory, in Section 5.1.2.

Finally, let us look at the definition of new term constants via our new generalised rule for constant specification. The update ConstSpec *eqs prop*, where *eqs* are equations with variables (varsof *eqs*) on the left, signifies introduction of a new constant for each of the variables which together share the defining specification *prop*. Thus *prop* (after substituting the new constants for the variables) is the sole new axiom:

$$\begin{array}{l}\mathsf{axioms\_of\_upd}\ (\mathsf{ConstSpec}\ \mathit{eqs}\ \mathit{prop}) =\\ \quad (\mathtt{let}\ \mathit{ilist}\ =\ \mathsf{consts\_for\_vars}\ \mathit{eqs}\ \mathtt{in}\ [\mathsf{subst}\ \mathit{ilist}\ \mathit{prop}])\end{array}$$

The purpose of the equations is to provide witnesses that *prop* is satisfiable, so the rule takes as input a theorem concluding *prop* assuming the equations. The complete list of conditions can be seen below:

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt},\mathsf{map}\ (\lambda\ (s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t)\ \mathsf{==}\ t)\ \mathit{eqs})\ \mathsf{|\text{-}}\ \mathit{prop}\\ \mathsf{every}\ (\lambda\ t.\ \mathsf{closed}\ t \wedge \mathsf{closed\_tyvars}\ t)\ (\mathsf{map}\ \mathsf{snd}\ \mathit{eqs})\\ \forall x\ ty.\ \mathsf{vfree\_in}\ (\mathsf{Var}\ x\ ty)\ \mathit{prop} \Rightarrow \mathsf{member}\ (x,ty)\ (\mathsf{varsof}\ \mathit{eqs})\\ \forall s.\ \mathsf{member}\ s\ (\mathsf{map}\ \mathsf{fst}\ \mathit{eqs}) \Rightarrow s \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{eqs})\end{array}}{\mathsf{ConstSpec}\ \mathit{eqs}\ \mathit{prop}\ \mathsf{updates}\ \mathit{ctxt}}$$

Here closed_tyvars t means that type variables appearing in t also appear in typeof t. The design of this principle of constant specification is explained in detail by Arthan [4]. We prove its soundness in Section 5.1.2. The rule of constant definition, which defines a new constant x to be equal to a term t, can be recovered as an instance of constant specification:

$$\mathsf{ConstDef}\ x\ t = \mathsf{ConstSpec}\ [(x,t)]\ (\mathsf{Var}\ x\ (\mathsf{typeof}\ t)\ \mathsf{==}\ t).$$

4.4 Axioms

We need some logical connectives and quantifiers to state two of the axioms asserted in HOL Light. Since they are generally useful, it is convenient to define them first before asserting the axioms.

4.4.1 Embedding Logical Operators

The connectives of propositional logic and universal and existential quantifiers (ranging over HOL types) can be defined as constants[11] in HOL. We define a list of updates each of which defines a connective or quantifier as it is defined in HOL Light, and show, by calculating out the semantics, that they all behave as intended.

Each of the connectives and quantifiers can be defined by an equation, so we use the simple ConstDef *name term* version of the rule for constant specification. The following function extends a context with definitions of the Boolean constants[12].

```
mk_bool_ctxt ctxt =
 ConstDef "~" NotDef::ConstDef "F" FalseDef::
   ConstDef "\\/" OrDef::ConstDef "?" ExistsDef::
   ConstDef "!" ForallDef::ConstDef "==>" ImpliesDef::
   ConstDef "/\\" AndDef::ConstDef "T" TrueDef::ctxt
```

Here the definition terms are as in HOL Light, for example,

```
ForallDef =
 Abs (Var "P" (Fun (Tyvar "A") Bool))
  (Var "P" (Fun (Tyvar "A") Bool) ==
   Abs (Var "x" (Tyvar "A")) (Const "T" Bool))

FalseDef =
 Comb (Const "!" (Fun (Fun Bool Bool) Bool))
  (Abs (Var "p" Bool) (Var "p" Bool))
```

(Pretty-printed as shallow embeddings, they are $(\forall) = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))$ and $\mathsf{F} = \forall p.\ p$.) We also specify the expected signature for constants with these names, for example,

```
is_forall_sig tmsig ⟺
 lookup tmsig "!" = Some (Fun (Fun (Tyvar "A") Bool) Bool)
```

and we show, by simple calculation, that sigof (mk_bool_ctxt *ctxt*) has the right signatures.

For the desired semantics of the Boolean constants, we refer to the connectives and quantifiers in the meta-logic (that is, for us, the logic of HOL4). For example, for implication and universal quantification, we have

```
is_implies_interpretation γ ⟺ γ interprets "==>" on [] as (λ y. Boolrel (⇒))
is_forall_interpretation γ ⟺
 γ interprets "!" on ["A"] as
  (λ l.
    Abstract (Funspace (head l) Boolset) Boolset
     (λ P. Boolean (∀x. x ⋖ head l ⇒ P ' x = True)))
```

[11] In HOL theorem prover parlance these are sometimes collectively known as the theory of Booleans.

[12] The names, "\\/" and "/\\", associated with OrDef and AndDef may appear to include extra backslashes, because backslashes must be escaped in strings in HOL4. The names are intended to be textual representations of $\vee$ and $\wedge$.

where the following helper functions interpret meta-level Booleans and relations on Booleans in our set theory:

$$\begin{array}{l}\mathsf{Boolean}\ b = \texttt{if}\ b\ \texttt{then}\ \mathsf{True}\ \texttt{else}\ \mathsf{False}\\ \mathsf{Boolrel}\ R =\\ \quad \mathsf{Abstract\ Boolset}\ (\mathsf{Funspace\ Boolset\ Boolset})\\ \quad\quad (\lambda\, p.\ \mathsf{Abstract\ Boolset\ Boolset}\ (\lambda\, q.\ \mathsf{Boolean}\ (R\ (p = \mathsf{True})\ (q = \mathsf{True}))))\end{array}$$

The desired interpretations for all the Boolean constants are collected together as follows.

$$\begin{array}{l}\mathsf{is\_bool\_interpretation}\ (\delta,\gamma) \iff\\ \quad \mathsf{is\_std\_interpretation}\ (\delta,\gamma) \wedge \mathsf{is\_true\_interpretation}\ \gamma \wedge\\ \quad \mathsf{is\_and\_interpretation}\ \gamma \wedge \mathsf{is\_implies\_interpretation}\ \gamma \wedge\\ \quad \mathsf{is\_forall\_interpretation}\ \gamma \wedge \mathsf{is\_exists\_interpretation}\ \gamma \wedge\\ \quad \mathsf{is\_or\_interpretation}\ \gamma \wedge \mathsf{is\_false\_interpretation}\ \gamma \wedge\\ \quad \mathsf{is\_not\_interpretation}\ \gamma\end{array}$$

The theorem we prove about the definitions of the Boolean constants says they have the desired semantics, that is, any interpretation that models a theory containing the definitions interprets the constants as specified by is_bool_interpretation.

$$\begin{array}{l}\vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ (\mathsf{mk\_bool\_ctxt}\ \mathit{ctxt})) \wedge\\ \quad i\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_bool\_ctxt}\ \mathit{ctxt})) \Rightarrow\\ \quad\quad \mathsf{is\_bool\_interpretation}\ i\end{array}$$

The semantics of a constant defined by an equation is uniquely specified, since that equation must be satisfied by any model of the definition. So, proving the theorem above is simply a matter of calculating out the semantics of the definitions of each of the constants and observing that they match the specification.

4.4.2 Statement of the Axioms

The standard library of HOL Light appeals to NewAxiom exactly three times, to assert the basic axioms of HOL that make it a classical logic and allow it to define the natural numbers. The axioms are: functional extensionality, choice, and infinity. Since the deeply-embedded syntax for the statements of the axioms is somewhat verbose, let us first look at their statements at the meta level:

- extensionality: $(\lambda\, x.\, f\ x) = f$
- choice: $P\ x \Rightarrow P\ ((\varepsilon)\ P)$
- infinity: $\exists (f : \mathit{ind} \to \mathit{ind}).\ \mathsf{ONE\_ONE}\, f \wedge \mathsf{ONTO}\, f$

While extensionality can be asserted in the initial context, the other two need additional constants to be added to the signature. For choice, we need to define implication, and to introduce the choice operator (ε above, but named `"@"` in the deep embedding.) For infinity, we need to introduce the type *ind* of individuals, and to define the existential quantifier[13], conjunction, and the ONE_ONE and ONTO functions.

[13] Axioms do not need to universally quantify their variables: free variables act as if universally quantified because of the INST rule of inference.

We define context-updating functions for each of the axioms, asserting the axiom with NewAxiom after introducing new constants if necessary. These are defined below, with some of the deeply-embedded syntax abbreviated (SelectAx, InfinityAx, OntoDef, and OneOneDef).

```
mk_eta_ctxt ctxt =
 NewAxiom
  (Abs (Var "x" (Tyvar "A"))
    (Comb (Var "f" (Fun (Tyvar "A") (Tyvar "B"))) (Var "x" (Tyvar "A"))) ==
   Var "f" (Fun (Tyvar "A") (Tyvar "B")))::ctxt

mk_select_ctxt ctxt =
 NewAxiom SelectAx::NewConst "@" (Fun (Fun (Tyvar "A") Bool) (Tyvar "A"))::ctxt

mk_infinity_ctxt ctxt =
 NewAxiom InfinityAx::NewType "ind" 0::ConstDef "ONTO" OntoDef::
   ConstDef "ONE_ONE" OneOneDef::ctxt
```

4.5 Comparison to Stateless HOL

The semantics (and inference system) we have just described cleanly separates the semantics of types from the semantics of terms. It also uses an explicit theory, with an interpretation, to track which constants are defined, what their semantics are, and the axioms. By contrast, Stateless HOL [34] puts types and terms in mutual recursion and has no separate theory parameter. Stateless HOL constants carry their definitions as syntactic tags (rather than in a separate signature), and the semantics interprets those tags directly (instead of using a separate interpretation parameter). We described the semantics for Stateless HOL in the ITP paper [16] on which this paper is based. By keeping the theory and its interpretation separate in the present work, we gain the following advantages:

- The semantics of types and terms are no longer in mutual recursion, and are simpler to understand individually.
- We can more naturally use functions (typesem and termsem) for the semantics, instead of mutually recursive relations.
- Specific parameters to support the axioms of choice and infinity are no longer required within the semantics. Instead, they are handled generically by the type and term interpretations, applied to "ind" and "@".
- We can support new axioms, beyond the initial three axioms asserted in HOL Light, in the same manner as the initial ones; the initial axioms are not baked into the semantics.
- The semantics of constants defined by new specification now properly captures the abstraction intended to be provided by that rule. The semantics are not tied to the specific witnesses given when the definition is made.
  In the Stateless HOL semantics, the semantics of a defined constant needs to be given in terms of the tag on the constant which provides the witnesses. By contrast in the current setup, the witnesses are only used in proving that the rule is sound (see Section 5.1.2). Since we now have an explicit interpretation of the constants, it can vary over many possible interpretations, constrained only by the axiom produced by the definition.

The primary motivation for Stateless HOL is the ability to "undo" definitions (this is achieved by soundly allowing simultaneous distinct definitions of constants with the same name). We did not take advantage of this ability in our verified implementation built under a Stateless HOL semantics [16], since we first translated to a stateful implementation. If we wanted to support undoing definitions, Stateless HOL is still an option worth consideration, but based on our experience we would first consider adding undo support to the context-based approach.

Additional advantages of Stateless HOL are that its kernel is purely functional, and therefore, we thought, would be easier to understand theoretically. We now claim that the difficulty of verifying a stateful implementation (as we do in Section 6.2) is smaller than the difficulty of giving semantics to the mutually recursive datatypes of Stateless HOL especially when the rules of definition are included.

As an alternative approach to purely functional kernels, the OpenTheory [13] kernel achieves purity by a careful redesign of the interface to the kernel while maintaining the traditional idea of a context-extending mechanism for making definitions. The semantics of an OpenTheory article is specified via a stateful virtual machine, but the higher-level operations on the resulting OpenTheory packages are pure, and names are carefully managed, so definitions never accidentally collide or go out of date. We expect to be able to verify an OpenTheory proof checker against our HOL semantics.

5 Soundness and Consistency

We have now seen HOL's inference system, which provides rules for proving sequents within a theory and updating that theory, and we have seen a specification of the meaning of such sequents: in particular, when they are considered valid. The main results of this section are that every sequent proved by the inference system is valid (soundness), with the corollary that some sequents cannot be proved, and that non-axiomatic extensions of contexts containing HOL's axioms are modelled (consistency).

Soundness holds for both the inference rules and the rules for theory extension, with the exception of NewAxiom. For an extension rule to be sound, it must put the inference system in a state whereby it continues to produce only valid sequents. We ground this idea by proving the continued existence of a model of the theory. Since we cannot prove NewAxiom sound in general, we also need to prove the three axioms used in HOL Light to set up the initial HOL context sound on a case-by-case basis.

We prove consistency for any non-axiomatic extensions of the following contexts:

$$\begin{aligned}&\mathsf{fhol\_ctxt} = \mathsf{mk\_select\_ctxt}\;(\mathsf{mk\_eta\_ctxt}\;(\mathsf{mk\_bool\_ctxt}\;\mathsf{init\_ctxt}))\\&\mathsf{hol\_ctxt} = \mathsf{mk\_infinity\_ctxt}\;\mathsf{fhol\_ctxt}\end{aligned}$$

The name of the first context above stands for "finitary HOL" since it omits the axiom of infinity. We name it separately because the consistency theorem we can prove of it has no assumptions apart from is_set_theory *mem*, which we saw in Section 3.2 can be discharged. The consistency theorem for the full hol_ctxt requires the set-theoretic axiom of infinity as an additional assumption.

5.1 Soundness

5.1.1 Inference Rules

The main soundness result for a fixed theory context is that every provable sequent is valid:

$$\vdash\ (\mathit{thy},h)\ \texttt{|-}\ c \Rightarrow (\mathit{thy},h) \models c$$

Our proof of this does not differ substantially form Harrison's, apart from our indirect treatment of substitution and instantiation via de Bruijn terms. Recall that by convention we elide on the theorem above the assumption $\mathsf{is\_set\_theory}\ \mathit{mem}$ and the *mem* argument passed to the validity relation ($\models$).

The result above is proved by induction on the provability relation ($\texttt{|-}$). Thus we have a case for each of HOL's inference rules, for example for $\mathsf{EQ\_MP}$:

$$\vdash\ (\mathit{thy},h_1) \models p \texttt{==} q \land (\mathit{thy},h_2) \models p' \land \mathsf{aconv}\ p\ p' \Rightarrow (\mathit{thy},h_1 \cup h_2) \models q$$

The proof for each case typically expands out the semantics of the sequents involved then invokes properties of the set theory. The case for the rule allowing an axiom to be proved is trivial by the definition of validity which assumes the theory is modelled.

The main work in proving soundness of the inference rules is establishing properties of the semantics of the operations used by the inference rules in constructing their conclusions. For example, for instantiation of type variables in terms, we show that instead of instantiating the term we can instantiate the valuations:

$$\begin{array}{l}\vdash\ \mathsf{term\_ok}\ (\mathit{tysig},\mathit{tmsig})\ \mathit{tm} \Rightarrow \\ \quad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{inst}\ \mathit{tyin}\ \mathit{tm}) = \\ \quad\ \ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma) \\ \quad\quad ((\lambda\, x.\ \mathsf{typesem}\ \delta\ \tau\ (\mathsf{tysubst}\ \mathit{tyin}\ (\mathsf{Tyvar}\ x))), \\ \quad\quad\ (\lambda\, (x,\mathit{ty}).\ \sigma\ (x,\mathsf{tysubst}\ \mathit{tyin}\ \mathit{ty})))\ \mathit{tm}\end{array}$$

This lemma is the main support for the $\mathsf{INST\_TYPE}$ case of the soundness theorem

$$\begin{array}{l}\vdash\ \mathsf{every}\ (\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy}))\ (\mathsf{map}\ \mathsf{fst}\ \mathit{tyin}) \land (\mathit{thy},h) \models c \Rightarrow \\ \quad (\mathit{thy},\mathsf{map\_set}\ (\mathsf{inst}\ \mathit{tyin})\ h) \models \mathsf{inst}\ \mathit{tyin}\ c\end{array}$$

since we need to establish conclusions about instantiations of terms from hypotheses about the terms themselves.

5.1.2 Theory Extension

The definition of new types and constants extends the context in which sequents may be proved, in particular it changes the signature of the theory and introduces new axioms depending on the kind of definition. Intuitively, we do not want such extensions to invalidate previously proved sequents, nor do we want the definitions to introduce an inconsistency.

The first property, preserving existing sequents, is easy to prove because the only dependence of a term's semantics on the theory is via the signature of constants and type operators that appear in the term. Thus, as shown below, satisfaction is preserved as long as the context grows monotonically, that is, without changing the signature of existing constants and type operators (the syntax $f \sqsubseteq f'$ means that f' agrees with f on everything in $\mathsf{domain}\,f$).

$$\begin{array}{l}\vdash\ \mathit{tmsig} \sqsubseteq \mathit{tmsig}' \wedge \mathit{tysig} \sqsubseteq \mathit{tysig}' \wedge \mathsf{every}\ (\mathsf{term\_ok}\ (\mathit{tysig},\mathit{tmsig}))\ (c{::}h) \wedge \\ \quad i\ \mathsf{satisfies}\ ((\mathit{tysig},\mathit{tmsig}),h,c) \Rightarrow \\ \quad\ \ i\ \mathsf{satisfies}\ ((\mathit{tysig}',\mathit{tmsig}'),h,c)\end{array}$$

All of our context-updating rules are monotonic, since we do not allow redefinition.

The second desired property of an update, not introducing an inconsistency, is what we shall designate as making the update *sound*. To be precise, we call an update sound if any model of a theory before the update can be extended to a model of the theory with the update:

$$\begin{array}{l}\mathsf{sound\_update}\ \mathit{ctxt}\ \mathit{upd} \iff \\ \ \forall i. \\ \quad i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \Rightarrow \\ \quad\ \ \exists i'.\ \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt})\ i\ i' \wedge i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathit{upd}{::}\mathit{ctxt}))\end{array}$$

The constant $\mathsf{equal\_on}$ helps formalise what we mean by one interpretation being an extension of another: they must be equal on terms and types in the previous context.

$$\begin{array}{l}\mathsf{equal\_on}\ \mathit{sig}\ i\ i' \iff \\ \ (\forall \mathit{name}.\ \mathit{name} \in \mathsf{domain}\ (\mathsf{tysof}\ \mathit{sig}) \Rightarrow \mathsf{tyaof}\ i'\ \mathit{name} = \mathsf{tyaof}\ i\ \mathit{name}) \wedge \\ \ \forall \mathit{name}.\ \mathit{name} \in \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{sig}) \Rightarrow \mathsf{tmaof}\ i'\ \mathit{name} = \mathsf{tmaof}\ i\ \mathit{name}\end{array}$$

It is now simply a matter of showing that each of our rules for updating the context are sound when their side conditions are met.

It is straightforward to show that NewType and NewConst are sound, because they do not introduce any new axioms. We simply need to extend the interpretation with some plausible interpretation of the data. The extended interpretation cannot be completely arbitrary, because to be a model of a theory an interpretation must be well-formed (that is, must satisfy is_interpretation). But a well-formed extension is always possible: for example mapping each new type to the set of Booleans and each new constant to an arbitrary member of the interpretation of its type (which is non-empty since the original theory is modelled). We thereby prove the following theorems.

$$\begin{array}{l}\vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \wedge \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \Rightarrow \\ \qquad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{NewType}\ \mathit{name}\ \mathit{arity}) \\ \vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \wedge \mathit{name} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty} \Rightarrow \\ \qquad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{NewConst}\ \mathit{name}\ \mathit{ty})\end{array}$$

Soundness of Type Definition A type definition, TypeDefn *name pred abs rep*, is sound if the two axioms it introduces (asserting the *abs* and *rep* constants form a bijection between the new type and the range of *pred*) can be made true by extending the original model with well-formed interpretations for the new type and two new constants. Such an extension is always possible, thus we can prove the following:

$$\begin{array}{l}\vdash\ (\mathsf{thyof}\ \mathit{ctxt},[\,])\ \mathsf{|-}\ \mathsf{Comb}\ \mathit{pred}\ \mathit{witness} \wedge \mathsf{closed}\ \mathit{pred} \wedge \\ \quad \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \wedge \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \\ \quad \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \mathit{abs} \neq \mathit{rep} \Rightarrow \\ \quad\ \ \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{TypeDefn}\ \mathit{name}\ \mathit{pred}\ \mathit{abs}\ \mathit{rep})\end{array}$$

The idea behind the proof is to interpret the new type as the subset of the representing type delineated by the semantics of *pred*, and to interpret the new constants as inclusion maps. When the *abs* constant is applied to a member of the representing type that is not in the new type, it simply picks an arbitrary element of the new type. The new type is guaranteed not to be empty by the theorem saying *pred* holds for some witness, which is required to make the type definition.

The proof of soundness of type definitions is the longest of the proofs about the rules for extension, taking around 400 lines of proof script compared to around 200 for constant specifications below and 40 for each of the other (non-definitional) updates. The reason is not that the soundness argument is significantly more complicated; rather, it is because the rule introduces many things (two axioms, two constants, and a type operator), where by contrast constant specification only introduces one axiom and introduces its constants uniformly; some work is required to calculate out the semantics of the equations in the axioms introduced by a type definition, and to ensure that each piece of the extension is well-formed.

Soundness of Constant Specification Specification of new constants, via ConstSpec *eqs prop*, introduces a single axiom, namely *prop* with its term variables replaced by the new constants, and is sound if the new constants are interpreted so as to make this axiom true. Such an interpretation is always possible when the side-conditions of the rule are met, thus we have the following:

$$
\begin{array}{l}
\vdash \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \land \\
\quad (\mathsf{thyof}\ ctxt, \mathsf{map}\ (\lambda\ (s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t) \mathrel{==} t)\ eqs)\ \mathrel{|\!-}\ prop \land \\
\quad \mathsf{every}\ (\lambda\ t.\ \mathsf{closed}\ t \land \mathsf{closed\_tyvars}\ t)\ (\mathsf{map}\ \mathsf{snd}\ eqs) \land \\
\quad (\forall x\ ty.\ \mathsf{vfree\_in}\ (\mathsf{Var}\ x\ ty)\ prop \Rightarrow \mathsf{member}\ (x,ty)\ (\mathsf{varsof}\ eqs)) \land \\
\quad (\forall s.\ \mathsf{member}\ s\ (\mathsf{map}\ \mathsf{fst}\ eqs) \Rightarrow s \notin \mathsf{domain}\ (\mathsf{tmsof}\ ctxt)) \land \\
\quad \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ eqs) \Rightarrow \\
\quad\quad \mathsf{sound\_update}\ ctxt\ (\mathsf{ConstSpec}\ eqs\ prop)
\end{array}
$$

The idea behind the proof is to interpret the new constants as the semantics of the witness terms (that is, map snd *eqs*) given in the input theorem that concludes *prop*. This works because then substitution of the new constants for the variables in *prop* has the same effect, semantically, as discharging the hypotheses of the input theorem.

The key lemmas required are about how the term semantics interacts with the interpretation and valuation. In particular, term substitution can be moved into the valuation; and, we can ignore extensions made to the interpretation when considering the semantics of a term that does not mention the new constants, since the semantics only cares about the interpretation of things in the signature.

$$
\begin{array}{l}
\vdash \mathsf{welltyped}\ tm \land \mathsf{subst\_ok}\ ilist \Rightarrow \\
\quad \mathsf{termsem}\ tmsig\ i\ (\tau,\sigma)\ (\mathsf{subst}\ ilist\ tm) = \\
\quad\quad \mathsf{termsem}\ tmsig\ i\ (\tau,\sigma \uplus \mathsf{map\_subst}\ (\mathsf{termsem}\ tmsig\ i\ (\tau,\sigma))\ ilist)\ tm
\end{array}
$$

$$
\begin{array}{l}
\vdash \mathsf{is\_std\_sig}\ (tysig,tmsig) \land \mathsf{term\_ok}\ (tysig,tmsig)\ tm \land \mathsf{equal\_on}\ (tysig,tmsig)\ i\ i' \Rightarrow \\
\quad \mathsf{termsem}\ tmsig\ i'\ v\ tm = \mathsf{termsem}\ tmsig\ i\ v\ tm
\end{array}
$$

Above, $f \uplus ls$ means the function that maps according to a binding in ls if it exists else defaults to applying f; and map_subst g *ilist* modifies the substitution *ilist*, which binds variables to terms, by applying g to all the terms.

Using these lemmas, we can reduce showing that the new axiom is satisfied to showing that *prop* is true under a valuation assigning the variables to the interpretations of the new constants. Since we interpreted the new constants as the witness terms corresponding to each variable, this then follows directly from the input theorem.

Sequences of Definitions Combining the results in this subsection, which cover soundness of each kind update except for NewAxiom, we prove that well-formed updates are sound.

$$\begin{array}{l}\vdash\ \mathit{upd}\ \mathsf{updates}\ \mathit{ctxt} \wedge \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \wedge (\forall p.\ \mathit{upd} \neq \mathsf{NewAxiom}\ p) \Rightarrow \\ \quad \mathsf{sound\_update}\ \mathit{ctxt}\ \mathit{upd}\end{array}$$

It is then a straightforward induction to show that a sequence of updates that do not introduce any axioms except via definitions preserve the existence of a model.

$$\begin{array}{l}\vdash\ \mathit{ctxt}_2\ \mathsf{extends}\ \mathit{ctxt}_1 \wedge \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}_1) \wedge i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}_1) \wedge \\ \quad (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt}_2 \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt}_1) \Rightarrow \\ \quad\quad \exists i'.\ \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt}_1)\ i\ i' \wedge i'\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}_2)\end{array}$$

5.2 Consistency

5.2.1 Axioms

We show that each of the axioms is consistent by proving: if the axiom is asserted in a theory that has a model, there is an extended interpretation that models the resulting theory. (This is the same idea as was formalised for sound_update, which we do not reuse since it only applies to a single update).

At this point, we drop our convention of eliding the is_set_theory *mem* assumption from our theorems, to make clear which of the axioms depend on which facts about the set theory.

The semantics of the axiom of extensionality is true because set-theoretic functions are extensional, and HOL functions are interpreted as set-theoretic functions. No constants are introduced, so the interpretation does not need extending.

$$\begin{array}{l}\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\ \quad \mathsf{is\_std\_sig}\ (\mathsf{sigof}\ \mathit{ctxt}) \Rightarrow \\ \quad\quad \forall i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \Rightarrow i\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_eta\_ctxt}\ \mathit{ctxt}))\end{array}$$

For the axiom of choice, the soundness theorem asserts existence of a model of the context extension produced by mk_select_ctxt, presuming the original context has a model, does not already define `"@"`, and correctly interprets implication. The theorem is as follows

$$\begin{array}{l}\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\ \quad \texttt{"@"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \mathsf{is\_implies\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \wedge \\ \quad \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \Rightarrow \\ \quad\quad \forall i. \\ \quad\quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \wedge \mathsf{is\_implies\_interpretation}\ (\mathsf{tmaof}\ i) \Rightarrow \\ \quad\quad\quad\quad \exists i'. \\ \quad\quad\quad\quad\quad \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt})\ i\ i' \wedge \\ \quad\quad\quad\quad\quad i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_select\_ctxt}\ \mathit{ctxt}))\end{array}$$

To prove this theorem, we need to provide an interpretation of the Hilbert choice constant, `"@"`, that satisfies the axiom: given a predicate on some type it should return an element of the type satisfying the predicate if one exists, or else an arbitrary element of the type. A suitable interpretation can be constructed using the choice operator in the meta-logic, that is, the logic of HOL4 (whose properties imply the set-theoretic axiom of choice, as shown at the end of Section 3.1).

For the axiom of infinity, the statement of the soundness theorem follows essentially the same structure as for the axiom of choice, except it uses mk_infinity_ctxt instead of mk_select_ctxt and assumes the set-theoretic axiom of infinity. Additionally, there are more assumptions about the context—that it contains certain constants, and does not already contain others—so we can define ONE_ONE and ONTO correctly. The theorem is as follows:

$$\begin{array}{l}
\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \land (\exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}) \Rightarrow \\
\quad \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \texttt{"ONTO"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \texttt{"ONE\_ONE"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \texttt{"ind"} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \land \\
\quad \mathsf{is\_implies\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathsf{is\_and\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \mathsf{is\_forall\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathsf{is\_exists\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \mathsf{is\_not\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \Rightarrow \\
\quad\ \forall i. \\
\quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \land i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \\
\quad\quad \mathsf{is\_implies\_interpretation}\ (\mathsf{tmaof}\ i) \land \\
\quad\quad \mathsf{is\_and\_interpretation}\ (\mathsf{tmaof}\ i) \land \\
\quad\quad \mathsf{is\_forall\_interpretation}\ (\mathsf{tmaof}\ i) \land \\
\quad\quad \mathsf{is\_exists\_interpretation}\ (\mathsf{tmaof}\ i) \land \\
\quad\quad \mathsf{is\_not\_interpretation}\ (\mathsf{tmaof}\ i) \Rightarrow \\
\quad\quad\ \exists i'. \\
\quad\quad\quad \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt})\ i\ i' \land \\
\quad\quad\quad i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_infinity\_ctxt}\ \mathit{ctxt}))
\end{array}$$

To prove this theorem, we need to provide an interpretation of the type of individuals in such a way that the axiom of infinity is satisfied. We pick the infinite set *inf* whose existence is assumed. Then proving the theorem is simply a matter of calculating out the semantics and observing that the axiom holds because the set is infinite.

Having proved the soundness of each axiom separately, we can put them together within a single context and prove soundness for it and all its extensions (as long as they do not introduce further axioms). Recall the definitions of the contexts that assert the axioms:

$$\begin{array}{l}
\mathsf{fhol\_ctxt} = \mathsf{mk\_select\_ctxt}\ (\mathsf{mk\_eta\_ctxt}\ (\mathsf{mk\_bool\_ctxt}\ \mathsf{init\_ctxt})) \\
\mathsf{hol\_ctxt} = \mathsf{mk\_infinity\_ctxt}\ \mathsf{fhol\_ctxt}
\end{array}$$

We obtain the following results by combining the soundness theorems for the three axioms presented in this section with the result from Section 5.1.2 about theory extensions that do not add any further new axioms.

$$\begin{array}{l} \vdash \mathsf{is\_set\_theory}\ mem \Rightarrow \\ \quad \forall ctxt. \\ \quad\quad ctxt\ \mathsf{extends}\ \mathsf{fhol\_ctxt} \wedge \\ \quad\quad (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{fhol\_ctxt}) \Rightarrow \\ \quad\quad\ \ \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge \exists i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \end{array}$$

$$\begin{array}{l} \vdash \mathsf{is\_set\_theory}\ mem \wedge (\exists inf.\ \mathsf{is\_infinite}\ mem\ inf) \Rightarrow \\ \quad \forall ctxt. \\ \quad\quad ctxt\ \mathsf{extends}\ \mathsf{hol\_ctxt} \wedge \\ \quad\quad (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{hol\_ctxt}) \Rightarrow \\ \quad\quad\ \ \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge \exists i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt) \end{array}$$

The order in which the extensions are made ensure that the signature and interpretation assumptions of each of the soundness theorems for the axioms is satisfied.

5.2.2 Syntactic Consistency

We have seen that the inference system for HOL (as implemented by HOL Light) is sound in that every sequent it derives is semantically valid. As a corollary, we can show that there are some sequents which cannot be derived (since some sequents are not valid). Our strategy for proving this syntactic notion of consistency is to use the fact, sometimes called semantic consistency, that every theory produced by the inference system has a model (as proved in the previous section).

We define a consistent theory as one for which there are sequents one of which can be derived and the other which cannot. In fact, we choose particular sequents for this purpose, an equation of equal variables and an equation of potentially different variables:

$$\begin{array}{l} \mathsf{consistent\_theory}\ thy \iff \\ \quad (thy,[\,])\ \mathsf{|-}\ \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool} == \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool} \wedge \\ \quad \neg((thy,[\,])\ \mathsf{|-}\ \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool} == \mathsf{Var}\ \texttt{"y"}\ \mathsf{Bool}) \end{array}$$

Any theory with a model is consistent, as the following lemma demonstrates.

$$\begin{array}{l} \vdash \mathsf{is\_set\_theory}\ mem \Rightarrow \\ \quad \forall thy.\ \mathsf{theory\_ok}\ thy \wedge (\exists i.\ i\ \mathsf{models}\ thy) \Rightarrow \mathsf{consistent\_theory}\ thy \end{array}$$

We prove the lemma by appeal to soundness: if the sequent equating two different variables were derivable, it would be valid (by soundness), and since the theory has a model it would be true in that model under every valuation. But it is not true under the valuation that sends Var "x" Bool to True and Var "y" Bool to False, so it cannot be derivable. As for the sequent equating equal variables, it is derivable as an instance of the REFL rule.

Combining the lemma above with the results in the previous section, the following consistency theorems follow immediately.

$$
\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ mem \Rightarrow \\
\quad \forall ctxt. \\
\quad\ \ ctxt\ \mathsf{extends}\ \mathsf{fhol\_ctxt} \land \\
\quad\ \ (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{fhol\_ctxt}) \Rightarrow \\
\quad\ \ \ \mathsf{consistent\_theory}\ (\mathsf{thyof}\ ctxt)
\end{array}
$$

$$
\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ mem \land (\exists inf.\ \mathsf{is\_infinite}\ mem\ inf) \Rightarrow \\
\quad \forall ctxt. \\
\quad\ \ ctxt\ \mathsf{extends}\ \mathsf{hol\_ctxt} \land \\
\quad\ \ (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{hol\_ctxt}) \Rightarrow \\
\quad\ \ \ \mathsf{consistent\_theory}\ (\mathsf{thyof}\ ctxt)
\end{array}
$$

The free variable *mem* in these theorems only appears in the assumptions, but those assumptions are of course necessary since we appealed to soundness, which depends on *mem* via the *i* models *thy* relation (and ultimately the semantics of terms and types).

6 Verifying the Kernel in CakeML

We have now seen that the HOL inference system, as specified by the provability relation (|-) and the rules for updating the context, is sound and consistent. Next, we turn our attention to producing a verified theorem prover implementing this sound inference system. Recall that our strategy is to produce the implementation in two steps: first, we define a theorem-prover kernel as recursive functions in a state-exception monad within the logic of HOL4, then we use an automated proof-producing technique to translate these recursive functions into code in the CakeML programming language. A preliminary description of this strategy can be found in our short paper [26] at ITP 2013.

6.1 The Monadic Functions

In implementations of HOL theorem provers, including the original OCaml implementation of HOL Light, the kernel module defines a datatype of theorems whose values correspond to the provable sequents of the HOL inference system. Our theorem datatype is defined with a single constructor as follows.

$$\mathit{thm}\ =\ \mathsf{Sequent}\ (\mathit{term\ list})\ \mathit{term}$$

In the implementation, the theory part of a sequent is not included on the theorem values, being instead embodied by the state of the theorem prover and the history of computations that led it into that state. The state of the theorem prover consists of the following four values, which will be implemented as references in CakeML.

$$
\begin{array}{l}
\mathit{state} = \\
\ \langle\ \mathsf{the\_type\_constants} : ((\mathit{string} \times \mathit{num})\ \mathit{list}); \\
\quad \mathsf{the\_term\_constants} : ((\mathit{string} \times \mathit{type})\ \mathit{list}); \\
\quad \mathsf{the\_axioms} : (\mathit{thm\ list}); \\
\quad \mathsf{the\_context} : (\mathit{update\ list})\ \rangle
\end{array}
$$

The first three fields of the state correspond to references found in the original OCaml implementation of HOL Light. The fourth field represents the current context. As we saw when describing the inference system, the type constants, term constants, and axioms can all be calculated from the context, so it is redundant to include them all in the state. For efficiency, and faithfulness to the original, we do not discard the other three references in favour of the context; rather, we think of the context as a “ghost” variable, which we will prove is always consistent with the rest of the state but which is not actually required for the implementation. For clarity, we leave it in the implementation rather than as an existentially quantified variable on our correctness theorems.

The monadic functions only raise two kinds of exceptions: failure with an error message, and, in the implementation of instantiation of type variables within a term, a “clash” exception for backtracking when unintended variable capture is detected.

$$\mathit{exn} \;=\; \mathsf{Fail}\ \mathit{string} \mid \mathsf{Clash}\ \mathit{term}$$

With our models of state and exceptions in place, we define our state-exception monad (α *M*) as follows.

$$\begin{aligned}\alpha\ \mathit{result} \;&=\; \mathsf{HolRes}\ \alpha \mid \mathsf{HolErr}\ \mathit{exn}\\ \alpha\ M = \mathit{state} \;&\to\; \alpha\ \mathit{result} \times \mathit{state}\end{aligned}$$

We define monadic bind as would be expected (that is, we either compute with the result or propagate the exception, and propagate the state in both cases), and make use of HOL4’s support for `do` notation (as found also in Haskell) for composition of monadic binds.

Let us look now at how the monadic functions are defined. For example, here is the function implementing the ASSUME rule of inference.

```
ASSUME tm =
 do
  ty ← type_of tm;
  bty ← mk_type ("bool",[]);
  if ty = bty then return (Sequent [tm] tm)
  else failwith "ASSUME: not a proposition"
 od
```

Here type_of *tm* computes the type of *tm* (failing on ill-typed terms), mk_type (*name*,*args*) constructs a type operator (failing if the number of arguments does not match the current signature in the state’s type constants reference), and failwith *msg* raises the Fail *msg* exception. We define a function like the above for each of the rules of inference and of definition, as well as all the requisite helper functions (like type_of), following the original OCaml implementation closely.

The monadic functions operate over the *thm* datatype, and re-use the underlying terms and types from the inference system. What we prove about them is that every computation preserves invariants on the values being computed. Importantly, the invariant on theorem values states that they are provable within the HOL inference system. The full list of invariants we use, each of which is parametrised by the current context, is given below.

$$\begin{array}{l}
\mathsf{TYPE}\ \mathit{ctxt}\ \mathit{ty} \iff \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty} \\
\mathsf{TERM}\ \mathit{ctxt}\ \mathit{tm} \iff \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{ctxt})\ \mathit{tm} \\
\\
\mathsf{THM}\ \mathit{ctxt}\ (\mathsf{Sequent}\ h\ c) \iff (\mathsf{thyof}\ \mathit{ctxt},h)\ \mathsf{|\text{-}}\ c \\
\\
\mathsf{STATE}\ \mathit{ctxt}\ \mathit{state} \iff \\
\quad \mathit{ctxt} = \mathit{state}.\mathsf{the\_context} \wedge \mathit{ctxt}\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \\
\quad \mathit{state}.\mathsf{the\_type\_constants} = \mathsf{type\_list}\ \mathit{ctxt} \wedge \\
\quad \mathit{state}.\mathsf{the\_term\_constants} = \mathsf{const\_list}\ \mathit{ctxt}
\end{array}$$

The STATE invariant requires the current context to be a valid extension (of init_ctxt). Thus preserving the STATE invariant entails only making valid updates to the context.

For each monadic function, we prove that good inputs produce good output. For example, for the ASSUME function, we prove that, if the input is a good term and the state is good, then the state will be unchanged on exit and if the function returned successfully, the return value is a good theorem:

$$\begin{array}{l}
\vdash\ \mathsf{TERM}\ \mathit{ctxt}\ \mathit{tm} \wedge \mathsf{STATE}\ \mathit{ctxt}\ s \wedge \mathsf{ASSUME}\ \mathit{tm}\ s = (\mathit{res},s') \Rightarrow \\
\quad\ s' = s \wedge \forall \mathit{th}.\ \mathit{res} = \mathsf{HolRes}\ \mathit{th} \Rightarrow \mathsf{THM}\ \mathit{ctxt}\ \mathit{th}
\end{array}$$

This theorem is proved by stepping through the definition of ASSUME, and, at the crucial point where a Sequent value is created, observing that the assumptions for the ASSUME clause of the provability (|-) relation are satisfied, so the THM invariant holds.

We prove a similar theorem for each function in the kernel, showing that they implement the HOL inference system correctly. As another example, consider the rule for constant specification, which may update the state. We prove that the new state still satisfies our invariants, as does the returned theorem.

$$\begin{array}{l}
\vdash\ \mathsf{THM}\ \mathit{ctxt}\ \mathit{th} \wedge \mathsf{STATE}\ \mathit{ctxt}\ s \Rightarrow \\
\quad \mathtt{case}\ \mathsf{new\_specification}\ \mathit{th}\ s\ \mathtt{of} \\
\quad\ (\mathsf{HolRes}\ \mathit{th},s') \Rightarrow \exists \mathit{upd}.\ \mathsf{THM}\ (\mathit{upd}::\mathit{ctxt})\ \mathit{th} \wedge \mathsf{STATE}\ (\mathit{upd}::\mathit{ctxt})\ s' \\
\quad \mid\ (\mathsf{HolErr}\ \mathit{exn},s') \Rightarrow s' = s
\end{array}$$

6.2 Producing CakeML

The monadic functions constitute a *shallow embedding* of a theorem-prover-kernel implementation, because they are functions whose semantics is given implicitly by HOL (as implemented by HOL4): consider the fact that the ASSUME function has type *term* → *thm M*. In this section, we turn to production of a *deep embedding* of the same functions, with semantics given explicitly as the operational semantics of the CakeML programming language. In the deep embedding, the ASSUME function is a piece of syntax; its type is *dec*, that is, a CakeML declaration. Furthermore, since CakeML supports references and exceptions directly, the functions no longer need to be monadic.

We produce the deep embeddings from our shallow embeddings automatically, using the proof-producing translation technique described in Myreen and Owens [25]. The result of translation is syntax and a certificate theorem. For example, for the monadic ASSUME function, we obtain the following syntax (shown as abbreviated CakeML abstract syntax):

```
⊢ nth_element 99 ml_hol_kernel_decls =
    Dlet (Pvar "assume")
     (Fun "v3"
       (Let (Some "v2")
         (App Opapp [Var (Short "type_of"); Var (Short "v3")])
         (Let (Some "v1")
           (App Opapp [Var (Short "mk_type"); Con None [... ... ; ... ]])
           (If (App Equality [Var (... ... ); ... ... ])
             (Con (Some (Short "Sequent"))
               [Con (... ... ) [... ... ; ... ]; Var (... ... )])
             (Raise
               (Con (Some (Short "Fail"))
                 [Lit (StrLit "ASSUME: not a proposition")])))))))
```

The same code pretty-printed in CakeML concrete syntax:

```
fun assume v3 =
  let val v2 = type_of v3
      val v1 = mk_type ("bool",[])
  in
    if (v2 = v1)
    then (Sequent([v3],v3))
    else (raise Fail("ASSUME: not a proposition"))
  end;
```

The meaning of the declaration above is specified by CakeML's operational semantics. The certificate theorem produced by translation connects evaluation of the declaration to the monadic function ASSUME:

$$\begin{aligned}&\vdash \mathsf{DeclAssum}\ (\mathsf{Some}\ \texttt{"Kernel"})\ \mathsf{ml\_hol\_kernel\_decls}\ \mathit{env}\ \mathit{tys} \Rightarrow\\ &\quad \mathsf{EvalM}\ \mathit{env}\ (\mathsf{Var}\ (\mathsf{Short}\ \texttt{"assume"}))\\ &\quad\ ((\mathsf{PURE\ TERM\_TYPE} \xrightarrow{\mathsf{M}} \mathsf{HOL\_MONAD\ THM\_TYPE})\ \mathsf{ASSUME})\end{aligned}$$

Here, DeclAssum *mn decls env tys* means that (*env*,*tys*) is the environment (of declared values and types) obtained by evaluating the list *decls* of declarations within a module *mn*; and, EvalM *env exp P* means that evaluation of the expression *exp* in environment *env* terminates and produces a result satisfying the refinement invariant *P*. In the theorem above, the refinement invariant takes the form $(A \xrightarrow{\mathsf{M}} B)\,f$, which specifies a closure that, when applied to an input value satisfying *A*, terminates and produces an output value, which will satisfy *B*, according to the monadic function *f*.

To understand the guarantee provided by the certificate theorem, let us unpack the refinement invariant a little further. The thing to remember is that the refinement invariants specify the relationship between certain HOL terms (values in the shallow embedding) and deeply-embedded CakeML values. For example, the following fact demonstrates how the THM_TYPE invariant relates values of type *thm* to CakeML values (Conv denotes a CakeML value made from application of a CakeML constructor):

$$\begin{array}{l}\vdash \ \mathsf{LIST\_TYPE}\ \mathsf{TERM\_TYPE}\ []\ v_1 \land \mathsf{TERM\_TYPE}\ tm\ v_2 \Rightarrow \\ \quad \mathsf{THM\_TYPE}\ (\mathsf{Sequent}\ []\ tm) \\ \quad\ (\mathsf{Conv}\ (\mathsf{Some}\ (\texttt{"Sequent"},\mathsf{TypeId}\ (\mathsf{Long}\ \texttt{"Kernel"}\ \texttt{"thm"})))\ [v_1;\ v_2])\end{array}$$

Here, we have CakeML values, v_1 and v_2, that are related by the refinement invariants for terms (and lists of terms) to the empty list and a term *tm*, and they are used to put together a CakeML value that is related to the theorem Sequent [] *tm*. The operators PURE and HOL_MONAD extend these refinement invariants to also relate the CakeML store (that is, the contents of references) and result (normal termination or raised exception) to the corresponding parts of the state-exception monad. (PURE lifts non-monadic values into the monad while HOL_MONAD works directly on a monadic value.)

Finally, $(A \xrightarrow{\mathsf{M}} B)\, f$ is the refinement invariant for monadic functions as explained earlier. Thus using the certificate theorem for ASSUME we can prove in CakeML's operational semantics that the return value of any successful application of the deeply-embedded `assume` function will be related by the THM_TYPE invariant to the corresponding application of the monadic ASSUME function. And, as we saw in the previous section, the result of applying the monadic ASSUME function is related by the THM invariant to a sequent in the sound inference system (|-).

We have certificate theorems like this for every function in the CakeML implementation of the HOL Light kernel. It is on that basis that we make the claim that our kernel only produces theorem values that correspond to true sequents according to the semantics of HOL.

7 Related Work

The themes of the work described in this paper are formalising (and mechanising) the syntax and semantics of logic, and verifying (or producing verified) theorem-prover implementations. We factor our review of prior work in these areas by the particular logic under consideration.

Higher-Order Logic There has been prior work on producing a formal (mechanised) specification of the semantics of HOL. The documentation for HOL4 includes a description, originally due to Pitts [27], of the semantics of HOL. However, this description is given in the traditional semiformal style of the mathematical logic literature. In the early 1990s, the development of the ProofPower logical kernel was informed by a formal specification in ProofPower-HOL of the proof development system, including a formalisation of the HOL language, logic and semantics. However, no formal proofs were carried out. The present work found several errors in the ProofPower formalisation of the semantics (all now corrected [3]). Pioneering work by von Wright [31] includes a mechanised formalisation of the syntax of HOL and its inference system (though no semantics). As discussed in Section 1, Harrison's work [10] on a proof in HOL Light of the consistency of the HOL logical kernel without definitions formed the starting point for the present work (initially, [26, 16]).

Krauss and Schropp [15] have formalised a translation from HOL to set theory, automatically producing proofs in Isabelle/ZF [33]. Their motivation was to revive Isabelle/ZF by importing Isabelle/HOL proofs into it, but this task necessitates formalising an interpretation of HOL in set theory for which they use the standard approach (as we did) sending types to non-empty sets and terms to elements of their types. Although the Isabelle/HOL logic is slightly more complicated than the HOL we described, due to type classes and overloading, they remove the extra features in a preprocessing phase. They handle type definitions and (equational) constant definitions by making equivalent definitions in Isabelle/ZF, which supports Isabelle's general facility for definitions.

Dependent Type Theory Barras [5] has formalised a reduced version of the calculus of inductive constructions, the logic used by the Coq proof assistant [6], giving it a semantics in set theory and formalising a soundness proof in Coq itself. The approach is modular, and Wang and Barras [32] have extended the framework and applied it to the calculus of constructions plus an abstract equational theory.

Anand and Rahli [1] have formalised the semantics of NuPRL's type theory and proved soundness for its sequent calculus. The mechanisation is carried out within Coq. The semantics of NuPRL is rather more complex than of HOL, so its formalisation is impressive; on the other hand, they do not yet go so far as producing a verified implementation, but allude to the interesting possibility of producing it directly from the proof term for the soundness of the inference system.

First-Order Logic Myreen and Davis [24] formalised Milawa's ACL2-like first-order logic and proved it sound using HOL4. This soundness proof for Milawa produced a top-level theorem which states that the machine-code which runs the prover will only print theorems that are true according to the semantics of the Milawa logic. Since Milawa's logic is weaker than HOL, it fits naturally inside HOL without encountering of the delicate foundational territory necessitating our $\mathsf{is\_set\_theory}\ \mathit{mem}$ and $\exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}$ assumptions.

Other noteworthy prover verifications include a simple first-order tableau prover by Ridge and Margetson [28] and a SAT solver algorithm with many modern optimizations by Marić [19].

8 Conclusion

A theorem prover is a computer program whose correctness can be understood at many levels. At the highest level, we focus solely on the logic, which should be consistent, and the particular inference system, which should be sound. At the next level down, we consider whether the inference system is implemented correctly, that is, whether the (abstract) computations performed by the theorem prover correspond to construction of derivations in the inference system. The remaining levels all concern correct implementation of those computations at more concrete levels of abstraction, from a high-level programming language down to hardware. In this paper we have dealt with correctness of a theorem prover for higher-order logic (HOL) spanning all the levels between consistency of the logic itself and implementation in a high-level programming language (CakeML), within a single mechanically-checked formalisation.

We have gone further than the previous work in this vein in two directions: the coverage of the logic formalised and the concreteness of the theorem-prover implementation verified. Our formalisation, with full support for making extensions to the context, now covers all of HOL as it is implemented by real theorem provers. Our implementation is a deeply-embedded program verified against the operational semantics of a realistic programming language. On both fronts, however, the end of the line has not been reached: one might like to verify a more sophisticated approach to contexts (such as the one implemented by Isabelle [33]), and a more concrete implementation (for example, in machine code). Additionally, we have so far only verified the kernel of a theorem prover, and would like to extend the result to a complete theorem prover, which means formally validating the LCF design [22] by reasoning about the guarantees provided by a protected (abstract) type in CakeML.

In constructing a formal specification of the semantics of HOL that is suitable both for proofs about the logic and inference system, and for proofs about implementing that inference system, we faced several design decisions. The main theme of the lessons learned is to value explicitness and separation of concerns. Using an explicit theory context gives a simpler semantics than that of Stateless HOL, as was discussed in Section 4.5. Similarly, specifying the axioms of set theory with an explicit membership relation yields a development that is easier to work with than the theoretically equivalent approach based solely on a cardinality assumption. And by specifying the set theory separately from defining an instance of it, we obtain a conservative approach using isolated assumptions about free variables rather than global axiomatic extensions. On a smaller scale, our choice to factor our reasoning about substitution and instantiation, which is complicated with name-carrying terms, through a separate small theory about de Bruijn terms led to simplifications.

Continuing the self-verification project initiated by Harrison [10] for HOL Light, our formalisation of HOL is conducted within HOL itself. It is common to cite Gödel's incompleteness theorems as making it meaningless or impossible for a logical system to be used to prove its own properties. However, this objection applies only to proofs of consistency. In the present work, our primary concern is with soundness, and what we have done is analogous to proving the soundness of first-order logic within a first-order formalisation of ZF set theory, which as standard logic textbooks (e.g., Mendelson [20]) show is well-known and uncontroversial.

The theorem prover we use for our mechanisation (HOL4) is distinct from the verified implementation we produce (in CakeML of a kernel based on HOL Light's). The implementation of HOL4 is not itself verified; one might wonder whether we gain anything by trusting one theorem prover to verify another of a similar (in fact lesser) complexity. While we acknowledge this objection, our reply is that HOL4 can be seen merely as a tool to help us organise our development if we consider the fact that our proofs can be exported from HOL4 (for example, via OpenTheory [13]) for independent checking. Thus although something ultimately needs to be trusted, we do not require it to be HOL4. A second reply is that the exercise of developing the formalisation leads us to clarify our thinking about the systems under consideration, and, on the implementation side, uncovers the kinds of bugs that are likely to occur in theorem provers in practice.

In future work, one might like to import our verified kernel into HOL Light for independent checking, or, more interestingly, to replay the proof in the verified CakeML implementation itself. Checking a correctness proof about its own concrete implementation would be closer to true self-verification than any theorem prover has yet achieved. Of course, such a check does not rule out the possibility that the theorem prover is not sound, because it might be broken in such a way that it fails to detect an incorrect correctness proof. But we would have high confidence in a theorem prover with such an ability (alongside other evidence for soundness, like a readable, concise implementation) and would expect the practical facility required for self-verification to be useful for tackling more ambitious software verification challenges.

Acknowledgements We thank Mike Gordon for helpful comments on drafts of this paper. We are grateful to John Harrison and Freek Wiedijk for providing inspiration for this project. We also thank the anonymous reviewers for carefully reading our submission and suggesting improvements.

References

1. Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Klein and Gamboa [14], pages 27–44.
2. Peter B Andrews. *An introduction to mathematical logic and type theory*, volume 27. Springer, 2002.
3. Rob Arthan. HOL formalised: Semantics. `http://www.lemma-one.com/ProofPower/specs/spc002.pdf`.
4. Rob Arthan. HOL constant definition done right. In Klein and Gamboa [14], pages 531–536.
5. Bruno Barras. Sets in Coq, Coq in sets. *J. Formalized Reasoning*, 3(1):29–48, 2010.
6. Yves Bertot. A short presentation of Coq. In Mohamed et al. [23], pages 12–16.
7. Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.
8. Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte fr Mathematik und Physik*, 38(1):173–198, 1931.
9. Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.
10. John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.
11. John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.
12. Leon Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15:81–91, 1950.
13. Joe Hurd. The OpenTheory standard theory library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.
14. Gerwin Klein and Ruben Gamboa, editors. *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.
15. Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 323–338. Springer, 2010.
16. Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In Klein and Gamboa [14], pages 308–324.
17. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.
18. Casimir Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1):161–171, 1921.
19. Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.
20. Elliott Mendelson. *Introduction to mathematical logic. 5th ed.* Boca Raton, FL: CRC Press, 5th ed. edition, 2009.
21. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
22. Robin Milner. LCF: A way of doing proofs with a machine. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 146–159. Springer, 1979.
23. Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.

24. Magnus O. Myreen and Jared Davis. The reflective Milawa theorem prover is sound - (down to the machine code that runs it). In Klein and Gamboa [14], pages 421–436.
25. Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.
26. Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL Light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 490–495. Springer, 2013.
27. Andrew M. Pitts. *The HOL System: Logic*, 3rd edition. `http://hol-theorem-prover.org/documentation.html`.
28. Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first-order logic. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2005.
29. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [23], pages 28–32.
30. Robert L. Vaught. *Set theory: An introduction*. Basel: Birkhäuser, 2nd edition, 1994.
31. Joakim von Wright. Representing higher-order logic proofs in HOL. *Comput. J.*, 38(2):171–179, 1995.
32. Qian Wang and Bruno Barras. Semantics of intensional type theory extended with decidable equational theories. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 653–667. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
33. Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [23], pages 33–38.
34. Freek Wiedijk. Stateless HOL. In Tom Hirschowitz, editor, *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009.*, volume 53 of *EPTCS*, pages 47–61, 2009.

Polymorphic Higher-Order Recursive Path Orderings

Jean-Pierre Jouannaud
LIX, École Polytechnique
and
Albert Rubio
Technical University of Catalonia

This paper extends the termination proof techniques based on reduction orderings to a higher-order setting, by defining a family of recursive path orderings for terms of a typed lambda-calculus generated by a signature of polymorphic higher-order function symbols. These relations can be generated from two given well-founded orderings, on the function symbols and on the type constructors. The obtained orderings on terms are well-founded, monotonic, stable under substitution and include β-reductions. They can be used to prove the strong normalization property of higher-order calculi in which constants can be defined by higher-order rewrite rules using first-order pattern matching. For example, the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering is polymorphic, in the sense that a single comparison allows to prove the termination property of all monomorphic instances of a polymorphic rewrite rule. Many non-trivial examples are given which exemplify the expressive power of these orderings. All have been checked by our implementation.

This paper is an extended and improved version of [Jouannaud and Rubio 1999]. Polymorphic algebras have been made more expressive than in our previous framework. The intuitive notion of a polymorphic higher-order ordering has now been made precise. The higher-order recursive path ordering itself has been made much more powerful by replacing the congruence on types used there by an ordering on types satisfying some abstract properties. Besides, using a restriction of Dershowitz's recursive path ordering for comparing types, we can integrate both orderings into a single one operating uniformly on both terms and types.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda Calculus and related Systems*; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems; I.2.4 [**Artificial Intelligence**]: Knowledge representation Formalisms and Methods—*Representations (Procedural and Rule-based)*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Automated termination prover tool, Gödel's polymorphic recursor, Higher-Order Rewriting, Termination orderings, Typed lambda calculus

This work was partly supported by the RNTL project AVERROES, France-Telecom, and the CICYT project LOGICTOOLS, ref. TIN2004-07925.
Project LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

Author's address: J.-P. Jouannaud, LIX, École Polytechnique, 91400 Palaiseau, FRANCE;
A. Rubio, Dep. LSI, Universitat Politètica de Catalunya, Jordi Girona 1-3, 08034 Barcelona, SPAIN.

1. INTRODUCTION

Rewrite rules are used in programming languages and logical systems, with three main goals: defining functions by pattern matching; encoding rule-based decision procedures; describing computations over lambda-terms used as a suitable abstract syntax for encoding functional objects like programs or specifications. ML and the current version of Coq based on the Calculus of Inductive Constructions [Coquand and Paulin-Mohring 1990] exemplify the first use. A prototype version of Coq exemplifies the second use [Blanqui et al. 2005]. Alf [Coquand 1994] and Isabelle [Paulson 1990] exemplify the third use. In Isabelle, rules operate on terms in β-normal, η-expanded form and use higher-order pattern matching. In ML and Coq, they operate on arbitrary terms and use first-order pattern-matching. Both kinds of rules target different needs, and should, of course, coexist.

The use of rules in logical systems is subject to three main meta-theoretic properties : type preservation, local confluence, and strong normalization. The first two are usually easy. The last one is difficult, requiring the use of sophisticated proof techniques based, for example, on Tait and Girard's computability predicate technique [Girard et al. 1989]. Our ambition is to provide a remedy for this situation by developing for the higher-order case the kind of automatable termination proof techniques that are available for the first-order case, of which the most popular one is the recursive path ordering [Dershowitz 1982].

Our contribution to this program is a reduction ordering for typed higher-order terms following a typing discipline including polymorphic sort constructors, which conservatively extends β-reductions for higher-order terms on the one hand, and on the other hand Dershowitz's recursive path ordering for first-order unisorted terms. In the latter, the precedence rule allows to decrease from the term $s = f(s_1, \ldots, s_n)$ to the term $g(t_1, \ldots, t_n)$, provided that (i) f is bigger than g in the given precedence on function symbols, and (ii) s is bigger than every t_i. For typing reasons, in our ordering the latter condition becomes: (ii) for every t_i, either s is bigger than t_i or some s_j is bigger than or equal to t_i. Indeed, we can instead allow t_i to be obtained from the subterms of s by computability preserving operations. Here, computability refers to Tait and Girard's strong normalization proof technique which we have used to show that our ordering is well-founded.

In a preliminary version of this work presented at the Federated Logic Conference in Trento, our ordering could only compare terms of equal types (after identifying sorts such as Nat or List). In the present version, the ordering is capable of ordering terms of *decreasing types*, the ordering on types being simply a slightly weakened form of Dershowitz's recursive path ordering. Several other improvements have been made, which allow to prove a great variety of practical examples. To hint at the strength of our ordering, let us mention that the polymorphic version of Gödel's recursor for the natural numbers is easily oriented. And indeed, our ordering can prove at once the termination property of all monomorphic instances of a polymorphic rewrite rule. Many other examples are given which exemplify the expressive power of the ordering.

In the literature, one can find several attempts at designing methods for proving strong normalization of higher-order rewrite rules based on ordering comparisons. These orderings are either quite weak [Loría-Sáenz and Steinbach 1992; Jouannaud and Rubio 1998], or need a heavy user interaction [van de Pol and Schwichtenberg 1995]. Besides, they operate on terms in η-long β-normal form, hence apply only to the higher-order rewriting "à la Nipkow" [Mayr and Nipkow 1998], based on higher-order pattern matching modulo $\beta\eta$. To our knowledge, our ordering is the first to operate on arbitrary higher-order terms, therefore applying to the other kind of rewriting, based on plain pattern matching. It is also the first to be automatable: an implementation is provided which does not require user-interaction. And indeed we want to stress several important features of our approach. Firstly, it can be seen as a way to lift an ordinal notation operating on a first-order language (here, the set of labelled trees ordered by the recursive path ordering) to an ordinal notation of higher type operating on a set of well-typed λ-expressions built over the first-order language. Secondly, the analysis of our ordering, based on Tait and Girard's computability predicate proof technique, leads to hiding this technique away, by allowing one to carry out future meta-theoretical investigations based on ordering comparisons rather than by a direct use of the computability predicate technique. Thirdly, a very elegant presentation of the whole ordering machinery obtained by integrating both orderings on terms and types into a single one operating on both kinds shows that this presentation can in turn be the basis for generalizing the ordering to dependent type calculi. Last, a modification of our ordering described in a companion paper can be used to prove strong normalization of higher-order rewrite rules operating on terms in η-long β-normal form and using higher-order pattern matching for firing rules [Jouannaud and Rubio 2006].

Described in Section 2, our framework for rewriting includes several novel aspects, among which two important notions of polymorphic higher-order rewriting and of polymorphic higher-order rewrite orderings. In order to define these notions precisely, we first introduce polymorphic higher-order algebras. Types will therefore play a central role in this paper, but the reader should be aware that the paper is by no means about polymorphic typing : it is about polymorphic higher-order orderings. In particular, we could have also considered inductive types, without having to face unpredictable difficulties. We chose not to do so in the present framework, which, we think, is already quite powerful and complex, to a point that those readers who feel no particular interest in typing technicalities should probably restrict their first reading of this section to the type system of Figure 1, Definition 2.28 and Theorem 2.29. The rest of the paper, we think, can be understood without knowing the details of the typing apparatus. The basic version of our ordering is defined and studied in Section 3, where several examples are given. The notion of computability closure [Blanqui et al. 1999] used to boost the expressiveness of the ordering is introduced and studied in Section 4. Our prototype implementation is discussed in Section 5. Related work and potential extensions are discussed in Section 6.

The reader is expected to have some familiarity with the basics of term rewriting systems [Dershowitz and Jouannaud 1990; Klop 1992; Baader and Nipkow 1998; Bezem et al. 2003] and typed lambda calculi [Barendregt 1990; 1992].

2. POLYMORPHIC HIGHER-ORDER ALGEBRAS

This section describes polymorphic higher-order algebras, a higher-order algebraic framework with typing "à la ML" which makes it possible to precisely define what are polymorphic higher-order rewriting and polymorphic higher-order rewrite orderings. Our target is indeed to have higher-order rewrite rules in the calculus of constructions and check their termination property by means of orderings, but, although our results address polymorphic typing, they do not scale up yet to dependent types.

We define our typing discipline in Sections 2.1 to 2.5, before investigating their properties in Section 2.6, and then define higher-order rewriting in Section 2.8 and finally polymorphic higher-order rewrite orderings in Section 2.9. We conclude in Section 2.10 that polymorphic higher-order rewrite orderings allow showing that the derivation relation defined by a set of polymorphic rewrite rules is well-founded by checking the rules for decreasingness.

2.1 Types

Given a set $\mathcal{S}$ of *sort symbols* of a fixed arity, denoted by $s : *^n \Rightarrow *$, and a denumerable set $\mathcal{S}^\forall$ of *type variables*, the set $\mathcal{T}_{\mathcal{S}^\forall}$ of (first-order) *types* is generated by the following grammar:

$$\mathcal{T}_{\mathcal{S}^\forall} := \alpha \mid s(\mathcal{T}^n_{\mathcal{S}^\forall}) \mid (\mathcal{T}_{\mathcal{S}^\forall} \rightarrow \mathcal{T}_{\mathcal{S}^\forall})$$
$$\text{for } \alpha \in \mathcal{S}^\forall \text{ and } s : *^n \Rightarrow * \ \in \mathcal{S}$$

We denote by $\mathcal{V}ar(\sigma)$ the set of type variables of the type $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$, and by $\mathcal{T}_{\mathcal{S}}$ the set of *monomorphic* or *ground* types, whose set of type variables is empty. Types in $\mathcal{T}_{\mathcal{S}^\forall} \setminus \mathcal{T}_{\mathcal{S}}$ are *polymorphic*.

Types are *functional* when headed by the → symbol, and *data types* when headed by a sort symbol. As usual, → associates to the right. We will often make explicit the functional structure of an arbitrary type τ by writing it in the *canonical form* $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$, with $n \geq 0$, where n is the *arity* of τ and σ is a data type or a type variable called *canonical output type* of τ. A *basic type* is either a data type or a type variable.

A *type substitution* is a mapping from $\mathcal{S}^\forall$ to $\mathcal{T}_{\mathcal{S}^\forall}$ extended to an endomorphism of $\mathcal{T}_{\mathcal{S}^\forall}$. We use postfix notation for their application to types or to other type substitutions. We denote by $\mathcal{D}om(\xi) = \{\alpha \in \mathcal{S}^\forall \mid \alpha\xi \neq \alpha\}$ the *domain* of ξ, and by $\mathcal{R}an(\xi) = \bigcup_{\alpha \in \mathcal{D}om(\xi)} \mathcal{V}ar(\alpha\xi)$ its *range*. Note that all variables in $\mathcal{R}an(\xi)$ belong to $\mathcal{S}^\forall$ by assumption, that is, they must be declared beforehand. A type substitution σ is *ground* if $\mathcal{R}an(\sigma) = \emptyset$, and a *type renaming* if it is bijective.

We use α, β for type variables, $\sigma, \tau, \rho, \theta$ for arbitrary types and ξ for type substitutions.

A *unification problem* is a conjunction of equation among types such as $\sigma_1 = \tau_1 \wedge \ldots \sigma_n = \tau_n$. A *solution* is a ground type substitution ξ such that $\sigma_i\xi = \tau_i\xi$ for all $i \in [1..n]$. Solutions are ground instances of a unique (up to renaming of type variables in its range) type substitution called the *most general unifier* of the problem and denoted by $mgu(\sigma_1 = \tau_1 \wedge \ldots \sigma_n = \tau_n)$, a result due to Robinson.

2.2 Signatures

We are given a set of function symbols denoted by the letters f, g, h, which are meant to be algebraic operators equipped with a fixed number n of arguments (called the *arity*) of respective types $\sigma_1 \in \mathcal{T}_{\mathcal{S}^\forall}, \ldots, \sigma_n \in \mathcal{T}_{\mathcal{S}^\forall}$, and *output type* $\sigma \in \mathcal{T}_{\mathcal{S}^\forall}$. Let $\mathcal{F} = \biguplus_{\sigma_1,\ldots,\sigma_n,\sigma} \mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma}$ be the set of all function symbols. The membership of a given function symbol f to a set $\mathcal{F}_{\sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma}$ is called a *type declaration* and written $f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma$. This type declarations is not a type, but corresponds to the type $\forall \mathcal{V}ar(\sigma_1) \ldots \forall \mathcal{V}ar(\sigma_n) \forall \mathcal{V}ar(\sigma) \sigma_1 \to \ldots \to \sigma_n \to \sigma$, hence, the intended meaning of a polymorphic type declaration is the set of all its monomorphic instances. In case $n = 0$, the declaration is written $f : \sigma$. A type declaration is *first-order* if its constituent types are solely built from sort symbols and variables, and higher-order otherwise. It is *polymorphic* if it uses some polymorphic type, otherwise, it is *monomorphic*. $\mathcal{F}$ is said to be *first-order* if all its type declarations are first-order, and *higher-order* otherwise. It is *polymorphic* if some type declaration is polymorphic, and *monomorphic* otherwise.

The triple $\mathcal{S}; \mathcal{S}^\forall; \mathcal{F}$ is called the *signature*. We sometimes say that the signature is *first-order, higher-order, polymorphic, monomorphic* when $\mathcal{F}$ satisfies the corresponding property.

2.3 Raw terms

The set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of *raw algebraic λ-terms* is generated from the signature $\mathcal{F}$ and a denumerable set $\mathcal{X}$ of variables according to the grammar rules:

$$\mathcal{T} := \mathcal{X} \mid (\lambda \mathcal{X} : \mathcal{T}_{\mathcal{S}^\forall}.\mathcal{T}) \mid @(\mathcal{T}, \mathcal{T}) \mid \mathcal{F}(\mathcal{T}, \ldots, \mathcal{T}).$$

Raw terms of the form $\lambda x : \sigma.u$ are called *abstractions*, while the other raw terms are said to be *neutral*. $@(u, v)$ denotes the application of u to v. We may sometimes omit the type σ in $\lambda x : \sigma.u$ as well as the application operator, writing $u(v)$ for $@(u, v)$, in particular when u is a higher-order variable. As a matter of convenience, we may write $u(v_1, \ldots, v_n)$, or $@(u, v_1, \ldots, v_n)$ for $u(v_1) \ldots (v_n)$, assuming $n \geq 1$. The raw term $@(u, v_1, \ldots, v_n)$ is called a (partial) *left-flattening* of $s = u(v_1) \ldots (v_n)$, u being possibly an application itself (hence the word "partial").

Note that our syntax requires using explicit applications for variables, since they cannot take arguments. On the other hand, function symbols have arities, eliminating the need for an explicit application of a function symbol to its arguments. Curried function symbol have arity zero, hence must be applied.

Raw terms are identified with finite labeled trees by considering $\lambda x : \sigma.\_$, for each variable x and type σ, as a unary function symbol taking a raw term u as argument to construct the raw term $\lambda x : \sigma.u$. We denote the set of free variables of the raw term t by $\mathcal{V}ar(t)$, its set of bound variables by $\mathcal{BV}ar(t)$, its size (the number of symbols occurring in t) by $|t|$. The notation $\overline{s}$ will be ambiguously used to denote a list, or a multiset, or a set of raw terms $s_1, \ldots, s_n$.

Positions are strings of positive integers. Λ and $\cdot$ denote respectively the empty string (root position) and the concatenation of strings. We use $\mathcal{P}os(t)$ for the set of positions in t. The *subterm* of t at position p is denoted by $t|_p$, and we write $t \trianglerighteq t|_p$ for the subterm relationship. The result of replacing $t|_p$ at position p in t by u is denoted by $t[u]_p$. We sometimes use $t[x : \sigma]_p$ for a raw term with a (unique) hole of type σ at position p, also called a *context.*

Type substitutions are extended to terms as homomorphisms by letting $x\xi = x$, $@(u, v)\xi = @(u\xi, v\xi)$, $f(\bar{t})\xi = f(\bar{t}\xi)$ and $(\lambda x : \sigma.u)\xi = \lambda x : \sigma\xi.u\xi$.

2.4 Environments

Definition 2.1. A *variable environment* Γ is a finite set of pairs written as $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ such that $x_i \in \mathcal{X}$, $\sigma_i \in \mathcal{T}_{\mathcal{S}^\forall}$, and $x_i \neq x_j$ for $i \neq j$. $\mathcal{V}ar(\Gamma) = \{x_1, \ldots, x_n\}$ is the set of variables of Γ. Given two variable environments Γ and Γ', their *composition* is the variable environment $\Gamma \cdot \Gamma' = \Gamma' \cup \{x : \sigma \in \Gamma \mid x \notin \mathcal{V}ar(\Gamma')\}$. Two variable environments Γ and Γ' are *compatible* if $\Gamma \cdot \Gamma' = \Gamma \cup \Gamma'$.

We now collect all declarations into a single *environment* $\Sigma; \Gamma$, where the signature $\Sigma = \mathcal{S}; \mathcal{S}^\forall; \mathcal{F}$ is the fixed part of the environment. We assume that there is exactly one declaration for each symbol in an environment $\Sigma; \Gamma$.

Example 2.2. We give here the signature for the specification of Gödel's system T. In contrast with Gödel's formulation, we use polymorphism to have the recursor rule as a polymorphic rewrite rule instead of a rule schema.

$\Sigma = \{\mathbb{N} : *\};\ \{\alpha : *\};\ \{0 : \mathbb{N},\ s : \mathbb{N} \Rightarrow \mathbb{N},\ + : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N},\ rec : \mathbb{N} \times \alpha \times (\mathbb{N} \rightarrow \alpha \rightarrow \alpha) \Rightarrow \alpha\}$.

$\Gamma = \{x : \mathbb{N}, U : \alpha,\ X : \mathbb{N} \rightarrow \alpha \rightarrow \alpha\}$.

Gödel's recursor rules are given in Example 2.27.

2.5 Typing Rules

Typing rules restrict the set of raw terms by constraining them to follow a precise discipline. Our typing judgements are written as $\Gamma \vdash_\Sigma s : \sigma$, and read "$s$ has type σ in the environment Γ". The typing judgements are displayed in Figure 1.

Variables:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma}$$

Abstraction:

$$\frac{\Gamma \cdot \{x : \sigma\} \vdash_\Sigma t : \tau}{\Gamma \vdash_\Sigma (\lambda x : \sigma.t) : \sigma \rightarrow \tau}$$

Instantiation:

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma s : \sigma \\ \xi \text{ a groud type substitution} \\ \text{of domain } \mathcal{V}ar(\sigma)\end{array}}{\Gamma\xi \vdash_\Sigma s\xi : \sigma\xi}$$

Application:

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma s : \sigma \quad \Gamma \vdash_\Sigma t : \tau \\ \xi = mgu(\alpha \rightarrow \beta = \sigma \wedge \alpha = \tau)\end{array}}{\Gamma\xi \vdash_\Sigma @(s, t)\xi : \beta\xi}$$

Functions:

$$\frac{\begin{array}{c}f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma \in \mathcal{F} \\ \Gamma \vdash_\Sigma t_1 : \tau_1 \ \ldots\ \Gamma \vdash_\Sigma t_n : \tau_n \\ \xi = mgu(\sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n)\end{array}}{\Gamma\xi \vdash_\Sigma f(t_1, \ldots, t_n)\xi : \sigma\xi}$$

Fig. 1. Typing judgements in higher-order algebras

According to the intended meaning of type declarations, we assume that the declaration $f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma$ is renamed so as to ensure that $\mathcal{V}ar(\sigma_1, \ldots, \sigma_n, \sigma) \cap \mathcal{V}ar(\tau_1, \ldots, \tau_n, \tau) = \emptyset$. Further, since the last three rules introduce a type substitution ξ, we shall consider for uniformity reasons that the first two introduce the identity type substitution. Note also that the rule for applications is nothing but the Rule **Functions** applied to the symbol $@ : (\alpha \rightarrow \beta) \times \alpha \Rightarrow \beta$.

Given an environment $\Sigma; \Gamma$, a raw term s has type σ if the judgement $\Gamma \ \vdash_\Sigma s : \sigma$ is provable in our type system. Given an environment $\Sigma; \Gamma$, a raw term s is *typable*, and said to be a *term* if there exists a type σ such that $\Gamma \ \vdash_\Sigma s : \sigma$.

Example 2.3 *Example 2.2 continued.* Let us type check in the environment $\Sigma; \{\}$ the ground raw term
$rec(s(0), 0, rec(0, \lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y), \lambda\ x : \mathbb{N}\ y\ : T\ z : \mathbb{N}.y(+(x, z))))$, where T is an abbreviation for the type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$:

$$\dfrac{\dfrac{\dfrac{}{\{\} \vdash_\Sigma 0 : \mathbb{N}}}{\{\} \vdash_\Sigma s(0) : \mathbb{N}} \quad \dfrac{}{\{\} \vdash_\Sigma 0 : \mathbb{N}} \quad \text{the missing 3rd premise is proved below}}{\{\} \vdash_\Sigma rec(s(0), 0, rec(0, \lambda x : \mathbb{N}\ y : \mathbb{N}. + (x, y), \lambda x : \mathbb{N}\ y : T\ z : \mathbb{N}.y(+(x, z)))) : \mathbb{N}}$$

For the third required premise we have:

$$\dfrac{\dfrac{}{\{\} \vdash_\Sigma 0 : \mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{\{x, y : \mathbb{N}\} \vdash_\Sigma x : \mathbb{N}} \quad \dfrac{}{\{x, y : \mathbb{N}\} \vdash_\Sigma y : \mathbb{N}}}{\{x, y : \mathbb{N}\} \vdash_\Sigma \ + (x, y) : \mathbb{N}}}{\{\} \vdash_\Sigma \lambda x, y : \mathbb{N}. + (x, y) : T} \quad \text{missing premise below}}{\{\} \vdash_\Sigma rec(0, \lambda x, y : \mathbb{N}. + (x, y), \lambda x : \mathbb{N}\ y : T\ z : \mathbb{N}.y(+(x, z))) \colon T}$$

ending up with the missing premise, in which Θ is an abbreviation for the environment $\{x, z : \mathbb{N},\ y : T\}$:

$$\dfrac{\dfrac{\dfrac{}{\Theta \ \vdash_\Sigma y : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})} \quad \dfrac{\dfrac{}{\Theta \ \vdash_\Sigma x : \mathbb{N}} \quad \dfrac{}{\Theta \ \vdash_\Sigma z : \mathbb{N}}}{\Theta \ \vdash_\Sigma \ + (x, z) : \mathbb{N}}}{\{x, z : \mathbb{N}\ y : T\} \ \vdash_\Sigma y(+(x, z)) : \mathbb{N} \to \mathbb{N}}}{\{\} \ \vdash_\Sigma \lambda x : \mathbb{N}\ y : T\ z : \mathbb{N}\ .\ y(+(x, z)) : \mathbb{N} \to T \to T}$$

Note that the rule **Instantiation** is not used here, since all computed types are ground.

Classically, we consider the proof of a given judgement as a tree whose nodes are labelled by the judgements derived in the proof and the edges by the names of the rules used in the proof. We use as usual $\mathcal{P}os(P)$ for the set of positions of the proof tree P. We can therefore speak of the rule used at a position $p \in \mathcal{P}os(P)$ whose conclusion labels the node at p while its premises label its sons (the premises being displayed above the conclusion). By convention, we associate to each position the type substitution ξ used in the corresponding rule (the identity in case of **Variables** or **Abstraction**).

2.6 Typing properties

We now come to some simple properties of our type system which are instrumental to develop a theory of polymorphic higher-order rewriting. All these properties are fairly standard and easily proved by induction on the type derivation for most of them. We therefore skip their proofs.

LEMMA 2.4 WEAKENING. *Assume given an environment $\Sigma; \Gamma$, a term s and a type σ such that $\Gamma \ \vdash_\Sigma s : \sigma$ holds. Then, $\Gamma \cdot \Gamma' \ \vdash_\Sigma s : \sigma$ for all Γ' compatible with Γ.*

Lemma 2.5 strengthening. *Assume given an environment $\Sigma;\Gamma\cdot\{x:\tau\}\cdot\Gamma'$, a term s such that $x\notin\mathcal{V}ar(s)$ and a type σ such that $\Gamma\cdot\{x:\tau\}\cdot\Gamma' \ \vdash_\Sigma s:\sigma$. Then, $\Gamma\cdot\Gamma' \ \vdash_\Sigma s:\sigma$.*

Definition 2.6. The proof of a judgement $\Gamma \ \vdash_\Sigma s:\sigma$ is *canonical* if the rule **Instantiation** is used at most once, at the root of the proof tree. We write $\Gamma \ \vdash^c_\Sigma s:\sigma$ for a canonical proof of the judgement $\Gamma \ \vdash_\Sigma s:\sigma$ and say in short that $\Gamma \ \vdash^c_\Sigma s:\sigma$ is a *canonical* judgement.

Note that the set of canonical proofs is closed under weakening and strengthening.

Lemma 2.7. *A provable judgement $\Gamma \ \vdash_\Sigma s:\sigma$ has a unique canonical proof (up to renaming of type variables).*

In the rest of Section 2, we consider canonical judgements when canonical proofs are required. We therefore need using both notations $\Gamma \ \vdash_\Sigma s:\sigma$ and $\Gamma \ \vdash^c_\Sigma s:\sigma$.

Lemma 2.8. *Given an environment $\Sigma;\Gamma$ and a raw term s, whether s is typable is decidable in linear time in the size of s. When s is typable, there exists a unique type σ, (up to renaming of type variables), called the* principal type *of s, such that the canonical judgement $\Gamma \ \vdash^c_\Sigma s:\sigma$ is free of any use of* **Instantiation**. *Besides, all types τ such that $\Gamma \ \vdash_\Sigma s:\tau$, are type instances of σ.*

Lemma 2.9. *Let P denote the proof of the canonical judgement $\Gamma \ \vdash^c_\Sigma s:\sigma$ and p be a position in $\mathcal{P}os(s)$. Then, there exists a unique position $q(p)\in\mathcal{P}os(P)$ such that the subproof $P|_{q(p)}$ is a canonical proof of a judgement of the form $\Gamma_{s|_p} \ \vdash^c_\Sigma s|_p:\tau$ in which τ is the principal type of $s|_p$ in $\Gamma_{s|_p}$.*

Definition 2.10. Given a canonical judgement $\Gamma \ \vdash^c_\Sigma s:\sigma$ and a position $p\in\mathcal{P}os(s)$, we call *actual type* of $s|_p$, the type $\theta=\tau\xi$ instance of the principal type τ of $s|_p$ in the environment $\Gamma_{s|_p}$ by the type substitution ξ used at position $q(p)$.

In particular, the actual type of s in the judgement $\Gamma \ \vdash^c_\Sigma s:\sigma$ is σ, regardless whether σ is principal or not. Note also that the proof of the canonical judgement $\Gamma_{s|_p} \ \vdash^c_\Sigma s|_p:\tau\xi$ is obtained from the proof of the canonical judgement $\Gamma_{s|_p} \ \vdash^c_\Sigma s|_p:\tau$ by adding a last **Instantiation** step when ξ is not the identity.

Example 2.11 *Example 2.2 continued.* Let us type check the simple raw term $s=rec(0,0,\lambda\, x:\mathbb{N}\ y:\mathbb{N}.y)$ in the environment $\Sigma;\{\}$ with the rules of Figure 1, writing the type substitution used in **Applications** and **Function** to the right of the rule when it is not the identity:

$$\dfrac{\dfrac{}{\{\} \ \vdash^c_\Sigma 0:\mathbb{N}} \quad \dfrac{}{\{\} \ \vdash^c_\Sigma 0:\mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{\{x:\mathbb{N}, y:\alpha\} \ \vdash^c_\Sigma .y:\alpha}}{\{x:\mathbb{N}\} \ \vdash^c_\Sigma \lambda\, y:\alpha.y:\alpha\to\alpha}}{\{\} \ \vdash^c_\Sigma \lambda\, x:\mathbb{N}\ y:\alpha.y:\mathbb{N}\to\alpha\to\alpha}}{\{\} \ \vdash^c_\Sigma rec(0,0,\lambda\, x:\mathbb{N}\ y:\mathbb{N}.y):\mathbb{N}} \ \{\alpha\mapsto\mathbb{N}\}$$

The principal and actual types of all subterms of s coincide, except for its subterm $s|_3=\lambda\, x:\mathbb{N}\ y:\alpha.y$ whose principal and actual type in the judgement $\{\} \ \vdash^c_\Sigma s:\mathbb{N}$ are respectively $\mathbb{N}\to\alpha\to\alpha$ and $\mathbb{N}\to\mathbb{N}\to\mathbb{N}$. Their canonical proofs only differ by the last instantiation step applied to the canonical proof of $\{\} \ \vdash^c_\Sigma \lambda\, x:\mathbb{N}\ y:$

$\alpha.y : \mathbb{N} \to \alpha \to \alpha$ obtained as a subproof of the above proof, yielding the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x : \mathbb{N}, y : \alpha\} \ \vdash^c_\Sigma \ .y : \alpha}}{\{x : \mathbb{N}\} \ \vdash^c_\Sigma \ \lambda\, y : \alpha.y : \alpha \to \alpha}}{\{\} \ \vdash^c_\Sigma \ \lambda\, x : \mathbb{N}\ y : \alpha.y : \mathbb{N} \to \alpha \to \alpha}}{\{\} \ \vdash^c_\Sigma \ \lambda\, x : \mathbb{N}\ y : \alpha.y : \mathbb{N} \to \mathbb{N} \to \mathbb{N}} \ \alpha \mapsto \mathbb{N}$$

Given a canonical typing judgement $\Gamma \ \vdash^c_\Sigma \ s : \sigma$ and a position $p \in \mathcal{P}os(s)$, the actual type θ of $s|_p$ plays a key role throughout the paper.

Lemma 2.12 replacement. *Assume given an environment* $\Sigma; \Gamma$*, two terms* s *and* v*, two types* σ *and* τ*, and a position* $p \in \mathcal{P}os(s)$ *such that* $\Gamma \ \vdash^c_\Sigma \ s : \sigma$*,* $\Gamma_{s|_p} \ \vdash^c_\Sigma \ s|_p : \tau$ *where* τ *is the actual type of* $s|_p$*, and* $\Gamma_{s|_p} \ \vdash_\Sigma \ v : \tau$*. Then,* $\Gamma \ \vdash_\Sigma \ s[v]_p : \sigma$*.*

Note that the proof of the judgement $\Gamma \ \vdash_\Sigma \ s[v]_p : \sigma$ obtained by substituting the canonical proof of the judgement $\Gamma_{s|_p} \ \vdash_\Sigma \ v : \tau$ to the canonical subproof $\Gamma_{s|_p} \ \vdash^c_\Sigma \ s|_p : \tau$ into the canonical proof of the judgement $\Gamma \ \vdash^c_\Sigma \ s : \sigma$, may not be canonical in case τ is not the principal type of v in $\Gamma_{s|_p}$. As a consequence, σ may not be the principal type of $s[v]$ in Γ even if it is the principal type of s in Γ.

2.7 Substitutions

Definition 2.13. A *term substitution*, or simply *substitution* is a finite set $\gamma = \{(x_1 : \sigma_1) \mapsto (\Gamma_1, t_1); \ldots; (x_n : \sigma_n) \mapsto (\Gamma_n, t_n)\}$, whose elements are quadruples made of a variable symbol, a type, a term environment and a term, such that

(i) for each $i \in [1..n]$, t_i is typable in Γ_i with principal type τ_i;

(ii) $\forall i \neq j \in [1..n]$, Γ_i and Γ_j are compatible environments;

(iii) the type unification problem $\sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n$ has a most general unifier ξ_γ;

(iv) $\forall i \in [1..n]$, $t_i \neq x_i$ or $\sigma_i \xi_\gamma \neq \sigma_i$ and $\forall i \neq j \in [1..n]$, $x_i \neq x_j$;

The substitution γ is *type preserving* if ξ_γ is the identity, and a *specialization* if $t_i = x_i$ for all i.

The domain of the substitution γ is the environment $\mathcal{D}om(\gamma) = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ while its *range* is the environment $\mathcal{R}an(\gamma) = \bigcup_{i \in [1..n]} \Gamma_i$. We say that $x \in \mathcal{D}om(\gamma)$ whenever $x : \sigma \in \mathcal{D}om(\gamma)$ for some type σ. We sometimes omit the parentheses, the type σ_i and the environment Γ_i in $(x_i : \sigma_i) \mapsto (\Gamma_i, t_i)$.

Note that the set $\mathcal{R}an(\gamma)$ is indeed an environment by our compatibility assumption (iv), which is of course compatible with every environment Γ_i.

The need of a term environment Γ_i for each variable x_i comes from the requirement of typing the term t_i (with a type compatible with the type σ_i of the variable x_i that it will be substituted to, a property ensured by condition (iii)). In case t_i is ground, its typing environment is of course empty. This is the case in the coming example:

Example 2.14 *Example 2.11 continued.* Here is an example of a ground substitution, for which condition (i) have been proved in Example 2.11. The other conditions are easily checked here (condition (iii) is satisfied with the substitution $\xi_\gamma = \alpha \mapsto \mathbb{N}$):

$$\begin{array}{rcl}\{(x:\mathbb{N}) & \mapsto & (\{\}, s(0)); \\ (U:\mathbb{N}) & \mapsto & (\{\}, 0); \\ (X:\mathbb{N}\to\mathbb{N}\to\mathbb{N}) & \mapsto & (\{\}, \lambda\, x:\mathbb{N}\; y:\alpha.y)\}\end{array}$$

In general, condition (iii) may yield a non-trivial unification problem as in the following example:

Example 2.15. Let
$\Sigma = \{\mathbb{N}, \mathbb{B} : *, \mathit{pair} : *^2 \Rightarrow *, \mathit{list} : * \Rightarrow *\}; \{\alpha, \beta, \alpha' : *\}; \{\mathit{iter} : (\alpha \to \beta) \times \mathit{list}(\alpha) \Rightarrow \mathit{list}(\beta)\}$.

Consider the term $\mathit{iter}(F, l)$ in the environment $\{F : \alpha \to \beta, l : \mathit{list}(\alpha)\}$ which will be used in the rest of the example as the domain for a substitution γ. We have the typing derivation:
$\{F : \alpha \to \beta, l : \mathit{list}(\alpha)\} \vdash_\Sigma \mathit{iter}(F, l) : \mathit{list}(\beta)$

Let us now first consider a non-substitution fullfilling all conditions but (iii):

$$\begin{array}{rl}\gamma = \{ & (F : \alpha \to \beta) \mapsto (\{F : \mathit{pair}(\mathbb{N}, \alpha') \to \alpha'\}, F), \\ & (l : \mathit{list}(\alpha)) \mapsto (\{l : \mathit{list}(\mathit{pair}(\alpha', \mathbb{B}))\}, l)\}.\end{array}$$

Then, the unification problem originating from condition (iii),
$\alpha \to \beta = \mathit{pair}(\mathbb{N}, \alpha') \to \alpha' \wedge \mathit{list}(\alpha) = \mathit{list}(\mathit{pair}(\alpha', \mathbb{B}))$ fails by generating the unsolvable equation $\mathbb{N} = \mathbb{B}$.
Trying to type the term $\mathit{iter}(F, l)$ in the environment $\mathcal{R}an(\gamma) = \{F : \mathit{pair}(\mathbb{N}, \alpha') \to \alpha', l : \mathit{list}(\mathit{pair}(\alpha', \mathbb{B}))\}$ results again in a failure, since the **Functions** rule generates the very same unsolvable unification problem as before.

Let us finally consider the substitution (indeed, a specialization)

$$\begin{array}{rl}\gamma = \{ & (F : \alpha \to \beta) \mapsto (\{F : \mathit{pair}(\mathbb{N}, \alpha') \to \alpha'\}, F), \\ & (l : \mathit{list}(\alpha)) \mapsto (\{l : \mathit{list}(\mathit{pair}(\alpha', \mathbb{N}))\}, l)\}.\end{array}$$

We check condition (iii) by solving the unification problem
$\alpha \to \beta = \mathit{pair}(\mathbb{N}, \alpha') \to \alpha' \wedge \mathit{list}(\alpha) = \mathit{list}(\mathit{pair}(\alpha', \mathbb{N}))$, which has most general unifier $\xi_\gamma = \{\alpha \mapsto \mathit{pair}(\mathbb{N}, \mathbb{N}), \alpha' \mapsto \mathbb{N}, \beta \mapsto \mathbb{N}\}$.
Trying now to type the term $\mathit{iter}(F, l)$ in the environment $\mathcal{R}an(\gamma) = \{F : \alpha \to \beta, l : \mathit{list}(\alpha)\}$, we get:
$\{F : \mathit{pair}(\mathbb{N}, \alpha') \to \alpha', l : \mathit{list}(\mathit{pair}(\alpha', \mathbb{N}))\} \vdash_\Sigma \mathit{iter}(F, l) : \mathit{list}(\mathbb{N}))$
We observe that the obtained type for $\mathit{iter}(F, l)$ is the instance of its type in the environment $\mathcal{D}om(\gamma) = \{F : \alpha \to \beta, l : \mathit{list}(\alpha)\}$ by the type substitution ξ_γ.

Definition 2.16. A substitution γ is said to be *compatible* with an environment Γ if

(i) $\mathcal{D}om(\gamma)$ is compatible with Γ,

(ii) $\mathcal{R}an(\gamma)$ is compatible with $\Gamma \setminus \mathcal{D}om(\gamma)$.

We will also say that γ is compatible with the judgement $\Gamma \vdash_\Sigma s : \sigma$.

Compatibility of a term substitution with an environment Γ is a necessary condition for defining how the substitution operates on a term typable in the environment Γ to yield a term typable in the environment $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma)$:

Definition 2.17. A term substitution γ compatible with a judgement $\Gamma\ \vdash_\Sigma s:\sigma$ operates as an endomorphism on s (keeping its bound variables unchanged) and yields a term $s\gamma$ defined as follows:

$$\begin{array}{llll}
\text{If} & s = x \in \mathcal{X} \text{ and} & & \\
 & x \notin \mathcal{V}ar(\gamma) & \text{then} & s\gamma = x \\
\text{If} & s = x \in \mathcal{X} \text{ and} & & \\
 & (x:\sigma) \mapsto (\Theta, t) \in \gamma & \text{then} & s\gamma = t \\
\text{If} & s = @(u,v) & \text{then} & s\gamma = @(u\gamma, v\gamma) \\
\text{If} & s = f(u_1,\ldots,u_n) & \text{then} & s\gamma = f(u_1\gamma,\ldots,u_n\gamma) \\
\text{If} & s = \lambda x:\tau.u \text{ and} & & \\
 & y \text{ a fresh variable} & \text{then} & s\gamma = \lambda y:\tau\xi_\gamma.(u\{(x:\tau)\mapsto(\{y:\tau\xi_\gamma\},y)\})\gamma
\end{array}$$

Term substitutions can be factored out via a specialization (possibly the identity) as follows:

Lemma 2.18. *Every term substitution γ can be written in the form $\delta\theta$ where δ is a specialization and θ is type preserving, that is, for any term s, $s\gamma = (s\delta)\theta$.*

Proof. Let $\gamma = \{(x_i:\sigma_i)\mapsto(\Gamma_i,t_i)\}_i$. Take $\delta = \{(x_i,\sigma_i)\mapsto(\{x_i:\sigma_i\xi_\gamma\},x_i)\}$, and $\theta = \{(x_i,\sigma_i\xi_\gamma)\mapsto(\{\},x_i\gamma)\}$. □

When writing $s\gamma$, using postfixed notation for substitutions, we will always make the assumption that the domain of γ is compatible with the judgement $\Gamma\ \vdash_\Sigma s:\sigma$. We will use the letter γ for arbitrary substitutions and the notation $A\gamma$, where A is a set of terms, for the set of instances of the terms in A.

Example 2.19 *Example 2.2 continued.* Let us illustrate the use of term substitutions with the term (where T is an abbreviation for the type $\mathbb{N}\rightarrow\mathbb{N}\rightarrow\mathbb{N}$) $rec(s(0),0,rec(0,\lambda\ x,y:\mathbb{N}.+(x,y),\lambda\ x:\mathbb{N}\ y:T\ z:\mathbb{N}.y(+(x,z))))$, which happens to be an instance of the lefthand side of the second Gödel's recursor rule $rec(s(x),U,X)\rightarrow @(X,x,rec(x,U,X))$ for which $x:\mathbb{N}, U:\alpha$ and $X:\mathbb{N}\rightarrow\alpha\rightarrow\alpha$, by the substitution

$$\begin{array}{rl}
\{(x:\mathbb{N}) & \mapsto (\{\},s(0)) \\
(U:\alpha) & \mapsto (\{\},0) \\
(X:\mathbb{N}\rightarrow\alpha\rightarrow\alpha) & \mapsto (\{\},rec(0,\lambda\ x,y:\mathbb{N}.+(x,y),\lambda\ x:\mathbb{N}\ y:T\ z:\mathbb{N}.y(+(x,z))))\}
\end{array}$$

while its subterm $rec(0,\lambda\ x:\mathbb{N}\ y:\mathbb{N}.+(x,y),\lambda\ x:\mathbb{N}\ y:T\ z:\mathbb{N}.y(+(x,z)))$ is an instance of the same lefthand side by the substitution

$$\begin{array}{rcl}
\{(x:\mathbb{N}) & \mapsto & (\{\},0) \\
(U:\alpha) & \mapsto & (\{\},\lambda\ x:\mathbb{N}\ y:\mathbb{N}.+(x,y)) \\
(X:\mathbb{N}\rightarrow\alpha\rightarrow\alpha) & \mapsto & (\{\},\lambda\ x:\mathbb{N}\ y:T\ z:\mathbb{N}.y(+(x,z)))\}.
\end{array}$$

These substitutions are factored out by the respective specializations
$\{(U:\alpha)\mapsto(\{U:\mathbb{N}\},U);\ (X:\mathbb{N}\rightarrow\alpha\rightarrow\alpha)\mapsto(\{X:T\},X)\}$ and
$\{(U:\alpha)\mapsto(\{U:\mathbb{N}\},U);\ (X:\mathbb{N}\rightarrow\alpha\rightarrow\alpha)\mapsto(\{X:\mathbb{N}\rightarrow T\rightarrow T\},X)\}$, whose associated type instantiations are respectively $\alpha\mapsto\mathbb{N}$ and $\alpha\mapsto T$.

The next lemma makes precise the action of substitutions on typing:

Lemma 2.20. *Assume given an environment $\Sigma;\Gamma$ and a substitution γ compatible with the judgement $\Gamma\ \vdash_\Sigma s:\sigma$. Then, $(\Gamma\setminus\mathcal{D}om(\gamma))\cdot\mathcal{R}an(\gamma)\ \vdash_\Sigma s\gamma:\sigma\xi_\gamma$.*

Proof. The proof is done by induction on the derivation of the judgement $\Gamma \ \vdash_\Sigma s : \sigma$. We carry out the **Abstraction** case, which shows the need for instantiating the types of bound variables.

Assume that $s = \lambda x : \tau.u$ with $\sigma = \tau \to \sigma'$, hence $\Gamma \cdot \{x : \tau\} \ \vdash_\Sigma u : \sigma'$ by the rule **Abstraction**. Given now a fresh variable y, let us consider the substitution $\gamma' = \gamma \cup \{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\}$. Since Γ is compatible with γ and y is a fresh variable, Γ is compatible with γ'. By induction hypothesis, $(\Gamma \setminus \mathcal{D}om(\gamma')) \cdot \mathcal{R}an(\gamma') \ \vdash_\Sigma u\gamma' : \sigma\xi'_\gamma$. By construction, $\xi_{\gamma'} = \xi_\gamma$, and since y is a fresh variable, $(\Gamma \setminus \mathcal{D}om(\gamma')) \cdot \mathcal{R}an(\gamma') = (\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \cdot \{y : \tau\xi_\gamma\}$. Therefore, our typing judgement becomes $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \cdot \{y : \tau\xi_\gamma\} \ \vdash_\Sigma u\{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\}\gamma : \sigma'\xi_\gamma$. By the rule **Abstraction**, we get $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \ \vdash_\Sigma \lambda y : \tau\xi_\gamma.u\{(x : \tau) \mapsto (\{y : \tau\xi_\gamma\}, y)\gamma : (\tau\xi_\gamma \to \sigma'\xi_\gamma) = \sigma\xi_\gamma$. Definition 2.17 now yields the expected result. □

Note that Lemma 2.5 allows us to clean the environment from the variables which do not occur in $s\gamma$.

As corollaries of Lemma 2.20, we obtain:

Corollary 2.21. *Let γ be a type preserving substitution compatible with the judgement $\Gamma \ \vdash_\Sigma s : \sigma$. Then, $(\Gamma \setminus \mathcal{D}om(\gamma)) \cdot \mathcal{R}an(\gamma) \ \vdash_\Sigma s\gamma : \sigma$.*

Corollary 2.22. *Let δ be a specialization which is compatible with the judgement $\Gamma \ \vdash_\Sigma s : \sigma$. Then, $s\delta = s\xi_\delta$ and $(\Gamma \setminus \mathcal{D}om(\delta)) \cdot \mathcal{R}an(\delta) \ \vdash_\Sigma s\delta : \sigma\xi_\delta$.*

2.8 Plain higher-order rewriting

We now come to the definition of plain higher-order rewriting based on plain pattern-matching, as considered in [Jouannaud and Okada 1991] or in the Calculus of Inductive Constructions [Coquand and Paulin-Mohring 1990].

Definition 2.23. Given a signature Σ, a *higher-order rewrite rule* or simply *rewrite rule* is a triple written $\Gamma \ \vdash \ l \to r$, where Γ is an environment and l, r are higher-order terms such that

(i) $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \subseteq \mathcal{V}ar(\Gamma)$,

(ii) for all substitutions γ such that $\Gamma \ \vdash_\Sigma l\gamma : \sigma$, then $\Gamma \ \vdash_\Sigma r\gamma : \sigma$.

The rewrite rule is said to be *polymorphic* if σ is a polymorphic type for some substitution γ.

A *plain higher order rewriting system*, or simply *rewriting system* is a set of higher-order rewrite rules.

Example 2.24. Here is an example of a triple which is not a rewrite rule because it violates condition (ii): let $\mathcal{F} = \{f : \alpha \Rightarrow \alpha, g : \beta \Rightarrow \beta, 0 : \mathbb{N}\}$, $\mathcal{T}_{\mathcal{S}^\forall} = \{\alpha, \beta\}$, and consider the triple $\{x : \alpha\} \ \vdash \ f(x) \to g(0)$. We have $\{x : \alpha\} \ \vdash_\Sigma f(x) : \alpha$, $\{x : \alpha\} \ \vdash_\Sigma g(0) : \mathbb{N}$, and α has instances different from $\mathbb{N}$ such as $\mathbb{N} \to \mathbb{N}$. Therefore, the triple $\{x : \alpha\} \ \vdash \ f(x) \to g(0)$ is not a rule. On the other hand, the instance $\{x : \alpha\} \ \vdash \ f(0) \to g(0)$ is a rule.

One may wonder whether a given triple $\Gamma \ \vdash \ l \to r$ is a rewrite rule, since condition (ii) quantifies over all possible substitutions. One can indeed answer the question, since types are first order terms and the existence of instances of a type σ (the principal type of s) which are not instances of τ (the principal type of r) can be expressed by the first-order formula $\exists\alpha.\alpha = \sigma \wedge \neg(\alpha = \tau)$, with α a fresh variable, interpreted over the set of ground types. Validity of such formulas is decidable by a result of Mal'cev [1961], but it is clear that there is no α satisfying $\alpha = \sigma \wedge \neg(\alpha = \tau)$ iff σ is an instance of τ, that is, the type of the righthand side is more general that the type of the lefthand one. There are useful examples of such rules in practice.

We shall sometimes omit the environment Γ in the rule $\Gamma \ \vdash_\Sigma l \rightarrow r$ when it can be inferred from the context. The β- and η-rules below are two examples of polymorphic rewrite rule schemas:

$$\begin{array}{rl}\{u : \alpha,\ v : \beta\} \vdash_\Sigma @(\lambda x : \alpha.v, u) \longrightarrow_\beta v\{x \mapsto u\} & \\ \{x : \alpha,\ u : \alpha \rightarrow \beta\} \vdash_\Sigma \lambda x.@(u, x) \longrightarrow_\eta u & \text{if } x \notin \mathcal{V}ar(u)\end{array}$$

Making these rule schemas true rewrite rules is possible to the price of including variables with arities in our framework, an idea due to Klop [1980].

We are now ready for defining the rewrite relation :

Definition 2.25. Given a plain higher-order rewriting system R and an environment Γ, a term s such that $\Gamma \ \vdash_\Sigma s : \sigma$ rewrites to a term t at position p with the rule $\Theta \ \vdash \ l \rightarrow r$ and the term substitution γ, written $\Gamma \ \vdash \ s \underset{\Theta \ \vdash \ l \rightarrow r}{\xrightarrow{p}} t$, or simply $\Gamma \ \vdash \ s \rightarrow_R t$, or even $s \rightarrow_R t$ assuming the environment Γ, if the following conditions are satisfied :

(i) $\mathcal{D}om(\gamma) \subseteq \Theta$; (ii) $\Theta \cdot \mathcal{R}an(\gamma) \subseteq \Gamma_{s|_p}$; (iii) $s|_p = l\gamma$; (iv) $t = s[r\gamma]_p$.

Note that, by definition of higher-order substitutions, condition (iii) implies that variables which are bound in the rule do not occur free in γ, therefore avoiding capturing variables when rewriting.

Type checking the rewritten term is not necessary, thanks to the following property:

LEMMA 2.26 TYPE PRESERVATION. *Assume that* $\Gamma \ \vdash_\Sigma s : \sigma$ *and* $\Gamma \ \vdash \ s \rightarrow_R t$. *Then* $\Gamma \ \vdash_\Sigma t : \sigma$.

PROOF. Let $\Gamma \ \vdash \ s \underset{\Theta \ \vdash \ l \rightarrow r}{\xrightarrow{p}} t$. By Lemma 2.9, $\Gamma_{s|_p} \ \vdash_\Sigma s|_p : \tau$, with τ the actual type of $s|_p$ in $\Gamma \ \vdash_\Sigma s : \sigma$. By condition (iii), $\Gamma_{s|_p} \ \vdash_\Sigma l\gamma : \tau$. By conditions (i) and (ii), the substitution γ is compatible with the environment Θ, and by condition (i) in Definition 2.23, $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(\Theta) \subseteq \mathcal{V}ar(\Theta \cdot \mathcal{R}an(\gamma))$. By Lemma 2.5 (repeated) it follows that $\Theta \cdot \mathcal{R}an(\gamma) \ \vdash_\Sigma l\gamma : \tau$. By condition (ii) of Definition 2.23, $\Theta \cdot \mathcal{R}an(\gamma) \ \vdash_\Sigma r\gamma : \tau$. By conditions (ii) and Lemma 2.4, $\Gamma_{s|_p} \ \vdash_\Sigma r\gamma : \tau$. By Lemma 2.12, $\Gamma \ \vdash_\Sigma s[r\gamma]_p : \sigma$. We conclude with condition (iv). □

Example 2.27 *Example 2.2 continued.* Here are Gödel's recursor rules for natural numbers:

$$\begin{array}{rcl}\{U : \alpha,\ X : \mathbb{N} \rightarrow \alpha \rightarrow \alpha\} & \vdash & rec(0, U, X) \rightarrow U \\ \{x : \mathbb{N},\ U : \alpha,\ X : \mathbb{N} \rightarrow \alpha \rightarrow \alpha\} & \vdash & rec(s(x), U, X) \rightarrow @(X, x, rec(x, U, X))\end{array}$$

Rewriting $rec(s(0), 0, rec(0, \lambda\, x : \mathbb{N}\ y : \mathbb{N}. + (x, y), \lambda\, x : \mathbb{N}\ y : T\ z : \mathbb{N}.y(+(x, z))))$ (where T is an abbreviation for the type $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$) with a call-by-value strategy (redexes are underlined) until a normal form is obtained, we get:

$$\begin{array}{l}
rec(s(0), 0, \underline{rec(0, \lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y), \lambda\ x : \mathbb{N}\ y : T\ z : \mathbb{N}.y(+(x, z)))})\\
\longrightarrow^3_{\{U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\}\ \vdash\ rec(0,U,X)\to U} \quad \underline{rec(s(0), 0, \lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y))}\\
\longrightarrow^\epsilon_{\{x:\mathbb{N},\ U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\}\ \vdash\ rec(s(x),U,X)\to @(X,x,rec(x,U,X)}\\
\qquad\qquad\qquad \underline{@(\lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y), 0, rec(0, 0, \lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y)))}\\
\longrightarrow^\epsilon_\beta \quad \underline{@(\lambda\ y : \mathbb{N}. + (0, y), rec(0, 0, \lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y)))}\\
\longrightarrow^\epsilon_\beta \quad +(0, \underline{rec(0, 0, \lambda\ x : \mathbb{N}\ y : \mathbb{N}. + (x, y))})\\
\longrightarrow^2_{\{U:\alpha,\ X:\mathbb{N}\to\alpha\to\alpha\}\ \vdash\ rec(0,U,X)\to U} \quad \underline{+(0, 0)}\\
\longrightarrow^\epsilon_{\{x:\mathbb{N}\}\ \vdash\ +(x,0)\to x} \quad 0
\end{array}$$

As a general benefit, the use of polymorphic signatures allows us to have only one recursor rule, instead of infinitely many rules described by one rule schema as in Gödel's original presentation.

Other examples of higher-order rewrite systems are developed in section 3.

Because plain higher-order rewriting is type preserving, we may often omit the environment in which a term is type checked as well as its type, and consider the sequence of terms originating from a given term s type checked in an environment Γ by rewriting with a given set R of higher-order rules. In other words, we will often consider rewriting as a *relation on terms*, which will allow us to simplify our notations when needed.

A term s such that $s \xrightarrow[R]{p} t$ is called *reducible* (with respect to R). $s|_p$ is a *redex* in s, and t is the *reduct* of s. Irreducible terms are said to be in *R-normal form.* A substitution γ is in R-normal form if $x\gamma$ is in R-normal form for all x. We denote by $\xrightarrow[R]{*}$ the reflexive, transitive closure of the rewrite relation $\xrightarrow[R]{}$, and by $\longleftrightarrow^*_R$ its reflexive, symmetric, transitive closure. We are indeed interested in the relation $\longrightarrow_{R\beta\eta} = \longrightarrow_R \cup \longrightarrow_\beta \cup \longrightarrow_\eta$.

Given a rewrite relation $\longrightarrow$, a term s is *strongly normalizable* if there is no infinite sequence of rewrites issuing from s. A substitution γ is *strongly normalizable* if all terms $x\gamma$ such that $x \in \mathcal{D}om(\gamma)$ are strongly normalizable.

The rewrite relation itself is *strongly normalizing*, or *terminating*, if all terms are strongly normalizable. It is confluent if $s \longrightarrow^* u$ and $s \longrightarrow^* v$ implies that $u \longrightarrow^* t$ and $v \longrightarrow^* t$ for some t.

2.9 Higher-Order Reduction Orderings

We will make intensive use of well-founded relations for proving strong normalization properties, using the vocabulary of rewrite systems for these relations. For our purpose, these relations may not be transitive, hence are not necessarily orderings, although their transitive closures will be well-founded orderings, which justifies to sometimes call them orderings by abuse of terminology.

For our purpose, a *strict ordering* $>$ is an irreflexive transitive relation, an *ordering* $\geq$ is the union of its strict part $>$ with equality $=$, and a *quasi-ordering* $\succeq$ is the union of its strict part $\succ$ with its equivalence $\simeq$. Because we identify α-convertible higher-order terms, their equality contains implicitly α-conversion. The following results will play a key role, see [Dershowitz and Jouannaud 1990]:

Assume $\succ, \succ_1, \ldots, \succ_n$ are relations on n given sets $S, S_1, \ldots, S_n$. Let

- $(\succ_1, \ldots, \succ_n)_{lex}$ be the relation on $\bigcup_{k \leq n} S_1 \times \ldots \times S_k$ defined as $(s_1, \ldots, s_m)(\succ_1, \ldots, \succ_n)_{lex}(t_1, \ldots, t_p)$ iff $\exists i \in [1..min(m,p)]$ such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and ($s_i \succ_i t_i$ or ($s_i = t_i$ and $i = p < m$);
- $\succ_{mul}$ be the relation on the set of multisets of elements of S defined as $M \cup \{x\} \succ_{mul} N \cup \{y_1, \ldots, y_n\}$ iff $\forall i \in [1..n]$ $x \succ y_i$ and $M = N$ or $M \succ_{mul} N$.

It is well known that both operations preserve the well-foundedness of the relations $\succ, \succ_1, \ldots, \succ_n$, and that they also preserve transitivity, hence orderings.

We end up this section by defining the notion of a reduction ordering operating on higher-order terms.

Definition 2.28. A *higher-order reduction ordering* $\succ$ is a well-founded ordering of the set of judgements which is

(i) *monotonic*: $(\Gamma \ \vdash_\Sigma s : \sigma) \succ (\Gamma \ \vdash_\Sigma t : \sigma)$ implies that for all Γ' compatible with Γ such that $\Gamma' \ \vdash_\Sigma u[x : \sigma] : \tau$, then $(\Gamma \cdot \Gamma' \ \vdash_\Sigma u[s] : \tau) \succ (\Gamma \cdot \Gamma' \ \vdash_\Sigma u[t] : \tau)$;

(ii) *stable*: $(\Gamma \ \vdash_\Sigma s : \sigma) \succ (\Gamma \ \vdash_\Sigma t : \sigma)$ implies that for all type preserving substitution γ whose domain is compatible with Γ, then $(\Gamma \cdot \mathcal{R}an(\gamma) \ \vdash_\Sigma s\gamma : \sigma) \succ (\Gamma \cdot \mathcal{R}an(\gamma) \ \vdash_\Sigma t\gamma : \sigma)$;

(iii) *polymorphic*: $(\Gamma \ \vdash_\Sigma s : \sigma) \succ (\Gamma \ \vdash_\Sigma t : \sigma)$ implies that for all specialization δ whose domain is compatible with Γ, then $(\Gamma \cdot \mathcal{R}an(\delta) \ \vdash_\Sigma s\delta : \sigma\xi_\delta) \succ (\Gamma \cdot \mathcal{R}an(\delta) \ \vdash_\Sigma t\gamma : \sigma\xi_\delta)$;

(iv) *compatible*: $(\Gamma \ \vdash_\Sigma s : \sigma) \succ (\Gamma \ \vdash_\Sigma t : \sigma)$ implies $(\Gamma' \ \vdash_\Sigma s : \sigma) \succ (\Gamma' \ \vdash_\Sigma t : \sigma)$ for all environments Γ' such that Γ and Γ' are compatible, $\Gamma' \ \vdash_\Sigma s : \sigma$ and $\Gamma' \ \vdash_\Sigma t : \sigma$;

(v) *functional*: $(\Gamma \ \vdash_\Sigma s : \sigma \longrightarrow_\beta \cup \longrightarrow_\eta t : \sigma)$ implies $(\Gamma \ \vdash_\Sigma s : \sigma) \succ (\Gamma \ \vdash_\Sigma t : \sigma)$.

Note that (iii) implies that $\Gamma\xi \ \vdash_\Sigma s\xi : \sigma\xi$ for all type substitutions ξ, which justify the separation of (iii) from (ii).

We will often abuse notations by writing $(\Gamma \ \vdash_\Sigma s : \sigma \succ t : \tau)$ instead of $(\Gamma \ \vdash_\Sigma s : \sigma) \succ (\Gamma \ \vdash_\Sigma t : \tau)$, therefore sharing the environment Γ. This amounts to view the ordering as a relation on typed terms instead of a relation on judgements. We may even omit types and/or environments, considering the ordering as operating directly on terms, and write $\Gamma \ \vdash \ s \succ t$, $s : \sigma \succ t : \tau$ or $s \succ t$.

2.10 Termination of polymorphic higher-order rules

Theorem 2.29. *Let $\succeq$ be a polymorphic, higher-order reduction ordering and $R = \{\Gamma_i \ \vdash_\Sigma l_i \to r_i\}_{i \in I}$ be a higher-order rewrite system such that $\forall i \in I, (\Gamma_i \ \vdash_\Sigma l_i : \sigma_i) \succ (\Gamma_i \ \vdash_\Sigma r_i : \sigma_i)$, where σ_i is the principal type of l_i in Γ_i. Then the relation $\longrightarrow_R \cup \longrightarrow_\beta \cup \longrightarrow_\eta$ is strongly normalizing.*

Proof. Let $\Gamma \ \vdash_\Sigma s : \sigma$ and $\Gamma \ \vdash_\Sigma s \underset{\Gamma \ \vdash \ l \to r \in R}{\xrightarrow{p}} t$. We assume without loss of generality that $\mathcal{V}ar(l) \cap \mathcal{D}om(\Gamma) = \emptyset$. By Lemma 2.9, $\Gamma_{s|_p} \ \vdash_\Sigma s|_p : \tau$, the actual type of $s|_p$ in the environment $\Gamma_{s|_p}$. By Definition 2.25(iii,iv) and Lemma 2.18, $s|_p = l\delta\gamma$ and $t|_p = r\delta\gamma$, where δ is a specialization and γ is type preserving. Therefore, $\Gamma_{s|_p} \ \vdash_\Sigma l\delta\gamma : \tau$. Since $\Gamma \cdot \mathcal{R}an(\delta\gamma) \subseteq \Gamma_{s|_p}$ by our assumption and Definition 2.25(ii) and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \subseteq \mathcal{V}ar(\Gamma)$ by Definition 2.23(i), then $\Gamma \cdot \mathcal{R}an(\delta\gamma) \ \vdash_\Sigma l\delta\gamma : \tau$ by applications of lemma 2.5. Since γ is type preserving, $\Gamma \cdot \mathcal{R}an(\delta) \ \vdash_\Sigma l\delta : \tau$. By

Lemma 2.26, it follows that $\Gamma \cdot \mathcal{R}an(\delta) \vdash_\Sigma r\delta : \tau$, $\Gamma \cdot \mathcal{R}an(\delta\gamma) \vdash_\Sigma r\delta\gamma : \tau$, and $\Gamma \vdash_\Sigma t : \sigma$.

By polymorphism, $\Gamma\cdot\mathcal{R}an(\delta) \vdash_\Sigma l\delta : \tau \succ r\delta : \tau$. By stability, $\Gamma\cdot\mathcal{R}an(\delta\gamma) \vdash_\Sigma l\delta\gamma : \tau \succ r\delta\gamma : \tau$. By the previous remark that $\Gamma \cdot \mathcal{R}an(\delta\gamma) \subseteq \Gamma_{s|_p}$ and by compatibility, $\Gamma_{s|_p} \vdash_\Sigma l\delta\gamma : \tau \succ r\delta\gamma : \tau$. By monotonicity and Lemma 2.12, $\Gamma_{s|_p} \cdot \Gamma \vdash_\Sigma s[l\delta\gamma] : \sigma \succ s[r\delta\gamma] : \sigma$. By Definition 2.25(iii,iv), $\Gamma_{s|_p} \cdot \Gamma \vdash_\Sigma s : \sigma \succ t : \sigma$. By compatibility again, $\Gamma \vdash_\Sigma s : \sigma \succ t : \sigma$.

Finally, the case of a β- or an η-step follows from functionality. □

The polymorphic property of higher-order reduction orderings allows us to show termination in all monomorphic instances of the signature by means of a single comparison for each polymorphic rewrite rule. Polymorphic higher-order reduction orderings are therefore an appropriate tool in order to make termination proofs of polymorphic plain higher-order rewrite systems. In case of a monomorphic rewrite system, there is of course no need for a polymorphic ordering, even if the signature itself is polymorphic.

We are left with constructing polymorphic higher-order reduction orderings.

3. THE HIGHER-ORDER RECURSIVE PATH ORDERING

The higher-order recursive path ordering (HORPO) on higher-order terms is generated from three basic ingredients: a *type ordering*; a *precedence* on functions symbols; and a *status* for the function symbols. We describe these ingredients before defining the higher-order recursive path ordering, and study its properties, including strong normalization. We then consider a first set of examples which can all be run with our prototype implementation. Some proofs are omitted in this section, since they will be carried out in Section 4, in which a more advanced version of HORPO is studied.

3.1 The ingredients

—A quasi-ordering on types $\geq_{\mathcal{T}_\mathcal{S}}$ called *the type ordering* satisfying the following properties:

(1) *Well-foundedness*: $>_{\mathcal{T}_\mathcal{S}}$ is well-founded;

(2) *Arrow preservation*: $\tau \to \sigma =_{\mathcal{T}_\mathcal{S}} \alpha$ iff $\alpha = \tau' \to \sigma'$, $\tau' =_{\mathcal{T}_\mathcal{S}} \tau$ and $\sigma =_{\mathcal{T}_\mathcal{S}} \sigma'$;

(3) *Arrow decreasingness*: $\tau \to \sigma >_{\mathcal{T}_\mathcal{S}} \alpha$ implies $\sigma \geq_{\mathcal{T}_\mathcal{S}} \alpha$ or $\alpha = \tau' \to \sigma', \tau' =_{\mathcal{T}_\mathcal{S}} \tau$ and $\sigma >_{\mathcal{T}_\mathcal{S}} \sigma'$;

(4) *Arrow monotonicity*: $\tau \geq_{\mathcal{T}_\mathcal{S}} \sigma$ implies $\alpha \to \tau \geq_{\mathcal{T}_\mathcal{S}} \alpha \to \sigma$ and $\tau \to \alpha \geq_{\mathcal{T}_\mathcal{S}} \sigma \to \alpha$;

(5) *Stability*: (i) $\sigma >_{\mathcal{T}_\mathcal{S}} \tau$ implies $\sigma\xi >_{\mathcal{T}_\mathcal{S}} \tau\xi$ for all type substitution ξ;
(ii) $\sigma =_{\mathcal{T}_\mathcal{S}} \tau$ implies $\sigma\xi =_{\mathcal{T}_\mathcal{S}} \tau\xi$ for all type substitution ξ.

We show the existence of a type ordering generated by the above properties in Section 3.5.1, and give a particular type ordering in Section 5.1.

—A quasi-ordering $\geq_\mathcal{F}$ on $\mathcal{F}$, called the *precedence*, such that $>_\mathcal{F}$ is well-founded.

—A *status* $stat_f \in \{Mul, Lex\}$ for every symbol $f \in \mathcal{F}$, therefore partitioning $\mathcal{F}$ into $Mul \uplus Lex$. We say that f has a *multiset* or (left-to-right) *lexicographic* status if $f \in Mul$ and $f \in Lex$ respectively. We use a right-to-left status in one example.

3.2 The definition

The definition of the higher-order recursive path ordering builds upon Dershowitz's recursive path ordering (RPO) for first-order terms [Dershowitz 1982]. A major difference is that we do not compare terms, but terms together with their type in a given environment. A difficulty is that the environments in which two given terms are compared may change along recursive calls, and even change differently for the two recursively compared terms, which makes it impossible to factor the environment out. As a consequence, the ordering is defined on pairs of judgements $(\Gamma \ \vdash \ s : \sigma, \Sigma \ \vdash \ t : \tau)$ instead of on pairs of terms (s, t). Following Theorem 2.29, the starting comparison is of the form $\Gamma \ \vdash_\Sigma l : \sigma \succ \Gamma \ \vdash_\Sigma r : \sigma$ for some rule $\Gamma \ \vdash_\Sigma l \rightarrow r$, where σ is the principal type of l in Γ and a non-necessarily principal type of r in Γ. We are therefore using actual rather than principal types [1]. A second difficulty is that we need to compare subterms recursively, hence to have an appropriate typing judgement for a subterm in a term. Definition 2.10 provides the answer by giving us the actual type of a subterm in a term with respect to the typing judgement. In the sequel, all judgements $\Gamma \ \vdash_\Sigma s : \sigma$ we consider are therefore canonical, and σ is the actual type of s, allowing us to use the simpler notation $\vdash_\Sigma$ in place of the more precise $\vdash^c_\Sigma$.

Following the tradition, we consider equivalence classes of terms modulo α-conversion, using the syntactic equality symbol $=$ for the equivalence $=_\alpha$, and define our ordering $\succ_{horpo}$ by means of a set of rules, writing $\succeq_{horpo}$ for $\succ_{horpo} \cup =$. In contrast, the recursive path ordering contains a non-trivial equivalence allowing one to freely permute subterms below multiset function symbols [Dershowitz and Jouannaud 1990]. Using here a richer equivalence relation would raise technical complications for no practical benefit.

The definition starts with 4 cases reproducing Dershowitz's recursive path ordering for first-order terms, with one main difference when higher-order terms are compared: the rules can also take care of higher-order terms in the arguments of the smaller side by having a corresponding bigger higher-order term in the arguments of the bigger side. This is redundant for first-order terms because of the subterm property. This idea is captured in the following *weak subterm property* used throughout the definition:

$$A = \forall v \in \overline{t}\ (\Gamma \vdash_\Sigma s : \sigma) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma v : \rho) \text{ or } (\Gamma \vdash_\Sigma u : \theta) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma v : \rho) \text{ for some } u \in \overline{s}$$

Cases 5 and 6 allow one to use the subterm case for applications and abstractions. In the abstraction case, we may need to rename the bound variable x of s to prevent confusing it with a free variable of t. Cases 7 and 8 are precedence cases for comparing a term headed by an algebraic symbol with a term headed by an application or an abstraction. Case 7 is crucial for the usefulness of the ordering. Cases 9 and 10 allow one to compare terms headed both by applications or both by lambdas. They are essential for monotonicity. Cases 11 and 12 include respectively β- and η-reductions in the ordering.

[1] Using principal types would require that l and r have the same principal type in Γ to make sure that they are comparable. We have actually implemented both choices, and all our examples run on both implementations.

Definition 3.1. Given two judgements $\Gamma \vdash_\Sigma s : \sigma$ and $\Sigma \vdash_\Sigma t : \tau$,

$$(\Gamma \vdash_\Sigma s : \sigma) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma t : \tau) \text{ iff } \sigma \geq_{\mathcal{T}_S} \tau \text{ and}$$

(1) $s = f(\overline{s})$ with $f \in \mathcal{F}$, and $(\Gamma \vdash_\Sigma u : \theta) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$ for some $u \in \overline{s}$

(2) $s = f(\overline{s})$ with $f \in \mathcal{F}$, and $t = g(\overline{t})$ with $f >_\mathcal{F} g$, and A

(3) $s = f(\overline{s})$, $t = g(\overline{t})$ with $f =_\mathcal{F} g \in Mul$, and $(\Gamma \vdash_\Sigma \overline{s} : \overline{\sigma})\ (\underset{horpo}{\succ})_{mul}\ (\Sigma \vdash_\Sigma \overline{t} : \overline{\tau})$

(4) $s = f(\overline{s})$, $t = g(\overline{t})$ with $f =_\mathcal{F} g \in Lex$, and $(\Gamma \vdash_\Sigma \overline{s} : \overline{\sigma})\ (\underset{horpo}{\succ})_{lex}\ (\Sigma \vdash_\Sigma \overline{t} : \overline{\tau})$ and A

(5) $s = @(s_1, s_2)$, and $(\Gamma \vdash_\Sigma s_1 : \rho \rightarrow \sigma) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$ or

$$(\Gamma \vdash_\Sigma s_2 : \rho) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$$

(6) $s = \lambda x : \alpha.u$ with $x \notin \mathcal{V}ar(t)$, and $(\Gamma \cdot \{x : \alpha\} \vdash_\Sigma u : \theta) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$

(7) $s = f(\overline{s})$ with $f \in \mathcal{F}$, $t = @(\overline{t})$ is a partial left-flattening of t, and A

(8) $s = f(\overline{s})$ with $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ with $x \notin \mathcal{V}ar(v)$ and $(\Gamma \vdash_\Sigma s : \sigma) \underset{horpo}{\succ} (\Sigma \vdash_\Sigma v : \rho)$

(9) $s = @(s_1, s_2)$, $t = @(\overline{t})$ is a partial left-flattening of t, and $\{(\Gamma \vdash_\Sigma s_1 : \theta \rightarrow \sigma), (\Gamma \vdash_\Sigma s_2 : \theta)\}(\underset{horpo}{\succ})_{mul}(\Sigma \vdash_\Sigma \overline{t} : \overline{\tau})$

(10) $s = \lambda x : \alpha.u$, $t = \lambda x : \beta.v$, $\alpha =_{\mathcal{T}_S} \beta$, and $(\Gamma \cdot \{x : \alpha\} \vdash_\Sigma u : \theta) \underset{horpo}{\succ} (\Sigma \cdot \{x : \beta\} \vdash_\Sigma v : \rho)$

(11) $s = @(\lambda x : \alpha.u, v)$ and $(\Gamma \vdash_\Sigma u\{x \mapsto v\} : \sigma) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$

(12) $s = \lambda x : \alpha.@(u, x)$, $x \notin \mathcal{V}ar(u)$ and $(\Gamma \vdash_\Sigma u : \sigma) \underset{horpo}{\succeq} (\Sigma \vdash_\Sigma t : \tau)$

The definition is recursive, and, apart from case 11, recursive calls operate on judgements whose terms are subterms of the term in the starting judgement. This ensures the well-foundedness of the definition, since the union of β-reduction with subterm is well-founded, by comparing pairs of argument terms in the well-founded compatible relation $(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$. This will actually be used as an inductive argument in many proofs to come. It must be stressed that multiset and lexicographic extensions are defined for arbitrary relations and preserve well-foundedness of arbitrary (well-founded) relations. As a consequence, our definition is inductive.

Carrying judgements systematically would often make it look awkward. Therefore, from now on, *we indulge forgetting judgements when comparing terms, leaving environments and types implicit when possible.*

Example 3.2. (Example 2.27 continued). We use here the existence of an ordering on types generated by the properties of the type ordering introduced in Section 3.1, and assume a multiset status for *rec*. The first rule succeeds immediately by case 1. For the second, we apply case 7, and need to show recursively that (i) $X \succeq_{horpo} X$, (ii) $s(x) \succ_{horpo} x$, and (iii) $rec(s(x), u, X) \succ_{horpo} rec(x, u, X)$.
(i) is clear. (ii) is by case 1. (iii) is by case 3, calling again recursively for $s(x) \succ_{horpo} x$.

Note that we have proved termination of Gödel's polymorphic recursor, for which the output type of rec is any given type. This is so because our ordering is polymorphic, as we will show. This example was already proved in [Jouannaud and Rubio 1999], where polymorphism was claimed but neither formally defined nor proved.

We can of course now add some defining rules for sum and product for which we omit environments and types for bound variables which can be easily inferred by the reader:

$$\begin{array}{rcl} x+0 & \rightarrow & 0 \\ x+s(y) & \rightarrow & s(x+y) \\ x*y & \rightarrow & rec(y,0,\lambda z_1 z_2.x+z_2) \end{array}$$

The first two rules are easy work. For the third, we use the precedence $* >_{\mathcal{F}} rec$ to eliminate the rec operator. But the computation fails, since no subterm of $x*y$ can take care of the righthand side subterm $\lambda z_1 z_2.x+z_2$. We will come back to this example later, after having boosted the ordering.

Example 3.3. We consider here lists of natural numbers. Let

$$\begin{array}{l} \mathcal{S} = \{List, \mathbb{N}\} \\ \mathcal{F} = \{nil : List,\ cons : \mathbb{N} \times List \Rightarrow List,\ map : List \times (\mathbb{N} \rightarrow \mathbb{N}) \Rightarrow List\} \end{array}$$

The rules for map are:

$$\begin{array}{ccrcl} \{X : \mathbb{N} \rightarrow \mathbb{N}\} & \vdash & map(nil, X) & \rightarrow & nil \\ \{x : \mathbb{N}, l : List, X : \mathbb{N} \rightarrow \mathbb{N}\} & \vdash & map(cons(x,l),X) & \rightarrow & cons(@(X,x), map(l,X)) \end{array}$$

We use the ordering on types generated as previously by the properties of the type ordering and the additional equality $\mathbb{N} =_{\mathcal{T}_{\mathcal{S}}} List$. For the precedence, we take $map >_{\mathcal{F}} cons$, and $map \in Mul$ for the statuses.

The first rule is easily taken care of by case 1. For the second, since $map >_{\mathcal{F}} cons$, applying case 2 yields the subgoals $map(cons(x,l),X) \succ_{horpo} @(X,x)$ and $map(cons(x,l),X) \succ_{horpo} map(l,X)$. The latter is true by case 3, since, by case 1, $cons(x,l) \succ_{horpo} l$. The first is by case 7, as X is an argument of the first term and $cons(x,l) \succ_{horpo} x$ by case 1. Note that we use the type comparison $List =_{\mathcal{T}_{\mathcal{S}}} \mathbb{N}$ in this computation.

Example 3.4. We now consider a specification of polymorphic lists. Let

$$\begin{array}{c} \mathcal{S}^{\forall} = \{\alpha\}; \mathcal{S} = \{List : * \Rightarrow *\}; \\ \mathcal{F} = \{nil : \alpha, cons : \alpha \times List(\alpha) \Rightarrow List(\alpha), map : List(\alpha) \times (\alpha \rightarrow \alpha) \Rightarrow List(\alpha)\} \end{array}$$

The rules for map become:

$$\begin{array}{ccrcl} \{X : \alpha \rightarrow \alpha\} & \vdash & map(nil, X) & \rightarrow & nil \\ \{x : \alpha, l : List(\alpha), X : \alpha \rightarrow \alpha\} & \vdash & map(cons(x,l),X) & \rightarrow & cons(@(X,x), map(l,X)) \end{array}$$

Letting again $map \in Mul$ and $map >_{\mathcal{F}} cons$, the first rule is taken care of by case 1 as previously. For the second, we need to set $List(\alpha) >_{\mathcal{T}_{\mathcal{S}}} \alpha$ for the subgoal $map(cons(x,l),X) \succ_{horpo} @(X,x)$. The computation goes then as previously.

Important observations are the following:

—HORPO compares terms of comparable types, improving over [Jouannaud and Rubio 1999] where terms of equivalent types only could be compared, as done in Example 3.3. Without type comparisons being based on an ordering instead of an equivalence, we could never allow a subterm case for terms headed by an application or by an abstraction. Subterm cases, however, must be controlled by the type comparison for the ordering to be well-founded.

—Another important improvement for practice is the inclusion of both β- and η-reductions in the ordering via Cases 11 and 12. We will see several examples where it is used.

—Note that there is no precedence case for applications against abstractions and that partial-flattening is used in the righthand sides only, that is, in Cases 7 and 9, and indeed, our strong normalization proof does not go through if it is also used in the lefthand sides. Left-flattening is essential for stability. The choice of a particular partial left-flattening in practice is considered in Section 5.

—HORPO is unfortunately not transitive, therefore explaining our statement in Theorem 3.5. Removing Cases 11 and 12, as well as left-flattening in Cases 7 and 9 yields a transitive ordering having the weak-subterm property. Some examples will indeed need using one step of transitivity, which raises some difficulties for the implementation addressed in Section 5.

—When the signature is first-order, Cases 1, 2, 3 and 4 of the definition of $\succ_{horpo}$ together reduce to the usual recursive path ordering for first-order terms, whose complexity is known to be in $O(n^2)$ for an input of size n. Inferring a precedence for comparing two given terms is also known to be NP-complete. This is not a problem for practice since signatures and terms are usually small. We have not investigated the complexity of HORPO, because an unbounded use of Case 11 may result in an arbitrary high complexity. We think however that both results still hold for HORPO when dropping Case 11 or restricting its use.

We now state the main result of this section, whose proof is the subject of the coming two subsections:

Theorem 3.5. $(\succ_{horpo})^+$ *is a decidable polymorphic, higher-order reduction ordering.*

3.3 Candidate Terms

Because our strong normalization proof is based on Tait and Girard's reducibility technique, we need to associate to each type σ, actually to the equivalence class of σ modulo $=_{\mathcal{T}_S}$, a set of terms $[\![\sigma]\!]$ closed under certain operations. In particular, if $s \in [\![\sigma \to \tau]\!]$ and $t \in [\![\sigma]\!]$, then the raw term $@(s,t)$ must belong to the set $[\![\tau]\!]$ even if it is not typable. In this section, we give a more liberal type system in which all raw terms needed in the strong normalization proof become typable *candidate terms.*

Definition 3.6. A raw term is a *candidate terms* if it is typable in the type system of Figure 2.

The set of types of a typable candidate term of type σ is a union of type equivalence classes modulo $=_{\mathcal{T}_S}$:

Lemma 3.7. *$\forall \sigma\tau$ such that $\sigma =_{\mathcal{T}_S} \tau$, $\Gamma \vdash_\Sigma s :_C \sigma$ iff $\Gamma \vdash_\Sigma s : \tau$.*

This allows us to talk about *the types* of a candidate term up to type equivalence. Note finally five easy, important properties of candidate terms:

Variables:

$$\frac{x : \sigma \in \Gamma \quad \sigma' =_{\mathcal{T}_{\mathcal{S}}} \sigma}{\Gamma \ \vdash_{\Sigma} x :_{C} \sigma'}$$

Abstraction:

$$\frac{\Gamma \cdot \{x : \sigma\} \ \vdash_{\Sigma} t :_{C} \tau \quad \theta =_{\mathcal{T}_{\mathcal{S}}} \sigma}{\Gamma \ \vdash_{\Sigma} (\lambda x : \sigma.t) :_{C} \theta \rightarrow \tau}$$

Instantiation:

$$\frac{\begin{array}{c}\Gamma \ \vdash_{\Sigma} s :_{C} \sigma \\ \xi \text{ a ground type substitution} \\ \text{of domain } \mathcal{V}ar(\sigma) \\ \sigma\xi =_{\mathcal{T}_{\mathcal{S}}} \tau\end{array}}{\Gamma\xi \ \vdash_{\Sigma} s\xi :_{C} \tau}$$

Application:

$$\frac{\begin{array}{c}\Gamma \ \vdash_{\Sigma} s :_{C} \sigma \quad \Gamma \ \vdash_{\Sigma} t :_{C} \tau \\ \xi = mgu(\alpha \rightarrow \beta = \sigma \wedge \alpha = \tau) \\ \beta\xi =_{\mathcal{T}_{\mathcal{S}}} \tau\end{array}}{\Gamma\xi \ \vdash_{\Sigma} @(s,t)\xi :_{C} \tau}$$

Functions:

$$\frac{\begin{array}{c}f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma \in \mathcal{F} \\ \Gamma \ \vdash_{\Sigma} t_1 :_{C} \tau_1 \ \ldots \ \Gamma \ \vdash_{\Sigma} t_n :_{C} \tau_n \\ \xi = mgu(\sigma_1 = \tau_1 \wedge \ldots \wedge \sigma_n = \tau_n) \\ \tau =_{\mathcal{T}_{\mathcal{S}}} \sigma\xi\end{array}}{\Gamma\xi \ \vdash_{\Sigma} f(t_1, \ldots, t_n)\xi :_{C} \tau}$$

Fig. 2. Typing judgements for candidate terms

(1) Every term is a candidate term;

(2) Typing Lemmas 2.4, 2.5, 2.9, 2.12 and 2.21 hold for candidate terms;

(3) HORPO applies to candidate terms as well as to terms, by keeping the same definition;

(4) Because β-reduction is type preserving, the set of candidate terms is closed under β-reductions;

(5) Because β-reduction is strongly normalizing for a typed λ-calculus with arbitrary constants, the set of candidate terms is well-founded with respect to β-reductions (for the argument, consider a signature $\mathcal{F}'$ in which $f : \sigma'_1 \times \ldots \times \sigma'_n \Rightarrow \sigma' \in \mathcal{F}'$ provided $f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma \in \mathcal{F}$ with $\sigma'_i =_{\mathcal{T}_{\mathcal{S}}} \sigma_i$ for every $i \in [1..n]$ and $\sigma' =_{\mathcal{T}_{\mathcal{S}}} \sigma$).

To make sense of the third observation, we need to show that $\succ_{horpo}$, when comparing candidate terms, is compatible with type equivalence.

Lemma 3.8. *Assume given candidate terms s, t and types σ, σ', τ, τ' such that $s :_{C} \sigma \succ_{horpo} t :_{C} \tau$, $\sigma =_{\mathcal{T}_{\mathcal{S}}} \sigma'$ and $\tau =_{\mathcal{T}_{\mathcal{S}}} \tau'$. Then $s :_{C} \sigma' \succ_{horpo} t :_{C} \tau'$.*

Proof. By Lemma 3.7, $s :_{C} \sigma'$ and $t :_{C} \tau'$, hence the statement makes sense. Now, since $=_{\mathcal{T}_{\mathcal{S}}} \subseteq \geq_{\mathcal{T}_{\mathcal{S}}}$, then $\sigma \geq_{\mathcal{T}_{\mathcal{S}}} \tau$ implies $\sigma' \geq_{\mathcal{T}_{\mathcal{S}}} \tau'$ by transitivity.

The proof of the ordering statement proceeds by induction on $(\longrightarrow_{\beta} \cup \rhd, \rhd)_{lex}$, and by case on the definition of the ordering, using for each case the induction hypothesis together with Lemma 3.7. □

3.4 Ordering properties of HORPO

Because the strong normalization proof requires using candidate terms, two results of this section, monotonicity and stability, which are used in the strong normalization proof, must be stated and proved for both terms and candidate terms. Since terms are closed under taking subterms, proofs are indeed essentially the same in both cases. We do them for candidate terms.

Lemma 3.9 Stability. *$\succ_{horpo}$ is stable for candidate terms and for terms.*

PROOF. Let $\Gamma \ \vdash_{\Sigma} s :_C \sigma \succ_{horpo} t :_C \tau$ and γ be a type preserving substitution compatible with Γ. By Lemma 2.21, $\Gamma \cdot \mathcal{R}an(\gamma) \ \vdash_{\Sigma} s\gamma :_C \sigma$ and $\Gamma \cdot \mathcal{R}an(\gamma) \ \vdash_{\Sigma} t\gamma :_C \tau$. We show that $\Gamma \cdot \mathcal{R}an(\gamma) \ \vdash_{\Sigma} s\gamma :_C \sigma \succ_{horpo} t\gamma :_C \tau$ by induction on $(\longrightarrow_{\beta} \cup \rhd, \rhd)_{lex}$. Since type preserving substitutions preserve types, we will from now on only consider the term comparisons of the ordering and omit all references to types and judgements. There are 12 cases according to the definition:

(1) If $s \succ_{horpo} t$ by case 1 then $s_i \succeq_{horpo} t$, and by induction hypothesis $s_i\gamma \succeq_{horpo} t\gamma$ and therefore, $s\gamma \succ_{horpo} t\gamma$ by case 1.

(2) If $s \succ_{horpo} t$ by case 2 then $s = f(\overline{s})$, $t = g(\overline{t})$, $f >_{\mathcal{F}} g$ and for all $t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$. By induction hypothesis, for all $t_i\gamma \in \overline{t\gamma}$ either $s\gamma \succ_{horpo} t_i\gamma$ or $s_j\gamma \succeq_{horpo} t_i\gamma$ for some $s_j\gamma \in \overline{s\gamma}$. Therefore, $s\gamma = f(\overline{s\gamma}) \succ_{horpo} g(\overline{t\gamma}) = t\gamma$ by case 2.

(3) If $s \succ_{horpo} t$ by case 3, then $s = f(\overline{s})$, $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, $f, g \in Mul$ and $\overline{s}(\succ_{horpo})_{mul}\overline{t}$. By induction hypothesis we have $\overline{s\gamma}(\succ_{horpo})_{mul}\overline{t\gamma}$, and hence $s\gamma \succ_{horpo} t\gamma$ by case 3.

(4) If $s \succ_{horpo} t$ by case 4, then $s = f(\overline{s})$, $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, $f, g \in Lex$, $\overline{s}(\succ_{horpo})_{lex}\overline{t}$, and for all $t_i \in \overline{t}$ either $s \succ_{horpo} t_i$ or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$. By induction hypothesis $\overline{s\gamma}(\succ_{horpo})_{lex}\overline{t\gamma}$, and as in the precedence case, by induction hypothesis, for all $t_i\gamma \in \overline{t\gamma}$ either $s\gamma \succ_{horpo} t_i\gamma$ or $s_j\gamma \succeq_{horpo} t_i\gamma$ for some $s_j\gamma \in \overline{s\gamma}$. Therefore $s\gamma \succ_{horpo} t\gamma$ by case 4.

(5) If $s \succ_{horpo} t$ by case 5, the reasoning is similar to Case 1.

(6) If $s \succ_{horpo} t$ by case 6, then $s = \lambda x.u$ and $u \succeq_{horpo} t$. Let γ a substitution of domain $\mathcal{V}ar(s) \cup \mathcal{V}ar(t)$, hence $x \notin \mathcal{D}om(\gamma)$ by assumption on x. By induction hypothesis, $u\gamma \succeq_{horpo} t\gamma$, hence $s\gamma = \lambda x.u\gamma \succ_{horpo} t\gamma$ by case 6.

(7) If $s \succ_{horpo} t = @(\overline{t})$ by case 7, then for every $t_i \in \overline{t}$, either $s \succ_{horpo} t_i$, and $s\gamma \succ_{horpo} t_i\gamma$ by induction hypothesis, or $s_j \succeq_{horpo} t_i$ for some $s_j \in \overline{s}$, and $s_j\gamma \succ_{horpo} t_i\gamma$ by induction hypothesis. Therefore, since $@(\overline{t\gamma})$ is a partial left-flattening of $t\gamma$, $s\gamma \succ_{horpo} t\gamma$ by case 7.

(8) If $s \succ_{horpo} t$ by Case 8, then $t = \lambda x.v$ with $x \notin \mathcal{V}ar(v)$ and $s \succ_{horpo} v$. By induction hypothesis, $s\gamma \succ_{horpo} v\gamma$ for every substitution γ of domain $\mathcal{V}ar(v) \setminus \{x\}$ such that $x \notin \mathcal{V}ar(v\gamma)$, hence $s\gamma \succ_{horpo} t\gamma = \lambda x.v\gamma$ by Case 8.

(9) If $s = @(s_1, s_2) \succ_{horpo} t = @(\overline{t})$ by case 9, then the proof goes as in case 2, concluding by case 9.

(10) If $s \succ_{horpo} t$ by case 10, then $s = \lambda x.u$, $t = \lambda x.v$ and $u \succ_{horpo} v$. By induction hypothesis $u\gamma \succ_{horpo} v\gamma$. Assuming that $x \notin \mathcal{D}om(\gamma)$, then $s\gamma = \lambda x.u\gamma \succ_{horpo} \lambda x.v\gamma = t\gamma$ by case 10.

(11) If $s = \succ_{horpo} t$ by case 11, then the property holds by stability of β-reduction.

(12) If $s = \succ_{horpo} t$ by case 12, the property holds by stability of η-reduction. □

LEMMA 3.10 POLYMORPHISM. *$\succ_{horpo}$ is polymorphic for candidate terms and for terms.*

PROOF. Let $\Gamma \vdash_{\Sigma} s :_C \sigma \succ_{horpo} t :_C \tau$ and δ be a specialization compatible with Γ. By Lemma 2.22, $\Gamma \cdot \mathcal{R}an(\delta) \ \vdash_{\Sigma} s\delta :_C \sigma\xi_{\delta}$ and $\Gamma \cdot \mathcal{R}an(\delta) \ \vdash_{\Sigma} t\delta :_C \tau\xi_{\delta}$. We show that $\Gamma \cdot \mathcal{R}an(\delta) \ \vdash_{\Sigma} s\delta :_C \sigma\xi_{\delta} \succ_{horpo} t\delta :_C \tau\xi_{\delta}$ as before, by induction on

$(\longrightarrow_\beta \cup \rhd, \rhd)_{lex}$. Since the type ordering is stable, $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ implies $\sigma\xi_\delta \geq_{\mathcal{T}_\mathcal{S}} \tau\xi_\delta$ and the proof continues as previously, using the stability of the type ordering for each recursive comparison. □

Lemma 3.11 Monotonicity. *$\succ_{horpo}$ is monotonic for candidate terms and for terms.*

Proof. Omitting the environment, we prove that $s :_C \sigma \succ_{horpo} t :_C \sigma$ implies $u[s] :_C \tau \succ_{horpo} u[t] :_C \tau$ for all $u[x : \sigma] :_C \tau$.

We proceed by induction on the size of u. If u is empty it holds. Otherwise there are three cases:

—$u[x : \sigma]$ is of the form $u'[f(\ldots, x : \sigma, \ldots)]$ for some function symbol f. If $f \in Mul$, then $f(\ldots s \ldots) :_C \theta \succ_{horpo} f(\ldots t \ldots) :_C \theta$ follows by Case 3, and we conclude by induction since u' is smaller than u. Otherwise, $f \in Lex$, hence $\{\ldots s \ldots\}(\succ_{horpo})_{lex}\{\ldots t \ldots\}$ and, for every $v \in \{\ldots t \ldots\}$, there exists $u \in \{\ldots s \ldots\}$ such that $u \succeq_{horpo} v$. Therefore, $f(\ldots s \ldots) \succ_{horpo} f(\ldots t \ldots)$ by Case 4 and we conclude by induction hypothesis as before.

—$u[x : \sigma]$ is of the form $u'[@(u, x : \sigma)]$ (resp. $u'[@(x : \sigma, u)]$). Then, $@(s, u) : \theta \succ_{horpo} @(t, u) : \theta$ (resp. $@(u, s) : \theta \succ_{horpo} @(u, t) : \theta$) follows by Case 9. We conclude by induction hypothesis.

—$u[x : \sigma]$ is of the form $u'[\lambda y.x]$. The result follows similarly by Case 10. □

From Lemmas 3.11 and 3.8, we easily get a generalized monotonicity property of $\succ_{horpo}$ for candidate terms of equivalent type. In the rest of the paper we will make use of this generalized monotonicity property by referring to Lemma 3.11. Note also that the monotonicity property of $\succ_{horpo}$ implies the monotonicity of the ordering $\succeq_{horpo}$ on terms of equivalent type.

A key usual consequence of monotonicity is that every subterm of a strongly normalizable term is itself strongly normalizable. This is true when decreasing sequences of terms are type preserving, since an infinite decreasing sequence originating in a subterm can be lifted in an infinite decreasing sequence originating in the superterm. This becomes false otherwise, since typing prevents the lifting.

Lemma 3.12. *$\succ_{horpo}$ is compatible.*

This is so because types, hence comparisons, are preserved by the use of a compatible environment.

Lemma 3.13. *$\succ_{horpo}$ is functional.*

Proof. By definition and monotonicity. □

Although the above proofs are slightly more difficult technically than the usual proofs for the recursive path ordering, they follow the same kind of pattern (polymorphism was actually never considered before). This contrasts with the proof of strong normalization to come.

3.5 Strong Normalization

In this section, we consider the well-foundedness of the strict ordering $(\succ_{horpo})^+$, that is, equivalently, the strong normalization of the rewrite relation defined by the rules $s \longrightarrow t$ such that $s \succ_{horpo} t$. For the recursive path ordering, well-foundedness follows from Kruskal's tree theorem. Since we do not know of any non-trivial extension of Kruskal's tree theorem for higher-order terms that includes β-reductions (and wasted much of our time in looking for an appropriate one), we adopt a completely different method, the computability predicate proof method of Tait and Girard. To our knowledge, the use of this method for proving well-foundedness of a recursively defined relation is original. This proof method will suggest an important improvement of our ordering discussed in Section 4.

A proof based on that method starts by defining for each type σ, a set of terms called the computability predicate $[\![\sigma]\!]$. Terms in $[\![\sigma]\!]$ are said to be computable. So are substitutions made of computable terms. In practice, $[\![\sigma]\!]$ can be defined by the properties it should satisfy. The most important one is that computable terms must be strongly normalizable. Given an arbitrary computable, hence strongly normalizable substitution γ, one can then build an induction argument to prove that, for an arbitrary term t, the term $t\gamma$ is computable. The identity substitution being computable, we then conclude that t is computable, hence strongly normalizable. See [Girard et al. 1989] for a detailed exposition of the method in case of system F, and [Gallier 1990] for a discussion about the different possibilities for defining computability predicates in practice.

3.5.1 *Type orderings.* Two lemmas are needed to justify our coming construction of the candidate interpretations: preservation of groundness first; second, any type ordering is included into an ordering enjoying the subterm property.

LEMMA 3.14. *Assume σ is ground and $\sigma \geq_{\mathcal{T}_S} \tau$. Then τ is ground as well.*

PROOF. Because the type ordering is both stable and well-founded. □

LEMMA 3.15. *Let $\geq_{\mathcal{T}_S}$ be a quasi-ordering on types such that $>_{\mathcal{T}_S}$ is well-founded, arrow monotonic and arrow preserving. Then, the relation $\geq_{\overrightarrow{\mathcal{T}_S}} = (\geq_{\mathcal{T}_S} \cup \rhd_{\rightarrow})^*$ is a well-founded quasi-ordering on types extending $\geq_{\mathcal{T}_S}$ and $\rhd_{\rightarrow}$, whose equivalence coincides with $=_{\mathcal{T}_S}$.*

PROOF. First, we show that $\rhd_{\rightarrow}$ and $\geq_{\mathcal{T}_S}$ commute, i.e that $\geq_{\mathcal{T}_S} \cdot \rhd_{\rightarrow} \;\subseteq\; \rhd_{\rightarrow} \cdot \geq_{\mathcal{T}_S}$. Assuming first that $\tau \rightarrow \sigma \rhd_{\rightarrow} \sigma \geq_{\mathcal{T}_S} \rho$, then, by arrow monotonicity, $\tau \rightarrow \sigma \geq_{\mathcal{T}_S} \tau \rightarrow \rho \rhd_{\rightarrow} \rho$. Assuming now that $\tau \rightarrow \sigma \rhd_{\rightarrow} \tau$ yields a similar computation. By transitivity of $\geq_{\mathcal{T}_S}$, it follows that $\geq_{\overrightarrow{\mathcal{T}_S}} = \geq_{\mathcal{T}_S} \cdot \rhd^{*}_{\rightarrow}$.

We show now that $=_{\mathcal{T}_S} \;=\; =_{\overrightarrow{\mathcal{T}_S}}$. Assume that $\sigma \geq_{\mathcal{T}_S} \rho_1 \rhd^{n}_{\rightarrow} \tau \geq_{\mathcal{T}_S} \rho_2 \rhd^{m}_{\rightarrow} \sigma$ for types ρ_1 and ρ_2, and natural numbers n, m. By applying commutation twice followed by transitivity of $\geq_{\mathcal{T}_S}$, we get $\sigma \rhd^{m+n}_{\rightarrow} \theta \geq_{\mathcal{T}_S} \sigma$. Therefore, $\theta = u[\sigma]_p$ for some context u and position p with $|p| = m + n$, such that there is an arrow in u at each position $q < p$. Assume that $m + n > 0$. By arrow preservation $\sigma \neq_{\mathcal{T}_S} u[\sigma]_p$, hence $\sigma >_{\mathcal{T}_S} u[\sigma]_p$. By arrow monotonicity, $u[\sigma] \geq_{\mathcal{T}_S} u[u[\sigma]]$ and by arrow preservation again, $u[\sigma] >_{\mathcal{T}_S} u[u[\sigma]]$, and so on, resulting in an infinite sequence $\sigma >_{\mathcal{T}_S} u[\sigma] >_{\mathcal{T}_S} u[u[\sigma]] >_{\mathcal{T}_S} \ldots$ which contradicts the well-foundedness of $>_{\mathcal{T}_S}$. The property follows.

Since $\tau \rightarrow \sigma \geq_{\overrightarrow{\mathcal{T}_S}} \tau$ and $\tau \rightarrow \sigma \geq_{\overrightarrow{\mathcal{T}_S}} \sigma$ by definition of $\geq_{\overrightarrow{\mathcal{T}_S}}$, and $=_{\mathcal{T}_S} \;=\; =_{\overrightarrow{\mathcal{T}_S}}$, arrow preservation implies that $\tau \rightarrow \sigma >_{\overrightarrow{\mathcal{T}_S}} \tau$ and $\tau \rightarrow \sigma >_{\overrightarrow{\mathcal{T}_S}} \sigma$.

We are left with well-foundedness. From $=_{\mathcal{T}_S} \;=\; =_{\overrightarrow{\mathcal{T}_S}}$ and $\geq_{\overrightarrow{\mathcal{T}_S}} = \geq_{\mathcal{T}_S} \cdot \rhd^{*}_{\rightarrow}$, it follows that $>_{\overrightarrow{\mathcal{T}_S}} \subseteq >_{\mathcal{T}_S} \cup \geq_{\mathcal{T}_S} \cdot \rhd^{+}_{\rightarrow}$. We show that the latter is well-founded. Since

$>_{\mathcal{T}_{\mathcal{S}}}$ is well-founded and $>_{\mathcal{T}_{\mathcal{S}}} \subseteq \geq_{\mathcal{T}_{\mathcal{S}}}$, any infinite sequence with $(>_{\mathcal{T}_{\mathcal{S}}} \cup \geq_{\mathcal{T}_{\mathcal{S}}} \cdot \rhd^{+}_{\rightarrow})$ is an infinite sequence with $\geq_{\mathcal{T}_{\mathcal{S}}} \cdot \rhd^{+}_{\rightarrow}$. Commutation and well-foundeness of $\rhd_{\rightarrow}$ yield the contradiction. □

3.5.2 *Candidate interpretations.* Again, this section refers to candidate terms, rather than to candidate judgements. Computability of candidate terms of variable type is reduced to the computability of candidate terms of ground type by type instantiation. Our definition of computability for candidate terms of a ground type is standard, but we must make sure that it is compatible with the equivalence $=_{\mathcal{T}_{\mathcal{S}}}$ on types. Technically, we denote by $[\![\sigma]\!]$ the computability predicate of the type σ, which, by construction, will be equal to the predicate $[\![\tau]\!]$ for any type $\tau =_{\mathcal{T}_{\mathcal{S}}} \sigma$. Without loss of generality, we assume a global environment for variables.

Definition 3.16. The family of *candidate interpretations* $\{[\![\sigma]\!]\}_{\sigma \in \mathcal{T}_{\mathcal{S}}}$ is a family of subsets of the set of candidates whose elements are the least sets satisfying the following properties:

(i) If σ is a ground data type, then $s :_C \sigma \in [\![\sigma]\!]$ iff $t \in [\![\tau]\!]$ for all $t :_C \tau$ such that $s \succ_{horpo} t$

(ii) If σ is a ground functional type $\rho \rightarrow \tau$ then $s \in [\![\sigma]\!]$ iff $@(s,t) \in [\![\tau]\!]$ for all $t :_C \in [\![\rho]\!]$;

(iii) If σ is not ground, then $s \in [\![\sigma]\!]$ iff $s \in [\![\sigma\xi]\!]$ for all ground type substitutions ξ.

A candidate term s of type σ is said to be *computable* if $s \in [\![\sigma]\!]$. A vector $\overline{s}$ of terms of type $\overline{\sigma}$ is computable iff so are all its components. A (candidate) term substitution γ is computable if all candidate terms in $\{x\gamma \mid x \in \mathcal{D}om(\gamma)\}$ are computable.

Case (iii) in the definition allows us to eliminate non-ground types by appropriate type instantiations: all properties of candidate interpretations proved for ground types can then be easily lifted to non-ground ones. This is so because Cases (i) and (ii) of the definition refer to ground types only: in Case (ii), ρ and τ are ground since $\rho \rightarrow \tau$ is ground; in Case (i), τ is ground by Lemma 3.14. Therefore, $[\![\sigma]\!]$ is a subset of the set of candidate terms of type σ recursively defined in terms of itself in Case (i) when $\tau =_{\mathcal{T}_{\mathcal{S}}} \sigma$, and of other sets $[\![\tau]\!]$, in Case (i) and (ii) with $\sigma >_{\mathcal{T}_{\mathcal{S}}} \tau$, and $[\![\rho]\!]$ with $\sigma >^{\rightarrow}_{\mathcal{T}_{\mathcal{S}}} \rho$. Therefore, our definition splits into two: a definition of candidate interpretations on ground types, based on a lexicographic combination of an induction on the well-founded type ordering $>^{\rightarrow}_{\mathcal{T}_{\mathcal{S}}}$ (which includes $>_{\mathcal{T}_{\mathcal{S}}}$), and a fixpoint computation for data types; a definition of candidate interpretations for non-ground types which reduces to the definition on ground types.

Instead of our indexed family of sets, we could consider the function $F : \mathcal{T}_{\mathcal{S}} \longrightarrow 2^{\mathcal{T}}$ such that $F(\sigma) = [\![\sigma]\!]$. This function F is defined by induction on types (using $>^{\rightarrow}_{\mathcal{T}_{\mathcal{S}}}$). Besides, for each date type σ, $F(\sigma)$ is defined by a fixpoint computation (Case (i) with $\tau =_{\mathcal{T}_{\mathcal{S}}} \sigma$). Since Case (i) does not involve any negation, it is monotonic with respect to set inclusion, therefore ensuring the existence of a least fixpoint.

The choice of a particular formulation for the computability predicate is entirely driven by the *computability properties* one needs. Most of these are proved for ground types by an "induction on the definition of the candidate interpretations", by which we mean an outer induction on the type ordering $>^{\rightarrow}_{\mathcal{T}_{\mathcal{S}}}$ followed by an inner induction on the fixpoint computation. Since the ordering computations involve subterms, a potential difficulty is that a term of a ground type may have subterms of a non-ground type. Fortunately, type comparisons will imply the type-groundness property when needed.

We denote by $\mathcal{T}_{\mathcal{S}}^{min}$ the set of ground types which are minimal with respect to $>_{\mathcal{T}_{\mathcal{S}}}^{\rightarrow}$.

LEMMA 3.17. *Assuming that $\mathcal{S}$ contains a constant, then $\mathcal{T}_{\mathcal{S}}^{min}$ is a non-empty set of data types.*

PROOF. Because $>_{\mathcal{T}_{\mathcal{S}}}^{\rightarrow}$ is well-founded and arrow types cannot be minimal in $>_{\mathcal{T}_{\mathcal{S}}}^{\rightarrow}$. □

Preservation of data types follows easily from arrow preservation and stability:

LEMMA 3.18. *Assume that $\sigma =_{\mathcal{T}_{\mathcal{S}}} \tau$ and σ is a data type, then τ is a data type as well.*

The following property is a clear consequence of the definition, Lemma 3.7 and arrow preservation:

LEMMA 3.19. *Assume $\sigma =_{\mathcal{T}_{\mathcal{S}}} \tau$. Then $[\![\sigma]\!] = [\![\tau]\!]$.*

In the sequel, we assume that functional types are in canonical form and that $n > 0$ in $\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$.

We first give the properties of the interpretations. Properties (i) to (v) are standard. Recall that a term is neutral if it is not an abstraction. Property (vi) is adapted from the simple type discipline, in which case the property is true of all basic types. Property (vii) is void when the algebraic signature is empty. It appeared first in [Jouannaud and Okada 1991].

PROPERTY 3.20 COMPUTABILITY PROPERTIES.
(i) Every computable term is strongly normalizable;
(ii) Assuming that s is computable and $s \succeq_{horpo} t$, then t is computable;
(iii) A neutral term s is computable iff t is computable for every t such that $s \succ_{horpo} t$;
(iv) If $\bar{t}$ be a vector of computable terms such that $@(\bar{t})$ is a candidate term, then $@(\bar{t})$ is computable;
(v) $\lambda x : \sigma.u$ is computable iff $u\{x \mapsto w\}$ is computable for every computable term $w :_C \sigma$;
(vi) Let $s :_C \sigma \in \mathcal{T}_{\mathcal{S}}^{min}$. Then s is computable iff it is strongly normalizable.
(vii) Let $f : \overline{\sigma} \rightarrow \tau \in \mathcal{F}$ and $s = f(\overline{s}) :_C \sigma$. Then s is computable if $\overline{s} :_C \overline{\sigma}$ is computable.

Our definition does not explicitly state that variables are computable. Variables of a data type are of course computable by definition (item (i)) since they have no reduct. But variables of a functional type will be computable by computability property (iii). This will actually forbid us to prove the computability properties (i), (ii) and (iii) separately. A possible alternative would be to modify the definition of the computability predicates by adding the property that variables of a functional type are computable. This would make the properties (i), (ii) and (iii) independent to the price of other complications.

PROOF.

—Property (iv).
Straightforward induction on the length of $\bar{t}$ and use of Case (ii) of the definition.

—Properties (i), (ii), (iii).
Note first that the only if part of property (iii) is property (ii). We are left with (i), (ii) and the if part of (iii) which we now spell out as follows:
Given a ground type σ and a candidate term s such that $s :_C \sigma \in [\![\sigma]\!]$, we prove by induction on the definition of $[\![\sigma]\!]$ that
(i) s is strongly normalizable;
(ii) $\forall t :_C \tau$ such that $s \succ_{horpo} t$, t is computable;
(iii) $\forall u :_C \sigma$ neutral, u is computable if $w :_C \theta \in [\![\theta]\!]$ for every w such that $u \succ_{horpo} w$.
Since $s \succ_{horpo} t$ and $u \succ_{horpo} w$, it follows from the definition of the ordering that $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ and $\sigma \geq_{\mathcal{T}_\mathcal{S}} \theta$, implying that τ and θ are ground by Lemma 3.14. We prove each property in turn, distinguishing in each case whether σ is a data type or functional.

(i) (a) Assume first that σ is a data type. All reducts of s are computable by definition of the interpretations, hence strongly normalizable by induction hypothesis, implying that s is strongly normalizable.

(b) Assume now that $\sigma = \theta \to \tau$, and let $s_0 = s :_C \sigma = \theta_0 \succ_{horpo} s_1 :_C \theta_1 \ldots \succ_{horpo} s_n :_C \theta_n \succ_{horpo} \ldots$ be a derivation issuing from s. Note that θ_i must be ground by Lemma 3.14. Hence $s_n \in [\![\theta_n]\!]$ by assumption for $n = 0$ and repeated applications of induction property (ii) for $n > 0$. Such derivations are of the following two kinds:

i. $\sigma >_{\mathcal{T}_\mathcal{S}} \theta_i$ for some i, in which case s_i is strongly normalizable by induction hypothesis, hence the derivation issuing from s is finite;

ii. $\theta_n =_{\mathcal{T}_\mathcal{S}} \sigma$ for all n, in which case $\{@(s_n, y :_C \sigma_1) :_C \sigma_2 \to \ldots \sigma_n \to \tau\}_n$ is a sequence of candidate terms of ground types which is strictly decreasing with respect to $\succ_{horpo}$ by monotonicity. Since $\sigma >_{\mathcal{T}_\mathcal{S}}^{\rightarrow} \sigma_1$, $y :_C \sigma_1$ is computable by induction hypothesis (iii), hence, by definition, $@(s_n, y)$ is computable. By induction hypothesis (i), the above sequence is finite, implying that the starting sequence itself is finite.

(ii) (a) Assume that σ is a data type. The result holds by definition of the candidate interpretations.

(b) Let $\sigma = \theta \to \rho$, hence θ and ρ are ground. By arrow preservation and decreasingness properties, there are two cases:

i. $\rho \geq_{\mathcal{T}_\mathcal{S}} \tau$. Since s is computable, $@(s, u)$ is computable for every $u \in [\![\theta]\!]$. Let $y :_C \theta$. By induction hypothesis (iii), $y \in [\![\theta]\!]$, hence $@(s, y)$ is computable. Since $@(s, y) :_C \rho \succ_{horpo} t :_C \tau$ by case 5 of the definition, t is computable by induction hypothesis (ii).

ii. $\tau = \theta' \to \rho'$, hence θ' and ρ' are ground, with $\theta =_{\mathcal{T}_\mathcal{S}} \theta'$ and $\rho \geq_{\mathcal{T}_\mathcal{S}} \rho'$. Since s is computable, given $u \in [\![\theta]\!]$, then $@(s, u) \in [\![\rho]\!]$, hence, by induction hypothesis (ii) $@(t, u) \in [\![\rho']\!]$. Since $[\![\theta]\!] = [\![\theta']\!]$ by Lemma 3.19, $t \in [\![\tau]\!]$ by definition of the interpretations.

(iii) (a) Assume that σ is a data type. The result holds by definition of the candidate interpretations.

(b) Assume now that $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \tau$, where $n > 0$ and τ is a data type. Then, $\sigma_1, \ldots, \sigma_n, \tau$ are ground types. By property (iv) u is computable if $@(u, u_1, \ldots, u_n)$ is computable for arbitrary terms $u_1 \in [\![\sigma_1]\!], \ldots, u_n \in [\![\sigma_n]\!]$ which are strongly normalizable by induction property (i). By definition, as τ is a data type, $@(u, u_1, \ldots, u_n)$ is computable iff so are all its reducts.

We prove by induction on the multiset of computable terms $\{u_1, \ldots, u_n\}$ ordered by $(\succeq_{horpo})_{mul}$ the property (H) stating that all terms w strictly smaller than $@(u, u_1, \ldots, u_i)$ in $\succ_{horpo}$ are computable. Remark that w has a ground type by definition of the ordering and Lemma 3.14. Taking $i = n$ yields the desired property, implying that u is computable.

If $i = 0$, terms strictly smaller than u are computable by assumption. For the general case, let $i = (j+1) \leq n$. We need to consider all terms w strictly smaller than $@(@(u, u_1, \ldots, u_j), u_{j+1})$. Since u is neutral, hence is not an abstraction, there are two possible cases:

i. $@(@(u, u_1, \ldots, u_j), u_{j+1}) \succ_{horpo} w$ by Case 5. There are again two possibilities:

—$@(u, u_1, \ldots, u_j) \succeq_{horpo} w$, and therefore $@(u, u_1, \ldots, u_j) \succ_{horpo} w$ for type reason since w is also a reduct of $@(u, u_1, \ldots, u_{j+1})$. We then conclude by induction hypothesis (H).

—$u_{j+1} \succeq_{horpo} w$. We conclude by assumption and induction property (ii).

ii. $@(@(u, u_1, \ldots, u_j), u_{j+1}) \succ_{horpo} w$ by Case 9, hence $w = @(\overline{w})$. By definition of the multiset extension and for type reasons, there are the following two possibilities:

—for all $v \in \overline{w}$, either $@(u, u_1, \ldots, u_j) \succ_{horpo} v$, and v is computable by induction hypothesis (H), or $u_{j+1} \succeq_{horpo} v$, in which case v is computable by assumption and induction property (ii). It follows that w is computable by Property 3.20 (iv).

—$w_1 = @(u, u_1, \ldots, u_j)$ and $u_{j+1} \succ_{horpo} w_2$, implying that w_2 is computable by assumption and induction property (ii). By induction property (H), all reducts of w are computable. Since w is an application hence is neutral, it is computable by induction property (iii).

As a consequence, all reducts of $@(u, u_1, \ldots, u_n)$ are computable and we are done.

—Property (v), assuming that the type of $\lambda x : \sigma.u$ is ground. In particular, σ is a ground type, as well as the type of u.

The only if part is property (ii) together with the definition of computability. For the if part, we prove that $@(\lambda x.u, w)$ is computable for an arbitrary computable w of type σ such that $u\{x \mapsto w\}$ is computable, implying that $\lambda x.u$ is computable by Definition 3.16 (ii).

Since variables are computable by property (iii), $u = u\{x \mapsto x\}$ is computable by assumption. By property (i), u and w are strongly normalizable, hence they are strongly normalizable by property (i). We can now prove that $@(\lambda x.u, w)$ is computable by induction on the pair $\{u, w\}$ ordered by $(\succ_{horpo})_{lex}$. By property (iii), the neutral term $@(\lambda x.u, w)$ is computable iff v is computable for all v such that $@(\lambda x.u, w) \succ_{horpo} v$. Once more, the definition and Lemma 3.14 imply that the type of v is ground. There are several cases to be considered.

(1) If the comparison is by case 5, there are two cases:
- if $w \succeq_{horpo} v$, we conclude by property (ii).
- if $\lambda x.u \succ_{horpo} v$, there are two cases. If $u \succeq_{horpo} v$ by case 6, we conclude by property (ii) again. Otherwise, $v = \lambda x.u'$ and $u \succ_{horpo} u'$, implying that u' has a ground type, and $u\{x \mapsto w\} \succ_{horpo} u'\{x \mapsto w\}$ by Lemma 3.9. By assumption and property (ii), $u'\{x \mapsto w\}$ is therefore computable. Hence, $@(v, w)$ which has the same ground type as u' is computable by induction hypothesis applied to the pair $(v = \lambda x.u', w)$. We then conclude by definition of the interpretations that v is computable.

(2) If the comparison is by case 9, then $v = @(\overline{v})$ and all terms in $\{\overline{v}\}$ are smaller than w or $\lambda x.u$, hence have a ground type. There are two cases:
- $v_1 = \lambda x.u$ and $w \succ_{horpo} v_i$ for $i > 1$. Then v_i is computable by property (ii) and, since $u\{x \mapsto v_2\}$ is computable by the main assumption, $@(v_1, v_2)$ is computable by induction hypothesis. If $v = @(v_1, v_2)$, we are done, and we conclude by property (iv) otherwise.
- For all other cases, terms in $\overline{v}$ are reducts of $\lambda x.u$ and w. Note that reducts of w and reducts of $\lambda x.u$ which are themselves reducts of u are computable by property (ii). Therefore, if all terms in $\overline{v}$ are such reducts, v is computable by property (iv).
Otherwise, for typing reason, v_1 is a reduct of $\lambda x.u$ of the form $\lambda x.u'$ with $u \succ_{horpo} u'$, and all other terms in $\overline{v}$ are reducts of the previous kind. By the main assumption, $u\{x \mapsto v''\}$ is computable for an arbitrary computable v''. Besides, $u\{x \mapsto v''\} \succ_{horpo} u'\{x \mapsto v''\}$ by Lemma 3.9. Therefore $u'\{x \mapsto v''\}$ is computable for an arbitrary computable v'' by Property (ii). By induction hypothesis, $@(v_1, v_2)$ is again computable. If $v = @(v_1, v_2)$, we are done, otherwise v is computable by property (iv).

(3) Otherwise, $@(\lambda x.u, w) \succ_{horpo} v$ by case 11, then $u\{x \mapsto w\} \succeq_{horpo} v$. By assumption, $u\{x \mapsto w\}$ is computable, and hence v is computable by property (ii).

—Property (vi).
The only if direction is property (i). For the if direction, let s be a strongly normalizable term of ground type $\sigma \in \mathcal{T}_{\mathcal{S}}^{min}$. We prove that s is computable by induction on the definition of $\succ_{horpo}$. Since σ is a data type, s must be neutral. Let now $s \succ_{horpo} t :_{C} \tau$, hence $\sigma \geq_{\mathcal{T}_{\mathcal{S}}} \tau$ which is therefore ground. By definition of $\mathcal{T}_{\mathcal{S}}^{min}$, $\tau =_{\mathcal{T}_{\mathcal{S}}} \sigma$, hence, by Lemma 3.18, τ is a data type, and since σ is minimal, so is τ, hence $\tau \in \mathcal{T}_{\mathcal{S}}^{min}$. By assumption on s, t must be strongly normalizable, and by induction hypothesis, it is therefore computable. Since this is true of all reducts of s, by definition s is computable.

—Property (vii).
Assuming that the type σ of $f(\overline{s})$ is ground does not imply, unfortunately, that terms in $\overline{s}$ have a ground type. On the other hand, all other properties have been

already proved, hence hold for arbitrary types by the lifting argument. Because we do not need to refer to the definition of the candidate interpretations in the coming proof, but will use instead the proved computability properties, we will be able to carry out the proof without any groundness assumption. In particular, since terms in $\overline{s}$ are computable by assumption, they are strongly normalizable by property (i). We use this remark to build our induction argument: we prove that $f(\overline{s})$ is computable by induction on the pair $(f, \overline{s})$ ordered lexicographically by $(>_{\mathcal{F}}, (\succ_{horpo})_{stat_f})_{lex}$.
Since $f(\overline{s})$ is neutral, by property (iii), it is computable iff every t such that $f(\overline{s}) \succ_{horpo} t$ is computable, which we prove by an inner induction on the size of t. We discuss according to the possible cases of the definition of $\succ_{horpo}$.

(1) Let $f(\overline{s}) \succ_{horpo} t$ by case 1, hence $s_i \succeq_{horpo} t$ for some $s_i \in \overline{s}$. Since s_i is computable, t is computable by property (ii).

(2) Let $s = f(\overline{s}) \succ_{horpo} t$ by case 2. Then $t = g(\overline{t})$, $f >_{\mathcal{F}} g$ and for every $v \in \overline{t}$ either $s \succ_{horpo} v$, in which case v is computable by the inner induction hypothesis, or $u \succeq_{horpo} v$ for some $u \in \overline{s}$ and v is computable by property (ii). Therefore, $\overline{t}$ is computable, and since $f >_{\mathcal{F}} g$, t is computable by the outer induction hypothesis.

(3) If $f(\overline{s}) \succ_{horpo} t$ by case 3, then $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, and $\overline{s}(\succ_{horpo})_{mul}\overline{t}$. By definition of the multiset comparison, for every $t_i \in \overline{t}$ there is some $s_j \in \overline{s}$, s.t. $s_j \succeq_{horpo} t_i$, hence, by property (ii), t_i is computable. This allows us to conclude by the outer induction hypothesis that t is computable.

(4) If $f(\overline{s}) \succ_{horpo} t$ by case 4, then $t = g(\overline{t})$, $f =_{\mathcal{F}} g$, $\overline{s}(\succ_{horpo})_{lex}\overline{t}$ and for every $v \in \overline{t}$ either $f(\overline{s}) \succ_{horpo} v$ or $u \succeq_{horpo} v$ for some $u \in \overline{s}$. As in the precedence case, this implies that $\overline{t}$ is computable. Then, since $\overline{s}(\succ_{horpo})_{lex}\overline{t}$, t is computable by the outer induction hypothesis.

(5) If $f(\overline{s}) \succ_{horpo} t$ by case 7, let $@(t_1, \ldots, t_n)$ be the partial left-flattening of t used in that proof. By the same token as in case 2, every term in $\overline{t}$ is computable, hence t is computable by property (iv).

(6) If $f(\overline{s}) \succ_{horpo} t$ by case 8, then $t = \lambda x.u$ with $x \notin \mathcal{V}ar(u)$, and $f(\overline{s}) \succ_{horpo} u$. By the inner induction hypothesis, u is computable. Hence, $u\{x \mapsto w\} = u$ is computable for any computable w, and therefore, $t = \lambda x.u$ is computable by property (v). □

3.5.3 *Strong normalization proof.* We are now ready for the strong normalization proof.

LEMMA 3.21. *Let γ be a type-preserving computable substitution and t be an algebraic λ-term. Then $t\gamma$ is computable.*

PROOF. The proof proceeds by induction on the size of t.

(1) t is a variable x. Then $x\gamma$ is computable by assumption.

(2) t is an abstraction $\lambda x.u$. By Property 3.20 (v), $t\gamma$ is computable if $u\gamma\{x \mapsto w\}$ is computable for every well-typed computable candidate term w. Taking $\delta = \gamma \cup \{x \mapsto w\}$, we have $u\gamma\{x \mapsto w\} = u(\gamma \cup \{x \mapsto w\})$ since x may not occur in γ. Since δ is computable and $|t| > |u|$, by induction hypothesis, $u\delta$ is computable.

(3) $t = @(t_1, t_2)$. Then $t_1\gamma$ and $t_2\gamma$ are computable by induction hypothesis, hence t is computable by Property 3.20 (iv).

(4) $t = f(t_1, \ldots, t_n)$. Then $t_i\gamma$ is computable by induction hypothesis, hence $t\gamma$ is computable by Property 3.20 (vii). □

We can now easily conclude the proof of well-foundedness needed for our main result, Theorem 3.5, by showing that every term is strongly normalizable with respect to $\succ_{horpo}$.

Proof. Given an arbitrary term t, let γ be the identity substitution. Since γ is type-preserving and computable, $t = t\gamma$ is computable by Lemma 3.21, and strongly normalizable by Property 3.20 (i). □

The restriction of our proof to the first-order sublanguage reduces essentially to Property 3.20 (vii), in which computability is simply identified with strong normalizability. This simple proof of well-foundedness of Dershowitz's recursive path ordering is spelled out in full detail in [Jouannaud 2005a]. The same proof technique had previously been used in a first order context first by Buchholz [1995], and later in [Fernández and Jouannaud 1995] and [Goubault-Larrecq 2001]. This simple well-foundedness proof of RPO proof does not use Kruskal's tree theorem, and of course, does not show that the recursive path ordering is a well-order. It appears therefore that proving the property of well-foundedness of the recursive path ordering is quite easy while proving the slightly stronger (and useless) property of well-orderedness becomes quite difficult.

3.6 Examples

We now illustrate both the expressive power of HORPO and its weaknesses with more examples. We start with a polymorphic example that shows the power of Case 7 of HORPO.

Example 3.22. (adapted from [Loría-Sáenz and Steinbach 1992])
Let $\mathcal{S} = \{\}, \mathcal{S}^\forall = \{\alpha, \beta\}$ and

$$\mathcal{F} = \left\{ \begin{array}{l} nil : List(\alpha), cons : \alpha \times List(\alpha) \Rightarrow List(\alpha), \\ dapply : \beta \times (\beta \to \beta) \times (\beta \to \beta) \Rightarrow \beta, \\ lapply : \beta \times List(\beta \to \beta) \Rightarrow \beta \end{array} \right\}$$

$$\begin{array}{ccrl} \{F, G : \beta \to \beta,\ x : \beta\} & \vdash & dapply(x, F, G) & \to F(G(x)) \\ \{x : \alpha\} & \vdash & lapply(x, nil) & \to x \\ \{F : \beta \to \beta,\ x : \beta,\ l : List(\beta \to \beta)\} & \vdash & lapply(x, cons(F, L)) & \to F(lapply(x, L)) \end{array}$$

The first rule follows by applying Case 7 twice. The second one holds by Case 1. For the third, we need $List(\alpha) >_{\mathcal{T}_\mathcal{S}} \alpha$ for every type α. Using Case 7, we show that $lapply(x, cons(F, L)) \succ_{horpo} F$ which holds by applying Case 1 twice (note that types decrease) and $lapply(x, cons(F, L)) \succ_{horpo} lapply(x, L)$ which holds by applying Case 3 (taking $lapply \in Mul$) and Case 1 for $cons(F, L) \succ_{horpo} L$.

The following classical example defines insertion for a polymorphic list. Polymorphism is then exploited by applying the algorithm to particular ascending and descending sorting algorithms for lists of natural numbers.

Example 3.23. Insertion Sort. Let $\mathcal{S}^{\forall} = \{\alpha\}$; $\mathcal{S} = \{\mathbb{N} : *, List : * \Rightarrow *\}$

$$\mathcal{F} = \left\{ \begin{array}{l} nil : List(\alpha); cons : \alpha \times List(\alpha) \Rightarrow List(\alpha); \\ insert : \alpha \times List(\alpha) \times (\alpha \to \alpha \to \alpha) \times (\alpha \to \alpha \to \alpha) \Rightarrow List(\alpha); \\ sort : List(\alpha) \times (\alpha \to \alpha \to \alpha) \times (\alpha \to \alpha \to \alpha) \Rightarrow List(\alpha); \\ ascending\_sort, descending\_sort : List(\mathbb{N}) \Rightarrow List(\mathbb{N}); \\ max, min : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N} \end{array} \right\}$$

Let $\Gamma_1 = \{X, Y : \alpha \to \alpha \to \alpha, n : \alpha\}$ and $\Gamma_2 = \{X, Y : \alpha \to \alpha \to \alpha, n, m : \alpha, l : List(\alpha)\}$.

$$\begin{array}{lcrcl} \Gamma_1 & \vdash & insert(n, nil, X, Y) & \to & cons(n, nil) \\ \Gamma_2 & \vdash & insert(n, cons(m, l), X, Y) & \to & cons(X(n, m), insert(Y(n, m), l, X, Y)) \end{array}$$

The first *insert* rule is easily taken care of by applying Case 2 with the precedence *insert* $>_{\mathcal{F}}$ *cons*, and then Case 1. For the second rule, we use Case 2 and recursively need to show two subgoals:

$insert(n, cons(m, l), X, Y) \succ_{horpo} @(X, n, m)$,

which follows by Case 7, and then Case 1 recursively (using $List(\alpha) >_{\mathcal{T}_S} \alpha$);

$insert(n, cons(m, l), X, Y) \succ_{horpo} insert(Y(n, m), l, X, Y)$,

which follows by Case 4 with a right-to-left lexicographic status for *insert*, calling recursively for the subgoal $insert(n, cons(m, l), X, Y) \succ_{horpo} @(Y, n, m)$ which is solved by Case 7 and Case 1. We now give the rules for sorting.

Let $\Gamma_3 = \{X, Y : \alpha \to \alpha \to \alpha\}$ and $\Gamma_4 = \{X, Y : \alpha \to \alpha \to \alpha, n : \alpha, l : List(\alpha)\}$.

$$\begin{array}{lcrcl} \Gamma_3 & \vdash & sort(nil, X, Y) & \to & nil \\ \Gamma_4 & \vdash & sort(cons(n, l), X, Y) & \to & insert(n, sort(l, X, Y), X, Y) \end{array}$$

These rules are easily oriented by $\succ_{horpo}$, with the precedence *sort* $>_{\mathcal{F}}$ *insert*. We now introduce rules for computing *max* and *min* on natural numbers (with either $\{x : \mathbb{N}\}$ or $\{x, y : \mathbb{N}\}$ as environment):

$$\begin{array}{lclclclclcl} max(0, x) & \to & x & max(x, 0) & \to & x & max(s(x), s(y)) & \to & s(max(x, y)) \\ min(0, x) & \to & 0 & min(x, 0) & \to & 0 & min(s(x), s(y)) & \to & s(min(x, y)) \end{array}$$

We simply need the precedence $max, min >_{\mathcal{F}} s$ for these first-order rules. We come finally to the ascending and descending sort functions for lists of natural numbers, for which typing the righthand sides (with type $\mathbb{N}$) illustrates the use of type substitutions:

$$\begin{array}{lcrcl} \{l : List(\mathbb{N})\} & \vdash & ascending\_sort(l) & \to & sort(l, \lambda xy.min(x, y), \lambda xy.max(x, y)) \\ \{l : List(\mathbb{N})\} & \vdash & descending\_sort(l) & \to & sort(l, \lambda xy.max(x, y), \lambda xy.min(x, y)) \end{array}$$

Unfortunately, $\succ_{horpo}$ fails to orient these two seemingly easy rules. This is so, because the term $\lambda xy.min(x, y)$ occurring in the righthand side has type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, which is not comparable to any lefthand side type. We will come back to this example in Section 4.

We now come to a more tricky example, for which we actually need the transitive closure of the ordering to show termination of a rule, that is, we will need to invent a *middle term* s such that $l \succ_{horpo} s \succ_{horpo} r$ for some rule $l \to r$.

Example 3.24. (Surjective Disjoint Union, taken from [van de Pol and Schwichtenberg 1995]). Let

$\mathcal{S} = \{A, B, U\}$, $\alpha \in \{A, B, U\}$,
$\mathcal{F} = \{inl : A \Rightarrow U; inr : B \Rightarrow U; case_\alpha : U \times (A \rightarrow \alpha) \times (B \rightarrow \alpha) \Rightarrow \alpha\}$.
Let $\Gamma_1 = \{X : A, F : A \rightarrow \alpha, G : B \rightarrow \alpha\}$, $\Gamma_2 = \{Y : B, F : A \rightarrow \alpha, G : B \rightarrow \alpha\}$ and $\Gamma_3 = \{Z : U, F : U \rightarrow \alpha\}$.

$$\begin{array}{lrrcl} \Gamma_1 & \vdash & case_\alpha(inl(X), F, G) & \rightarrow & @(F, X) \\ \Gamma_2 & \vdash & case_\alpha(inr(Y), F, G) & \rightarrow & @(G, Y) \\ \Gamma_3 & \vdash & case_\alpha(Z, \lambda x.H(inl(x)), \lambda y.H(inr(y))) & \rightarrow & @(H, Z) \end{array}$$

The type U here is the disjoint union of types A and B, while the $case_\alpha$ function is the associated recursor. This construction exists in functional languages with sum types.

Note that we write $@(F, X)$ instead of $F(X)$ to make clear that @ is the top function symbol of the term $F(X)$. For the type ordering we need $A =_{\mathcal{T}_\mathcal{S}} B =_{\mathcal{T}_\mathcal{S}} U$. The first two rules are taken care of by Case 7. For the last one, we apply Case 7, and then prove $\lambda x.H(inl(x)) \succ_{horpo} H$ by showing

$$\lambda x.H(inl(x)) \underset{horpo}{\succ} \lambda x.H(x) \underset{horpo}{\succ} H$$

The last comparison holds by Case 12. The first one holds by Cases 10, then Case 9 and finally Case 1.

4. COMPUTABILITY CLOSURE

The ordering is quite sensitive to innocent variations of the rules to be checked, like adding (higher-order) dummy arguments to righthand sides or η-expanding higher-order variables in the righthand sides. We will solve these problems by improving our definition in the light of the strong normalization proof. In Case 1 of Definition 3.1, we assume that the righthand side term t is indeed smaller or equal to a subterm u of the lefthand side $f(\overline{s})$. Assuming that $\overline{s}$ is computable, we conclude by Property 3.20 (ii) that t is computable. This argument comes again and again in the proofs of the computability properties. Let us now change the definition of Case 1, and require that $u = @(v, w) \succ_{horpo} t$, for some subterms v and w of s. By assumption, v and w are computable, making $@(u, v)$ computable by Property 3.20 (iv), and we are back to the previous case We see here that u may not be a term in $\overline{s}$, but must be built from terms in $\overline{s}$ by computability preserving operations. For example, since a higher-order variable X is computable if and only if $\lambda x.X(x)$ is computable, we may have X as a subterm of the lefthand side of a rule, and $\lambda x.X(x)$ in the righthand side. This discussion is formalized in subsection 4.1 with the notion of a computability closure borrowed from [Blanqui et al. 1999], with a slightly enhanced formulation.

4.1 The Computability Closure

Given a set of computable terms, the computability closure builds a superset of computable terms by using computability preserving operations such as those listed in Property 3.20. First, we need a technical definition allowing us to control the type of a selected subterm:

Definition 4.1. Let $>_{\mathcal{T}_\mathcal{S}}$ be a type ordering. The (strict) *type-decreasing subterm relation*, denoted by $\rhd_{\geq_{\mathcal{T}_\mathcal{S}}}$, is defined as: $s : \sigma \rhd_{\geq_{\mathcal{T}_\mathcal{S}}} t : \tau$ iff $s \rhd t$, $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ and

$\mathcal{V}ar(s) \subseteq \mathcal{V}ar(t)$.

Definition 4.2. Given a term $t = f(\overline{t})$ with $f \in \mathcal{F}$, we define its *computability closure* $\mathcal{CC}(t)$ as $\mathcal{CC}(t, \emptyset)$, where $\mathcal{CC}(t, \mathcal{V})$, with $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$, is the smallest set of typable terms containing all variables in $\mathcal{V}$, all terms in $\bar{t}$, and closed under the following operations:

(1) subterm of minimal type: let $s \in \mathcal{CC}(t, \mathcal{V})$, and $u : \sigma$ be a subterm of s such that $\sigma \in \mathcal{T}_{\mathcal{S}}^{min}$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$; then $u \in \mathcal{CC}(t, \mathcal{V})$;

(2) precedence: let g such that $f >_{\mathcal{F}} g$, and $\overline{s} \in \mathcal{CC}(t, \mathcal{V})$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$;

(3) recursive call: let $\overline{s}$ be a sequence of terms in $\mathcal{CC}(t, \mathcal{V})$ such that the term $f(\overline{s})$ is well typed and $\overline{t}(\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_{\mathcal{S}}}})_{stat_f}\overline{s}$; then $g(\overline{s}) \in \mathcal{CC}(t, \mathcal{V})$ for every $g =_{\mathcal{F}} f$;

(4) application: let $s : \sigma_1 \to \ldots \to \sigma_n \to \sigma \in \mathcal{CC}(t, \mathcal{V})$ and $u_i : \sigma_i \in \mathcal{CC}(t, \mathcal{V})$ for every $i \in [1..n]$; then $@(s, u_1, \ldots, u_n) \in \mathcal{CC}(t, \mathcal{V})$;

(5) abstraction: let $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$; then $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$;

(6) reduction: let $u \in \mathcal{CC}(t, \mathcal{V})$, and $u \succeq_{horpo} v$; then $v \in \mathcal{CC}(t, \mathcal{V})$;

(7) weakening: let $x \notin \mathcal{V}ar(u, t) \cup \mathcal{V}$. Then, $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ iff $u \in \mathcal{CC}(t, \mathcal{V})$.

As an illustration of the definition, we show that abstraction as well as extensionality are equivalences:

Example 4.3. Assume that $\lambda x.s \in \mathcal{CC}(t, \mathcal{V})$ and $x \notin \mathcal{V}ar(t) \cup \mathcal{V}$. By weakening, $\lambda x.s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. Since $x \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ by base case of the definition, $@(\lambda x.s, x) \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ by application case. Therefore, $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$ by reduction case.

Example 4.4. Assuming that $x \notin \mathcal{V}ar(u) \cup \mathcal{V}$, we show that $\lambda x.@(u, x) \in \mathcal{CC}(t, \mathcal{V})$ iff $u \in \mathcal{CC}(t, \mathcal{V})$.

For the if direction, assuming without loss of generality that $x \notin \mathcal{V}ar(t)$, by weakening, $u \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. By basic case, $x \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$, hence, by application, $@(u, x) \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$, and by abstraction, we get $\lambda x.@(u, x) \in \mathcal{CC}(t, \mathcal{V})$.

Conversely, assuming that $\lambda x.@(u, x) \in \mathcal{CC}(t, \mathcal{V})$ with $x \notin \mathcal{V}ar(u)$, then $u \in \mathcal{CC}(t, \mathcal{V})$ by reduction, since $\lambda x.@(u, x) \succ_{horpo} u$ by Case 12 of Definition 3.1.

An important remark is that we use the previously defined ordering $\succeq_{horpo}$ in Case 6 and the relation $\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_{\mathcal{S}}}}$ in Case 3 of the closure definition instead of simply β-reductions and $\beta \cup \rhd$-reductions respectively as in [Jouannaud and Rubio 1999]. And indeed, we will consequently use $\succ_{horpo}$ and $\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_{\mathcal{S}}}}$ as induction arguments in our proofs. The derivation of the extensionality rule shows the usefulness of rule 6. A price has to be paid for the added expressive power: because $\succ_{horpo}^{+}$ is not order-isomorphic to the natural numbers, the computability closure may be infinite, and since $\succ_{horpo}^{+}$ is probably undecidable, so is the membership of a term to the computability closure. But this membership remains decidable if the number of times Rules 6 and 3 are used is bound. In practice, we can restrict their use by allowing a single step only, which is enough for all examples of the paper. More complex examples to come illustrate how checking membership of a term to a closure can be done by a goal-oriented use of the rules of Defining 4.2.

The following property of the computability closure is shown by induction on the definition:

LEMMA 4.5. *Assume that $u \in \mathcal{CC}(t)$. Then, $u\gamma \in \mathcal{CC}(t\gamma)$ for all type-preserving substitutions γ.*

PROOF. We prove that if $u \in \mathcal{CC}(t, \mathcal{V})$ with $\mathcal{V} \subseteq \mathcal{X} \setminus (\mathcal{V}ar(t) \cup \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma))$, then $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$. We proceed by induction on the definition of $\mathcal{CC}(t, \mathcal{V})$. Note that the property on $\mathcal{V}$ depends on t and γ, but not on u. It will therefore be trivially satisfied in all cases but abstraction and weakening. And indeed, these are the only cases in the proof which are not routine, hence we do them in detail as well as Case 1 to show its simplicity.

Case 1: let $s = f(\overline{s}) : \sigma \in \mathcal{CC}(t, \mathcal{V})$ with $f \in \mathcal{F}$ and $u : \tau$ be a subterm of s with $\sigma \in \mathcal{T}_{\mathcal{S}}^{min}$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. By induction hypothesis, $s\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$. By assumption on u, $u\gamma : \tau$ is a subterm of $s\gamma : \sigma$ and $\mathcal{V}ar(u\gamma) \subseteq \mathcal{V}ar(t\gamma)$. Therefore, $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$ by Case 1.

Case 5: let $u = \lambda x.s$ with $x \in \mathcal{X} \setminus (\mathcal{V} \cup \mathcal{V}ar(t))$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. To the price of renaming the variable x in s if necessary, we can assume in addition that $x \notin \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma)$, and therefore $\mathcal{V} \cup \{x\} \subseteq \mathcal{X} \setminus (\mathcal{V}ar(t) \cup \mathcal{V}ar(t\gamma) \cup \mathcal{D}om(\gamma))$. By induction hypothesis, $s\gamma \in \mathcal{CC}(t\gamma, \mathcal{V} \cup \{x\})$. Since $x \notin \mathcal{V}ar(t\gamma) \cup \mathcal{V}$, by Case 5 of the definition $\lambda x.s\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$ and, since $x \notin \mathcal{D}om(\gamma)$, we have $u\gamma = \lambda x.s\gamma$.

Case 7: let $\mathcal{V}' = \mathcal{V} \cup \{x\}$. By definition, $u \in \mathcal{CC}(t, \mathcal{V}')$, with $x \notin \mathcal{V}ar(u, t) \cup \mathcal{V}$. By induction hypothesis, $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V}')$. Since $x \notin \mathcal{V}ar(u\gamma, t\gamma) \cup \mathcal{V}$, $u\gamma \in \mathcal{CC}(t\gamma, \mathcal{V})$ by Case 7. □

LEMMA 4.6. *Assume that $u : \sigma \in \mathcal{CC}(t : \tau)$. Then, $u\delta : \sigma\xi_\delta \in \mathcal{CC}(t\delta : \tau\xi_\delta)$ for every specialization δ.*

PROOF. The proof is similar as previously, but uses the polymorphic property of $\succeq_{horpo}$ in Case 6. □

4.2 The Higher-Order Recursive Path Ordering with Closure

Both orderings $\succ_{horpo}$ and $\succ_{chorpo}$ will coexist from now on, until they get merged into a single one in Section 4.5. They differ only by the definition of the property A, and by the first, subterm case. As a way to stress their similarities, we do not reformulate the entire ordering. Environments and types are again omitted here.

$$A = \forall v \in \bar{t}\ s \underset{chorpo}{\succ} v \text{ or } u \underset{chorpo}{\succeq} v \text{ for some } u \in \mathcal{CC}(s)$$

Definition 4.7.

$$s : \sigma \underset{chorpo}{\succ} t : \tau \text{ iff } \sigma \geq_{\mathcal{T}_{\mathcal{S}}} \tau \text{ and}$$

(1) $s = f(\overline{s})$ with $f \in \mathcal{F}$, and (i) $u \underset{chorpo}{\succeq} t$ for some $u \in \overline{s}$ or (ii) $t \in \mathcal{CC}(s)$

Cases 2 to 12 are kept unchanged, $\succ_{horpo}$ (resp. $\succeq_{horpo}$) being replaced by $\succ_{chorpo}$ (resp. $\succeq_{chorpo}$).

The definition is recursive, since recursive calls operate on pairs of terms which decrease in the well-founded ordering $(\succ_{horpo}, \rhd)_{right-to-left-lex}$.

As an easy consequence of Case 1, we obtain that $s = f(\overline{s}) : \sigma \succ_{chorpo} t : \tau$ with $f \in \mathcal{F}$ if $\sigma = \tau$ and $t \in \mathcal{CC}(s)$. This shows that the technique based on the general schema, essentially based on the closure mechanism introduced by Blanqui, Jouannaud and Okada [1999], is a very particular case of the present method. Besides, this new method inherits all the advantages of the recursive path ordering, in particular it is possible to combine it with interpretation based techniques [Borralleras and Rubio 2001].

The proofs of Lemmas 3.11, 3.9 and 3.10 are easy to adapt (using now Lemma 4.5 for the proof of the new version of Lemma 3.9 and Lemma 4.6 for Lemma 3.10) to the ordering with closure:

Lemma 4.8. *$\succ_{chorpo}$ is monotonic, stable, and polymorphic.*

4.3 Strong Normalization

We first show that terms in the computability closure of a term are computable.

First, note that the computability predicate is of course now defined with respect to $\succ_{chorpo}$. To show that the computability properties remain valid, we simply observe that there is no change whatsoever in the proofs, since the rules applying to application-headed terms did not change. For this, it is crucial to respect the given formulation of Case 5, avoiding the use of the closure as in Case 1. Actually, we could have defined a specific, weaker closure for terms headed by an application, to the price of doing again the two most complex proofs of the computability properties. We did not think it was worth the trouble.

Second, we are still using the previous ordering $\succ_{horpo}$ in the definition of the new ordering $\succ_{chorpo}$, via the closure definition in particular. We therefore need to prove that $\succ_{horpo} \subseteq \succ_{chorpo}$ in order to apply the induction arguments based on the well-foundedness of $\succ_{chorpo}$ on some appropriate set of terms:

Lemma 4.9. $\succ_{horpo} \ \subseteq \ \succ_{chorpo}$.

The proof is easily done by induction since all cases in the definition of $\succ_{horpo}$ appear in the definition of $\succ_{chorpo}$. The importance of this lemma comes from the computability properties. By the above inclusion, any term smaller than a computable term in the ordering $\succ_{horpo}$ will therefore be computable by Property 3.20 (ii).

We may wonder whether $\succ^{+}_{horpo} \ \subseteq \succ_{chorpo}$. Let $s_1 \succ_{horpo} s_2 \succ_{horpo} \ldots \succ_{horpo} s_n$. Then $s_1 \succ_{chorpo} s_n$ in case $s_n \in \mathcal{CC}(s_1)$. This is in particular true when $s_2 \in \mathcal{CC}(s_1)$, since Case 6 of the closure definition then implies that $s_n \in \mathcal{CC}(s_1)$. This is not true in general since $s_1 \notin \mathcal{CC}(s_1)$.

We now show that $\succ_{horpo}$, which is monotonic for terms of equivalent types and well-founded, remains well-founded when combined with $\rhd_{\geq_{\mathcal{T}_S}}$:

Lemma 4.10. *Let $>$ be a well-founded relation on terms which is monotonic for terms of equivalent types (in $=_{\mathcal{T}_S}$). Then, $> \cup \rhd_{\geq_{\mathcal{T}_S}}$ is well-founded.*

Proof. Since $>$ is well-founded, it is enough to show the absence of infinite decreasing sequences for terms of equivalent types, which follows from the fact that $\rhd_{\geq_{\mathcal{T}_S}}$ and $>$ commute for such terms by monotonicity of $>$. □

We now come to the main property of terms in the computability closure, which justifies its name:

Property 4.11. *Assume $\overline{t} : \overline{\tau}$ is computable, as well as every term $g(\overline{s})$ with $\overline{s}$ computable and smaller than $t = f(\overline{t})$ in the ordering $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_{\mathcal{S}}}})_{stat_f})_{lex}$ operating on pairs $\langle f, \overline{t}\rangle$. Then every term in $\mathcal{CC}(t)$ is computable.*

The precise formulation of this statement arises from its forthcoming use in the proof of Lemma 4.12.

Proof. We prove that $u\gamma : \sigma$ is computable for every computable substitution γ of domain $\mathcal{V}$ and every $u \in \mathcal{CC}(t, \mathcal{V})$ such that $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$. We obtain the result by taking $\mathcal{V} = \emptyset$. We proceed by induction on the definition of $\mathcal{CC}(t, \mathcal{V})$.

For the basic case: if $u \in \mathcal{V}$, then $u\gamma$ is computable by the assumption on γ; if $u \in \overline{t}$, we conclude by the assumption that $\overline{t}$ is computable, since $u\gamma = u$ by the assumption that $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$; and if u is a subterm of $f(\overline{t})$, we again remark that γ acts as the identity on such terms, and therefore, they are computable by assumption.

For the induction step, we discuss the successive operations to form the closure:

case 1: u is a subterm of type in $\mathcal{T}_{\mathcal{S}}^{min}$ of some $v \in \mathcal{CC}(t, \mathcal{V})$. By induction hypothesis, $v\gamma$ is computable, hence strongly normalizable by Property 3.20 (i). By monotonicity of $\succ_{chorpo}$, its subterm $u\gamma$ is also strongly normalizable, and hence computable by Property 3.20 (vi).

case 2: $u = g(\overline{u})$ where $\overline{u} \in \mathcal{CC}(t, \mathcal{V})$. By induction hypothesis, $\overline{u}\gamma$ is computable. Since $f >_{\mathcal{F}} g$, $u\gamma$ is computable by our assumption that terms smaller than $f(\overline{t})$ are computable.

case 3: $u = g(\overline{u})$ where $f =_{\mathcal{F}} g$, $\overline{u} \in \mathcal{CC}(t, \mathcal{V})$ and $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(t)$. By induction hypothesis, $\overline{u}\gamma$ is computable. By assumption, and stability of the ordering under substitutions, $\overline{t}\gamma(\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_{\mathcal{S}}}})_{stat_f}\overline{u}\gamma$. Note that $\overline{t}\gamma = \overline{t}$ by our assumption that $\mathcal{V} \cap \mathcal{V}ar(t) = \emptyset$. Therefore $u\gamma = g(\overline{u}\gamma)$ is computable by our assumption that terms smaller than $f(\overline{t})$ are computable.

case 4: by induction and Property 3.20 (iv).

case 5: let $u = \lambda x.s$ with $x \notin \mathcal{V}$ and $s \in \mathcal{CC}(t, \mathcal{V} \cup \{x\})$. To the price of possibly renaming x, we can assume without loss of generality that $x \notin \mathcal{D}om(\gamma) \cup \mathcal{V}ar(t)$. As a first consequence, $\mathcal{V} \cup \{x\} \cap \mathcal{V}ar(t) = \emptyset$; as a second, given an arbitrary computable term w, $\gamma' = \gamma \cup \{x \mapsto w\}$ is a computable substitution of domain $\mathcal{V} \cup \{x\}$. By induction hypothesis, $s\gamma'$ is therefore computable, and by Property 3.20 (v), $(\lambda x.s)\gamma$ is computable.

case 6: by induction, stability and Property 3.20 (ii).

case 7: by induction hypothesis. □

We now restate Property 3.20 (vii) and show in one case how the proof makes use of Property 4.11.

Lemma 4.12. *Let $f \in \mathcal{F}$ and let $\overline{t}$ be a set of terms. If $\overline{t}$ is computable, then $f(\overline{t})$ is computable.*

Proof. We prove that $f(\overline{t})$ is computable if terms in $\overline{t}$ are computable by an outer induction on the pair $\langle f, \overline{t}\rangle$ ordered lexicographically by the ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_{\mathcal{S}}}})_{stat_f})_{lex}$, and an inner induction on the size of the reducts of t. Since terms in $\overline{t}$ are computable by assumption, they are strongly normalizable by

Property 3.20(ii), hence, by Lemma 4.10 the ordering $(>_{\mathcal{F}}, ((\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_S}})_{stat})_{lex}$ is well founded on the pairs satisfying the assumptions.

By Lemma 4.9, the set of pairs smaller than the pair $\langle f, \overline{t} \rangle$ for the ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_S}})_{stat})_{lex}$ contains the set of pairs that are smaller than that pair for the ordering $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_S}})_{stat})_{lex}$. This key remark allows us to use Property 4.11.

The proof is actually similar to the proof of Property 3.20 (vii), except for Case 1 and for the cases using property A. We therefore do Cases 1 and 2, the latter using property A.

case 1: let $f(\overline{t}) \succ_{chorpo} s$ by case 1, hence $t_i \succeq_{horpo} s$ for some $t_i \in \overline{t}$ or $s \in \mathcal{CC}(t)$. In the former case, since t_i is computable, s is computable by Property 3.20 (ii); in the latter case, all terms smaller than $f(\overline{t})$ with respect to $(>_{\mathcal{F}}, (\succeq_{horpo})_{stat})_{lex}$ are computable by induction hypothesis and the above key remark, hence s is computable by Property 4.11.

case 2: let $t = f(\overline{t}) \succ_{chorpo} s$ by case 2. Then $s = g(\overline{s})$, $f >_{\mathcal{F}} g$ and for every $s_i \in \overline{s}$ either $t \succ_{chorpo} s_i$, in which case s_i is computable by the inner induction hypothesis, or $v \succeq_{chorpo} s_i$ for some $v \in \mathcal{CC}(t)$, in which case v is computable by Property 4.11 and hence s_i is computable by Property 3.20 (ii). We conclude that s is computable by the outer induction hypothesis since $f >_{\mathcal{F}} g$. □

Theorem 4.13. *$\succ^{+}_{chorpo}$ is a polymorphic higher-order reduction ordering.*

Proof. Thanks to Property 4.11, the strong normalization proof of this improved ordering is exactly the same as the one for HORPO, using now Lemma 4.12 instead of Property 3.20 (vii). Using now Lemma 4.8, we therefore conclude that the transitive closure of $\succ_{chorpo}$ is a higher-order polymorphic reduction ordering. □

4.4 Examples

This new definition of the ordering is much stronger than the previous one. In addition to allowing us proving the strong normalization property of the remaining rules of the sorting example, it also allows proving termination of the following rule, which is added to the rules of Example 2.27:

Example 4.14.

$$x * y \ \rightarrow \ rec(y, 0, \lambda z_1 z_2 . x + z_2)$$

This rule can be proved terminating with the precedence $* >_{\mathcal{F}} \{rec, +, 0\}$. Indeed, $x * y \succ_{chorpo} rec(y, 0, \lambda z_1 z_2 . x + z_2)$ by Case 2 of $\succ_{chorpo}$, since $x * y \succ_{chorpo} y$ by case 1, $x * y \succ_{chorpo} 0$ by case 2 again, and $\lambda z_1 z_2 . x + z_2 \in \mathcal{CC}(x * y)$:
applying Case 5 of the definition of the computability closure twice, we need to show $x + z_2 \in \mathcal{CC}(x * y, \{z_1, z_2\})$. By Case 2 we are left with $x \in \mathcal{CC}(x * y, \{z_1, z_2\})$ and $z_2 \in \mathcal{CC}(x * y, \{z_1, z_2\})$, which both hold by base Case of the definition of the computability closure.

The coming example is quite classical too.

Example 4.15. Let

$$\mathcal{S} = \{List : * \Rightarrow *\};\ \mathcal{S}^{\forall} = \{\alpha\}$$

$$\mathcal{F} = \left\{ \begin{array}{l} 0, 1 : \alpha; + : \alpha \times \alpha \Rightarrow \alpha; nil : List(\alpha); cons : \alpha \times List(\alpha) \Rightarrow List(\alpha); \\ foldl : (\alpha \rightarrow \alpha \rightarrow \alpha) \times \alpha \times List(\alpha) \Rightarrow \alpha; sum : List(\alpha) \Rightarrow \alpha; \\ +^c : \alpha \rightarrow \alpha \Rightarrow \alpha \end{array} \right\}$$

Let $\Gamma = \{x, y : \alpha,\ F : \alpha \rightarrow \alpha \rightarrow \alpha,\ l : List(\alpha)\}$ in

$$\begin{array}{ccrcl} \{x : \alpha,\ F : \alpha \rightarrow \alpha \rightarrow \alpha\} & \vdash & foldl(F, x, nil) & \rightarrow & x \\ \Gamma & \vdash & foldl(F, x, cons(y, l)) & \rightarrow & foldl(F, (F\ x\ y), l) \\ \{\} & \vdash & +^c & \rightarrow & \lambda xy.x + y \\ \{l : List(\alpha)\} & \vdash & sum(l) & \rightarrow & foldl(+^c, 0, l) \end{array}$$

The first rule is by subterm case. For the second, we set a right-to-left lexicographic status for $foldl$, and, applying Case 4, we recursively have to show that (i) $F \succeq_{chorpo} F$; (ii) $foldl(F, x, cons(y, l)) \succ_{chorpo} @(F, x, y)$, which succeeds easily by rule 7; and (iii) $cons(y, l) \succ_{chorpo} l$, which succeeds by subterm case provided $List(\alpha) \geq_{\mathcal{T}_\mathcal{S}} \alpha$.

For the third, we show that $\lambda xy.x + y$ is in the closure of $+^c$, provided $+^c >_{\mathcal{F}} +$, by successively applying Case 5 twice, Case 2 and, finally, the base Case twice.

For the last, we add the precedence $sum >_{\mathcal{F}} \{foldl, +^c, 0\}$ in order to prove it by Case 2 of $\succ_{chorpo}$, followed by Case 2 for $sum(l) \succeq_{chorpo} 0$ and showing that $+^c \in \mathcal{CC}(sum(l))$, by Case 2.

The following example, a definition of formal derivation, illustrates best the power of the computability closure.

Example 4.16. Let $\mathcal{S} = \{\Re\}$ be the sort of real numbers.

$$\mathcal{F} = \left\{ \begin{array}{l} D : (\Re \rightarrow \Re) \Rightarrow (\Re \rightarrow \Re); \\ 0, 1 :\rightarrow \Re;\ -, sin, cos, ln : \Re \Rightarrow \Re; \\ +, \times, / : \Re \times \Re \Rightarrow \Re \end{array} \right\}$$

Let $\Gamma_1 = \{F, G : \Re \rightarrow \Re\}$ and $\Gamma_2 = \{F : \Re \rightarrow \Re\}$ in

$$\begin{array}{ccrcl} & & D(\lambda x.y) & \rightarrow & \lambda x.0 \\ & & (\lambda x.x) & \rightarrow & \lambda x.1 \\ & & D(\lambda x.sin(x)) & \rightarrow & \lambda x.cos(x) \\ & & D(\lambda x.cos(x)) & \rightarrow & \lambda x. - sin(x) \\ \Gamma_1 & \vdash & D(\lambda x.(F\ x) + (G\ x)) & \rightarrow & \lambda x.(D(F)\ x) + (D(G)\ x) \\ \Gamma_1 & \vdash & D(\lambda x.(F\ x) \times (G\ x)) & \rightarrow & \lambda x.(D(F)\ x) \times (G\ x) + (F\ x) \times (D(G)\ x) \\ \Gamma_2 & \vdash & D(\lambda x.ln(F\ x)) & \rightarrow & \lambda x.(D(F)\ x)/(F\ x) \end{array}$$

We take $D >_{\mathcal{F}} \{0, 1, \times, -, +, /\}$ for precedence and assume that the function symbols are in Mul. Apart from the first rule, this example makes a heavy use of the closure. The computations involved are quite complex, and can be followed by using our implementation. The but-last rule, in particular, is entirely processed by the closure mechanism (together with Case 1). This does not mean that it could be already solved with the same proof by the technique developed in [Blanqui et al.

1999], where the computability closure was first introduced without any other mechanism. The point is that the closure defined here is more powerful thanks to case 6 based on $\succ_{horpo}$ instead of simply using β-reductions as in [Blanqui et al. 1999].

We are not completely satisfied with this computation, though, because the structural definition of the ordering has been completely lost here. We would like to improve the ordering so as to do more computations with the ordering and delegate the hard parts only to the more complex closure mechanism. For the time being, however, the closure mechanism is the only one which applies to rules whose righthand side is an abstraction with the bound variable occurring in the body, assuming that the lefthand side of rule is not itself an abstraction. We discuss its implementation in Section 5.2.

The next example (currying/uncurrying) shows again the use of a middle term.

Example 4.17. First, to a signature $\mathcal{F}$, we associate the signature

$$\mathcal{F}^{curry} = \left\{ \begin{array}{l} f_i : \sigma_1 \times \ldots \times \sigma_i \Rightarrow \sigma_{i+1} \to \ldots \to \sigma_p \to \tau \text{ for every } i \in [0..p] \mid \\ f : \sigma_1 \times \ldots \times \sigma_n \Rightarrow \sigma_{n+1} \to \ldots \to \sigma_p \to \tau \in \mathcal{F} \text{ where } \tau \text{ is a data-type} \end{array} \right\}$$

and we introduce the following sets of rewrite rules for currying/uncurrying:

$$j \in J \subseteq [1..p] : \{t_1 : \sigma_1, \ldots, t_{j+1} : \sigma_{j+1}\} \vdash f_{j+1}(t_1, \ldots, t_{j+1}) \to @(f_j(t_1, \ldots, t_j), t_{j+1})$$

$$i \in I \subseteq [1..p] : \{t_1 : \sigma_1, \ldots, t_{i+1} : \sigma_{i+1}\} \vdash @(f_i(t_1, \ldots, t_i), t_{i+1}) \to f_{i+1}(t_1, \ldots, t_{i+1})$$

When $i = j$, the above rules are clearly non-terminating since they originate from the same equation oriented in both directions. Termination therefore requires that the two subsets I and J of $[0..p]$ are disjoint. We postpone their precise definition.

Starting with the first rule, we set $f_{j+1} >_{\mathcal{F}} f_j$. Applying case 7 we have to show that $f_j(t_1, \ldots, t_j) \in \mathcal{CC}(f_{j+1}(t_1, \ldots, t_{j+1}))$ which holds by the precedence case followed by the base case for all the arguments.

Moving to the second rule, we set $f_i >_{\mathcal{F}} f_{i+1}$. The only possible case is subterm for application, i.e. Case 5, but the subgoal $f_i(t_1, \ldots, t_i) \succ_{chorpo} f_{i+1}(t_1, \ldots, t_{i+1})$ cannot succeed since there is a new free variable (t_{i+1}) in the righthand side. The computability closure does not help either here, since it is defined for terms headed by an algebraic function symbol.

The trick is to invent a middle term, and show that the lefthand side is bigger than the righthand one in the *transitive closure* of the ordering. The convenient middle term here is $@(\lambda x. f_{i+1}(t_1, \ldots, t_i, x), t_{i+1})$, which reduces to the righthand side by the use of Case 11. We therefore simply need to show that the lefthand side is bigger than the middle term. Since both terms are headed by an abstraction, by Case 9 we have to prove that $f_i(t_1, \ldots, t_i) \succ_{chorpo} \lambda x. f_{i+1}(t_1, \ldots, t_i, x)$, which we prove now by showing that the righthand side term is in the closure of the lefthand side one: by abstraction case we need $f_{i+1}(t_1, \ldots, t_i, x) \in \mathcal{CC}(f_i(t_1, \ldots, t_i), \{x\})$, and by precedence case, we are left with $t_1, \ldots t_i, x \in \mathcal{CC}(f_i(t_1, \ldots, t_i), \{x\})$, which holds by the base case.

Although the use of a middle terms looks like a lucky trick, the conditions for applying it successfully can be easily characterized and hence implemented, making it transparent for the user. This is described in Section 5.

We can accommodate a mixture of currying and uncurrying by choosing an appropriate well-founded precedence for selecting the right number of arguments desired for a given function symbol: let $J = [1..m]$ and $I = [m+1..p]$ for some $m \in [1..p]$, and let us add the rule

$$\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \ \vdash \ f(x_1, \ldots, x_n) \to f_n(x_1, \ldots, x_n)$$

which is easily proved terminating by adding $f >_{\mathcal{F}} f_n$ to the precedence. The resulting set of rules will then replace all occurrences of f by f_m while adding or eliminating the necessary application operators.

4.5 A mutually inductive definition of the ordering and the computability closure

So far, we restrained ourselves using $\succ_{chorpo}$ in the computability closure definition, and used instead $\succ_{horpo}$ in order to avoid a mutually inductive definition of the ordering and the closure. This allowed us to present both separately, starting with the computability closure which is built on a simple intuitive idea. Using $\succ_{chorpo}$ recursively in Cases 3 and 6 of the closure definition would of course yield a stronger relation. In this section, we show that this indeed yields a well-founded ordering.

First, we need to show that the mutually recursive definition yields a least fixpoint. This is the case since the underlying functional is defined by a set of Horn clauses, hence is monotone.

Then, we need to show that the computability properties remain true when using this new definition. Indeed, there is no impact on any of the proofs but the proof of Lemma 4.11. However, it suffices to replace the assumption based on the ordering $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ by the very same assumption based now on the stronger ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$. The proof then goes exactly as before.

Finally, we remark that the proof of Lemma 4.12 is itself based on an induction with the ordering $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$, hence can be kept as it is. Note however that we needed to prove that $\succ_{horpo}$ was included into $\succ_{chorpo}$. This was Lemma 4.9, and it is needed to relate both inductions, the one with respect to $(>_{\mathcal{F}}, (\succ_{horpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ in Lemma 4.11, and the one with respect to $(>_{\mathcal{F}}, (\succ_{chorpo} \cup \rhd_{\geq_{\mathcal{T}_\mathcal{S}}})_{stat_f})_{lex}$ in Lemma 4.12. This is no more needed, hence the need for lemma 4.9 disappears.

5. IMPLEMENTATION

A PROLOG implementation is available from our web page whose principles are described here. We define first an ordering on types satisfying the required properties before explaining how to approximate $(\succ_{horpo})^*$ and the closure mechanism.

5.1 The recursive path ordering on types

Given a partial quasi-ordering $\geq_{\mathcal{S}}$ on sort constructors again called a *precedence*, as well as a status, we define the following rpo-like quasi ordering on types:

$$\text{Let } B = \forall v \in \overline{\tau} \ \sigma >_{\mathcal{T}_\mathcal{S}} v$$

Definition 5.1. $\sigma \geq_{\mathcal{T}_\mathcal{S}} \tau$ iff

(1) $\sigma = c(\overline{\sigma})$ for some $c \in \mathcal{S}$ and $\sigma_i \geq_{\mathcal{T}_\mathcal{S}} \tau$ for some $\sigma_i \in \overline{\sigma}$

(2) $\sigma = c(\overline{\sigma})$ and $\tau = d(\overline{\tau})$ for some $c, d \in \mathcal{S}$ such that $c >_{\mathcal{S}} d$, and B

(3) $\sigma = c(\overline{\sigma})$ and $\tau = d(\overline{\tau})$ for some $c, d \in \mathcal{S}$ such that $c =_{\mathcal{S}} d$ and $\overline{\sigma}(\geq_{\mathcal{T}_{\mathcal{S}}})_{mul}\overline{\tau}$

(4) $\sigma = c(\overline{\sigma})$ and $\tau = d(\overline{\tau})$ for some $c, d \in \mathcal{S}$ such that $c =_{\mathcal{S}} d$ and $\overline{\sigma}(\geq_{\mathcal{T}_{\mathcal{S}}})_{lex}\overline{\tau}$, and B

(5) $\sigma = \alpha \rightarrow \beta$, and $\beta \geq_{\mathcal{T}_{\mathcal{S}}} \tau$

(6) $\sigma = \alpha \rightarrow \beta$, $\tau = \alpha' \rightarrow \beta'$, $\alpha =_{\mathcal{T}_{\mathcal{S}}} \alpha'$ and $\beta \geq_{\mathcal{T}_{\mathcal{S}}} \beta'$

Note that, because of the subterm property for sort symbols (since they belong to a first-order unisorted structure), our above definition of B ($\forall v \in \overline{\tau}\ \sigma >_{\mathcal{T}_{\mathcal{S}}} v$) is equivalent to the former definition of A applied to types ($\forall v \in \overline{\tau}\ \sigma >_{\mathcal{T}_{\mathcal{S}}} v$ or $u \geq_{\mathcal{T}_{\mathcal{S}}} v$ for some $u \in \overline{\sigma}$). This remark will be used in Section 6.3 to build a single uniform ordering operating on both terms and types.

Proposition 5.2. *$\geq_{\mathcal{T}_{\mathcal{S}}}$ is a type ordering and $>_{\mathcal{T}_{\mathcal{S}}}^{\rightarrow}$ is the recursive path ordering generated by the given precedence on $\mathcal{S}$.*

Proof. It is clear that adding the subterm property for the arrow to the definition of $>_{\mathcal{T}_{\mathcal{S}}}$ yields the recursive path ordering (note that the precedence is not changed, hence symbols in $\mathcal{S}$ do not compare with the arrow). This shows the latter property and implies therefore well-foundedness of $>_{\mathcal{T}_{\mathcal{S}}}$. Arrow-preservation and arrow-decreasingness are built-in. Stability is proved by induction on types. □

All examples given so far can use the above type ordering, provided the appropriate precedence on type constructors is given by the user as well as their status.

5.2 Implementable approximation of $\succ_{horpo}$ and $\succ_{chorpo}$

Among the many issues that must be dealt with, we consider here only the *new* ones. Guessing a precedence and a status for the function symbols and the type constructors is by no means different from what it is for Dershowitz's recursive path ordering, and the technology is therefore well-known. Accordingly, this is not done by our current implementation: the user is supposed to input the precedence to the system which can then *check* a set of rules. On the other hand, guessing a middle term or how to implement the closure mechanism effectively are new implementation problems that we consider here.

Property ***A*****.** The non-deterministic or comparison of proposition A used in cases 2, 4, and 7, can be replaced by the equivalent deterministic one:

$$\forall v : \rho \in \overline{t} \quad \begin{array}{l} \text{if } \rho \leq_{\mathcal{T}_{\mathcal{S}}} \tau \ \text{ then } s \succ_{horpo} v \\ \text{otherwise } u \succeq_{horpo} v \text{ for some } u : \theta \in \overline{s} \text{ such that } \theta \geq_{\mathcal{T}_{\mathcal{S}}} \rho \end{array}$$

Left-flattening. There is a tradeoff between the increase of types and the decrease of size when using left-flattening in Case 9: moving from $@(@(a, b), c))$ to $@(a, b, c)$ replaces the subterm $@(a, b)$ by the two smaller subterms a, b, but a has a bigger type than $@(a, b)$. Type considerations can therefore be used to drive the choice of the amount of flattening.

Guessing middle terms. A weakness of our definition is that the relation $\succ_{horpo}$ does not satisfy transitivity. This is why our statement in Theorem 3.5 uses $(\succ_{horpo})^+$. In most examples, we have used $\succ_{horpo}$, and indeed, the lack of transitivity has two origins: left-flattening and Cases 11 and 12. Therefore, the use of these two cases will usually come together with the need of guessing a middle term u such that $s \succ_{horpo} u \succ_{horpo} t$, the second comparison being done by one of the above two cases. It turns out that it is quite easy to guess when such a middle term may be necessary: when no case other than 11 and 12 applies to compare s and t. More specifically, when we need to compare a term s headed by an application with a term t headed by a function symbol. In this case, a middle term u can be generated by β-expanding t, and we can now try applying Case 9 between s and u and Case 11 between u and t. Similarly, in case s is an abstraction whose λ cannot be pulled out, we can η-expand t before comparing s and u by Case 10 and then u and t by Case 12.

Approximation of the closure. Our implementation of the closure mechanism is an approximation as well, by weakening proposition A defined in Section 4.2 so as to be the same as in the subterm case (Case 1), that is $A = \forall v \in \vec{t}\ s \succ_{chorpo} v$ or $v \in \mathcal{CC}(s)$, which amounts to have a simple guess of the term u defined in the afore mentioned version of A to be t itself. Reduction is only applied to the arguments of the starting term itself, it can therefore now be seen as a new basic case. The implementation of the other rules defining the closure is goal-directed. We have implemented a subset of them, namely application, abstraction, precedence, recursive call and reduction which turned out to suffice for all our examples -we of course had proved these examples by hand before, but the hand-made proofs were indeed more complex and used a larger set of rules. In the current version of the implementation, we use $\succ_{chorpo}$ recursively in the computability closure definition, rather than $\succ_{horpo}$.

Examples All examples considered in the paper have been checked with our implementation, therefore supporting our claim that HORPO is automatable. The only user inputs are the following: the type constructors, their arity and their precedence and status; the function symbols, their type and their precedence and status; the terms to be compared and the type of their free variables. The implementation checks the type of both terms to be compared in the environment formed by the type of the variables, and proceeds with checking the ordering with the approximation of $\succ_{chorpo}$ that we have just described.

6. FROM PAST TO FUTURE WORK

6.1 Comparisons

Higher-order termination is a difficult problem with several instances.

Higher-order rules using plain pattern matching are routinely used in proof systems with inductive types, at the object level (Gödel's recursor is one example), and at the type level as well (under the name of strong elimination in the Calculus of Inductive Constructions [Coquand and Paulin-Mohring 1990]). These rules have a specific format, hence their strong normalization property can be proved by ad'hoc arguments. By allowing rule-based definitions in proof systems, one provides with more general induction schemas than the usual structural induction schema for inductive types. This however requires user-defined plain higher-order rules with the recursor rules for inductive types as particular cases. Then, showing that proof-checking is decidable, and the obtained proof system is sound, requires proving that these rules are strongly normalizing (together with the built-in $\beta\eta$ rules). Our program aims at proving the strong normalization property of such rules, therefore including the recursor rules for inductive types, and provide a generic tool for that purpose. So far, we have answered this question within the framework of polymorphic higher-order algebras only. Our future target is a framework including dependent types, therefore containing the Calculus of Inductive Constructions as a special case.

Higher-order rewrite rules are also used in logical systems to describe computations over lambda-terms used as a suitable abstract syntax for encoding functional objects like programs or specifications. This approach has been pioneered in this context by Nipkow [Mayr and Nipkow 1998] and is available in Isabelle [Paulson 1990]. Its main feature is the use of higher-order pattern matching for firing rules. A recent generalization of Nipkow's setting allows one for rewrite rules of polymorphic, higher-order type [Jouannaud et al. 2005; Jouannaud 2005b]. Proving the strong normalization property of such rules is a different problem, somewhat harder, since requiring a well-founded ordering compatible with $\beta\eta$-equivalence classes of terms, and it is well known that building compatible well-founded orderings is a hard problem as exemplified with the case of associativity and commutativity in the first-order case.

There is still a third way of defining higher-order rewriting, due to Klop, which came actually first [Klop 1980]. Retrospectively, this approach appears to be a sort of instance of Nipkow's more recent approach, in the sense that, given a set of rules, a typed term rewrites in Klop sense when it does so in Nipkow's sense. Since Klop was more interested in confluence than in termination properties, the question of proving termination of Klop's rewrite relation has not been investigated.

Surprisingly, the question of designing orderings for proving higher-order termination attracted attention first within Nipkow's framework. The initial attempts, however, including ours, were not very convincing. All methods were based either on very weak orderings [Jouannaud and Rubio 1998], or required interaction with a general purpose higher-order theorem prover [van de Pol and Schwichtenberg 1995]. Only recently were we able to answer the question with a decidable ordering able to prove all examples we have found in the literature [Jouannaud and Rubio 2006]. What is interesting to note is that this ordering builds on the present one in a decisive way.

Let us therefore now compare the different attempts to prove termination of plain higher-order rules.

First, it should be noted that the present definition is by far superior to the one given in [Jouannaud and Rubio 1999], which could only compare terms of the same type, or of equivalent types. Consequently, we are able to solve many more examples. Besides, the definition of the ordering has been enriched considerably, by adding new cases. We are convinced that Definition 3.1 cannot be substantially improved, except by: handling explicitly bound variables in the recursive calls; using the computability properties of subterms in presence of inductive types; using interpretations as it was successfully done for the recursive path ordering [Kamin and Levy 1980] and for the first version of the higher-order recursive path ordering [Borralleras and Rubio 2001]. On the other hand, we hope that the more complex closure mechanism can be improved and integrated to the ordering. These improvements are carried out in part in [Blanqui et al. 2006].

There is an alternative to using the higher-order recursive path ordering for proving termination of plain higher-order rewriting, based on the computability closure and the general schema [Blanqui et al. 2002; Blanqui 2003]. Because the main construction in [Blanqui 2003] is the computability closure, it can indeed be obtained by restricting $\succ_{chorpo}$ to its first case, and is therefore much weaker for our framework of polymorphic algebras. On the other hand, this method is more advanced insofar it allows for dependent types and even for rules, like strong elimination, operating at the type level. This lets us hope that generalizing the higher-order recursive path ordering to dependent types may be at reach.

Let us finally mention two completely different methods by Jones and Bohr [2004] and by Giesl, Thiemann and Schneider-Kamp [2005]. They differ from ours by targeting functional programs rather than proof systems, and encoding these programs as a set of termination-preserving applicative rules obtained by Currying [Kennaway et al. 1996]. Termination of these applicative rules is then analyzed by various methods. However, since the structure of terms is lost because of the translation, these methods must be based on an analysis of the flow of redexes in terms. This analysis must of course cope with the flow of beta-redexes via the translation. In presence of polymorphic or dependent types, there is no method known, apart from Girard's, which is able to account for the flow of redexes in lambda-terms. Therefore, we do not think that these method can help us solving our grand challenge. In the simpler case of functional programs, these methods may be quite effective whenever termination does not depend upon the type structure used. We believe that the orderings methods considered here are inherently more powerful since they can in principle easily integrate semantic information such as that obtained from an analysis of the flow of redexes.

6.2 Future work

The higher-order recursive path ordering should be seen as a firm step to undergo further developments in different directions. Two of them have been investigated in the first order framework: the case of associative commutative operators, and the use of interpretations as a sort of elaborated precedence operating on function symbols. The second extension has been carried out for a restricted higher-order framework [Borralleras and Rubio 2001]. Both deserve immediate attention. Other extensions are specific to the higher-order case: incorporating the computability closure into the ordering definition; enriching the type system with inductive types, a problem considered in the framework of the general schema by Blanqui [Blanqui et al. 2002; Blanqui 2003]; enriching the type system with dependent types, considered for the original version of the higher-order recursive path ordering [Jouannaud and Rubio 1999] by Walukiewicz [2003]. We suggest below a path to an extension of the higher-order recursive path ordering defined here to a dependent type framework.

6.3 A uniform ordering on terms and their types

We are going here to give a single definition working uniformly on terms and types. Its restriction to types will be the type ordering defined in Section 3, while its restriction to terms will be the higher-order recursive path ordering using this type ordering. This ensures that the new definition is equivalent to Definition 3.1 when using the type ordering introduced in Section 5, since terms and types do not interfere in our framework. The definition is very uniform, working by induction, as it is usual for the recursive path ordering, with cases based on the top operator of the lefthand and righthand expressions to be compared. What makes this uniform definition possible is that the type ordering is a restriction of Dershowitz's recursive path ordering for first-order terms, and that the higher-order recursive path ordering restricts to Dershowitz's recursive path ordering when comparing first-order terms.

For making uniformity possible, we add a new constant in our language, $*$, such that all types have themselves type $*$. We omit the straightforward type system for typing types, which only aims at verifying arities of sorts symbols. $*$ will therefore be the only non-typable term in the language.

6.3.1 *Statuses and Precedence.* We assume given
- a partition $Mul \uplus Lex$ of $\mathcal{F} \cup \mathcal{S}$;
- a well-founded precedence $\geq_{\mathcal{FS}}$ on $\mathcal{F} \cup \mathcal{S}$ such that $\geq_{\mathcal{FS}}$ is the union of a quasi-ordering $\geq_{\mathcal{F}}$ on $\mathcal{F}$ and a quasi-ordering $\geq_{\mathcal{T_S}}$ on $\mathcal{S}$ whose strict parts are well-founded; variables are considered as constants incomparable in $>_{\mathcal{FS}}$ among themselves and with other function symbols.

6.3.2 *Definition of the ordering*

Definition 6.1. Let $A = \forall v \in \overline{t}\ s \underset{horpo}{\succ} v$ or $u \underset{horpo}{\succeq} v$ for some $u \in \overline{s}$

Given $s : \sigma$ and $t : \tau$, $\quad s \underset{horpo}{\succ} t$ iff $\sigma = \tau = *$ or $\sigma \underset{horpo}{\succeq} \tau$ and

(1) $s = f(\overline{s})$ with $f \in \mathcal{FS}$, and $u \underset{horpo}{\succeq} t$ for some $u \in \overline{s}$

(2) $s = f(\overline{s})$ and $t = g(\overline{t})$ with $f >_{\mathcal{FS}} g$, and A

(3) $s = f(\overline{s})$ and $t = g(\overline{t})$ with $f =_{\mathcal{FS}} g \in Mul$ and $\overline{s}(\underset{horpo}{\succ})_{mul}\overline{t}$

(4) $s = f(\overline{s})$ and $t = g(\overline{t})$ with $f =_{\mathcal{FS}} g \in Lex$ and $\overline{s}(\underset{horpo}{\succ})_{lex}\overline{t}$, and A

(5) $s = @(s_1, s_2)$ and $s_1 \underset{horpo}{\succeq} t$ or $s_2 \underset{horpo}{\succeq} t$

(6) $s = \lambda x : \sigma.u$, $x \notin \mathcal{V}ar(t)$ and $u \underset{horpo}{\succeq} t$

(7) $s = f(\overline{s})$, $@(\overline{t})$ is an arbitrary left-flattening of t, and A

(8) $s = f(\overline{s})$ with $f \in \mathcal{F}$, $t = \lambda x : \alpha.v$ with $x \notin \mathcal{V}ar(v)$ and $s \underset{horpo}{\succeq} v$

(9) $s = @(s_1, s_2)$, $@(\overline{t})$ is an arbitrary left-flattening of t and $\{s_1, s_2\}(\underset{horpo}{\succ})_{mul}\overline{t}$

(10) $s = \lambda x : \alpha.u,\ t = \lambda x : \beta.v$, $\alpha{=}\beta$ and $u \underset{horpo}{\succ} v$

(11) $s = @(\lambda x.u, v)$ and $u\{x \mapsto v\} \underset{horpo}{\succeq} t$

(12) $s = \lambda x.@(u, x)$ with $x \notin \mathcal{V}ar(u)$ and $u \underset{horpo}{\succeq} t$

(13) $s = \alpha \rightarrow \beta$, and $\beta \underset{horpo}{\succeq} t$

(14) $s = \alpha \rightarrow \beta,\ t = \alpha' \rightarrow \beta'$, $\alpha \underset{horpo}{=} \alpha'$ and $\beta \underset{horpo}{\succ} \beta'$

This uniform higher-order recursive path ordering operating on terms and types opens the way to its generalization to dependent type calculi such as the Calculus of Constructions.

7. CONCLUSION

We have defined a powerful mechanism for defining polymorphic reduction orderings on higher-order terms that include β- and η-reductions. To the best of our knowledge, this is the first such ordering ever. Moreover, these orderings can be implemented without much effort, as witnessed by our own prototype implementation and the many examples proved. And because these orderings restrict to Dershowitz's recursive path ordering on first-order terms, a hidden achievement of our work is a new, simple, easy to teach well-foundedness proof for Dershowitz's recursive path ordering itself.

Our construction includes the computability closure mechanism originating from [Blanqui et al. 1999], where it was used to define a syntactic class of higher-order rewrite rules that are compatible with beta reductions and with recursors for arbitrary positive inductive types. The language there is indeed richer than the one considered here, since it is the calculus of inductive constructions generated by a monomorphic signature. The usefulness of the notion of closure in these different context shows the strength of the concept.

Using the computability technique instead of the Kruskal theorem to prove the strong normalization property of the ordering is intriguing, and raises the question whether it is possible to exhibit a suitable extension of Kruskal's theorem that would allow proving that the higher-order recursive path ordering is a well-order of the set of higher-order terms. Here, we must confess that we actually failed in our initial quest to find one. In retrospect, the reason is that we were looking for too strong a statement. It may be that a version of Kruskal's theorem holds based on an adequate notion of subterm such as the weak subterm property satisfied by HORPO when dropping Case 11. We do not think that any Kruskal-like theorem holds when Case 11 is included.

ACKNOWLEDGMENTS

We are grateful to Frédéric Blanqui, Adam Koprowski and Femke van Ramsdoonk for several useful remarks and to a careful referee for scrutinizing the paper and making important suggestions.

REFERENCES

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press.

Barendregt, H. 1990. Functional programming and lambda calculus. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. Elsevier Science Publishers B.V.

Barendregt, H. 1992. Typed lambda calculi. In *Handbook of Logic in Computer Science*, A. et al., Ed. Vol. 2. Oxford University Press.

Bezem, M., Klop, J., and de Vrijer, R., Eds. 2003. *Term Rewriting Systems.* Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Blanqui, F. 2003. Inductive types in the calculus of algebraic constructions. In *6th International Conf. on Typed Lambda Calculi and Applications.* Lecture Notes in Computer Science, vol. 2701. Springer, 46–59.

Blanqui, F., Jouannaud, J.-P., and Okada, M. 1999. The Calculus of Algebraic Constructions. In *10th International Conf. on Rewriting Techniques and Applications*. Lecture Notes in Computer Science, vol. 1631. Springer, 301–316.

Blanqui, F., Jouannaud, J.-P., and Okada, M. 2002. Inductive data types. *Theoretical Computer Science 272,* 1-2, 41–68.

Blanqui, F., Jouannaud, J.-P., and Rubio, A. 2006. Higher-order termination: From kruskal to computability. In *13th International Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science, vol. 4246. Springer, 1–14.

Blanqui, F., Jouannaud, J.-P., and Strub, P.-Y. 2005. The Calculus of Congruent Constructions. draft available from the web at `http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud`.

Borralleras, C. and Rubio, A. 2001. A monotonic, higher-order semantic path ordering. In *8th International Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Computer Science, vol. 2250. Springer, 531–547.

Buchholz, W. 1995. Proof-theoretic analysis of termination proofs. *Annals Pure Applied Logic 75,* 1-2, 57–65.

Coquand, T. 1994. Inductive definitions and type theory.

Coquand, T. and Paulin-Mohring, C. 1990. Inductively defined types. In *International Conf. on Computer Logic, 1988*. Lecture Notes in Computer Science, vol. 417. Springer, 50–66.

Dershowitz, N. 1982. Orderings for term rewriting systems. *Theoretical Computer Science 17,* 3, 279–301.

Dershowitz, N. and Jouannaud, J.-P. 1990. *Rewrite systems*. Vol. B. North-Holland, 243–309.

Fernández, M. and Jouannaud, J.-P. 1995. *Modular termination of term rewriting systems revisited*. Lecture Notes in Computer Science, vol. 906. Springer, 255–272.

Gallier, J. 1990. On girard's "candidats de reductibilité". In *Logic and Computer Science*. Academic Press, 123–203.

Giesl, J., Thiemann, R., and Schneider-Kamp, P. 2005. Proving and disproving termination of higher-order functions. In *5th International Workshop on Frontiers of Combining Systems*. Lecture Notes in Computer Science, vol. 3717. Springer, 216–231.

Girard, J.-Y., Lafont, Y., and Taylor, P. 1989. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Goubault-Larrecq, J. 2001. Well founded recursive relations. In *15th International Workshop on Computer Science Logic*. Lecture Notes in Computer Science, vol. 2142. Springer, 484–497.

Jones, N. and Bohr, N. 2004. Termination analysis of the untyped lambda-calculus. In *15th International Conf. on Rewriting Techniques and Applications*. Lecture Notes in Computer Science, vol. 3091. Springer, 1–23.

Jouannaud, J.-P. 2005a. Extension orderings revisited. Available from the web at `http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud`.

Jouannaud, J.-P. 2005b. Higher-order rewriting: framework, confluence and termination. In *Processes, Terms and Cycles: Steps on the road to infinity. Essays Dedicated to Jan Willem Klop on the occasion of his 60th Birthday.* Springer Verlag.

Jouannaud, J.-P. and Okada, M. 1991. A computation model for executable higher-order algebraic specifications languages. In *6th Annual IEEE Symp. on Logic in Computer Science.* IEEE Computer Society Press, 350–361.

Jouannaud, J.-P. and Rubio, A. 1998. Rewrite orderings for higher-order terms in η-long β-normal form and the recursive path ordering. *Theoretical Computer Science 208,* 1-2, 3–31.

Jouannaud, J.-P. and Rubio, A. 1999. The higher-order recursive path ordering. In *14th Annual IEEE Symp. on Logic in Computer Science.* IEEE Computer Society Press, 402–411.

Jouannaud, J.-P. and Rubio, A. 2006. Higher-order orderings for normal rewriting. In *17th International Conf. on Rewriting Techniques and Applications.* Lecture Notes in Computer Science, vol. 4098. Springer, 387–399.

Jouannaud, J.-P., van Raamsdonk, F., and Rubio, A. 2005. Higher-order rewriting with types and arities. Available from the web at `http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud`.

Kamin, S. and Levy, J.-J. 1980. Two generalizations of the recursive path ordering. Technical report, Department of computer science, University of Illinois at Urbana-Champaign.

Kennaway, J., J.W. Klop, M. S., and de Vries, F. 1996. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation 21*, 15–39.

Klop, J. 1980. *Combinatory Reduction Systems.* Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam.

Klop, J. W. 1992. Term rewriting systems. In *Handbook of Logic in Computer Science*, A. et al., Ed. Vol. 2. Oxford University Press.

Loría-Sáenz, C. and Steinbach, J. 1992. Termination of combined (rewrite and λ-calculus) systems. In *3rd International Workshop on Conditional Term Rewriting Systems.* Lecture Notes in Computer Science, vol. 656. Springer, 143–147.

Mal'cev, A. 1961. On the elementary theories of locally free universal algebras. *Soviet Math. Doklady 2,* 3, 768–771.

Mayr, R. and Nipkow, T. 1998. Higher-order rewrite systems and their confluence. *Theoretical Computer Science 192,* 1, 3–29.

Paulson, L. C. 1990. Isabelle: The next 700 theorem provers. In *Logic and Computer Science.* Academic Press, 361–386.

van de Pol, J. and Schwichtenberg, H. 1995. Strict functional for termination proofs. In *2nd International Conf. on Typed Lambda Calculi and Applications.* Lecture Notes in Computer Science, vol. 902. Springer, 350–364.

Walukiewicz-Chrzaszcz, D. 2003. Termination of rewriting in the calculus of constructions. *Journal of Functional Programming 13,* 2, 339–414.

Received July 2005; revised July 2006; accepted November 2006

Crash course on higher-order logic[*]

Theodore Sider December 2, 2022

Contents

[*] Thanks to Daniel Berntson, Eliya Cohen, Cian Dorr, John Keller, Brock Sides, Eric Winsberg, and Alessandro Torza for comments. This was written for a graduate seminar I taught in the spring of 2020. Despite its flaws (such as lack of rigor and comprehensiveness), it may have some value as a first port of call. Other introductions to this material have started to appear, such as the first chapter of Dorr et al. (2021) and Bacon (2022).

1. Introduction

“Higher order metaphysics” is a hot topic, and an excellent one. The work is at a high level, and is intrinsically interesting. It is relatively new, so there is still much to be done. And it connects to adjacent areas, such as logic, philosophy of logic, philosophy of mathematics, and philosophy of language. Getting up to speed on higher-order matters is a good way to broaden your horizons.

But much of the literature on higher-order metaphysics is very hard to understand, if you’re coming in from the outside. It is often technical, and familiarity with various issues in logic, philosophy of logic, and philosophy of mathematics is often presupposed.

For any subject, if you lack the background, the experience of reading an advanced paper can be very scary. Nothing makes any sense! And this is especially true if a lot of the paper is written in symbols that you don’t understand, or barely understand. It is easy to just assume that the topic is not for you, and move on to something else. But the situation is rarely as bad as it seems; usually, all you need is a little background. Then the papers won’t be anywhere near as scary, and you’ll understand most of what is going on; and you’ll be able to pick up the rest as you continue.

The necessary background for higher-order metaphysics isn't that difficult. But it's spread out in many different places, and (as far as I know) isn't presented anywhere in a concise and introductory way. This document will go through this background slowly and informally, without (I hope!) presupposing knowledge of anything more than introductory logic. This background will include some basic logic, philosophy of logic, philosophy of mathematics, and philosophy of language.

This is intended to be a very informal introduction. I will sacrifice rigor and comprehensiveness to make things accessible.[1]

2. Importance of syntax to logic

In the literature on higher-order metaphysics, syntax—that is, grammar—often matters a great deal. This might seem strange. So I'd like to start by talking about why syntax is important in logic.

Logic is largely about logical implication, about what follows from what.

"Logical implication" means something like: truth preservation in virtue of logical form. For example, 'Jones is a sister' does not logically imply 'Jones is a sibling', because even though the latter is true whenever the former is, this is not because of their logical forms; it is because of the meanings of 'sister' and 'sibling'.

Since logical implication is by virtue of form, our statements of logical rules, such as conjunction elimination:

$$\frac{A \text{ and } B}{A}$$

talk about logical form. The statement of the logical rule says: any sentence that has the logical form above the line (namely, "A and B") logically implies the corresponding sentence below the line. Other rules concern sentences with other logical forms, such as "A or B", "If A then B", and so on.

But we need a clear account of syntax in order to state such rules properly. Why isn't the following an instance of conjunction elimination?

[1] For instance, when sketching formal semantics for quantified languages, I won't deal with variables properly; I'll say things like "$\forall vA$ is true iff A is true of every member of the domain".

Daisy and Luke are siblings

Therefore, Daisy

It's because the variables A and B in the rule of conjunction elimination are supposed to stand for *sentences*, and the word 'and' in the rule is assumed to occur between sentences. Thus conjunction elimination and other logical rules use certain syntactic concepts (such as that of a sentence) and presuppose a certain syntax for the language to which they're applied.[2]

The success of a proposed logic depends crucially on the quality of its assumptions about syntax. To take a famous example, consider how Frege's logic supplanted Aristotle's.[3] Aristotle's syllogistic logic had been the dominant conception of logic for centuries, but in the late nineteenth century, Frege (and others) showed that Aristotle's logic was too weak, and developed a much more powerful logic that we now know as predicate logic. Aristotle's logic was limited in power precisely because of its too-crude syntactic assumptions, namely that the sentences that can occur in syllogisms always have subject-predicate form ('a is a G', 'All Fs are Gs', 'Some Fs are Gs', 'No Fs are Gs', etc.), with a single subject and single predicate. Frege's main innovations were syntactic. Frege allowed predicates to have multiple subjects, as in 'Jones respects Smith', in which the predicate is 'respects' and the subjects are 'Jones' and 'Smith', and replaced Aristotelian general subjects 'All Fs', 'Some Fs', and 'No Fs', with quantifiers $\forall x$ and $\exists x$ (not his notation), which could be attached to sentences containing sentential connectives such as $\wedge$, $\vee$, $\rightarrow$, and $\sim$. Without developing this new syntax, one couldn't even state familiar logical rules of inference such as conjunction elimination, existential generalization, and so on. With the new syntax, Frege was able to explain the correctness of arguments that could not be explained by Aristotle, such as:

Someone respects everyone

Therefore, everyone is respected by someone (or other)

The best representation Aristotle could have given these sentences would have

[2] In typical formal languages, the symbol for 'and' (e.g., '$\wedge$') can only occur between sentences. 'And' in English has a more flexible syntax. For example, in addition to connecting sentences, it can connect names to form complex plural subjects. Stating systematic rules of inference for a natural language like English is much harder than stating rules for formal languages with a simpler, and stipulated, syntax.

[3] The history here is actually more complex than the following caricature suggests.

been:

> Some Fs are Gs
>
> Therefore, every F is an H

where F stands for 'is a thing' (a predicate true of everything), G stands for 'respects-everyone', and H stands for 'is-respected-by-someone'. This is obviously not a valid syllogism. The problem is that Aristotle has no way of "breaking up" the predicates G and H. His syntax allows him no way of recognizing the further logical structure they contain. But Frege's syntax does let him recognize this structure; he can represent the sentences thus:

> $\exists x \forall y Rxy$
>
> $\forall y \exists x Rxy$

In Frege's logic, the second sentence does indeed follow from the first.

2.1 Syntax in formal languages

Syntax has to do with what combinations of symbols "make sense", or are "well-formed". In English, 'Sally owns a dog' is a grammatical sentence, and 'owns a dog' is a grammatical verb phrase. In contrast, 'Sally dog a owns' isn't a grammatical sentence, and 'dog a owns' isn't a grammatical verb phrase. Those strings of words don't make sense; they aren't English. Concepts like *sentence* and *verb phrase* are syntactic concepts.

In a modern approach to logic, one doesn't deal with natural languages, because their syntax is too complex. Rather, one develops formal languages. The syntax of a formal language is typically similar to that of natural languages in certain ways, but simpler, free of ambiguity, and stipulated.

To give a syntax for a formal language, one gives a rigorous definition of what counts as a grammatical, or "well-formed" formula—or just "formula" for short. The syntactic concept of *formula* is the analog in the formal language of the syntactic concept of being a grammatical sentence of English. Here is a typical definition:

1. “$Rt_1 \dots t_n$” is a formula, for any n-place predicate R and any n terms (i.e., names or variables), $t_1, \dots, t_n$
2. If A is a formula, so is “$\sim A$”
3. If A and B are formulas, so are “$(A \wedge B)$”, “$(A \vee B)$”, “$(A \rightarrow B)$”, and “$(A \leftrightarrow B)$”
4. If A is a formula then so are “$\forall v A$” and “$\exists v A$”, where v is any variable
5. A string of symbols is a formula only if it can be shown to be a formulas using rules 1–4

To illustrate: $\forall x(Fx \rightarrow Gx)$ is a formula (assuming that x is a variable and F and G are one-place predicates), since: by clause 1, Fx and Gx are formulas; and so by clause 3, $(Fx \rightarrow Gx)$ is a formula; and so, by clause 4, $\forall x(Fx \rightarrow Gx)$ is a formula. But $\forall F \rightarrow)x$ is not a formula, since it can’t be shown to be a formula by sequential application of rules 1–4.

Note, by the way, the use of the concepts of *predicate* and *term* in this definition. Like the concept of a formula, these are syntactic concepts. (They are partly, but only partly, analogous to the natural-language syntactic concepts of *verb phrase* and *noun phrase*.)

In logic, then, expressions fall under syntactic categories (predicate, term, formula). Moreover, we make reference to those categories when we state logical principles. For example, the rule of conjunction elimination, for a formula language like the one we just developed, is:

$$\frac{A \wedge B}{A}$$

A fuller statement makes the reliance on the syntax clear:

> For any formulas A and B, the formula $A \wedge B$ logically implies A.

3. First- versus second-order logic

3.1 Syntax

The kind of logic usually taught in introductory philosophy classes is what is known as "first-order logic". A different kind, "second-order" logic (and indeed, third- and higher-order logic) will be important for us.

The difference between first- and second-order logic begins with a difference in syntax. The definition of a formula we considered in the previous section was the one for first-order logic. Given that definition, there are formulas like these:

$$\forall x Gx \qquad \exists x \exists y Bxy$$

but there are no formulas like the following:

$$\exists F Fa \qquad \forall R(Rab \rightarrow Rba)$$

in which 'F' and 'R' are variables. That is, variables are allowed to occur in *subject* position only (the syntactic positions in which you're allowed to fill in names), not predicate position. But in second-order logic, variables *are* allowed to occur in predicate position, and so the second pair of expressions do count as formulas.

To state the syntax for second-order logic, first, we make a distinction amongst variables. Variables now come in two (disjoint) types: *individual variables* $x, y, z \ldots$ (these are the old style) and, for each natural number $n \geq 1$, n-place *predicate variables*: $F, X, R, \ldots$. Second, we change the first clause in the definition of a formula:

1′. For any n-place predicate *or predicate variable* R and any n terms (i.e., names or *individual* variables), $t_1, \ldots, t_n$, "$Rt_1 \ldots t_n$" is a formula.

Other than that, everything stays the same.

What do formulas like $\exists F Fa$ and $\forall R(Rab \rightarrow Rba)$ *mean*? Well, there isn't really a fixed answer—different people use such formulas to mean different things. But on one common usage, these formulas quantify over properties.

Thus the first means, to a first approximation, that a has some property; and the second means that b bears every relation to a that a bears to b.

This is only an approximation; and saying something more accurate would be getting ahead of ourselves. But right here at the start we can make an important point. Consider:

> a has some property

Let's make a distinction between a *first-order* and a *second-order* symbolization of this sentence. The second-order symbolization is the one we've just met: $\exists FFa$. The first-order symbolization, on the other hand, is:

$$\exists x(Px \wedge Hax)$$

where 'P' is a one-place predicate symbolizing 'is a property' and 'H' is a two-place predicate symbolizing 'has' (i.e., instantiates). This second symbolization really is a first-order sentence, not a second-order sentence, since its variable (namely, x) only occurs in subject position.

Thus the difference between first-order and second-order logic *isn't* that the latter is needed to talk about properties. One can talk about any entities one wants, including properties, using first-order logic, since there is no limit to the kinds of entities that can be in the range of the first-order quantifiers '$\forall x$' and '$\exists x$'. If there are such things as properties, we can quantify over them; and if it makes sense to speak of objects *having* these properties, then we can introduce a predicate H for having in a first order language.

(Caveat: sometimes the term 'second-order' (or 'higher-order') is used loosely, to refer to all talk about properties. But in this document I will be using the term in the narrowly logical sense, to refer to sentences in which quantified variables occur in predicate position—and later, other non-subject positions.)

But this naturally raises the question of what the difference in meaning is, between the second-order sentence $\exists FFa$ and the first-order sentence $\exists x(Px \wedge Hax)$. We'll be getting to that soon; but to preview: some people think that there is no difference, whereas others regard second-order quantifiers as being *sui generis*, and thus regard such sentences as not being equivalent.

3.2 Formal logic and logical consequence

Modern mathematical logic is a branch of mathematics. You set up a bunch of definitions—for instance, the notion of a formula in a given formal language, the notion of an interpretation, the notion of a formula being true in an interpretation, etc.—and then you give mathematical proofs of various facts given the definitions—e.g., that the formula $\forall x(Hx \vee {\sim}Hx)$ is true in all interpretations.

Unless you think there is something wrong with mathematics itself, there can be no disputing the results of mathematical logic. But these results are about stipulatively defined notions. It's a *non*mathematical claim that if a formula (such as $\forall x(Hx \vee {\sim}Hx)$) is true in all interpretations in some stipulatively defined sense, then meaningful natural language sentences represented by that formula (such as 'Everything is either a human or not a human') are *logical truths*, in a sense that is *not* stipulatively defined. Similarly, it is an indisputable mathematical truth that in every interpretation in which a formula ${\sim}{\sim}Ha$ is true, the formula H is true; but it is a nonmathematical claim that, for example, "It's not the case that it's not the case that Ted is human" *logically implies* (again, in a nonstipulated sense) "Ted is human". And indeed, some people deny these claims about logical truth and logical implication, despite admitting the mathematical results. For example, there is a school of thought about logic called intuitionism that rejects both double negation elimination and the law of the excluded middle.

One of the main points of logic is to study the notions of logical truth and logical implication. These notions are not stipulatively defined. They are objects of philosophical reflection, just like knowledge or persistence or goodness or beauty. In mathematical logic we devise various formal gizmos, but it is a philosophical claim that these formal gizmos accurately represent or depict or model logical truth and logical consequence.

(When I say that there are philosophical questions about logical truth and implication, I don't mean to prejudge the question of how deep or substantive those questions are. As I mentioned earlier, it is common to say that 'Jones is a sister' does not *logically* imply 'Jones is a sibling' since this implication does not hold in virtue of "form", but rather in virtue of the meanings of 'sister' and 'sibling'. But a paradigm of logical implication, such as the implication of 'Jones is a sister' by 'Jones is a sister and Jones is a lawyer', also holds by virtue of meaning (in whatever sense the first implication does), namely, by virtue of the meaning of ‘and’. Why do we count ‘and’, but not ‘sister’ or ‘sibling’, as being part of the “forms” of sentences? Put another way, why is ‘and’, but not ‘sister’ or ‘sibling’, a *logical constant*?[4] It’s an open question how deep the answers to such questions are, and thus an open question how deep the questions of logical truth and logical consequence are along certain dimensions.[5])

To sum up: it’s important to distinguish the notions of truth and logical consequence—whose meanings are not stipulated—from the stipulative notions we construct in mathematical logic (such as truth-in-all-interpretations-of-a-certain-sort), which may or may not yield an accurate model of the former notions.

3.3 Semantics

There are two main kinds of formal/mathematical models of the notions of logical truth and logical consequence: *semantic* models and *proof-theoretic* models.

Let’s start with semantic models. Here is the overall idea:

Semantic approach

1. Define *interpretations*—mathematically precise “pictures” of logical possibilities
2. Define the notion of a sentence’s being *true-in* a given interpretation
3. Use these notions to define metalogical concepts. E.g., a sentence is *valid* iff it is true in all interpretations; a set of sentences Γ *semantically implies* a sentence S iff S is true in every interpretation in which every member of Γ is true.

[4] See MacFarlane (2005) for an overview of this issue.

[5] But note that a failure of depth about what is, e.g., *logically true* wouldn’t imply a failure of depth about what is *true*. The disagreement between intuitionists and classical logicians isn’t just about what is logically true, e.g., whether ‘the decimal expansion of π either does or does not contain some sequence of 666 consecutive 6s’ is a logical truth. It also is about, e.g., whether the decimal expansion of π either does or does not contain some sequence of 666 consecutive 6s; and the substantivity of this latter disagreement isn’t undermined by nonsubstantivity of what counts as a logical constant.

For first- and second-order logic, the usual definition of interpretation is exactly the same:

Intepretation
(for both first- and second-order logic)

A nonempty set, D (the "domain"), plus an assignment of an appropriate denotation based on D to every nonlogical expression in the language. Names are assigned members of D; one-place predicates are assigned sets of members of D; two-place predicates are assigned sets of ordered pairs of D; and so on.

But the definition of truth in an interpretation differs a little between first- and second-order logic. In the case of first-order logic, the definition looks roughly like this:[6]

Definition of truth

i) A sentence Fa is true in an interpretation I if and only if the denotation of the name a is a member of the denotation of the predicate F; a sentence Rab is true if and only if the ordered pair of the denotation of the name a and the denotation of the name b is a member of the denotation of the predicate R; etc.

ii) A sentence $\sim A$ is true in I if and only if A is not true in I; a sentence $A \wedge B$ is true in I if and only if A is true in I and B is true in I; etc.

iii) A sentence $\forall v A$ is true in I if and only if A is true of every member of D; a sentence $\exists v A$ is true in I if and only if A is true of some member of D.

In the case of second-order logic, we keep these three clauses in the definition of truth in an interpretation, and add a clause for quantified sentences with second-order quantifiers:

[6] These and other such definitions in this document are not formally rigorous, particularly in the clauses for quantified sentences.

iv) Where R is an n-place predicate variable, a sentence $\forall RA$ is true in I if and only if A is true of every set of n-tuples of members of D; a sentence $\exists RA$ is true in I if and only if A is true of some set of n-tuples of members of D.

Here is the informal summary. In semantics, first- and second-order logic use the same notion of an interpretation: namely, a domain together with denotations for names and predicates. As for the definition of truth-in-an-interpretation, the second-order definition is just like the first-order one, except that we add a provision for second-order sentences: the second-order quantifiers $\forall R$ and $\exists R$ range over the n-tuples of the domain.

3.4 Proof theory

The second main kind of formal/mathematical model of the notions of logical truth and logical implication is the proof-theoretic model. Here the idea is that logical implication is *provability*: a set of formulas Γ logically implies a formula A if and only if there exists a *proof* of A from Γ.

The idea of a proof can be made precise in a number of ways. Here is a particularly simple one (called a Hilbert system of proof). First one chooses a set of formulas called *axioms*, and a set of relations over formulas called *rules*. We say that a formula S *follows via rule* R from other formulas $S_1, \dots, S_n$ iff the formulas $S_1, \dots, S_n, S$ stand in R. (For example, the rule *modus ponens* is the relation that holds between any three formulas of the form A, $A \rightarrow B$, and B.) And then we define proofs thus:

A *proof of A from* Γ is a finite series of formulas, the last of which is A, in which each formula is either i) a member of Γ, ii) an axiom, or iii) follows from earlier lines in the series by a rule

And finally we say that Γ proves A iff there exists some proof of A from Γ

One more bit of terminology: a *theorem* is a formula, A, such that there exists a proof of A from the empty set $\varnothing$. What is a proof from $\varnothing$? Well, if you think about the definition of a proof from Γ, a proof from ∅ is a proof in which no premises (other than axioms) are allowed. Thus a theorem is a formula that can be proven from the axioms alone—this is the proof-theoretic notion of a logical truth.

The idea is to choose axioms that are obviously logical truths—statements such as:

$$\big((A \vee B) \wedge {\sim}A\big) \rightarrow B$$

and to choose rules that are obviously logical implications, such as modus ponens. Thus a proof of A from Γ is just a chain of good arguments, starting from Γ and culminating in A.

There are other ways of making provability precise.[7] For instance, many other systems of proof allow one to make temporary assumptions, as when one assumes the antecedent of a conditional in a conditional proof, or when one assumes the opposite of what one is trying to prove in a reductio. But let's stick with the Hilbert approach.

The Hilbert approach to provability can be taken with both first- and second-order logic. For example, in first-order logic, one axiom system looks like this:[8]

[7] But all ways of defining what counts as a proof share the feature of being *mechanically checkable* in a certain sense. For example, to check whether a finite series of formulas counts as a legal proof in a given Hilbert system, start with the first formula of the series. Is it a member of Γ or an axiom? If not, we don't have a legal proof. Otherwise move on to the second formula. Is it a member of Γ or an axiom? If not, does it follow from earlier formulas in the series by some rule? If not, then we don't have a legal proof. Otherwise, on to formula three. Eventually we'll reach the final formula, and will have checked whether the series counts as a proof. Note that in any Hilbert system, it is always required that it be mechanically decidable what counts as an axiom or rule of that system.

[8] See Shapiro (1991, section 3.2) for this system and the one for second-order logic. That book is a generally useful although not easy source of information on second-order logic. In universal instantiation, t must be a term that is "free for v in A"; in universal generalization, v must be a variable that doesn't occur freely in A or any member of Γ.

Axioms:

$$
\begin{gathered}
A \to (B \to A) \\
\big(A \to (B \to C)\big) \to \big((A \to B) \to (A \to C)\big) \\
(\sim A \to \sim B) \to (B \to A) \\
\forall v A \to A_{v \mapsto t} \qquad \text{(universal instantiation)}
\end{gathered}
$$

Rules:

$$\frac{A \qquad A \to B}{B} \quad \text{(modus ponens)} \qquad\qquad \frac{A \to B}{A \to \forall v B} \quad \text{(universal generalization)}$$

In the axioms, "$A_{v \mapsto t}$" means: the result of starting with A, and then replacing every free occurrence of v with an occurrence of t. For example, if A is Fv, then $A_{v \mapsto t}$ would be Ft.

The displayed "axioms" are in fact not axioms. Rather, they are axiom *schemas*. Take the first schema, $A \to (B \to A)$. The letters "A" and "B" are schematic variables, which stand for any formulas. When you replace them with formulas, the result is called an *instance* of the schema. Here are some instances of the schema $A \to (B \to A)$:

$$
\begin{gathered}
P \to (Q \to P) \\
R \to (S \to R) \\
\sim(P \to Q) \to \big((Q \to R) \to \sim(P \to Q)\big) \\
\text{etc.}
\end{gathered}
$$

The idea, then, is that *every* instance of $A \to (B \to A)$ counts as an axiom; and similarly for the other three axiom schemas. Thus there are infinitely many axioms, not just four; but each axiom falls under one of four patterns.

To get various proof systems for second-order logic, we can just add new axioms and/or rules to those of the above system for first-order logic. For instance:

New axiom schemas

$$\forall X A \rightarrow A_{X \mapsto F} \qquad \text{(second-order universal instantiation)}$$

$$\exists X \forall v_1 \dots \forall v_n (X v_1 \dots v_n \leftrightarrow A) \qquad \text{(Comprehension)}$$

(X is any n-place predicate variable, F is any n-place predicate, and A is a schematic variable for any formula, which cannot have X free.)

Second-order universal instantiation is just like the old version of universal instantiation, except that the variable is second-order. Here is an example of an instance:

$$\forall R(Rab \rightarrow Rba) \rightarrow (Tab \rightarrow Tba)$$

where R is a two-place predicate variable and T is a two-place predicate constant. (We might similarly add a second-order version of the rule of universal generalization.)

To get a feel for the Comprehension axiom, let A be the following formula:

$$Hx \wedge \exists y(Cy \wedge Oxy)$$

which might symbolize, e.g.,

x is a hipster who owns at least one chicken

We can then write down the following instance of Comprehension:

$$\exists X \forall x\big(Xx \leftrightarrow Hx \wedge \exists y(Cy \wedge Oxy)\big)$$

where X is a one-place predicate variable. In other words: there exists a property which is had by an object if and only if that object is a hipster who owns at least one chicken.

Thus what Comprehension says, roughly, is that to any formula A there is a corresponding property or relation. (Thus this is a lot like the principle of naïve comprehension from set theory. As we will see later, it doesn't lead to the same trouble.)

3.5 Metalogic

Here's what we have done so far. For each of first- and second-order logic, we've laid out i) a syntax for the language, ii) a semantics, and iii) a proof theory. We are now in a position to understand some of the dramatic differences between first- and second-order logic. The differences have to do with the behavior of the metalogical notions introduced above: e.g., provability, semantic consequence, and so on.

(The notions are called "metalogical" to distinguish them from logical notions such as "and", "not", and "all". Metalogical notions are notions that we employ when we are talking about formal logical languages and investigating their properties.)

3.5.1 Completeness

Let's start with the relationship between theoremhood and validity. As we saw earlier, these are two different ways of formalizing the intuitive notion of logical truth. However, as Kurt Gödel proved in his dissertation in 1929, the two formalizations are equivalent in the case of first-order logic. Gödel proved:

Completeness If a first-order formula is valid then it is a theorem

It's relatively straightforward to prove the converse:

Soundness If a first-order formula is a theorem then it is valid

Thus the two results together tell us that a first order theorem is a theorem *if and only if* it is valid—i.e., that for first-order logic, theoremhood is equivalent to validity.

However, for second-order logic, the situation is very different. The axiom system mentioned above is indeed sound: every theorem is valid. However, that system isn't complete: some valid formulas aren't theorems. Moreover, this isn't just a failing of the particular system mentioned above. It follows from Gödel's first *in*completeness theorem (which he proved two years later, in 1931) that there can be *no* sound and complete axiom system for second-order logic.

There are some details here that matter. First, there is a trivial sense in which we *can* come up with a complete axiom system for second-order logic, if we allow systems with “rules” like the following:

$$\frac{A_1, A_2, \ldots}{B} \quad \text{(where } B \text{ is any formula that is semantically implied by } A_1, A_2, \ldots\text{)}$$

or if we are allowed to define the “axioms” as being all the valid formulas. In a proper axiomatic system, it is required that there be a mechanical procedure—in a certain precise sense—for deciding what counts as a rule or axiom.

Second, the claim that there can be no sound and complete axiom system for second-order logic is tied to the definition of an interpretation for second-order logic that was given in section 3.3. The interpretations defined there are often called “standard” or “full” interpretations, since in them the second-order quantifiers range over *all* sets of n-tuples drawn from the domain. But one can define other notions of interpretation, in which the second-order quantifiers range over only some such sets; and there is no guarantee that metalogical facts that hold under one notion of interpretation will continue to hold under other notions of interpretation. For example, if “Henkin interpretations” are substituted for full interpretations, then one can have a sound and complete axiom system after all. Similar remarks apply to other metalogical differences between first- and second-order logic to be discussed below.

This is philosophically important. It means that, in order to draw substantive philosophical conclusions from metalogical facts like the incompleteness of second-order logic (a procedure which is philosophically fraught anyway, as we will see) one must be sure that the definition of interpretation involved in the metalogical facts matches (as much as possible!) the intended interpretation of the second-order quantifiers.

3.5.2 Compactness

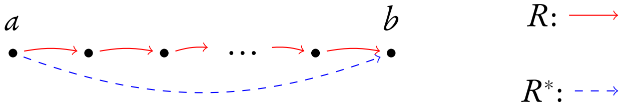
Call a set of formulas, Γ, *satisfiable* if and only if there is some interpretation in which every member of Γ is true. (This is a semantic notion of consistency.) The following holds for first-order logic (it is a fairly immediate consequence of the completeness theorem):

Compactness If every finite subset of Γ is satisfiable, Γ is satisfiable.

To get a sense of why this is important, consider the following. Let R be any two-place predicate. (Not a predicate variable; a predicate constant.) Informally put, the *ancestral* of R, R^*, is a predicate obeying the following rule:

R^*ab iff: Rab, or
Rax and Rxb, for some x, or
Rax and Rxy and Ryb, for some x and y, or …

That is: a is an R-ancestor of b iff there is some "finite R-chain" from a to b:



('Ancestral' because ancestor-of is the ancestral of parent-of.)

In fact there is no way to define R^* using first-order predicate logic. For suppose otherwise—suppose there is some sentence, R^*ab, of predicate logic that says that a is an R-ancestor of b. Now consider the following infinite set, Γ, of sentences:

$$\Gamma = \{R^*ab, A_1, A_2, A_3, \ldots\}$$

where the sentences $A_1, A_2, A_3, \ldots$ are the following:

$A_1: \quad \sim Rab$ ("There is no one-link R-chain")
$A_2: \quad \sim\exists x(Rax \wedge Rxb)$ ("There is no two-link R-chain")
$A_3: \quad \sim\exists x\exists y(Rax \wedge Rxy \wedge Ryb)$ ("There is no three-link R-chain")
etc.

Notice two things about Γ: 1. Γ is unsatisfiable—that is, there is no interpretation in which every member of Γ is true. For if R^*ab is true in an interpretation then there is some finite R-chain from a to b, in which case one of the A_is must be false in that interpretation. 2. Every finite subset of Γ is satisfiable. For any finite subset of the A_is merely rules out finite chains between a and b up to some particular length (the length being whatever the highest of the A_is is in the finite subset), and R^*ab can still be true if there is a chain longer than that between a and b.

But 1 and 2 together contradict Compactness. Therefore there can exist no such sentence R^*ab. You can't define the ancestral of a predicate using first-order logic.

Thus compactness tells us something very important about first-order logic: that language is expressively weak in a certain way. (There are other ways in which first-order logic is expressively weak. One other related example is that you can't express in first-order logic the idea that there are only finitely many things.)

Second-order logic is very different in this respect. Compactness does *not* hold for second-order logic. And in second-order logic you *can* define the ancestral of a predicate:

$$R^{*}ab \leftrightarrow \forall F\Big(\Big(\forall x(Rax \rightarrow Fx) \wedge \forall x \forall y((Fx \wedge Rxy) \rightarrow Fy)\Big) \rightarrow Fb\Big)$$

What this says, intuitively, is this:

> a is an R-ancestor of b if and only if for every property, F: IF i) everything that a bears R to is an F, and ii) whenever an F bears R to something, that something is also an F, THEN b is an F

or, to put it differently:

> a is an R-ancestor of b if and only if b has every property that i) is had by every "R-child" of a, and ii) is "closed under" R

Thus there is a sense in which second-order logic is more expressively powerful than first-order logic: you can say more with the language of second-order logic.

3.5.3 Clarifying differences in expressive power

But it's important to be clear about the exact sense in which second-order logic is more expressively powerful. After all, consider the following first-order sentence, in which '∈' is a two-place predicate for set-membership:

$$R^{*}ab \leftrightarrow \forall z\Big(\Big(\forall x(Rax \rightarrow x \in z) \wedge \forall x \forall y\big((x \in z \wedge Rxy) \rightarrow y \in z\big)\Big) \rightarrow b \in z\Big)$$

What this says is that b is a member of every set, z, that contains every R-child of x and is closed under R. This clearly seems like an acceptable definition of R^{*}; but it's first-order. So what is going on?

Here is the answer. When I said above that no sentence R^*ab "says" that a is an R-ancestor of b, what I meant was that there is no sentence that says this in *every interpretation*. A little more exactly:

> There is no first-order sentence R^*ab, such that in any interpretation I, R^*ab is true in I if and only if, where r is the set of ordered pairs that is denoted by R in I, there is a finite chain of members of the domain of I, pairwise connected by r, leading from the denotation of a to the denotation of b.

The set-theoretic sentence above does succeed in saying that a is an R-ancestor of b *in set-theoretic interpretations*—that is (roughly) in interpretations in which the domain contains a suitable number of sets, and in which '$\in$' denotes set-membership. But there are plenty of non-set-theoretic interpretations in which the set-theoretic sentence doesn't express the ancestral, for instance interpretations in which '$\in$' means something that has nothing to do with sets.

Thus there is a sense in which the usual semantics for second-order logic "stacks the deck" in its favor. It is hard-wired into that semantics that the second-order quantifier $\forall F$ ranges over subsets of the domain, and that second-order predications Fx express set membership. We simply don't consider second-order interpretations in which Fx means anything different from "the denotation of x is a member of the set denoted by F", whereas we do consider second-order interpretations in which '$x \in y$' does not mean 'the denotation of x is a member of the denotation of y'—the reason, in the latter case, is that $\in$ is not treated as a logical constant. It's *not* hard-wired into the semantics for first-order logic that the nonlogical constant $\in$ expresses set-membership, or that the domain contains the sets it would need for the set-theoretic definition to work. We will return to this sort of issue later.

3.6 Metamathematics

There are further important differences between first- and second-order logic, which emerge when we consider the differences between first- and second-order formulations of mathematical theories.

3.6.1 Skolem's paradox

Call a *model* of a set of sentences, an interpretation in which every sentence in the set is true. In first-order logic, the following holds:[9]

Löwenheim-Skolem theorem If a set of sentences, Γ, has a model, it has a model whose domain is at most countably infinite.

Thus, for example, no matter what first-order axioms we write down for set theory, say, or for the theory of real numbers, provided those axioms are satisfiable at all, there is some interpretation whose domain has no more elements than the natural numbers, but in which all the axioms are true.

But how can that be? After all, you can prove in set theory, for example, that there exist sets that are larger than the set of natural numbers.

This is called "Skolem's paradox". Really it isn't a paradox at all. You can prove in set theory a sentence in the language of set theory—containing the predicate '$\in$'—that says, *given its intended interpretation*, that there are sets larger than the set of natural numbers. The interpretations that the Löwenheim-Skolem theorem tells us about, in which the domain is the size of the natural numbers, aren't the intended interpretation. '$\in$' doesn't mean set-membership in such interpretations, and the domain won't contain all the sets.[10]

Skolem's paradox does bring out something important, however, about first-order logic: that sets of first-order sentences can't pin down the intended interpretation of their vocabulary solely by virtue of logic. Take any set of sentences about the real numbers. If it has an intended model, in which the domain is the set of real numbers, it must also have another nonisomorphic model in which the domain has the size of the natural numbers.

Second-order logic is different in this respect. The Löwenheim-Skolem theorem doesn't hold for second-order logic. (But note again that it is just hardwired into the standard definition of a second-order interpretation that, e.g., "monadic predication means set-membership".)

3.6.2 Nonstandard models of arithmetic

First some terminology.

[9] This is just one of a number of "Löwenheim-Skolem" theorems.

[10] Indeed, in *no* interpretation can the domain contain all the sets.

First-order language of arithmetic: the first-order language with symbols $\mathtt{0}$, $'$, $+$, and $\cdot$.

Second-order language of arithmetic: the second-order language with those symbols.

Standard interpretation: the interpretation whose domain is the set of natural numbers, and in which ‘$\mathtt{0}$’ denotes the number $\mathtt{0}$, ‘$'$’ denotes the successor (or add-one) function, ‘$+$’ denotes the addition function, and ‘$\cdot$’ denotes the multiplication function.

So: you can write things like these this in the first-order language of arithmetic: ‘$\mathtt{0}=\mathtt{0}$’, ‘$\mathtt{0}'+\mathtt{0}'=\mathtt{0}''$’, ‘$\exists x\, x\cdot\mathtt{0}''=\mathtt{0}''''$’, and so on. And in the standard interpretation, they mean, respectively, that the number $\mathtt{0}$ is self-identical, that $1+1=2$, and that there is some number that when multiplied by 2 yields the number 4; and they are all true in that interpretation.

The second-order language of arithmetic is the same, but allows predicate variables in addition to individual variables. Thus ‘$\exists F\, F\mathtt{0}$’ (“the number zero has at least one property”) is a formula in the second-order language of arithmetic, though not the first-order.

OK. Here is a question. Is there any set of sentences in the first-order language of arithmetic whose *only* model is the standard interpretation?

The answer to this is trivially no. Suppose the standard interpretation is a model of some set of arithmetic sentences, Γ. We can construct a new interpretation in which all the members of Γ are also true, by “swapping” the numbers 2 and $\mathtt{0}$: let ‘$\mathtt{0}$’ denote the number 2 in the new interpretation, let ‘$'$’ denote the function that maps 2 to 1, 1 to $\mathtt{0}$, $\mathtt{0}$ to 3, and so on; and similarly for ‘$+$’ and ‘$\cdot$’. For that matter, we could construct an interpretation just like the standard interpretation but in which Julius Caesar is swapped in for $\mathtt{0}$, both in the domain and in the interpretations of $\mathtt{0}$, $'$, $+$, and $\cdot$; and the members of Γ will all be true in this interpretation as well.

In general, exactly the same sentences are true in “isomorphic” models: models that have the same pattern, but in which the identities of the objects have been altered.

OK, so the standard interpretation can’t be a unique model of any set of sentences. But let’s ask a further question: is there any set of sentences whose models are all *isomorphic* to the standard interpretation?

More surprisingly, the answer to this question is also no.[11] Let Γ be any set of sentences that are all true in the standard interpretation. In fact, let Γ be the set of *all* such sentences. In addition to having the standard interpretation as a model, Γ has what are called “nonstandard models”, which are models that are structured as follows. They consist in part of a copy of the standard model: there is the model’s “zero”—that is, the member of the domain that is denoted by ‘0’. Then there is the model’s “one”—that is, the member of the domain that is the successor of the model’s zero (that is: the member of the domain that results by applying the model’s denotation of ‘′’ to the model’s denotation of ‘0’). And so on. But in addition to these “standard numbers”, the model also contains infinitely many (and densely ordered) other series of numbers, each of which is structured like the negative and positive integers: discrete, but without beginning or end.

$$\underbrace{\mathtt{0},1,2,\ldots}_{\text{standard numbers}} \qquad \ldots \qquad \underbrace{\ldots a_{-1},a_0,a_1,\ldots}_{\text{some nonstandard numbers}} \qquad \ldots \qquad \underbrace{\ldots b_{-1},b_0,b_1,\ldots}_{\text{more nonstandard numbers}} \quad \ldots$$

Thus we again have an example of the expressive weakness of the language of first-order logic. Using first-order sentences in the language of arithmetic, there is no way to “force” models to look like the standard interpretation.

The situation is very different with second-order logic. There is a second-order sentence in the language of arithmetic, all of whose models are isomorphic to the standard interpretation. (That’s not to say that this sentence gives us an axiomatic basis for arithmetic: since the completeness theorem fails for second-order logic, there is no axiomatic method we can use to draw out the consequences of this sentence.)

3.6.3 Schematic and nonschematic axiomatizations

There is a further difference between first- and second-order logic in the formulation of mathematical theories that will be very important going forward.

[11] There is actually a further reason for the answer being no: because of the “upward” Löwenheim-Skolem theorem, if the standard interpretation is a model of Γ, Γ must also have models in which the domain has cardinality greater than that of the natural numbers. The nonstandard models discussed in the text have domains of the same size as the natural numbers.

Let's start with arithmetic. Consider the project of giving axioms for arithmetic: axioms whose consequences (perhaps proof-theoretic consequences, perhaps semantic consequences—we'll confront this question later) are some, perhaps all, of the truths of arithmetic. How will we do this?

Here are some of the axioms we will want:

$$\begin{array}{l}
\forall x \forall y (x' = y' \to x = y) \\
\forall x\, \mathtt{0} \neq x' \\
\forall x (x \neq \mathtt{0} \to \exists y\, x = y') \\
\forall x\, x + \mathtt{0} = x \\
\forall x \forall y\, x + y' = (x + y)' \\
\forall x\, x \cdot \mathtt{0} = \mathtt{0} \\
\forall x \forall y\, x \cdot y' = (x \cdot y) + x
\end{array}$$

Never mind the details (though these should look like intuitively correct and very basic statements about arithmetic). What's important is that these axioms are not enough. A further key principle about arithmetic is the principle of *induction*. This says, roughly, that if i) the number $\mathtt{0}$ has a certain property, and ii) whenever a number has that property, so does that number's successor, then: every number has the property.

But how to state the principle of induction? You do it differently depending on whether your language is first- or second-order:

$$\forall F\Big(\big(F\mathtt{0} \land \forall x(Fx \to Fx')\big) \to \forall x Fx\Big)$$

(second order induction *principle*)

$$\Big(A(\mathtt{0}) \land \forall x\big(A(x) \to A(x')\big)\Big) \to \forall x A(x)$$ (first order induction *schema*)

Thus in second-order logic, you just add one further axiom: the second-order induction principle is a single formula, that actually fits the gloss above—it begins with a quantifier $\forall F$—"for every property…". But in first-order logic, you can't say "for every property". Rather, what you do is state an induction axiom *schema* and say that every instance of this schema is an axiom. In this schema, A is some formula with some variable v free; $A(\mathtt{0})$ is the result of changing free vs in A to $\mathtt{0}$s, $A(x)$ is the result of changing free vs in A to xs, etc.[12]

So in first-order logic, if we want to state principles of induction, we can't do it with one sentence; we need infinitely many. However, even those infinitely many sentences don't really capture the full principle of induction. What they in effect say, taken together, is that induction holds for every *statable* property—that is, every property of natural numbers corresponding to some formula A in the first-order language of arithmetic. But there are many more properties of natural numbers than that (there are only as many formulas as there are natural numbers, but there are as many properties of natural numbers as there are real numbers). So there is a sense in which we can't really state the principle of induction in a first-order language.

(This fact is closely connected to the existence of nonstandard models. It is because we can include nothing stronger than the instances of the induction schema that we get nonstandard models of first-order arithmetic. Including the second-order induction principle rules them out.)

As with many of these differences between first- and second-order logic, there are philosophical questions about the significance of this distinction involving schemas. It isn't as if one avoids the need for schemas by adopting second-order logic. Even though second-order axiomatizations of arithmetic and set theory don't require schemas, one still needs schemas in the axioms of second-order logic itself, such as the comprehension schema. These latter logical axioms are essential to the use of the second-order theories of arithmetic and set theory. For instance, one conclusion we should be able to draw from any induction principle is that "if $\mathtt{0}$ is either even or odd, and if whenever n is even or odd, $n+1$ is also either even or odd, then every number is either even or odd". Now, the second-order principle of induction says that for any property, if $\mathtt{0}$ has it and if $n+1$ has it whenever n has it, then every number has that property. But in order

[12] In the terminology I was using earlier, the schema can be written this way:

$$\big(A_{v\mapsto\mathtt{0}} \wedge \forall x(A_{v\mapsto x} \rightarrow A_{v\mapsto x'})\big) \rightarrow \forall x A_{v\mapsto x}$$

to apply this principle to get the desired result, we need to know that there is a property of being either even or odd; that's what the comprehension schema tells us. So one must tread carefully with arguments that second-order logic's ability to avoid schemas in certain axiomatizations constitutes an advantageous sort of expressive power.

4. Paradoxes

4.1 Abstract mathematics and set-theoretic foundations

In the nineteenth century, mathematics underwent a dramatic transformation, moving toward an "abstract" approach. Traditionally, mathematics had been associated with relatively "concrete" matters: numbers were for counting or measuring; geometry was about physical points and lines; etc. But in the nineteenth century, mathematicians moved toward a conception of mathematics in which one studies the properties of arbitrary mathematical *structures*, regardless of their association with anything concrete.

In very abstract mathematics, one must be wary of relying on intuitions. In order to investigate what is true in an arbitrary structure, one must avoid smuggling in assumptions that seem intuitively correct but are not made explicit in the assumptions about the structure. This requires sophistication about *logic*, since what is needed is the ability to draw only the logical consequences from the assumptions laid down about the structure. Not coincidentally, it was in the late nineteenth and early twentieth century when modern logic was developed.

Modern logic was also key to a second development around the same time: a surge of interest in the foundations of mathematics. (This surge naturally accompanied the increase in abstraction, for it is natural to wonder what we ultimately doing, when we are studying arbitrary "structures".)

A second key tool in the foundations of mathematics was set theory (which was also discovered around the same time). An arbitrary "structure" was understood to be a certain sort of *set*. Thus on the set-theoretic point of view, what we are ultimately doing, when investigating mathematical structures, is investigating the properties of various kinds of sets.

What is a set? Well, it's something that contains other things; those other things are its *members*. We name a set by enclosing a list of its members within set braces, { and }. Thus the set containing Ted Sider and Barack Obama would be named this way:

$$\{\text{Ted Sider}, \text{Barack Obama}\}$$

But sets can also have infinitely many members, and we may not be able to list the members. In such a case, in order to name the set we might give the conditions under which something is in that set. If E is the set of all and only the even natural numbers, we might write this:

$$x \in E \text{ if and only if } x \text{ is an even natural number}$$

Alternatively, we might write:

$$E = \big\{x | x \text{ is an even natural number}\big\}$$

(In general, "$\big\{x|\phi(x)\big\}$" means: "the set whose members are exactly those things, x, such that $\phi(x)$".)

In addition to containing things like people, sets can also contain other sets. For instance, the set $\big\{\{\text{Ted Sider}, \text{Barack Obama}\}\big\}$ has just one member. (Note the two set braces on each side.) That one member, $\{\text{Ted Sider}, \text{Barack Obama}\}$, is itself a set, the set whose members are Ted Sider and Barack Obama.

There is also such a thing as the *null set*, the set with no members at all: $\varnothing$.

Sets have members, but not in any particular order. Thus

$$\{\text{Ted Sider}, \text{Barack Obama}\} = \{\text{Barack Obama}, \text{Ted Sider}\}$$

But sometimes it is helpful to talk about set-like things that do have an order. These are called *ordered sets*. We write them using angle-braces $\langle$ and $\rangle$ instead of the set-braces $\{$ and $\}$ for unordered sets. Since the order of the members of an ordered set is significant, reordering the members results in a different ordered set:

$$\langle\text{Ted Sider}, \text{Barack Obama}\rangle \neq \langle\text{Barack Obama}, \text{Ted Sider}\rangle$$

How could all of mathematics be regarded as just being about sets? The reason is that we can *construct* any mathematical objects as sets. For example, the natural numbers. We can regard the number 0 as just being the null set, the number 1 as just being the set containing the null set, the number 2 as just being the set containing those previous two sets, and so on:

$$0 \Rightarrow \varnothing \qquad 1 \Rightarrow \{\varnothing\} \qquad 2 \Rightarrow \big\{\varnothing, \{\varnothing\}\big\} \qquad \ldots$$

We can regard a rational number as just being the ordered pair of its numerator and denominator:[13]

$$\frac{1}{2} \Rightarrow \langle 1,2\rangle$$

Ordered pairs can in turn be defined as sets:

$$\langle x,y\rangle \Rightarrow \big\{\{x\},\{x,y\}\big\}$$

We can regard real numbers as just being certain sets of rational numbers (e.g., “Dedekind cuts”). We can regard complex numbers as being ordered pairs of real numbers:

$$a+bi \Rightarrow \langle a,b\rangle$$

And so on.[14]

Here is one more reason for the importance of sets. The development of calculus in the 17th century was a major step forward for mathematics. But as developed by Newton and Leibniz, the foundations of calculus were profoundly unclear (cf. Berkeley’s famous critique). Gradually the foundations of calculus became clearer (as concepts like limits and epsilon-delta definitions were developed). As this happened, it became clearer that *functions* were an important part of the story. (What a derivative is, is a derivative of a function.) At first, functions were just regarded as being formulas in languages, but later it became clear that they must rather be regarded as “arbitrary mappings”. But what are those? With set theory we have an answer: a function is just a set of ordered pairs, in which no two ordered pairs in the set share the same first member.

4.2 Russell’s paradox

So sets increasingly became important in the foundations of mathematics. At first, mathematicians weren’t very clear about what they were. Some didn’t clearly distinguish sets from formulas picking them out. Many thought of sets as being completely unproblematic, or (relatedly) as just being part of logic.

But gradually it became clear that set theory itself is a substantial part of mathematics.

[13] Really it’s better to regard rational numbers as equivalence classes of such ordered pairs.

[14] See a textbook on set theory, e.g., Enderton (1977), for a discussion of these constructions.

There were many reasons for this, including the discovery by Cantor of different sizes of infinity (which we'll discuss in a bit), and the discovery of the need for the axiom of choice. But most worrisome was the discovery of certain paradoxes. The kinds of assumptions mathematicians had been making about sets, it turned out, were contradictory!

The simplest of these paradoxes (though not the first to be discovered) was Bertrand Russell's. The following assumption had been made (sometimes implicitly):

Naïve comprehension for any "condition", there exists a corresponding set—a set of all and only those things that satisfy the condition

We use this principle implicitly when we speak of, for example, "the set of all even natural numbers": we form the condition "even natural number" and infer from Naïve comprehension that there exists a set of all and only those things that satisfy that condition.

But Russell then said: well, what about the condition "is a set that is not a member of itself"? Naïve comprehension implies that there exists a set *R* (the "Russell set") containing all and only the sets that aren't members of themselves. Thus:

i) Any set that is *not* a member of itself *is* a member of *R*

ii) Any set that *is* a member of itself is *not* a member of *R*

But now, Russell asked, is *R* a member of itself?

The claim that *R* *isn't* a member of itself leads to a contradiction: if *R* isn't a member of itself, then by i), *R* would be a member of *R*, and so it would be a member of itself after all—contradiction.

But the claim that *R* *is* a member of itself also leads to a contradiction: if *R* is a member of itself, then by ii) it wouldn't be a member of *R*, and so wouldn't be a member of itself—contradiction.

So each possibility is ruled out: that *R* is a member of itself and that *R* is not a member of itself. But one of those possibilities has to be correct—either *R* is a member of itself or it isn't. Thus the principle of Naïve comprehension has let to a contradiction.

Russell's argument can be put more simply. Naïve comprehension implies that there exists a set, R, such that:

$$\text{for all } z, z \in R \text{ iff } z \notin z$$

But this implies, instantiating the variable z to R:

$$R \in R \text{ iff } R \notin R$$

which is a contradiction.

So the principle of Naïve comprehension isn't true. But that principle seems to be at the core of the very idea of a set. The very idea of a set, the main idea in the foundations of mathematics, seems to be contradictory!

4.3 Axiomatic set theory and ZF

In the decades after the paradoxes were discovered, mathematicians gradually found ways to develop a consistent approach to set theory, which avoids Russell's and other paradoxes. The approach that is now standard in mathematics is called "Zermelo-Frankel", or "ZF" set theory.

The ZF method for avoiding Russell's paradox is to reject the principle of Naïve comprehension. However, we can't just stop talking about sets of things satisfying various conditions we specify—the ability to do that is what makes sets so useful. So in place of Naïve comprehension, ZF includes new principles telling us that various sets exist. These new principles will be unlike Naïve comprehension in being consistent, but like Naïve comprehension in implying the existence of all the sorts of sets we need in the foundations of mathematics.

The main principles of ZF are these:

Extensionality Sets with the same members are identical

Null set There exists a set $\varnothing$ containing no members

Pairing For any sets a and b, there exists a set $\{a, b\}$ containing just a and b

Unions For any sets a and b, there exists a set $a \cup b$ containing all and only those things that are members of a or members of b (or both)

Infinity There exists a set, A, that i) contains the null set, and i) is such that for any $a \in A$, $a \cup \{a\}$ is also a member of A. (Any such set A must be infinite, since it contains all of these sets: $\varnothing, \{\varnothing\}, \{\{\varnothing\}, \varnothing\}, \ldots$.)

Power set For any set, A, there exists a set containing all and only the subsets of A (this is called A's "power set")

Separation Suppose some set x exists, and let $\mathscr{C}$ be any condition. Then there exists a set y consisting of all and only the members of x that satisfy $\mathscr{C}$.

Here is the rough idea of how they avoid Russell's paradox. Instead of a naïve comprehension principle telling us that certain sets exist, we have a two-step process telling us that certain sets exist. First, we have "expansion" principles which assert the existence of certain kinds of sets: Null set, Pairing, Unions, Infinity, and Power set.[15] Second, we have the principle of Separation, which lets us "contract" the sets that we obtain from the expansion principles.

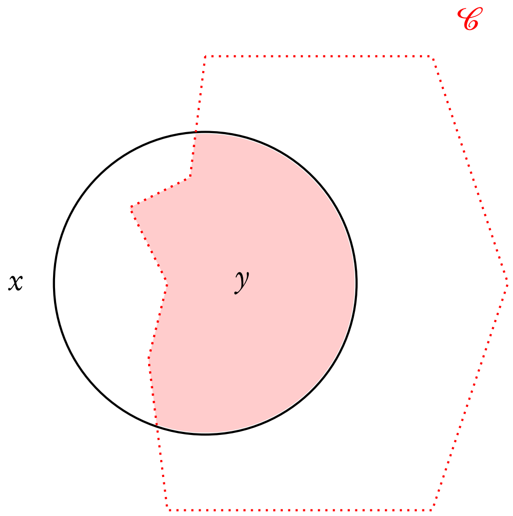
Separation is like the principle of Naïve comprehension, in that it assures us that there exist sets corresponding to various conditions; but it does this in a way that doesn't lead to contradiction. It says, roughly:

Separation Suppose some set x exists, and let $\mathscr{C}$ be any condition. Then there exists a set y consisting of all and only the members of x that satisfy $\mathscr{C}$.

So the principle of separation doesn't quite say that there exists a set corre-

[15] Actually there is also a stronger schema, called "Replacement".

sponding to each condition, since to apply it, we need to already know that a certain set x exists (say, by proving its existence using the expansion axioms). We can then use any chosen condition to pick out a subset of the given set x:



Here is why Separation doesn't lead to Russell's paradox. Suppose you start with a set, x. You can then use the principle of separation, choose the condition "is not a member of itself", and conclude that there exists a *subset* of x, call it y, that contains all and only the non-self-membered members of x:

$$\text{For all } z : z \in y \text{ iff } z \in x \text{ and } z \notin z$$

But this doesn't lead to a contradiction. It does imply this:

$$y \in y \text{ iff } y \in x \text{ and } y \notin y$$

But now we can consistently suppose that $y \notin y$. There is no need to suppose[16] that $y \in x$, so we can't use the right-to-left-hand direction of this biconditional to infer $y \in y$.

Suppose there existed a *universal set*—a set, U, that contains *every* set. We could then use the principle of separation to pick out the subset, R, of U containing all and only the nonself-membered members of U:

$$\text{For all } z : z \in R \text{ iff } z \in U \text{ and } z \notin z$$

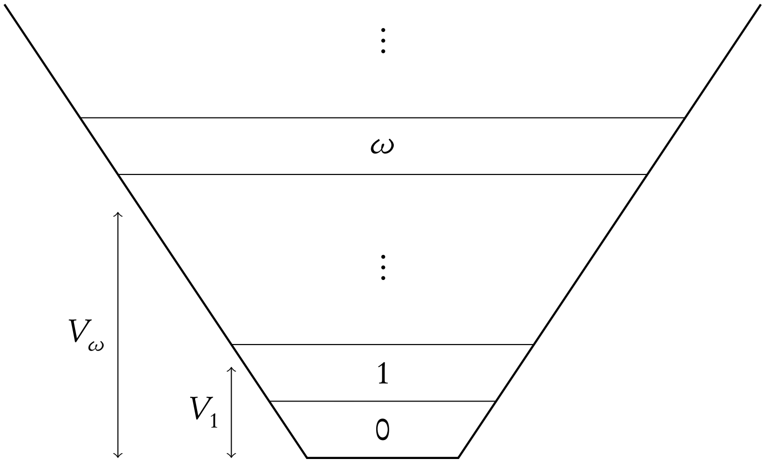
[16] In fact, other principles of ZF imply that $y \notin x$. The axiom of regularity implies that no set is a member of itself. Thus this principle implies that $y = x$ (given y's definition); and then, applying this principle again, we have that $y \notin x$.

But since *every* z is a member of U, this implies:

$$\text{For all } z : z \in R \text{ iff } z \notin z$$

which is Russell's contradiction.

So what we've learned is that in ZF set theory, there does not exist a universal set. What, then, is the picture of sets according to ZF? It is that of an "iterative hierarchy":



V_0 is the set of urelements

You first start with a set of "urelements"—a set of things that don't have members. (In "pure" ZF, you don't have urelements, and so the bottom of the diagram is "pointy"; in applied mathematics, nonmathematical objects will be the urelements.) Then you form a new level of sets, V_1, which adds to V_0 all the sets of urelements. Then you form a new level, V_2, which adds all sets you can form from V_1. And (in a certain sense) you keep going infinitely.[17]

One final thing about ZF. My statement above of the principle of separation wasn't quite right. An axiom of set theory must only talk about things like sets; it can't talk about things like "conditions". In fact, you state Separation differently in second- and first-order logic:

[17] See Boolos (1971).

$$\forall x \exists y \forall z \big(z \in y \leftrightarrow (z \in x \wedge A)\big) \quad \text{(first-order separation \textit{schema})}$$

$$\forall x \forall X \exists y \forall z \big(z \in y \leftrightarrow (z \in x \wedge Xz)\big)$$

(second-order separation *principle*)

Thus in *first-order* ZF—that is, ZF set theory stated using first-order logic—there is no single axiom of separation. Rather, there is an axiom schema. ("A" is a schematic variable; whenever you replace 'A' with a formula in the language of set theory, the result is an axiom.) But in *second-order* set theory, there is a single axiom of separation—instead of the schematic variable A, there is a universally quantified second-order variable F.

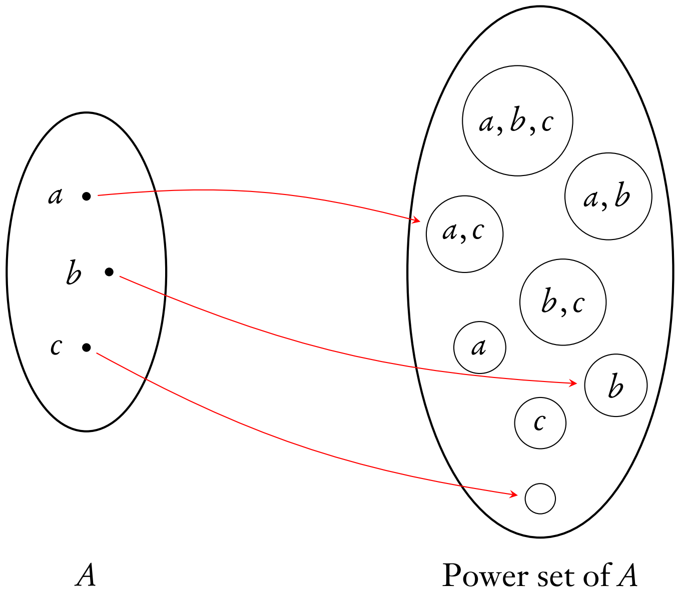
Thus the situation is parallel to the situation with the principle of induction. In first-order set theory, we need infinitely many separation axioms; but even then, as a group they say, in effect, merely that a set has subsets corresponding to properties we can formulate in our language. This is much weaker than the second-order separation axiom, which says that the set has subsets corresponding to every property, not just those properties we can formulate. (As a consequence of this weakness, first-order set theory has all sorts of unwanted models.)

4.4 Other paradoxes, other solutions

Philosophers are generally aware that sets are entities that are "susceptible to paradoxes", that one needs to be cautious when reasoning about them, that natural-seeming principles about sets (such as the principle of naïve comprehension) need to be restricted in some way, and so on. But many other kinds of entities are just as susceptible to paradoxes, and this isn't always appreciated.

For example, propositions and properties face paradoxes similar to those confronting sets. Consider, for example, the property of *being a non-self-exemplifying property*: it seems that it exemplifies itself if and only if it does not exemplify itself—a contradiction. The problem even arises for predicates: the predicate 'heterological', meaning *is a predicate that does not apply to itself*, seems to apply to itself if and only if it does not apply to itself—again a contradiction.

Here is another paradox confronting properties and propositions, which demonstrates the need for caution in reasoning about such entities. First a bit of background. The mathematician Georg Cantor famously showed that the power set of a set is always larger than that set, in the sense that if you take each member of the set and pair it with a unique member of the powerset, some members of the powerset will be left over, not paired with any member of the set. (See the diagram. In the diagram, A is a finite set; but Cantor's argument applies to all sets, even infinite ones.)



Here is the argument. Let f be any function that maps every member of a set, A, to some subset of A. We'll show that some subset of A isn't in the range of this function—the function doesn't map any member of A to that subset. (In the diagram, this means showing that there is a circle in the right-hand-side of the diagram to which no red arrow is pointing.)

Let's begin by noting that some members of A have the following feature: they are members of the sets to which they are mapped by f. For example, in the diagram, a and b have the feature, since a is mapped to $\{a, c\}$ and b is mapped to $\{b\}$. But c does not have the feature, since it is mapped to the empty set.

Let's form the set, D, of members of A that do *not* have the feature. That is, let:

$$\text{for any } x,\ x \in D \text{ iff: } x \in A \text{ and } x \notin f(x) \qquad (*)$$

So in the diagram, $D = \{c\}$.

Now, suppose for reductio that every member of the powerset of A is in the range of f. That is, suppose that for every subset, X, of A, some member of A is mapped to X by f. It follows that some member of A is mapped to D by f. Let's call that member of A "d". Thus $f(d)=D$.

But there can be no such d. (This is true in the diagram: no red arrow is pointing to $\{c\}$; but we will give a general argument that doesn't rely on the diagram.) By (*),

$$d \in D \text{ iff: } d \in A \text{ and } d \notin f(d)$$

But $f(d)=D$, so:

$$d \in D \text{ iff: } d \in A \text{ and } d \notin D$$

But $d \in A$, so:

$$d \in D \text{ iff } d \notin D$$

which is a contradiction. (Note the similarity to Russell's argument. Russell thought of his argument by reflecting on Cantor's.)

Back, now, to propositions. Consider any set, S of propositions. It would be natural to assume that there exists a unique proposition that *some god is currently entertaining all and only the propositions that are members of S*. But that can't be: it would mean that there are at least as many propositions as there are sets of propositions. But how could there fail to be such a proposition? Or how could it fail to be unique? The argument seems to show that our ordinary ways of thinking about propositions lead to contradictions.[18]

Back to set theory and Russell's paradox. We discussed the ZF solution, but it isn't the only one. One might, for instance, give up classical logic, and say that $R \in R \leftrightarrow R \notin R$ isn't a contradiction. Others have developed other set theories, in which there is a universal set (e.g., Quine's "New foundations" system). There is also the "indefinite extensibility" tradition. On this view, there is something wrong with the idea that one can quantify over absolutely everything. One can accept the principle of naïve comprehension, and so accept a set of all and only the non-self-members. That is, given the current range of our quantifiers, we can then introduce a set that includes all and only the members in that range which are not self-members. However, in doing so we

[18] See Kaplan (1994). We'll also discuss the Russell-Myhill paradox in more detail later, which is in this vicinity.

have expanded the domain of quantification, since that set we just introduced wasn't in the original domain.[19]

And finally, and more important for our purposes, there are attempts to solve paradoxes by broadly "syntactic" means: by adopting languages in which attempts to raise the paradox are not grammatical. As an example, take second-order logic. In second-order logic, we can think of the predicate variables as standing for properties, but we can't even grammatically raise the question of whether one of those properties applies to itself. The reason is that the way you attribute properties in second-order logic is by predicating: e.g., you say that object a has some property by saying $\exists F Fa$. So the way to say that some property has itself would seem to be this: $\exists F F F$. But that's not grammatical: the syntax of second-order logic says that if F is a one-place predicate, it needs to combine with a *term* to form a sentence; it can't combine with another predicate. So you can't even grammatically formulate anything like Russell's paradox here.

We'll look into this more when we develop a stronger higher-order logic.

5. Higher-order logic and λ-abstraction

In the following sections we'll develop a very expressively powerful language—the language in which higher-order metaphysics takes place.

5.1 Third-order logic and beyond

Let's consider languages that are even more expressively powerful than that of second-order logic.

In second-order logic, we have expressions which attach to terms in order to form sentences; these are predicates (whether predicate variables or predicate constants). However, we don't have any predicates of predicates: expressions that attach to predicates in order to form sentences.

We could add them—we could modify the definition of a formula as follows:

> In addition to the ordinary kind of predicates (both constant and variable), there are some "predicates of predicates": $\boldsymbol{F}, \boldsymbol{G}, \ldots$. For any predicate of predicates, $\boldsymbol{F}$, and any one-place ordinary predicate G, "$\boldsymbol{F}G$" is a formula.

[19] See Fine (2007).

And we could modify the semantics in the obvious way:

> In any interpretation, the denotation of a predicate of predicates is a set of sets of members of the domain.
>
> The formula "$\boldsymbol{F}G$" is true in an interpretation if and only if the denotation of the ordinary predicate G is a member of the denotation of the predicate of predicates $\boldsymbol{F}$.

This might be useful, for example, in symbolizing a sentence like:

> Sally and John have exactly the same virtues
>
> $\forall X(\boldsymbol{V}X \to (Xs \leftrightarrow Xj))$

(Although notice that the syntax of the English sentence 'Sally and John have exactly the same virtues' seems more like a first-order sentence quantifying over properties and employing a predicate 'has' for instantiation.)

Once we have predicates of predicates, it is natural to introduce quantifiers binding variables in that syntactic position, so that in addition to saying "Sally and John have exactly the same virtues" as above, we can also express a sort of existential generalization of that sentence:

> There is some type of property such that Sally and John have exactly the same properties of that type
>
> $\exists \boldsymbol{Y}\forall X(\boldsymbol{Y}X \to (Xs \leftrightarrow Xj))$

This is third-order logic.

And we could keep iterating, introducing predicates that can apply to predicates of predicates, and variables for such predicates, resulting in fourth-order logic, even higher-order predicates, and so on. Also we could allow these higher-order predicates to be multi-place. But there is a further sort of syntactic generalization that we will explore.

5.2 Higher-order logic and types

So far we have discussed languages in which one can quantify into predicate position, the position of predicates of predicates, and so on. I want next to discuss an even more expressive sort of language, in which one can quantify into many more grammatical positions. In a sense, quantification can be into *any* grammatical position, in a broad sense of grammatical position.

5.2.1 More about syntax

To introduce this language, let's think a little more about grammar/syntax in general.

Syntax is about well-formedness—about what kinds of expressions "make sense". Syntactic rules tell you what kinds of expressions combine with what other kinds of expressions to form still other kinds of expressions. We have seen a few such rules, such as:

> If you take a one-place predicate, F, and attach it to a term, t, the result Ft is a formula
>
> If you take the expression $\sim$, and attach it to a formula, A, the result $\sim A$ is a formula

There is a common pattern here:

> If you take an expression of category X, and attach it to an expression of category Y, the result is an expression of category Z

For the first rule, X was *one-place predicate*, Y was *term*, and Z was *formula*; for the second rule, X was *the negation sign*, Y was *formula*, and Z was *formula*.

This makes it natural to ask whether it would make sense to introduce new syntactic categories other than those we have been examining so far. For example, in each of the two rules above, the "output" of the rule (the expression of category Z) was a formula. But could we have a rule letting us form complex expressions, in which the output (Z) was, say, the category *one-place predicate*?

We can. We can, for instance, introduce a syntactic category of *predicate functors*, which combine with one-place predicates to form new one-place predicates. The syntactic rule governing predicate functors would be this:

> If you take a predicate functor $\boldsymbol{q}$, and attach it to a one-place predicate, F, the result $\boldsymbol{q}F$ is a one-place predicate

Here X is *predicate functor*, Y is *one-place predicate*, and Z is *one-place predicate*. Natural language has something like predicate functors, namely adverbs:

'quickly' can combine with the predicate 'runs' to form the predicate 'runs quickly'.

Once we have seen syntactic categories and rules in this light, a possibility for generalizing them suggests itself. For *any* syntactic categories Y and Z, we ought to be able to introduce a new category, X, with this feature: expressions of category X combine with expressions of category Y to form expressions of category Z. Since this process can be iterated, there will be infinitely many syntactic categories. Moreover, in all categories, we will allow quantified variables of that category. The result is a certain sort of "higher-order logic", based on the "theory of types".[20]

5.2.2 Types

If we are going to develop a language with infinitely many syntactic categories, we need a way of talking about syntactic categories as entities, so that we can make generalizations about all syntactic categories ("for any syntactic categories, a and b, ...".) Syntactic categories, thought of as entities, are called *types*.

Let's first get a handle on what these types are supposed to do for us. Each one is an entity representing a syntactic category. Thus we will speak of expressions in our language of higher-order logic as being *of* types: an expression is *of* type a if and only if that expression falls under the grammatical category that the type a represents. For instance, as we'll see in the next paragraph, the type that represents the grammatical category of *formulas* is: t. Thus expressions that are formulas, such as $Fa \wedge Gb$ or $\exists x F x$, are expressions *of* type t. (What *is* the entity t? It doesn't matter; think of it as being the letter 't' if you like. What matters about the types is what they represent, not what they are.)

OK, let's define the infinite class of types:

[20] The theory of types we'll be discussing derives from Alonzo Church (1940), who was improving the theory of types in Russell and Whitehead's *Principia Mathematica*. There are a number of different kinds of systems that go by the name of "type theory"; this is just one. For a quick peek at some of the alternatives, see section 5.4.

There are two undefined types, e and t

> (e is the type of singular terms; t is the type of formulas)

For any types, $a_1,\dots,a_n$ and b, there is another type $\langle a_1,\dots,a_n,b\rangle$

> ($\langle a_1,\dots,a_n,b\rangle$ is the type of expressions that combine with expressions of types $a_1,\dots,a_n$ to form an expression of type b)

The undefined types e and t are so-named because they are the types of expressions that name *e*ntities and can be *t*rue or false, respectively. Thus individual variables $(x,y,\dots)$ and names $(a,b,\dots)$ are of type e; and formulas $(Fx,\forall x(Fx\to Gx),\dots)$ are of type t. As for complex types $\langle a_1,\dots,a_n,b\rangle$, an expression E that is of this type is the sort of expression that can combine with n expressions, of types $a_1,\dots,a_n$, respectively, to form an expression of type b. Thus such an expression E is an "n-place" expression: in order to make a syntactically well-formed result, E needs to combine with n other expressions, of the appropriate types. The "appropriate types" are $a_1,\dots,a_n$—the types of the expressions with which E can combine. And b is the type of the resulting expression—the type of the expression that results when E is combined with those n expressions of appropriate types. One might think of E as a kind of function, mapping expressions to expressions: $a_1,\dots,a_n$ are the types of its "inputs", and b is the type of its "output".

In the simplest case where $n=1$, the type $\langle a,b\rangle$ is that of an expression which can combine with an expression of type a to make an expression of type b. In the terminology of the previous section, X is $\langle a,b\rangle$, Y is a, and Z is b.

Let's look at some examples.

Example 1: the type $\langle e,t\rangle$. Expressions of this type combine with something of type e to make an expression of type t. That is: they combine with terms to make formulas. Thus they are one-place predicates! (That is, they are one-place predicates of individuals.)

Example 2: the type $\langle e,e,t\rangle$. Expressions of this type combine with *two* expressions of type e to make an expression of type t. That is, they combine with pairs of terms to make formulas. Thus they are two-place predicates (of individuals). Similarly, a three place-predicate is of type $\langle e,e,e,t\rangle$, and so on.

Example 3: the type $\langle t, t\rangle$. Expressions of this type combine with a formula to make a formula. An example is the symbol for negation, $\sim$. If you combine it with a formula, A, the result $\sim A$ is another formula. (The operators $\Box$ and $\Diamond$ from modal logic are further examples.) Expressions of type $\langle t, t\rangle$ are thus one-place sentence operators.

Example 4: the type $\langle t, t, t\rangle$. Expressions of this type combine with two formulas to make a formula. That is, they are two-place sentence operators. Examples include the symbol for conjunction, $\wedge$, the symbol for disjunction, $\vee$, and the conditional symbol $\rightarrow$. (Another example is the counterfactual conditional symbol $\Box\!\!\rightarrow$ in counterfactual logic.)

Example 5: the type $\langle\langle e, t\rangle, t\rangle$. Expressions of this type combine with an expression of type $\langle e, t\rangle$ to make an expression of type t. That is, they combine with predicates to make formulas. Thus they are what we called higher-order predicates in section 5.1.[21] For example, we might represent 'is patient' as an ordinary predicate P, and 'is a virtue' as a higher-order predicate V, and then represent (with a little violation to English syntax) 'patience is a virtue' as "VP".

[21] There is a confusing terminological issue I'd like to mention. Recall the "higher-order" predicates of section 5.1: predicates of predicates, predicates of predicates of predicates, and so on. What numerical orders should be assigned to such predicates? Different authors use different terminology here, and there is no happy choice. On one hand, it's natural to call predicates of predicates *second-order* predicates, since they're "one jump up" from ordinary predicates. On the other hand, if we called them second-order predicates, we would expect them to have the same syntactic type as the variables that we call "second-order variables"; but what are usually called second-order variables are variables of type $\langle e, t\rangle$—the "predicate variables" of section 3.1. (After all, the latter are the variables that are distinctive of second-order logic.) Relatedly, if predicate constants of type $\langle\langle e, t\rangle, t\rangle$ are "second-order", then presumably predicate constants of individuals (such as P for 'is patient' or 'F' for 'is a fox') should be called "first-order". But that means we're calling an expression of type $\langle e, t\rangle$ "first order" if it's a constant and "second-order" if it's a variable, which is unfortunate.

The problem is that there are two distinct sorts of "jumps" that we feel inclined to call an increase in "order". On one hand, there is the move from a language in which you can't quantify into (ordinary) predicate position to a language in which you can. The first sort of language is standardly called "first-order logic" and the second is standardly called "second-order logic". On the other hand, there is the jump from a language in which you only have predicates of individuals—that is, expresions of syntactic type $\langle e, t\rangle$ (whether constant or variable)—to a language in which you have predicates of predicates—that is, expressions of syntactic type $\langle\langle e, t\rangle, t\rangle$ (whether constant or variable). The latter also feels like a jump in order, which explains the impulse to call predicates of predicates "second-order". But it's a different sort of jump than the first jump; and the two aren't "in sync".

Example 6: the type $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$. Expressions of this type combine with expressions of type $\langle e,t\rangle$ and make expressions of type $\langle e,t\rangle$. That is, they combine with one-place predicates to form one-place predicates. Adverbs! (Or: predicate functors.) The adverb ‘quickly’ combines with the one-place predicate ‘runs’ to form the one-place predicate ‘runs quickly’.

All this, then, inspires the following syntax for the language of our higher-order logic.

1. For each type, there are infinitely many variables which are expressions of that type, and there may also be zero or more constants which are expressions of that type. (Any chosen set of constants is called a “signature”, and determines a language.)
2. $\sim$ is a constant expression of type $\langle t,t\rangle$; $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ are constant expressions of type $\langle t,t,t\rangle$
3. If $E_1,\dots,E_n$ are expressions of types $a_1,\dots,a_n$, and E is an expression of type $\langle a_1,\dots,a_n,b\rangle$, then $E(E_1,\dots,E_n)$ is an expression of type b
4. If v is a variable of type a and E is an expression of type t then $\forall vE$ and $\exists vE$ are expressions of type t

Notice, by the way, a difference between this syntax and, e.g., the syntax for first-order logic in section 2.1. In that earlier syntax, we didn’t assign $\sim$ any syntactic category (since the only syntactic categories mentioned there were *term* and *formula*, and $\sim$ isn’t either of those). Rather, we simply gave a rule that specifies the syntactic role of $\sim$—a rule saying how $\sim$ combines with an expression of an appropriate syntactic category to yield another expression of a certain syntactic category. That rule was the following: “if A is a formula then so is $\sim A$”. But here, we don’t have any such rule for $\sim$, or for any of the other propositional connectives. Why not? Because we have instead explicitly assigned types to them. $\sim$, for example, is said in clause 2 to be of type $\langle t,t\rangle$: it combines with one expression of type t to form an expression of type t. As a result, clause 3 covers the syntax of $\sim$ and the other propositional connectives (along with covering the syntax of variables and nonlogical constants). For instance, since $\sim$ is of type $\langle t,t\rangle$, it follows from clause 3 that for any expression

E of type t (i.e., any formula), $\sim(E)$ is an expression of type t.[22]

5.2.3 Meaning: Frege and functions

What do all these expressions of different types *mean*?

For the types that have corresponding expressions in natural language, this is comparatively easy. For example, since we are already familiar with predicates/verb phrases (like 'runs') and adverbs/predicate functors (like 'quickly'), we already have a handle on what sorts of meanings are possessed by expressions of types $\langle e, t\rangle$ and $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$. But what about expressions of much more complex types, for which there are no natural-language counterparts?

In such cases, it's natural to think of the meaning of an expression in terms of what it *does*—in terms of how the meaning of that expression interacts with the meanings of other expressions with which it is combined, to generate the meanings of more complex expressions. And there is a systematic way of thinking about meaning in this way: Frege's (1997*b*).

Let's begin by thinking about meaning informally. (Later on we'll consider a Frege-inspired formal semantics for higher-order logic.) For Frege, in one sense of meaning, the meaning of a proper name (like 'Ted') is an individual (me), and the meaning of a sentence (like 'Ted is a philosopher') is a truth value, of which there are two: T (or 1, or "The True"); and F (or 0, or "The False").

Now, Frege (1952/1892) famously pointed out that expressions that mean the same, in this sense of meaning, can mean different things in another sense of meaning. 'Ted is a philosopher' and 'Barack Obama was born in Hawaii' are both true sentences, and so each denote the same thing, namely T (The True); nevertheless there is clearly some sense of meaning in which they have different meanings. Similarly, '$2+3$' and '$1+4$' both denote the number 5, but again, they have different meanings in some sense. Frege dealt with this by distinguishing one kind of meaning, called *Bedeutung*, or reference, or as I will say, denotation, from another sort of meaning called *Sinn*, or sense (which is roughly the rule by which the denotation is determined). Here I will consider only denotation, and ignore sense. So I can rephrase the claims about meaning from the previous paragraph: the *denotation* of a name is an individual, and the denotation of a sentence is a truth value.

[22] Note that we are still treating quantifiers in the old way. But they too can be assigned types once we have introduced λ-abstraction; see section 5.4.3.

(It's admittedly somewhat awkward to think of sentences as denoting their truth values, particularly if, when we get to quantification into sentence position, we think of $\forall P$ (variable of type t) as meaning "for all propositions…". But let's stick with Frege's original approach for now, for simplicity.)

So: names denote individuals and sentences denote truth values. What about predicates and connectives? For Frege they denote *functions*.

The notion of a function is one that we've mentioned a few times before. It's a familiar notion from mathematics. A function is a rule that yields an output if you give it appropriate inputs. For example, $f(x) = x^2 + 4$ is a function that yields the output 4 if you give it the input 0, and yields the output 8 if you give it the input 2. Another way of putting this: the function f "maps 0 to 4" and "maps 2 to 8". Now, f is a "one-place" function, since it requires one input in order to yield an output. Other functions have more than one place. For example, the addition function, which we might represent as $g(x,y) = x + y$, maps two numbers to a single output: it maps, e.g., $1,2$ to 3, and maps $4,7$ to 11. The examples so far have been functions of numbers, but functions can have objects of any sort as inputs and outputs. The biological mother of function, for example, is a function from persons to persons, which maps each person to his or her biological mother.[23]

According to Frege, predicates denote functions from *individuals to truth values*. For example, the one-place predicate 'runs', for Frege, denotes the function that maps any object to T if it runs and to F if it does not run. That is, 'runs' denotes this function:

$$r(x) = \begin{cases} T \text{ if } x \text{ runs} \\ F \text{ if } x \text{ does not run} \end{cases}$$

[23] In section 4.1 we mentioned the set-theoretic definition of a function, according to which a function is a set of ordered pairs. But in some contexts one might prefer some other conception of what functions are. i) According to the set-theoretic definition, any "two" functions that assign the same values to all arguments are in fact identical. For instance, the "add one" set-theoretic function on natural numbers is the very same as the "add two and then subtract one" set-theoretic function. One might prefer a more "fine-grained" conception of function in certain circumstances. ii) According to the set-theoretic definition, functions are sets. Thus, given ZF, there cannot exist functions defined on absolutely all entities (including all sets). Consider, for example, the function that Frege would call the denotation of the predicate '$\in$' of set-membership: the two-place function that maps any a and any set b to T if and only if $a \in b$. Given ZF there is no such set-theoretic function.

Similarly, the two-place predicate ‘bites’ denotes this function:

$$b(x,y)=\begin{cases} T \text{ if } x \text{ bites } y \\ F \text{ if } x \text{ does not bite } y \end{cases}$$

In general, n-place predicates for Frege denote n-place functions from entities to truth values.

As for connectives, they denote functions from *truth values to truth values*. For instance, the one-place connective ‘not’ denotes this function, n:

$$n(T)=F$$
$$n(F)=T$$

This function “reverses” truth values, mapping truth to falsity and falsity to truth; it is the *negation* function. Similarly, according to Frege, the two-place connectives ‘and’ and ‘or’ denote the *conjunction* and *disjunction* functions c and d:

$$\begin{array}{ll} c(T,T)=T & d(T,T)=T \\ c(T,F)=F & d(T,F)=T \\ c(F,T)=F & d(F,T)=T \\ c(F,F)=F & d(F,F)=F \end{array}$$

These should be familiar from introductory logic; Frege’s idea is that the functions that truth tables depict are the meanings of the connectives.

5.2.4 Formal semantics for higher-order logic

In this section I’ll give a partial sketch of a formal semantics for our language of higher-order logic inspired by Frege’s functional approach to denotation.[24]

As in first-order logic, an interpretation will consist in part of a domain—a nonempty set. And as before, an interpretation assigns denotations to nonlogical expressions. In the case of names—constant expressions of type e, these denotations will be exactly what they were earlier: members of the domain. But

[24] The definition sketched here is not the only possible one. There are alternatives, for instance alternatives in the spirit of Henkin interpretations for second-order logic (section 3.5.1).

for expressions of other types, denotations will be different sorts of entities—albeit always entities that are "based" on the domain.

First let's give a general statement of the kind of entity denoted by expressions of any type in a given interpretation. For each type, a, we'll specify what kind of object counts as an "a denotation" (or: "denotation of type a")—the kind of thing that can be denoted by an expression of type a.[25] The definition first specifies the kinds of denotations for the undefined types, and then it gives a rule covering complex types:

Kinds of denotations, for a given domain D

e denotations are members of D

t denotations are truth values (i.e., T, F)

An $\langle a_1,\dots,a_n,b\rangle$ denotation is an n-place function that maps an a_1 denotation, an a_2 denotation, …, and an a_n denotation, to a b denotation

Consider, for example, the type $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$—the type of predicate functors. Given the above definition, an $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$ denotation for a domain D is a function that maps $\langle e,t\rangle$ denotations to $\langle e,t\rangle$ denotations. And $\langle e,t\rangle$ denotations are themselves functions—functions from D to truth values. Thus an $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$ denotation is a function that maps any function from D to truth values to another function that maps D to truth values.

Recall from the end of section 5.2.2 that in our syntax for higher order logic, the logical constants $\wedge$, $\vee$, etc. were treated as being expressions of certain types.

Now we can give a proper definition of an interpretation:

[25] In some contexts, types are thought of as being, in the first instance, types of semantic values, rather than as types of expressions, as I have been thinking of them.

Definition of interpretation for higher-order logic

An interpretation consists of a domain, D, and a specification, for each nonlogical expression of type a, of an a denotation for D.

Additionally, any interpretation assigns to the propositional connectives as denotations their Fregean truth-functions. (For example, the denotation of $\sim$ in any interpretation is the one-place truth function mapping T to F and F to T.)

So, for example, if G is some nonlogical predicate (type $\langle e, t\rangle$), then any interpretation must assign to G some particular function from its domain to truth values.

Next we give a general rule for computing the denotations of complex expressions based on the denotations of their parts:

Denotations of complex expressions (in a given interpretation)

If $E_1, \dots, E_n$ are expressions of types $a_1, \dots, a_n$, with denotations $d_1, \dots, d_n$, and expression E has type $\langle a_1, \dots, a_n, b\rangle$ and denotes a function f, then the denotation of the complex expression $E(E_1, \dots, E_n)$ (which has type b) is: $f(d_1, \dots, d_n)$

Thus denotations of complexes are derived by applying functions to arguments: the function that is the "outer" expressions's denotation is applied to the arguments that are the "inner" expressions' denotations. Indeed, the attraction of the Fregean functional approach to denotation is that it leads to such a simple rule.

To see all this in action, let's work through an example. Consider an interpretation in which the following formula symbolizes "Ted doesn't run quickly":

$$\sim \boldsymbol{q}(R)(c)$$

Thus c is a name symbolizing "Ted", R is a one-place predicate symbolizing "runs", and $\boldsymbol{q}$ is a predicate functor symbolizing "quickly". Thus the denotations of these expressions will be of the following sorts ('fx' abbreviates 'function'):

	type	kind of denotation	particular denotation
c	e	member of the domain	Ted
R	$\langle e,t\rangle$	fx from entities to truth values	the fx r that maps o to T iff o runs
$\boldsymbol{q}$	$\langle\langle e,t\rangle,\langle e,t\rangle\rangle$	fx from $\langle e,t\rangle$ fxs to $\langle e,t\rangle$ fxs	the fx q that maps g to the "g-ing quickly" fx
$\boldsymbol{\sim}$	$\langle t,t\rangle$	fx from TVs to TVs	the fx n that maps T to F and F to T

Now let's figure out what the denotation—i.e., truth value—of the entire sentence $\boldsymbol{\sim q}(R)(c)$ is. To make this more readable, let's write "$|E|$" for "the denotation of expression E". Thus, what we're trying to figure out is what $|\boldsymbol{\sim q}(R)(c)|$ is. We'll start by writing down the denotations of simple expressions, and then work up to the denotations of more complex expressions.

$$\begin{aligned}
|R| &= r && \text{(see table)}\\
|\boldsymbol{q}| &= q && \text{(see table)}\\
|\boldsymbol{q}(R)| &= q(r) && \text{(rule for denotations of complexes)}\\
|c| &= \text{Ted} && \text{(see table)}\\
|\boldsymbol{q}(R)(c)| &= q(r)(\text{Ted}) && \text{(rule for denotations of complexes)}\\
|\boldsymbol{\sim}| &= n && \text{(see table)}\\
|\boldsymbol{\sim q}(R)(c)| &= n(q(r)(\text{Ted})) && \text{(rule for denotations of complexes)}
\end{aligned}$$

Thus $\boldsymbol{\sim q}(R)(c)$ denotes $n(q(r)(\text{Ted}))$. But what is $n(q(r)(\text{Ted}))$? Well, $q(r)$ is the function that maps any member of the domain to T iff it runs quickly. (This is so because q is the function that maps a function from the domain to truth values to the corresponding "quickly function" from members of the domain to truth values.) In fact, I do run quickly. (Very quickly.) So $q(r)(\text{Ted}) = T$. And so, $n(q(r)(\text{Ted})) = F$. Our sentence denotes The False.

In a full presentation of the semantics, we would need to give a definition of the denotation for all expressions of all types, including expressions containing variables (see next section) and connectives. I won't do this here, since my goal has been to say just enough to give the intuitive feel for meaning in higher-order logic. However, it's important to note that there are philosophical questions about how formal, set-theoretic semantics of the type sketched here relates to intended meaning. Such questions are especially pressing for higher-order logic. For instance, under one common interpretation of the language of higher-order logic, it is common to assume that a comprehension principle holds, so that (for example) the sentence $\exists F\forall x(Fx \longleftrightarrow x = x)$ is true. But then, if the quantifier $\forall x$ includes all sets in its range, it would seem that the quantifier $\exists F$ cannot range over sets, since a set of all self-identical sets would be a set of all sets whatsoever, which does not exist given ZF set theory. So: take the semantics sketched in this section as being a sort of model (in the informal, philosophy-of-science sense) of one possible intended meaning of the language of higher-order logic—a model which may not be fully accurate but may nevertheless be useful for certain purposes.

5.2.5 Variables

Back to syntax. Consider the symbolization of "Ted is sitting and Ted is eating":

$$Sc \wedge Ec$$

In first-order logic we can generalize into subject position:

$$\exists x(Sx \wedge Ex)$$

And in second-order logic we can generalize into predicate position:

$$\exists X(Xc \wedge Ec)$$

So why not also allow generalization into the 'and' position:

$$\exists \bigcirc (Sc \bigcirc Ec)$$

to mean, roughly, that there is some relation between propositions that holds between the proposition that Ted is sitting and the proposition that Ted is eating? '$\bigcirc$' here would be a variable with the same type as '$\wedge$', namely: $\langle t, t, t\rangle$.

In higher-order logic, we do allow this. More generally, since we are allowing constants of any of the infinitely many types, we will also allow *variables* of each of those types, together with quantifiers binding them.

For example, we will have a variable $\triangle$ of the same type as $\sim$, namely $\langle t, t\rangle$, so that we can write:

$$\exists \triangle \, \triangle Sc$$

meaning, roughly, that there is some property of propositions had by the proposition that Ted is sitting. ('$\triangle$' and '$\bigcirc$' and other symbols I'm using here aren't standard; I'll talk later about how we write variables of different types.)

Similarly, suppose $\boldsymbol{q}$ is a constant predicate functor (type $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$), representing the adverb ‘quickly’, so that we can symbolize ‘Ted runs quickly’ as:

$$\boldsymbol{q}(R)(c)$$

Then we can also write:

$$\exists \boldsymbol{x}\, \boldsymbol{x}(R)(c)$$

where $\boldsymbol{x}$ is a variable of type $\langle\langle e,t\rangle\langle e,t\rangle\rangle$—a “predicate functor variable”.

We can even have variables of type t, i.e., the type of formulas. If P is such a variable, then “$\exists P\, P$” means roughly that there is some true proposition.

5.2.6 Typing symbols

We need some way of writing symbols, whether variables or constants, that indicates their types, and our ad hoc method for doing this so far (x, X, $\triangle$, $\boldsymbol{x}$, P) won’t work in general. There are infinitely many types (for any types $a_1,\dots,a_n,b$, there is a new type $\langle a_1,\dots,a_n,b\rangle$), so we can’t go on picking a different font or symbol set for each type—we’ll run out.

Instead, in official contexts anyway, let’s write all variables as “x”, and all constants as “c”, and indicate the type of the variable or constant by including that type as a superscript on the symbol. So for any type, a, x^a will be a variable of type a, and c^a will be a constant of type a. We can get as many variables and constants of any type as we like by using subscripts, e.g.: $x^a_1, x^a_2, \dots$.

Unofficially, however, we’ll continue to use the old ad-hoc methods, to improve readability. Thus we’ll write:

official	unofficial	
x^e	$x,y,\dots$	individual variables
c^e	$c,d,\dots$	names
x^t	$P,Q\dots$	sentence variables
$x^{\langle e,t\rangle}, x^{\langle e,e,t\rangle}$	$X,F,R\dots$	predicate variables
$x^{\langle t,t,t\rangle}, x^{\langle t,t\rangle}$	$\triangle,\bigcirc$	sentence-operator variables
$c^{\langle\langle e,t\rangle,t\rangle}$	$\boldsymbol{F},\boldsymbol{G},\dots$	higher-order predicate constants
$x^{\langle\langle e,t\rangle,t\rangle}$	$\boldsymbol{X},\boldsymbol{Y},\dots$	higher-order predicate variables
$c^{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$	$\boldsymbol{q},\boldsymbol{r},\dots$	predicate-functor constants
$x^{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$	$\boldsymbol{x},\boldsymbol{y},\dots$	predicate-functor variables

5.3 λ-abstraction

5.3.1 Complex predicates

In standard logic, the symbol for conjunction, $\wedge$, is a sentential connective: the only place that it can grammatically occur is between two formulas: $A \wedge B$.

The natural-language word 'and' is more flexible. In addition to functioning as a sentential connective, as in 'Ted is sitting and Ted is eating' (and in addition to constructing plural terms such as 'Daisy and Luke'—recall section 2), it can also occur in *complex predicates*, as in the sentence 'Ted is sitting and eating'. Here 'is sitting and eating' is, grammatically, a predicate (it attaches to 'Ted' to form a sentence); but it is a complex predicate, since it is made up of two other predicates, 'sitting' and 'eating', which are combined with 'and'.[26]

So: natural languages (some, anyway) have complex predicates. Logical languages can be constructed which also have complex predicates. Under a common way of doing so, a complex predicate like 'is sitting and eating' is represented using a new symbol, λ ("lambda"), as follows:

$$\lambda x(Sx \wedge Ex)$$

(Or "$\lambda x.(Sx \wedge Ex)$", "$\lambda x[Sx \wedge Ex]$", or "$(\lambda x.Sx \wedge Ex)$"—different authors use different notation.) It can be read in any of the following ways:

> "is an x such that Sx and Ex"
>
> "is such that: it is sitting and it is eating"
>
> "is sitting and eating"

$\lambda x(Sx \wedge Ex)$ is a one-place complex predicate; but we can also allow complex predicates with more than one place. For example, if B is a two-place predicate meaning "bites", then the following is a two-place predicate meaning "bites or is bitten by":

$$\lambda xy(Bxy \vee Byx)$$

In general, the syntax of λ is:

[26] When I'm teaching introductory logic students how to symbolize natural language sentences like 'Ted is sitting and eating', I tell them that this sentence "means the same as" or "is short for" the sentence 'Ted is sitting and Ted is eating', and hence can be symbolized as $Sc \wedge Ec$. These claims are perhaps correct in some sense, but not if taken as claims about English syntax. In 'Ted is sitting and eating', 'is sitting and eating' is a syntactic unit.

Where $x_1,\ldots,x_n$ are any variables and A is any formula, $\lambda x_1\ldots x_n A$ is an n-place predicate

$\lambda x_1\ldots x_n A$ can be read as meaning "are $x_1,\ldots,x_n$ such that A".

Expressions formed with λ are often called "λ abstracts", and the process by which they are formed is called "λ abstraction".

How are λ abstracts used in sentences? Just like other predicates: you attach them to terms to form sentences. For example, if c is a name referring to me, then the following is a sentence symbolizing "Ted is such that he is sitting and eating":

$$\lambda x(Sx \wedge Ex)c$$

(I'll sometimes add parentheses to sentences involving λ-abstracts to improve readability; e.g.: $\lambda x(Sx \wedge Ex)(c)$.)

It's important to be clear about the syntax of λ expressions, so let me re-emphasize it: $\lambda x A$ is a *predicate*. Thus we should not think of '$\lambda x(Sx \wedge Ex)$' as symbolizing 'the property of sitting and eating'. It rather symbolizes 'is sitting and eating'.

What is the difference? 'The property of sitting and eating' is a *term*, whose function is to name something, whereas 'is sitting and eating' is a *predicate*, whose function is to describe something. In order to construct a sentence using the former, you need to combine it with a predicate. For instance, you can use the two-place predicate 'instantiates' to say:

> Ted instantiates the property of sitting and eating

You *can't* combine it with a term to form a sentence:

> # Ted the property of sitting and eating

That's ungrammatical—terms can't combine with terms to form sentences. But you *can* combine the *predicate* 'is sitting and eating' with a term to form a sentence:

> Ted is sitting and eating

What might a formal semantics for λ abstracts look like? Well, they will have meanings just like predicates. Let's continue with the Fregean approach from section 5.2.4, in which (first-order) predicates denote functions from the domain to truth values. Then λvA, also being a predicate, also denotes a function from the domain to truth values: the function that maps any member of the domain, o, to T if and only if the formula A is true of o.

For example, consider an interpretation in which S represents "sitting" and E represents "eating" (i.e., S denotes the function mapping any o in the domain to T if and only if it is sitting, and E denotes the function mapping any o in the domain to T if and only if it is eating). Then $\lambda x(Sx \wedge Ex)$ denotes the function that maps any o in the domain to T if and only if '$Sx \wedge Ex$' is true of o—that is, if and only if o is sitting and o is eating. So if c names me in that interpretation, then the sentence $\lambda x(Sx{\wedge}Ex)c$ is true if and only if Ted is sitting and Ted is eating—i.e., if and only if the sentence $Sc \wedge Ec$ is true.

Similarly, let B be a two-place predicate representing "bites" in some interpretation. Then $\lambda x\exists yByx$ is a predicate symbolizing "being an x such that something bites x", and denotes the function that maps o to T if and only if something bites o; and so, the sentence $\lambda x\exists yByx(c)$ is true if and only if something bites Ted—i.e., if and only if the sentence $\exists yByc$ is true.

Note, then, that $\lambda x(Sx \wedge Ex)c$ is, in a sense, just a long-winded way of saying $Sc \wedge Ec$, and $\lambda x\exists yByx(c)$ is just a long-winded way of saying $\exists yByc$. (Similarly, "Ted is such that he is: sitting-and-eating" is just a long-winded way of saying "Ted is sitting and Ted is eating", and "Ted is such that: something bites him" is just a long-winded way of saying "Something bites Ted".) This fact helps a lot in getting the feel for λ abstraction: it's helpful to think of $\lambda xA(c)$ as just meaning $A_{x\mapsto c}$ (i.e., the result of substituting c in for all the free xs in A), and similarly for multi-place complex predicates. Thus we can think of $\lambda x(Sx \wedge Ex)c$ as just meaning $Sc \wedge Ec$, $\lambda x\exists yByx(c)$ as just meaning $\exists yByc$, $\lambda xyBxy(cd)$ as just meaning Bcd, and so on.

If λ just lets us say things more long-windedly, what's the point? We'll think about this more later, but the point is to introduce single syntactic units with complex meanings. Without λ, you can't formulate a *single predicate* which means "is sitting and eating".

5.3.2 Generalizing λ: syntax

Let's think about what the previous section accomplished. In standard predicate logic, all predicates are simple: predicates like $F, G, R, \ldots$. But once we have λ, we can construct complex predicates. Thus for the syntactic category of *predicate*, λ allows us to construct complex expressions of that category.

Once we move from standard predicate logic to higher-order logic, with its very general notion of a syntactic category (type), it is natural to introduce a corresponding generalization of λ abstraction, which lets us construct complex expressions of *any* category.

Let's think in more detail about the syntax of the λ abstracts we introduced in the previous section. An example was:

$$\lambda x(Sx \wedge Ex)$$

There are three important syntactic elements here. (1) The variable attached to λ, namely x. Call this the "abstraction variable". This is an *individual variable*—type e. (2) The expression coming after λx, namely $Sx \wedge Ex$. Call this the "abstraction matrix". This is a *formula*—type t. (3) The entire λ abstract, namely $\lambda x(Sx \wedge Ex)$. This is a *predicate*—type $\langle e, t\rangle$. So: our old λ abstracts were formed by attaching λ to an abstraction variable of type e, and then appending an abstraction matrix of type t, resulting in a λ abstract of type $\langle e, t\rangle$.

To generalize this for higher-order logic: (1) we let the abstraction variables be of any types, $a_1, \ldots, a_n$; (2) we let the abstraction matrix also be of any type, b; and then (3) the resulting λ abstract has type $\langle a_1, \ldots, a_n, b\rangle$. To summarize:

Syntax for λ generalized

For any variables, $x_1^{a_1}, \ldots, x_n^{a_n}$, of types $a_1, \ldots, a_n$, and any expression, E, of type b, the expression

$$\lambda x_1^{a_1} \ldots x_n^{a_n}\, E$$

is of type $\langle a_1, \ldots, a_n, b\rangle$

(Notice how the complex predicates of section 5.3.1 are special cases of this general rule.)

5.3.3 Generalizing λ: semantics

What do lambda abstracts of arbitrary type mean?

As we saw in section 5.3.1, a λ abstract λxA of type $\langle e,t\rangle$ means "is an x such that A". We can give similar glosses in some other cases. For instance, we can think of λXA as meaning "is a property, X, such that A", and λPA as meaning "is a proposition, P, such that A". (In the previous sentence, the first λ abstract was type $\langle\langle e,t\rangle,t\rangle$, and the second was type $\langle t,t\rangle$.)

But this only gets us so far. The natural language gloss "is a … such that…" is apt only when the abstraction matrix is a formula (type t), since what comes after "such that" needs to be the kind of thing that can be true or false. We still need an explanation of the meanings of λ abstracts whose matrices are not of type t (i.e., λ abstracts of type $\langle a_1,\dots,a_n,b\rangle$ where $b\neq t$).

Sometimes the best strategy for understanding expressions of complex types is *not* trying to find natural-language glosses for them (since such glosses might not exist), but rather trying to understand what they *do*. By this I mean: understanding how such expressions help determine the meanings of larger expressions of which they are parts.

To understand this for λ abstracts, let's return to a lesson from the end of section 5.3.1. If you attach $\lambda x(Sx\wedge Ex)$ to the name c, the result:

$$\lambda x(Sx\wedge Ex)c$$

is (I said) just a long-winded way of saying:

$$Sc\wedge Ec$$

In general, $\lambda xA(c)$ means the same as $A_{x\mapsto c}$ (i.e., what you get if you start with A and change all the free xs to cs), and similarly for multi-place complex predicates. So what $\lambda x(Sx\wedge Ex)$ *does* is this: when attached to a name, c, it results in a sentence meaning $Sc\wedge Ec$.

The process of changing, e.g., $\lambda x(Sx\wedge Ex)c$ to $Sc\wedge Ec$ is called "β conversion" or "β reduction". In general it works as follows.[27]

[27] Important qualification: no free variables in $A_1,\dots A_n$ may be "captured" by quantifiers in $E_{x_1^{a_1}\mapsto A_1,\dots,x_n^{a_n}\mapsto A_n}$. To illustrate the importance of this restriction, consider $\lambda x\exists yBxy$, a complex predicate meaning "bites someone". Without the restriction, $\lambda x\exists yBxy(y)$ would reduce by β

β conversion

The result of applying the λ abstract $\lambda x_1^{a_1} \dots x_n^{a_n} E$ to expressions $A_1, \dots, A_n$ (of types $a_1, \dots, a_n$), namely:

$$\lambda x_1^{a_1} \dots x_n^{a_n} E(A_1, \dots, A_n)$$

reduces by β conversion to:

$$E_{x_1^{a_1} \mapsto A_1, \dots, x_n^{a_n} \mapsto A_n}$$

Actually it's contentious that the result of β conversion means the same as the original. But clearly there is some very close semantic relationship between the two; and in any case thinking of them as meaning the same is a useful heuristic for grasping the meanings of λ abstracts.

β conversion can give us an intuitive handle on the meanings of other λ abstracts. Take another example, $\lambda X(Xc \vee Xd)$ (where c and d are names and X is a type $\langle e, t\rangle$ variable). In this case we do have a natural-language gloss: "being a property that is had either by c or by d". But we can also think about its meaning via what it does. If you attach it to a one-place predicate F, you get this sentence:

$$\lambda X(Xc \vee Xd)F$$

which means the same thing as (via β conversion):

$$Fc \vee Fd$$

Thus what $\lambda X(Xc \vee Xd)$ does is this: it converts any predicate, F, into a sentence meaning that c is F or d is F.

For more practice, let's consider some further examples.

conversion to $\exists y Byy$. But the former means "y is such that it bites someone" (with y a free variable) whereas the latter means "something bites itself".

There are other kinds of conversions that also result in expressions that are equivalent (in some sense). "α conversion" is what is often called "relettering of bound variables" in logic: $\lambda x Fx$ α-converts to $\lambda y Fy$. "η conversion" is less familiar: $\lambda x Fx$ η-converts to F.

Example 1:

$$\lambda P\, P$$

(P is a variable of type t.) Since both the abstraction variable and the abstraction matrix are of type t, the entire λ abstract has type $\langle t, t\rangle$. That is, it combines with a formula to make a formula. So it's a one-place sentence operator (like $\sim$).

What does it mean? Since the matrix is of type t, we can gloss it using "such that": it means "is a proposition, P, such that P". Or, one might say, "is a true proposition" (though really the concept of truth is not involved).

But let's also think about what it does. If you attach it to a formula, A, you get this formula:

$$\lambda P\, P(A)$$

which reduces by β conversion to:

$$A$$

Thus what $\lambda P\, P$ does is attaches to a sentence A to form a sentence that means A. So it is a redundant sentence operator.

Example 2:

$$\lambda x\left(\boldsymbol{q}(R)(x) \wedge \boldsymbol{g}(R)(x)\right)$$

where x is a variable of type e, $\boldsymbol{q}$ and $\boldsymbol{g}$ are predicate functor constants (type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$, and R is a one-place predicate constant (type $\langle e, t\rangle$). Since x is type e, and $\boldsymbol{q}(R)(x) \wedge \boldsymbol{g}(R)(x)$ is type t, the entire λ abstract is type $\langle e, t\rangle$—it's a one-place predicate of individuals. But what does it mean?

Since the matrix expression is of type t, we can gloss the λ abstract using "such that". It means: "is an entity, x, such that $\boldsymbol{q}(R)(x)$ and $\boldsymbol{g}(R)(x)$". Suppose we think of $\boldsymbol{q}$ as meaning "quickly", $\boldsymbol{g}$ as meaning "gracefully", and R as meaning "runs". Then $\boldsymbol{q}(R)$ is a one-place predicate meaning "runs quickly", and $\boldsymbol{g}(R)$ is a one place predicate meaning "runs gracefully"; and so, the entire λ abstract means "is an x such that x runs quickly and x runs gracefully". In other words, it is a complex one-place predicate meaning "runs quickly and gracefully".

We can reach the same conclusion by thinking about what the λ abstract does. If you attach it to a name, c, symbolizing "Ted", say, you get:

$$\lambda x\left(\boldsymbol{q}(R)(x) \wedge \boldsymbol{g}(R)(x)\right)(c)$$

which reduces by β conversion to:

$$\boldsymbol{q}(R)(c) \wedge \boldsymbol{g}(R)(c)$$

which means that Ted runs quickly and runs gracefully. That is exactly what the complex predicate ‘runs quickly and gracefully’ does.

Example 3:

$$\lambda Y\, \lambda x \big(\boldsymbol{q}(Y)(x) \wedge \boldsymbol{g}(Y)(x)\big)$$

This one is a little more complicated. Here the matrix variable Y is of type $\langle e, t\rangle$, and the matrix expression is itself a λ abstract:

$$\lambda x \big(\boldsymbol{q}(Y)(x) \wedge \boldsymbol{g}(Y)(x)\big)$$

This “inner” λ abstract is of type $\langle e, t\rangle$, since its abstraction variable x is type t and its matrix $\boldsymbol{q}(Y)(x) \wedge \boldsymbol{g}(Y)(x)$ is type t. Thus the type of the “outer” λ abstract (i.e., the entire original λ abstract $\lambda Y\, \lambda x \big(\boldsymbol{q}(Y)(x) \wedge \boldsymbol{g}(Y)(x)\big)$) is $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$. It’s a predicate functor.

We can’t gloss the outer λ abstract using “such that” since its matrix isn’t of type t. We’ll need to think instead about what it does. If you attach it to a one-place predicate, R, you get:

$$\lambda Y\, \lambda x \big(\boldsymbol{q}(Y)(x) \wedge \boldsymbol{g}(Y)(x)\big)(R)$$

which β-reduces to:

$$\lambda x \big(\boldsymbol{q}(R)(x) \wedge \boldsymbol{g}(R)(x)\big)$$

This is just Example 2. If we think of R as meaning “runs”, then this is a predicate meaning “runs quickly and gracefully”. So, here is what the outer λ abstract did: it converted a predicate meaning “runs” into a predicate meaning “runs quickly and gracefully”. Similarly, if attached to a predicate meaning “walks” it will result in a predicate meaning “walks quickly and gracefully”. So what it does in general is convert any predicate G into a predicate meaning “Gs quickly and gracefully”.

We now have a pretty good handle on what the λ abstract does. In many cases, that will be the best we can do. However, in this case, we can supply a gloss: the λ abstract symbolizes this *complex adverb*: “quickly and gracefully”!

Let’s end by examining how the formal semantics of section 5.2.4 can be applied to λ abstracts. In any interpretation, the denotation of a λ abstract of a certain type will just be the same sort of animal that is generally denoted by expressions of that type: a function of the appropriate sort. Which function? The rule is this:

Semantics for λ abstracts

In any interpretation, the λ abstract

$$\lambda x_1^{a_1} \dots x_n^{a_n} E$$

denotes the n-place function that maps any n denotations, of types $a_1, \dots, a_n$, respectively, to the object that E denotes when the variables $x_1^{a_1}, \dots, x_n^{a_n}$ are assigned those denotations

For example, $\lambda x(Cx \wedge Ex)$ denotes the one-place function that maps any denotation of type e—i.e., any member of the domain—to the object that the formula $Cx \wedge Ex$ denotes when x is assigned that individual. But formulas denote truth values, so this amounts to saying that $\lambda x(Cx \wedge Ex)$ denotes the function that maps a member of the domain to T if and only if the formula $Cx \wedge Ex$ is true of that entity—which is exactly what we said in section 5.3.1.

5.4 Alternate systems

I have been presenting one sort of lambda abstraction and higher-order logic, but there are others out there which you may encounter.

5.4.1 Binary-only types and schönfinkelization

In the approach to types that I have introduced, complex types look like this: $\langle a_1, \dots, a_n, b\rangle$. A complex type can have any finite number of "constituent types". But in some other systems, complex types are only allowed to have two constituent types; they must always look like this: $\langle a, b\rangle$. That is, complex types must always be binary.

In fact this is not a substantive restriction, since there is a way of simulating the more complex types using only binary types. Let's illustrate this with a two-place predicate, L, for "loves". In standard predicate logic, you can write things like Lcd, meaning that c loves d. The two-place predicate L attaches to two names to form a sentence. Thus L is of type $\langle e, e, t\rangle$. How would we represent L if we were only allowed to use expressions of binary types?

Here is the trick. (It's called "schönfinkelization" or "schoenfinkelization", after Moses Schönfinkel, or "currying", after Haskell Curry.) We represent L as being of type $\langle e, \langle e, t\rangle\rangle$. Thus it combines with a name or individual variable (type e) to form a *predicate* (type $\langle e, t\rangle$). $L(c)$ therefore is a predicate, meaning *is loved by* c; and we can then attach this predicate to d to form a sentence, $L(c)(d)$, meaning that d is such that it is loved by c.[28]

In a theory allowing only binary types, we would play this same trick with other "multi-place" expressions, such as two-place sentential connectives like $\wedge$, $\vee$, and $\rightarrow$. These are usually treated as having the type $\langle t, t, t\rangle$: they attach to a pair of formulas to make a formula. But in a binary type theory they are treated as having type $\langle t, \langle t, t\rangle\rangle$: they attach to an expression of type t to make an expression of type $\langle t, t\rangle$. That is, they attach to formulas to make one-place sentence operators, which can in turn attach to formulas to make formulas. So, for example, instead of writing $A \wedge B$ (where A and B are formulas), we would instead write $\wedge(A)(B)$.

5.4.2 Relational types

The types we have been discussing are sometimes called "functional" types. There is an alternative system of types, which are often called "relational" types, in which the only types in addition to the types of names and formulas are predicate types, albeit of arbitrarily high level.

To define relational types, you begin with just one primitive type, e. Then, for any types $a_1, \ldots a_n$, there is another type: $(a_1, \ldots, a_n)$.[29] This further type is to be understood as the type of expressions that combine with n expressions, of types $a_1, \ldots, a_n$, respectively, to make a *formula*. The fact that the resulting expression is a formula needn't be explicitly represented in relational type $(a_1, \ldots, a_n)$ (in contrast to the functional type $\langle a_1, \ldots, a_n, b\rangle$, which explicitly represents the type b of the resulting expression), because the resulting expression for

[28] We made an arbitrary choice here, in deciding that $L(c)$ is to mean *is loved by* c. We could have instead decided that it would mean *loves* c. In that case, $L(c)(d)$ would have meant that d loves c. See Heim and Kratzer (1998, section 2.4) on "left-" and "right-" schönfinkelizing.

[29] Relational types are often written with angled rather than rounded brackets: $\langle a_1, \ldots, a_n\rangle$. I'm using a different notation to distinguish them from functional types (which are, by the way, often written as $(a \rightarrow b)$ when they're binary).

relationally typed expressions is *always* a formula; this is hard-wired into the idea of relational types.

For relational types $(a_1,\ldots,a_n)$, the case where $n = 0$ is allowed. That is, there is a type (). This is the type of expressions that combine with zero expressions to make formulas. How can an expression combine with *no* formulas to make a formula? Easy: by already *being* a formula. So what I am saying is that () is the type of expressions that are formulas.

Thus the definition of relational types is as follows:

Definition of relational types

Undefined type: e.

> (The type of singular terms)

For any types $a_1,\ldots a_n$ (where n may be 0), $(a_1,\ldots,a_n)$ is also a type.

> (The type of expressions that combine with n expressions, of types $a_1,\ldots,a_n$, respectively, to make a formula)

There is an easy to miss, but very important, difference between relational and functional types. Expressions of functional type $\langle a,b\rangle$ convert an a into a b, so to speak, whereas expressions of relational type (a,b) convert an a and a b into a formula. The former are one-place expressions whereas the latter are two-place; and the latter are always predicates whereas the former are not unless b happens to be t. For instance, the functional type $\langle e,e\rangle$ is the type of *one-place function symbols*. Although we haven't discussed this type explicitly, we have met one expression of this type: the successor sign $'$ from the language of arithmetic, which combines with a term (such as 0) to form a term ($0'$). But the similar-looking relational type (e,e) is the type of two place (first-order) predicates, such as B for "bites".

Just as we introduced a logical language based on functional types at the end of section 5.2.2, we can also introduce a logical language based on relational types. As before, for each relational type there will be variables and perhaps constants; propositional connectives will be assigned appropriate types (for instance, $\sim$ will have type $(())$ and $\wedge$ will have type $((),())$; and for any variable v of any type and any expression E of type $()$ (i.e., any formula), $\forall vE$ and $\exists vE$ will be expressions of type $()$. But the rule for forming complex expressions will be a little different. The old rule for functionally typed expressions looked like this:

> If $E_1, \ldots, E_n$ are expressions of types $a_1, \ldots, a_n$, and E is an expression of type $\langle a_1, \ldots, a_n, b\rangle$, then $E(E_1, \ldots, E_n)$ is an expression of type b

whereas the new rule for relationally typed expressions looks like this:

> If $E_1, \ldots, E_n$ are expressions of types $a_1, \ldots, a_n$, and E is an expression of type $(a_1, \ldots, a_n)$, then $E(E_1, \ldots, E_n)$ is an expression of type $()$.

This new language is in many ways similar to the language introduced at the end of section 5.2.2, though not perfectly so. Each language has names, formulas, sentential connectives, quantifiers, n-place first-order predicates for each n, n-place second-order predicates for each n, and so on. (Notice that in most cases, these expressions are assigned different-looking types in the two systems. For instance, formulas are of functional type t, but relational type $()$; one-place predicates are of functional type $\langle e, t\rangle$, but relational type (e); binary sentential connectives (like $\wedge$) are of functional type $\langle t, t, t\rangle$, but relational type $((), ())$.) But the functional language has some additional expressions without direct counterparts in the relational language. For example, only the functional language has one-place predicate-functors (i.e., expressions that combine with one-place predicates to form one-place predicates; i.e., expressions of type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$; i.e., adverbs). The relational language lacks such expressions because every complex relational type is that of an expression which, when combined with appropriate other expressions, produces a *formula*, whereas predicate-functors combine with predicates to produce *predicates*.

So there is a sense in which functional types are "syntactically more expressive" than relational types: on the natural way of associating types with syntactic categories, relational syntactic categories are a proper subset of functional syntactic categories. But in a way this is superficial, since there is a sense in which relational types are just as *semantically* expressive as functional types. See Dorr (2016, Appendix 2) for details, but the basic idea is that we can use a trick akin to schönfinkelization to simulate the meanings of, e.g., predicate functors with relationally typed expressions. Given the semantics of section

5.2.4, a predicate functor (type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$) expresses a function from predicate meanings to predicate meanings—i.e., a one-place function from (i) functions-from-entities-to-truth-values, to (ii) functions-from-entities-to-truth-values. But the information encoded in any such function could also be encoded in a function that moves the argument of the function in (ii) into (i)—i.e., by a two place function from (ia) functions-from-entities-to-truth-values and (ib) entities, to (ii) truth values. For example, the functionally typed adverb 'quickly' expresses the function, q, that maps any function g (from entities to truth values) to the "g-ing quickly" function—the function that maps any entity x to T iff x "gs quickly". We can simulate this function with the closely related function q' that maps any g and any entity x to T if and only if x gs quickly. And since this is a function to truth values, it is the kind of meaning that can be expressed by a predicate (if we modify the semantics of section 5.2.4 in the obvious way for relational types), namely a predicate with two arguments, the first of which is a predicate and the second of which is a name. And such predicates do exist in the relationally typed language: they have relational type $((e), e)$.

5.4.3 Quantifiers as higher-order predicates

In first-order logic, quantifiers do two things: bind variables and express quantity. λ abstracts also bind variables, and in fact can take over variable-binding from quantifiers. Here is how.[30]

Why not think of a quantified natural language sentence like:

Something is sitting and eating

as meaning:

Sitting-and-eating has at least one instance

? Here 'sitting-and-eating' is a (complex) predicate, and 'has at least one instance' is a higher-order predicate—a predicate of a predicate. This in turn suggests a symbolization like this:

$$\text{has-at-least-one-instance}\big(\lambda x(Sx \wedge Ex)\big)$$

where 'has-at-least-one-instance' is a predicate applied to the predicate $\lambda x(Sx \wedge Ex)$ representing 'sitting-and-eating'. Only, to save writing, let's write $\exists$ instead

[30] See Stalnaker (1977). This is inspired by Frege, e.g. (1892, p. 187).

of ‘has-at-least-one-instance’, and drop the outer parentheses:

$$\exists\lambda x(Sx \wedge Ex)$$

Compare, now, the standard symbolization:

$$\exists x(Sx \wedge Ex)$$

In the standard symbolization, the quantifier both binds the variable in the matrix formula $Sx \wedge Ex$, and also makes a statement of quantity: that at least one thing satisfies the matrix. In the new symbolization, λ binds the variable, and then all the quantifier $\exists$ has to do is make the statement of quantity: that the predicate formed using the λ has at least one instance.

So our new syntax for the quantifiers is this:

> If Π is a one-place predicate then $\forall\Pi$ and $\exists\Pi$ are formulas

As we’ve seen, instead of writing $\exists x(Sx \wedge Ex)$ in the new syntax we now write $\exists x\, \lambda x(Sx \wedge Ex)$. Similarly, instead of writing $\forall x(Fx \to Gx)$ we write $\forall\lambda x(Fx \to Gx)$. Instead of writing $\exists x Fx$, we write $\exists x\, \lambda x Fx$, or better, just $\exists F$. And instead of writing $\exists x \forall y Rxy$ for “there is someone who respects everyone”, we instead write:

$$\exists\lambda x \forall\lambda y Rxy$$

Let’s think about this last example carefully, working backwards: $\lambda y Rxy$ is a predicate meaning “is a y that x respects”, or more concisely, “is respected by x” (x is free here), so $\forall\lambda y Rxy$ is a sentence meaning “everyone is respected by x”, that is, “x respects everyone” (x is still free), so $\lambda x \forall\lambda y Rxy$ is a predicate (in which x is no longer free) meaning “is an x that respects everyone”, or more concisely, “respects everyone”; and so, finally, $\exists\lambda x \forall\lambda y Rxy$ is a sentence saying that there is someone who respects everyone.

We can play this trick in higher-order logic too. Instead of symbolizing “Ted has at least one property” as:

$$\exists X Xc$$

we can instead symbolize it thus:

$$\exists\lambda X Xc$$

meaning: "the property of *being a property had by Ted* has at least one instance—i.e., is had by at least one property". And instead of symbolizing "every proposition is either true or false" as:

$$\forall P(P \vee {\sim}P)$$

we can instead write:

$$\forall \lambda P(P \vee {\sim}P)$$

meaning: "the property of *being a proposition that is either true or false* is such that every proposition has it".

In general, for any type a, instead of writing:

$$\exists x^a A \qquad\qquad \forall x^a A$$

(where A is a formula) we now write instead:

$$\exists^a \lambda x^a A \qquad\qquad \forall^a \lambda x^a A$$

(In the previous two examples I left out the superscripts on $\exists$ or $\forall$, and we might continue to do so in informal contexts, but the idea here is that they are officially required.[31]) $\exists^a$ and $\forall^a$ are of type $\langle\langle a,t\rangle,t\rangle$: they attach to predicates of type

[31] To require the superscripts is to treat quantifiers of different types as being distinct logical constants—e.g., $\forall^t$ is not the same symbol as $\forall^e$. This isn't mandatory; one could instead continue with just two quantifiers, $\exists$ and $\forall$, and write the semantics so that the truth conditions of sentences with quantifiers depends on the type of the variable to which the quantifiers are attached. But one nice thing about requiring the superscripts is that quantifiers can then be treated "categorematically". A categorematic expression is one that has a meaning in isolation; a "syncategorematic" expression has no meaning in isolation; rather, larger expressions containing it have a meaning. ("Having a meaning" needs to be understood narrowly in order for this to make sense, as signifying that the expression is assigned an entity as a meaning by a certain semantic theory.) In the standard semantics for first-order logic, quantifiers and sentential connectives are syncategorematic, since instead of assigning them denotations (in the way that we assign predicates and names denotations) we state rules that govern sentences containing them, such as "$A \wedge B$ is true if and only if A is true and B is true" or "$\forall v A$ is true if and only if A is true of every member of the domain". But in higher-order logic, sentential connectives can be treated categorematically. E.g., recall from section 5.2.4 that $\sim$ (which is of type $\langle t,t\rangle$, denotes (in any interpretation) the function that maps T to F and F to T. Now, if in higher-order logic we had just the two untyped quantifiers, they would still need to be syncategorematic. But with typed quantifiers, each one can receive a denotation in isolation, as in the text. Note, however, that even in higher-order logic we still need a syncategoramic expression: λ.

$\langle a, t\rangle$ (which is the type of the λ abstracts they're attached to) to make formulas. We can gloss $\exists^a$ and $\forall^a$ as meaning "applies to at least one a-entity" and "applies to every a-entity", respectively. Given the formal semantics for higher-order logic we have been developing, in any interpretation the denotations of these logical constants would be the following:

> $\exists^a$ denotes the function that maps any $\langle a, t\rangle$ denotation, d, to T if and only if for some a denotation, d', $d(d') = T$
>
> $\forall^a$ denotes the function that maps any $\langle a, t\rangle$ denotation, d, to T if and only if for every a denotation, d', $d(d') = T$

References

Bacon, Andrew (2022). *A Philosophical Introduction to Higher-order Logics*. Routledge. Forthcoming.

Boolos, George (1971). "The Iterative Conception of Set." *Journal of Philosophy* 68: 215–31. Reprinted in Boolos 1998: 13–29.

— (1998). *Logic, Logic, and Logic*. Cambridge, MA: Harvard University Press.

Church, Alonzo (1940). "A Formulation of the Simple Theory of Types." *Journal of Symbolic Logic* 5: 56–68.

Dorr, Cian (2016). "To Be F is to Be G." *Philosophical Perspectives* 30(1): 39–134.

Dorr, Cian, John Hawthorne and Juhani Yli-Vakkuri (2021). *The Bounds of Possibility: Puzzles of Modal Variation*. Oxford: Oxford University Press.

Enderton, Herbert (1977). *Elements of Set Theory*. New York: Academic Press.

Fine, Kit (2007). "Relatively Unrestricted Quantification." In Agustín Rayo and Gabriel Uzquiano (eds.), *Absolute Generality*, 20–44. Oxford: Oxford University Press.

Frege, Gottlob (1892). "On Concept and Object." In Frege (1997*a*), 181–93.

— (1952/1892). "On Sense and Reference." In Peter Geach and Max Black (eds.), *Translations of the Philosophical Writings of Gottlob Frege*. Oxford: Blackwell.

— (1997*a*). *The Frege Reader*. Ed. Michael Beany. Oxford: Blackwell.

— (1997*b*). "Function and Concept." In Frege (1997*a*), 130–48.

Heim, Irene and Angelika Kratzer (1998). *Semantics in Generative Grammar*. Malden, MA: Blackwell.

Kaplan, David (1994). "A Problem in Possible Worlds Semantics." In Walter Sinnott-Armstrong (ed.), *Modality, Morality, and Belief*, 41–52. New York: Cambridge University Press.

MacFarlane, John (2005). "Logical Constants." Stanford Encyclopedia of Philosophy. Available at `http://plato.stanford.edu/entries/logical-constants/`.

Shapiro, Stewart (1991). *Foundations without Foundationalism: A Case for Second-Order Logic*. Oxford: Clarendon Press.

Stalnaker, Robert (1977). "Complex Predicates." *The Monist* 60: 327–39.