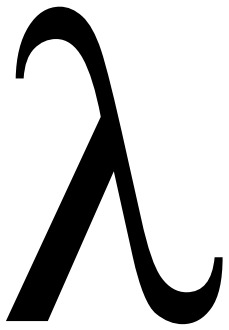
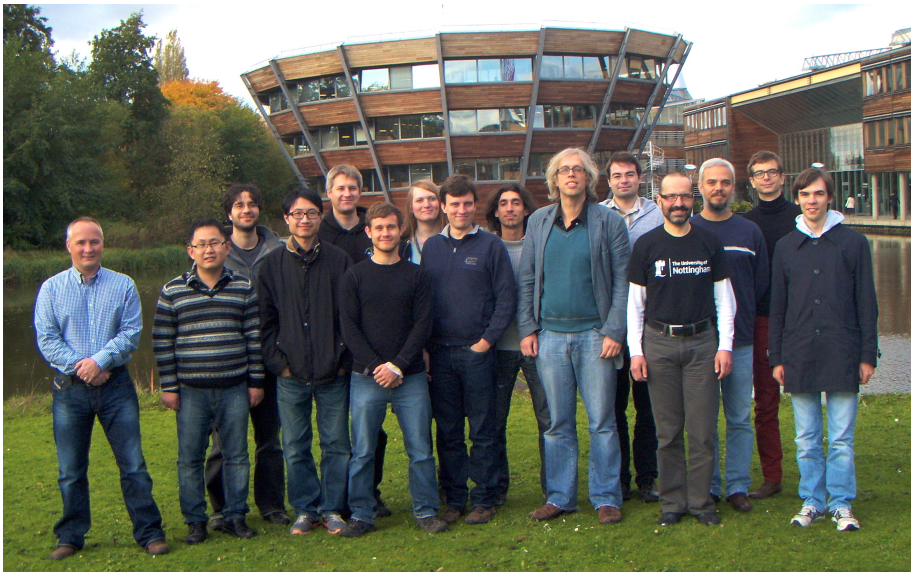
Proceedings of the

Functional Programming Lab
Away Day 2013



13-14 June, 2013
Cressbrook Hall, Buxton, England

The University of Nottingham



Functional Programming Lab
The University of Nottingham
2013

Table of Contents

From High-School- to University-Algebra

Thorsten Altenkirch

1 Introduction

Wilkie [Wil01] showed that there are identities in Tarski's Highschool Algebra which are not provable from the laws. At the core of his counterexample is the observation that $AD = BC$ implies

$$(A^y + B^y)^x (C^x + D^x)^y = (A^x + B^x)^y (C^y + D^y)^x$$

but this is unprovable with the laws. Di Cosmo et al observed that this identity corresponds to an isomorphism present in all bicartesian closed categories, i.e. typed λ-calculus with $\to, 1, \times, +$ which cannot be decomposed into the basic isomorphisms of biCCCs [FDCB02].

The situation becomes different, if we move to a setting with dependent types, i.e. we consider locally cartesian closed categories with a disjoint boolean object — this corresponds to Type Theory with $\Pi, \Sigma, 1, 2$. We give a collection of isomorphisms for this Type Theory, which we call *University Algebra* and show that Wilke's counterexample can be derived. We conjecture that this collection is complete for deriving isomorphisms and decidable.

By moving from equations to isomorphisms we move from ordinary algebra to a form of 2-algebra, where we don't only state that an equation hold but also use constants (i.e. isomorphisms) witnessing equations on which further equations may depend.

2 University algebra

University algebra is presented in the language of finitary Type Theory with $\Pi, \Sigma, 1, 2$. The Type Theory is extensional, i.e. all the η-rules are present. We conjecture that equality is decidable (in the presence of type variables), the idea is to extend the technique we have explored for simple types with strong sums [ADHS01] to dependent types. Adding 0 would make the theory undecidable, because the equations $\mathrm{tt} = \mathrm{ff}$ only holds in inconsistent contexts which can be used to encode consistency of first order intuitionistic logic.

Disjointness of 2 corresponds to the existence of a large if, i.e. if there are types $A, B : \mathbf{Type}$ and $b : 2$ we can form a new type: if $b\,A\,B : \mathbf{Type}$.

University algebra is given by the following basic isomorphisms:

$$
\begin{array}{ll}
\Phi_{2C} & : \quad 2 \simeq 2 \\
\Phi_{2A} & : \Sigma x : 2.\mathrm{if}\, x\, A\, \Sigma y : 2.\mathrm{if}\, y\, B\, C \simeq \Sigma x : 2.\mathrm{if}\, x\, (\Sigma y : 2.\mathrm{if}\, y\, A\, B)\, C \\
\Phi_{\Sigma A} & : \quad \Sigma a : A.\Sigma b : B\, a.C\, a\, b \simeq \Sigma(a,b) : (\Sigma a : A.B\, a).C\, a\, b \\
\Phi_{\Pi 1} & : \quad \Pi - : A.1 \simeq 1 \\
\Phi_{1\Pi} & : \quad \Pi x : 1.B\, x \simeq B\, () \\
\Phi_{2\Pi} & : \quad \Pi b : 2.B\, b \simeq (B\, \mathrm{tt}) \times (B\, \mathrm{ff}) \\
\Phi_{1\Sigma} & : \quad \Sigma x : 1.B\, x \simeq B\, () \\
\Phi_{\Sigma\Pi} & : \quad \Pi a : A.\Pi b : B\, a.C\, a\, b \simeq \Pi(a,b) : (\Sigma a : A.B\, a).C\, a\, b \\
\Phi_{\Pi\Sigma} & : \quad \Pi a : A.\Sigma b : B\, a.C\, a\, b \simeq \Sigma f : (\Pi a : A.B\, a).\Pi a : A.C\, a\, (f\, a)
\end{array}
$$

All the isomorphisms are definable in the language and give rise to definitional isomorphisms, i.e. there is $\phi : A \to B$ and $\phi^{-1} : B \to A$ such that $\phi^{-1} \circ \phi = I_A$ and $\phi \circ \phi^{-1} = I_B$, which can be expanded to $\lambda a : A.\phi^{-1}(\phi\, a) = \lambda a : A.a$ and $\lambda b : B.\phi(\phi^{-1}\, b)$. We sketch this construction by giving the left-to-right morphisms:

$$
\begin{array}{ll}
\Phi_{2C}\, \mathrm{tt} & = \mathrm{ff} \\
\Phi_{2C}\, \mathrm{ff} & = \mathrm{tt} \\
\Phi_{2A}\, (\mathrm{tt}, a) & = (\mathrm{tt}, (\mathrm{tt}, a)) \\
\Phi_{2A}\, (\mathrm{ff}, (\mathrm{tt}, b) & = (\mathrm{tt}, (\mathrm{ff}, b)) \\
\Phi_{2A}\, (\mathrm{ff}, (\mathrm{ff}, c) & = (\mathrm{ff}, c) \\
\Phi_{\Sigma A}\, (a, (b, c)) & = ((a, b), c) \\
\Phi_{1\Pi}\, f & = f\, () \\
\Phi_{2\Pi}\, f & = (f\, \mathrm{tt}, f\mathrm{ff}) \\
\Phi_{\Sigma 1}\, ((), b) & = b \\
\Phi_{\Pi\Pi}\, f & = \lambda(a,b).f\, a\, b \\
\Phi_{\Pi\Sigma}\, f & = (\lambda a.\pi_0\, (f a), \lambda a.\pi_1\, (f\, a))
\end{array}
$$

We also introduce structural operators which allow us to derive isomorphisms for complex types from isomorphisms for simpler ones:

$$
\frac{\phi : A \simeq A' \quad \psi : \Pi a : A.B\, a \simeq B'\, (\phi\, a)}{\Psi_\Sigma\, \phi\, \psi : \Sigma a : A.B\, a \simeq \Sigma a' : A'.B'\, a'} \qquad \frac{\phi : A' \simeq A \quad \psi : \Pi a : A'.B\, (\phi\, a) \simeq B'\, a}{\Psi_\Pi\, \phi\, \psi : \Pi a : A.B\, a \simeq \Pi a' : A'.B'\, a'}
$$

The rules for Π and Σ show that we have to define isomorphims wrt. a context of assumptions. To complete the list we also have:

$$
\frac{}{I_A : A \simeq A} \qquad \frac{b : 2 \quad \phi : A \simeq A' \quad \psi : B \simeq B'}{\Psi_{\mathrm{if}}\, b\, \phi\, \psi : \mathrm{if}\, b\, A\, B \simeq \mathrm{if}\, b\, A'\, B'}
$$

3 Deriving High School Algebra

We represent the operations of High School algebra — there are two isomorphic choices (using 2Π) for $\times$:

$$\begin{aligned} A + B &= \Sigma b : 2.\text{if}\, b\, A\, B \\ A \times B &= \Pi b : 2.\text{if}\, b\, A\, B \\ &\simeq \Sigma - : A.B \\ A \to B &= \Pi - : A.B \end{aligned}$$

The isomorphisms of High School Algebra are:

$$\begin{array}{ll} \Phi_{+C} : & A + B \simeq B + A \\ \Phi_{+A} : & A + (B + C) \simeq (A + B) + C \\ \Phi_{\times 1} : & 1 \times A \simeq A \\ \Phi_{\times A} : & B \times A \simeq B \times A \\ \Phi_{\times +} : & A \times (B + C) \simeq (A \times B) + (A \times C) \\ \Phi_{\to 1} : & A \to 1 \simeq 1 \\ \Phi_{1\to} : & 1 \to A \simeq A \\ \Phi_{+\to} : & (A + B) \to C \simeq (A \to C) \times (B \to C) \\ \Phi_{\to\times} : & A \to (B \times C) \simeq (A \to B) \times (A \to C) \\ \Phi_{\times\to} : & (A \times B) \to C \simeq A \to (B \to C) \end{array}$$

These isomorphisms are easily derivable from the ones in University Algebra: Φ_{+C} and $\Phi_{\times C}$ follow from Φ_{2C}. $\Phi_{+}A$ can be derived from Φ_{2A}. $\Phi_{\times +}$ follows from $\Phi_{\Sigma A}$:

$$\begin{aligned} A \times (B + C) &\simeq \Sigma_{:} A.\Sigma b : 2.\text{if}\, b\, B\, C \\ &\simeq \Sigma b : 2.\Sigma_{:} A.\text{if}\, b\, B\, C && (\Phi_{\Sigma A}) \\ &= \Sigma b : 2.\text{if}\, b\, (\Sigma - : A.B)\, (\Sigma - : A.C) \\ &\simeq (A \to C) \times (B \to C) \end{aligned}$$

$\Phi_{\to 1}$ is $\Phi_{\Pi 1}$ and $\Phi_{1\to}$ is an instance of $\Phi_{\Pi 1}$. $\Phi_{+\to}$ can be constructed from $\Phi_{\Sigma\Pi}$:

$$\begin{aligned} (A + B) \to C &\simeq \Pi - : (\Sigma b : 2.\text{if}\, b\, A\, B).C \\ &\simeq \Pi b : 2.\Pi - : \text{if}\, b\, A\, B.C && (\Phi_{\Sigma\Pi}) \\ &= \Pi b : 2.\text{if}\, b\, (\Pi - : A.B)\, (\Pi - : A.C) \\ &\simeq (A \to C) \times (B \to C) \end{aligned}$$

$\Phi_{\to\times}$ and $\Phi_{\times\to}$ are instances of $\Phi_{\Pi\Sigma}$ and $\Phi_{\Sigma\Pi}$ respectively.

4 Deriving the Wilkie isomorphism

We can derive a number of useful isomorphisms, essential for the Wilkie isomorphism is the following:

$$\begin{aligned} & X \to (Y \to A + Y \to B) \\ & \equiv X \to \Sigma b : 2.\text{if}\, b\, (Y \to A)\, (Y \to B) \\ & \simeq \Sigma f : X \to 2.\Pi x : X.\text{if}\, (f\, x)\, (Y \to A)\, (Y \to B) && (\Psi_{\Sigma\Pi}) \\ & \equiv \Sigma f : X \to 2.\Pi x : X.Y \to \text{if}\, (f\, x)\, A\, B \end{aligned}$$

Writing both sides of the Wilkie equality as types we can *normalize* both sides so that they only differ in the assumed isomorphism $\phi : A \times D \simeq B \times C$:

$$
\begin{array}{ll}
(X \to (Y \to A + Y \to B)) \times (Y \to (X \to C + X \to D) & \\
\simeq (\Sigma f : X \to 2.\Pi x : X.Y \to \mathrm{if}\,(f\,x)\,A\,B) \times (\Sigma g : Y \to 2.\Pi y : Y.X \to \mathrm{if}\,(g\,y)\,C\,D) & \\
\simeq \Sigma f : X \to 2.\Sigma g : Y \to 2. & \\
\qquad (\Pi x : X.Y \to \mathrm{if}\,(f\,x)\,A\,B) \times (X \to \Pi y : Y.\mathrm{if}\,(g\,y)\,C\,D) & (\Sigma\Sigma, \times C)) \\
\simeq \Sigma f : X \to 2.\Sigma g : Y \to 2. & \\
\qquad (\Pi x : X.\Pi y : Y.(\mathrm{if}\,(f\,x)\,A\,B)) \times (\mathrm{if}\,(g\,y)\,C\,D)) & (\Pi\Sigma) \\
\equiv \Sigma f : X \to 2.\Sigma g : Y \to 2. & \\
\qquad \Pi x : X.\Pi y : Y.(\mathrm{if}\,(f\,x)\,(\mathrm{if}\,(g\,y)\,(A \times C)\,(A \times D))\,(\mathrm{if}\,(g\,y)\,(B \times C)\,(B \times D)) &
\end{array}
$$

$$
\begin{array}{ll}
(X \to (Y \to A + Y \to B)) \times (Y \to (X \to C + X \to D) & \\
\simeq (\Sigma g : Y \to 2.\Pi y : Y.X \to \mathrm{if}\,(g\,y)\,A\,B) \times (\Sigma f : X \to 2.\Pi x : X.Y \to \mathrm{if}\,(f\,x)\,C\,D) & \\
\simeq \Sigma f : X \to 2.\Sigma g : Y \to 2. & \\
\qquad (X \to \Pi y : Y \to \mathrm{if}\,(g\,y)\,A\,B) \times (\Pi x : X.Y \to \mathrm{if}\,(f\,x)\,C\,D) & (\Sigma\Sigma, \times C)) \\
\simeq \Sigma f : X \to 2.\Sigma g : Y \to 2. & \\
\qquad (X \to \Pi y : Y \to \mathrm{if}\,(g\,y)\,A\,B) \times (\Pi x : X.Y \to \mathrm{if}\,(f\,x)\,C\,D) & (\Pi\Sigma) \\
\equiv \Sigma f : X \to 2.\Sigma g : Y \to 2. & \\
\qquad \Pi x : X.\Pi y : Y.(\mathrm{if}\,(f\,x)\,(\mathrm{if}\,(g\,y)\,(A \times C)\,(B \times C))\,(\mathrm{if}\,(g\,y)\,(A \times D)\,(B \times D)) &
\end{array}
$$

5 Questions

By finitary non-empty extensional Type Theory (FNETT), I mean the Type Theory with $1, 2$ and Π and Σ-types. We can also add Identity-types but they are actually definable in this theory. We intentionally leave out 0 because this would destroy a number of the properties we are interested in (in particular decidability).

We are considering this theory wrt a number of large assumptions, e.g. we may assume $X : \mathbf{Type}$ or $Y : X \to \mathbf{Type}$ which corresponds to assuming that X is a natural number or that Y is an X-tuple of natural numbers.

The development in this paper raises a number fo questions about thie Type Theory:

1. Is FNETT decidable?
2. Can we show that FNETT is complete wrt finite sets? Note that this would entail 1.
3. Can we show that all provable isomorphisms can be derived from the ones given here (or a similar collection)? In particular is there an algorithm which calculates such a canonical isomorphism from any given one?

The last point is related to finding a normal form for types. I have something like this in mind:

$$
\begin{array}{rl}
\mathrm{NF} & :: \Sigma x : \mathrm{NF}_\Pi.\mathrm{NF} \mid \mathrm{NF}_\Pi \\
\mathrm{NF}_\Pi & :: \Pi x : \mathrm{NF}.\mathrm{NF}_\Pi \mid \mathrm{NF}_0 \\
\mathrm{NF}_0 & :: X|n|\mathrm{T}[\mathrm{NF}]
\end{array}
$$

Here $T[X]$ stands for case-trees over X.

So unlike in the simply-typed case we have such a normal form which relies essentially on the presence of the axiom-of-choice-isomorphism.

References

[ADHS01] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, 2001.

[FDCB02] Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 147–156. IEEE, 2002.

[Wil01] Alex J Wilkie. On exponentiation-a solution to Tarski's high school algebra problem. 2001.

Programming Macro Tree Transducers

Patrick Bahr
Department of Computer Science
University of Copenhagen
paba@diku.dk

Laurence E. Day
Functional Programming Laboratory
University of Nottingham
led@cs.nott.ac.uk

Abstract

Tree transducers are tree automata defining functions from trees to trees. Put simply, a tree transducer is a set of mutually recursive functions transforming an input tree into an output tree. *Macro* tree transducers extend this recursion scheme by allowing each function to be defined in terms of an arbitrary number of accumulation parameters. In this paper, we show how macro tree transducers can be concisely represented in Haskell, and demonstrate the benefits of utilising such an approach with a number of examples. In particular, tree transducers afford a modular programming style as they can be easily composed and manipulated. Our Haskell representation generalises the original definition of (macro) tree transducers, abolishing a restriction on finite state spaces. However, as we demonstrate, this generalisation does not affect compositionality.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data Types and Structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and Recursion Schemes

General Terms Design, Languages, Performance

Keywords algebraic programming, reusability, deforestation, mutual recursion

1. Introduction

Tree automata [5] describe restricted forms of computations on trees. The simplest tree automata are known as acceptors, which – analogously to finite state string automata – are used to describe a set of (potentially infinite) trees. Other types of automata, called transducers, describe binary relations between trees, in general, or total functions on trees, in the case of total, deterministic transducers, which we consider in this paper.

Tree automata are widely applied in the fields of logic [35], term rewriting [8, 24, 36], XML processing [19], natural language processing [23]. The restrictions upon such automata affords them many desirable meta-properties such as decidability of equality [5] and reversability [28].

In this paper, we aim to utilise the properties of tree automata in order to structure functional programs. In this sense, this paper follows a line of work [1–3] in which we explore the use of recursion schemes derived from tree automata to build highly modular programs that manipulate tree structures. In particular, our goal is to extend the expressive power of this approach by using *macro tree transducers* (MTTs) [11].

1.1 Contributions

In previous work [1], we considered both top-down tree transducers (DTTs) and bottom-up tree transducers (UTTs) [31, 34]. Intuitively, DTTs represent a set of mutually recursive functions, and MTTs generalise this recursion scheme by allowing each function to include an arbitrary number of additional accumulator parameters. In this paper we show how to represent MTTs in Haskell using a free monad type.

In particular, the contributions are as follows:

- Starting from the representation of DTTs in Haskell, we illustrate a restriction upon the state space derived from the types (Section 3.1). In contrast to the original intent of MTTs, this observation is independent from the availability of accumulation parameters.
- We show how this restriction of DTTs in Haskell is mitigated by moving to a polymorphic state space (Section 3.2).
- We illustrate that the generalisation to a polymorphic state space corresponds to the generalisation from DTTs to MTTs in automata theory (Section 4).
- Our representation of tree transducers in Haskell generalises their automata theoretic definitions, since we do not require a finite state space. We illustrate the benefit of this generalisation (Section 2.5) and show that the compositional properties of MTTs and DTTs are maintained in this setting (Section 6).
- We also represent a recursion scheme that combines top-down and bottom-up state propagation, namely *MTTs with regular look-ahead* (Section 7).

This paper has been compiled from a literate Haskell file, and can be downloaded from the authors' websites. Furthermore, the recursion schemes presented in this paper have been implemented as part of the `compdata` Haskell library [4]. This implementation combines the recursion schemes with Swierstra's *data types à la carte* [33] and other modular constructs [1].

1.2 Why Transducers?

Before diving into the technical details, we highlight the benefits of using tree transducers in Haskell in the first place.

As alluded to in the introduction, tree transducers lend themselves to a highly modular programming style. In previous work [1] we have demonstrated that tree transducers allow us to leverage modularity along multiple, independent dimensions.

The primary notion of modularity is that of *sequential composition*. Given two transformations, one of type $A \to B$ and one of

type $B \to C$, we are able to construct a single transformation of type $A \to C$ by combining the underlying transducers. This allows us to split complex transformations – as seen in compilers, for example – into sequential phases and combine them without incurring undue performance penalties. This proves a powerful mechanism when performing deforestation [26, 27, 40], for example.

Secondly, we can modularise the types of the input trees. Given a transducer defined over trees of signature $\mathcal{F}$ and one over $\mathcal{G}$, we can compose the two to obtain a transducer defined over trees of signature $\mathcal{F} \uplus \mathcal{G}$. This is done via the *data types à la carte* [33] technique of Swierstra. But one should note that this construction is not always straightforward, as subtle interactions between signatures may exist. However, our proposed technique allows for a flexible composition that accounts for these problems (see [1] for a number of examples).

The third aspect of modularity is the fact that we can compose simpler automata to obtain more complex ones. For example, a transducer may perform a transformation (e.g. inlining) that is dependent on information that is provided by other automata independently (e.g. information about the occurrences of bound variables). This flexible approach not only reduces the complexity of individual automata, but ensures that they can be substituted easily regardless of the context in which they operate.

Modularity aside, the structure of tree transducers permits other operations that manipulate computations on trees. A simple but powerful class of such operations is the automatic generation and propagation of annotations. For example, given an code generator (as an MTT) as part of a compiler, a lifting can be defined that automatically propagates annotations of the input (e.g. annotations indicating the originating line in a source file) [2]. Section 5 demonstrates this lifting for the case of MTTs.

2. Top-Down Tree Transducers

In this section, we briefly explain the concept of top-down tree transducers, which is the type of automata that we shall built upon subsequently.

2.1 First-Order Terms

The tree automata that we consider in this paper operate on terms over some signature $\mathcal{F}$. In the setting of tree automata, a signature $\mathcal{F}$ is simply a set of function symbols with a fixed arity and we write $f/n \in \mathcal{F}$ to indicate that f is a function symbol in $\mathcal{F}$ of arity n. Given a signature $\mathcal{F}$ and some set $\mathcal{X}$, the set of terms over $\mathcal{F}$ and $\mathcal{X}$, denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the smallest set T such that $\mathcal{X} \subseteq T$ and if $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in T$ then $f(t_1, \ldots, t_n) \in T$. Instead of $\mathcal{T}(\mathcal{F}, \emptyset)$ we also write $\mathcal{T}(\mathcal{F})$ and call elements of $\mathcal{T}(\mathcal{F})$ terms over $\mathcal{F}$. Tree automata run on terms in $\mathcal{T}(\mathcal{F})$.

We will use the words "tree" and "term" interchangeably. In general, we will use the word term and subterm, but in certain contexts it is beneficial to think of the tree representation of a term. For example, we shall use depictions of terms in the form of trees in order to illustrate the workings of the tree automata we consider. In this context it is convenient to talk about *nodes* and *edges* of the tree representation. For example, for a term of the form $f(t_1, \ldots, t_k)$, we think of a tree rooted in a node n labelled by f, with k outgoing ordered edges to nodes $n_1, \ldots, n_k$. These nodes are called the *successor nodes* of n, and each n_i is the root node of a tree that represents the term t_i.

Each of the tree automata that we describe in this paper consists at least of a finite set Q of states and a set of rules according to which an input term is transformed into an output term. While performing such a transformation, these automata maintain state information, which is stored in the intermediate results of the transformation. To this end each state $q \in Q$ is considered as a unary function symbol and a subterm t is annotated with state q by writing $q(t)$. For example, $f(q_0(a), q_1(b))$ represents the term $f(a, b)$, where the two subterms a and b are annotated with states q_0 and q_1, respectively.

The rules of the tree automata in this paper will all be of the form $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}', \mathcal{X})$, where $\mathcal{F}' = \mathcal{F} \uplus \{q/k_q \mid q \in Q\}$. For the simple tree automata, the arity k_q of each state q will be just 1. We shall see later in Section 4, that this restriction is lifted for macro tree transducers. The rules can be read as term rewrite rules, i.e. the variables in l and t are placeholders that are instantiated with terms when the rule is applied. Running an automaton is then simply a matter of applying these term rewrite rules to a term. The different kinds of tree automata only differ in the set of rules they allow.

In general, tree transducers induce a relation on trees that pairs each input tree with each of its correspondent outputs. That is, tree transducers are non-deterministic. Since we are only interested in tree transformations that are functions, in particular transformations that occur in compilers, we shall only consider automata which are deterministic and total. Roughly speaking, 'deterministic' means that, in each "situation", there is at most one rule applicable, while 'total' means that there is at least one rule applicable.

2.2 Top-Down Tree Transducers

Top-down tree transducers (DTTs) are able to produce transformations that depend on a top-down flow of information. To do this, the rules of a DTT specify for each function symbol f (1) how a state is propagated from a node (labelled with f) to its successor nodes, and (2) a tree that replaces the node (labelled with f). More formally, a (deterministic and total) DTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a finite set of states Q, an initial state $q_0 \in Q$ and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n)) \to u \quad \text{for each } f/n \in \mathcal{F} \text{ and } q \in Q$$

where the right-hand side $u \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X}))$ is a term over $\mathcal{G}$ and $Q(\mathcal{X}) = \{p(x_i) \mid p \in Q, 1 \leq i \leq n\}$. That is, the right-hand side is a term that may have subterms of the form $p(x_i)$ with x_i a variable from the left-hand side and p a state in Q. In other words, each *occurrence* of a variable on the right-hand side is given a successor state. We only consider DTTs that are deterministic and total, which means that for each $f \in \mathcal{F}$ and $q \in Q$ there is exactly one rule with left-hand side $q(f(x_1, \ldots, x_n))$.

Figure 1 illustrates the shape of the transduction rules of a DTT. On the left-hand side we find a node with a function symbol $f \in \mathcal{F}$ annotated by a state $q \in Q$. Furthermore, we find the successor nodes (indicated by orange circles), which are represented by variables x_i in the above notation. Intuitively, these variables are *placeholders*, which are instantiated when the rule is applied. On the right-hand side we find a tree over $\mathcal{G}$ that may also contain placeholders taken from the left-hand side. However, each such placeholder has to be annotated with a successor state.

When applying such a rule, the placeholders are instantiated by the actual successor nodes of an f-labelled node in the input tree. The application of a rule at the root of a tree is illustrated in Figure 2. The tree has an f-labelled root node annotated with a state q, and each of the root nodes successor nodes is the root of a tree that represents the corresponding subterm. The rule application replaces the root node with a tree that contains subtrees of the original tree as specified in the rule. After this first application of a transduction rule, the process is repeated recursively at each node annotated with a state. In Figure 2, this is the case for the states q_1 and q_2.

In order to run a DTT on a term $t \in \mathcal{T}(\mathcal{F})$, we have to provide an initial state q_0 and then apply the transduction rules to $q_0(t)$ in a top-down fashion. Eventually, this yields a result term $t' \in \mathcal{T}(\mathcal{G})$.

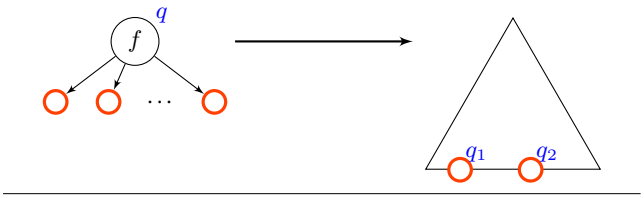


Figure 1. Top-down tree transduction rule.

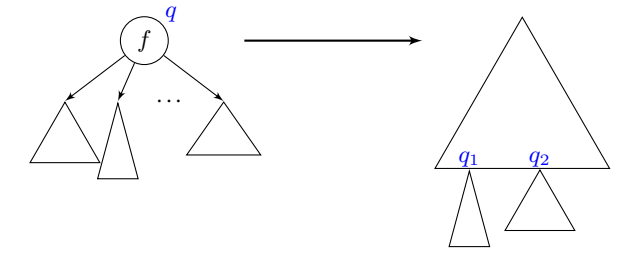


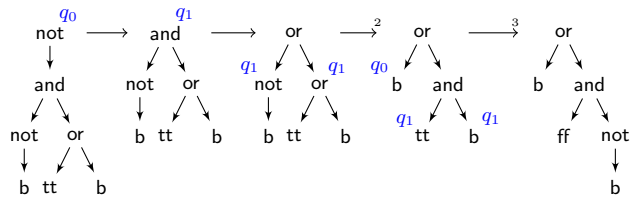
Figure 2. Application of top-down tree transduction rule.

Example 1. Consider the signature $\mathcal{F} = \{\mathsf{or}/2, \mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0, \mathsf{b}/0\}$. Terms over $\mathcal{F}$ are supposed to represent Boolean expressions with a single Boolean variable b. We construct a DTT from $\mathcal{F}$ to $\mathcal{F}$ that implements the transformation of Boolean expressions into negation normal form, in which negation is only allowed to be applied directly to a variable. The DTT operates on the set of states $Q = \{q_0, q_1\}$ with initial state q_0 and the following transduction rules:

$$
\begin{array}{ll}
q_0(\mathsf{and}(x,y)) \to \mathsf{and}(q_0(x), q_0(y)) & q_0(\mathsf{not}(x)) \to q_1(x) \\
q_1(\mathsf{and}(x,y)) \to \mathsf{or}(q_1(x), q_1(y)) & q_1(\mathsf{not}(x)) \to q_0(x) \\
q_0(\mathsf{or}(x,y)) \to \mathsf{or}(q_0(x), q_0(y)) & q_0(\mathsf{b}) \to \mathsf{b} \\
q_1(\mathsf{or}(x,y)) \to \mathsf{and}(q_1(x), q_1(y)) & q_1(\mathsf{b}) \to \mathsf{not}(\mathsf{b})
\end{array}
$$

$$
q_0(\mathsf{tt}) \to \mathsf{tt} \quad q_0(\mathsf{ff}) \to \mathsf{ff} \quad q_1(\mathsf{tt}) \to \mathsf{ff} \quad q_1(\mathsf{ff}) \to \mathsf{tt}
$$

The transformation that the above DTT performs is straightforward: it moves the operator not inwards, applying De Morgan's laws when it encounters conjunctions and disjunctions. For instance, applied to the expression $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{or}(\mathsf{tt}, \mathsf{b})))$, the automaton yields the following derivation (where superscripts on the arrows indicate the number of transition steps that have been performed simultaneously):



Note that while states appear as function symbols in the transduction rules we have indicated the states rather as annotations in the derivation above.

In order to start the run of a DTT, the initial state q_0 has to be explicitly inserted at the root of the input term. The run of the automaton is completed as soon as all states in the term have vanished; there is no final state.

2.3 Terms in Haskell

In Haskell, we represent signatures as (polynomial) functors. For instance the signature $\mathcal{F}$ from Example 1 is represented in Haskell as follows:

```
data F a = Or a a | And a a | Not a | TT | FF | B
```

Terms over F are then represented as the free monad F^* of F, which is defined as follows:

```
data f* a = Re a | In (f (f* a))
```

That is, given a functor F that represents the signature $\mathcal{F}$ and a type X that represents the set $\mathcal{X}$, we use the type $F^*\ X$ to represent the set of terms over $\mathcal{F}$ and $\mathcal{X}$.

For each functor f, the type f^* comes with the following generic fold operation:

```
type Alg f a = f a → a
fold* :: Functor f ⇒ (a → c) → Alg f c → f* a → c
fold* ret alg (In t)  = alg (fmap (fold* ret alg) t)
fold* ret _   (Re x) = ret x
```

Furthermore, we shall make use of the fact that f^* indeed forms a monad:

```
instance Functor f ⇒ Functor (f*) where
  fmap f = fold* (Re ∘ f) In
instance Functor f ⇒ Monad (f*) where
  return  = Re
  m >>= f = fold* f In m
```

The bind operation of this monad ($\gg\!=$) implements a substitution operation: $t \gg\!= f$ is obtained from t by replacing each subterm $Re\ x$ in t by $f\ x$.

Note that, while we use Haskell as an implementation language, we assume a set-theoretic semantics. That is, all functions are assumed to be total. This assumption is important since we are interested in representing *total* tree transducers only. The full importance of the set theoretic semantics will become apparent when we describe the compositionality of transducers since the compositionality results of (top-down) tree transducers do not hold for tree transducers that are not total [31].

Additionally, the set theoretic semantics allows us to derive the representation of the set $\mathcal{T}(\mathcal{F})$ from the free monad using an empty data type:

```
data Empty
type μf = f* Empty
```

The type $Empty$ comes with a canonical mapping $empty :: Empty \to a$, which allows us to embed the type μf into $f^*\ a$ for any a:

```
toFree :: Functor f ⇒ μf → f* a
toFree = fold* empty In
```

Moreover, we also get a fold operation for each μf:

```
fold_μ :: Functor f ⇒ Alg f c → μf → c
fold_μ = fold* empty
```

2.4 Top-Down Transduction Functions

We now turn to the representation of DTTs in Haskell. The transduction rules of a DTT use placeholder variables x_1, x_2, etc. in order to refer to arguments of function symbols. These placeholder variables can then be used on the right-hand side of a transduction rule. This mechanism makes it possible to rearrange, remove and duplicate the terms that are matched against these placeholder

variables. On the other hand, it is not possible to inspect them. For instance, in Example 1, $q_0(\mathsf{not}(\mathsf{ff})) \to \mathsf{tt}$ would not be a valid transduction rule as we are not allowed to pattern match on the argument of not. We can only observe the state.

When representing transduction rules as Haskell functions, we have to be careful in order to maintain this restriction on DTTs. In their categorical representation, Hasuo et al. [18] recognised that the restriction due to placeholder variables in the transduction rules can be enforced by a *naturality* condition. Naturality, in turn, can be represented in Haskell's type system as *parametric polymorphism*. Following this approach, we represent DTTs from signature functor f to signature functor g with state space q by the following type:

$$\mathbf{type}\ \mathit{Trans}_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ (q, f\ a) \to g^*\ (q, a)$$

In the definition of tree automata, states are used syntactically as a unary function symbol – an argument with state q is written as $q(x)$ in the left-hand side. In the Haskell representation, we use pairs and simply write (q, x).

In the type $\mathit{Trans}_{\mathsf{D}}$, the type variable a represents the type of the placeholder variables. The universal quantification over a makes sure that these placeholders cannot be scrutinised; they can only be taken from the left-hand side and inserted on the right-hand side without alteration.

Example 2. The representation of the signature $\mathcal{F}$ and the state space Q from Example 1 is straightforward:

$$\mathbf{data}\ F\ a = \mathit{Or}\ a\ a \mid \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \mid B$$
$$\mathbf{data}\ Q = \mathit{Q0} \mid \mathit{Q1}$$

For the definition of the transduction function, we use smart constructors iAnd, iNot, iTT, iFF and iB for the constructors of the signature F. These smart constructors additionally apply the constructor In of the free monad type:

$$\begin{aligned}&\mathit{iAnd} :: F^*\ a \to F^*\ a \to F^*\ a\\ &\mathit{iAnd}\ x\ y = \mathit{In}\ (\mathit{And}\ x\ y)\end{aligned}$$

The transduction function is then defined as follows:

$$\begin{aligned}&\mathit{trans} :: \mathit{Trans}_{\mathsf{D}}\ F\ Q\ F\\ &\mathit{trans}\ (\mathit{Q0}, \mathit{TT})\ = \mathit{iTT}; \qquad \mathit{trans}\ (\mathit{Q0}, \mathit{FF}) = \mathit{iFF}\\ &\mathit{trans}\ (\mathit{Q1}, \mathit{TT})\ = \mathit{iFF}; \qquad \mathit{trans}\ (\mathit{Q1}, \mathit{FF}) = \mathit{iTT}\\ &\mathit{trans}\ (\mathit{Q0}, B) = \mathit{iB}\\ &\mathit{trans}\ (\mathit{Q1}, B) = \mathit{iNot}\ \mathit{iB}\\ &\mathit{trans}\ (\mathit{Q0}, \mathit{Not}\ x) = \mathit{Re}\ (\mathit{Q1}, x)\\ &\mathit{trans}\ (\mathit{Q1}, \mathit{Not}\ x) = \mathit{Re}\ (\mathit{Q0}, x)\\ &\mathit{trans}\ (\mathit{Q0}, \mathit{And}\ x\ y) = \mathit{Re}\ (\mathit{Q0}, x)\ `\mathit{iAnd}`\ \mathit{Re}\ (\mathit{Q0}, y)\\ &\mathit{trans}\ (\mathit{Q1}, \mathit{And}\ x\ y) = \mathit{Re}\ (\mathit{Q1}, x)\ `\mathit{iOr}`\ \ \mathit{Re}\ (\mathit{Q1}, y)\\ &\mathit{trans}\ (\mathit{Q0}, \mathit{Or}\ \ x\ y) = \mathit{Re}\ (\mathit{Q0}, x)\ `\mathit{iOr}`\ \ \mathit{Re}\ (\mathit{Q0}, y)\\ &\mathit{trans}\ (\mathit{Q1}, \mathit{Or}\ \ x\ y) = \mathit{Re}\ (\mathit{Q1}, x)\ `\mathit{iAnd}`\ \mathit{Re}\ (\mathit{Q1}, y)\end{aligned}$$

The definition of the transduction function trans is a one-to-one translation of the transduction rules of the DTT from Example 1.

Running a top-down tree transducer on a term is a straightforward affair:

$$\begin{aligned}&[\![\cdot]\!]_{\mathsf{D}} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow\\ &\quad \mathit{Trans}_{\mathsf{D}}\ f\ q\ g \to q \to \mu f \to \mu g\\ &[\![tr]\!]_{\mathsf{D}}\ q\ t = \mathit{run}\ (q, t)\ \mathbf{where}\\ &\quad \mathit{run}\ (q, \mathit{In}\ t) = \mathit{join}\ (\mathit{fmap}\ \mathit{run}\ (\mathit{tr}\ (q, t)))\end{aligned}$$

A top-down transducer is run by applying its transduction function – $\mathit{tr}\ \ (q, t)$ – then recursively running the transformation in the "holes" of the produced context – $\mathit{fmap}\ \mathit{run}$ – and finally joining the context with the thus produced embedded terms – join. The function join is the multiplication of monads, which can be derived from the $\gg\!=$ operation:

$$\mathit{join}\ x = x \gg\!= \mathit{id}$$

Specialised to the free monad constructor, its type is

$$\mathit{join} :: \mathit{Functor}\ f \Rightarrow f^*\ (f^*\ a) \to f^*\ a$$

With the thus defined semantics of DTTs in Haskell we can derive the transformation into negation normal form as defined by the DTT in Example 2:

$$\begin{aligned}&\mathit{negNorm} :: \mu F \to \mu F\\ &\mathit{negNorm} = [\![\mathit{trans}]\!]_{\mathsf{D}}\ \ \mathit{Q0}\end{aligned}$$

2.5 Infinite State Space

The Haskell representation of DTTs generalises the definition of DTTs as it does not put any restriction on the state space: the state space q in the type $\mathit{Trans}_{\mathsf{D}}\ f\ q\ g$ does not have to be finite. As a consequence, we can express transformations that go beyond DTTs in the traditional sense.

An example of such a transformation is substitution:

Example 3. We consider a simple expression language with let bindings:

$$\mathbf{type}\ \mathit{Var} = \mathit{String}$$
$$\mathbf{data}\ \mathit{Sig}\ a = \mathit{Add}\ a\ a \mid \mathit{Val}\ \mathit{Int} \mid \mathit{Let}\ \mathit{Var}\ a\ a \mid \mathit{Var}\ \mathit{Var}$$

Like in Example 2, we assume corresponding smart constructors iAdd, iVal, iLet, and iVar.

The DTT in works on a state space of type $\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig}$, i.e. finite maps from variables to terms over Sig:

$$\begin{aligned}&\mathit{trans}_{\mathsf{subst}} :: \mathit{Trans}_{\mathsf{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig})\ \mathit{Sig}\\ &\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Var}\ v) \quad = \mathbf{case}\ m\,[v]\ \mathbf{of}\\ &\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{iVar}\ v\\ &\qquad\qquad\qquad\qquad\quad \mathit{Just}\ t \to \mathit{toFree}\ t\\ &\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Let}\ v\ b\ s) = \mathit{iLet}\ v\ (\mathit{Re}\ (m, b))\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{Re}\ (m \setminus v, s))\\ &\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Val}\ n) \quad = \mathit{iVal}\ n\\ &\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Add}\ x\ y)\ = \mathit{Re}\ (m, x)\ `\mathit{iAdd}`\ \mathit{Re}\ (m, y)\end{aligned}$$

The actual substitution function is obtained by applying the semantic function $[\![\cdot]\!]_{\mathsf{D}}$:

$$\begin{aligned}&\mathit{subst} :: \mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig} \to \mu\mathit{Sig} \to \mu\mathit{Sig}\\ &\mathit{subst} = [\![\mathit{trans}_{\mathsf{subst}}]\!]_{\mathsf{D}}\end{aligned}$$

The initial state is simply the substitution that is supposed to be applied.

The restriction to finite state spaces is necessary to obtain certain decidability results. For example, the equivalence of deterministic DTTs is decidable [10, 32, 41].

However, our interest in tree transducers here is not the decidability of such meta properties but rather the compositionality of tree transducers. In particular, tree transducers are closed under sequential composition [31]. That is, there is an effective procedure that takes two (total, deterministic, top-down) tree transducers A and B such that A is from $\mathcal{F}$ to $\mathcal{G}$ and B is from $\mathcal{G}$ to $\mathcal{H}$, and produces a tree transducer C from $\mathcal{F}$ to $\mathcal{H}$ with $[\![C]\!] = [\![B]\!] \circ [\![A]\!]$, i.e. executing C is equivalent to first executing A and then B. This construction provides a means to perform program transformations in order to avoid constructing intermediate results, which often results in improved performance [25–27, 38, 39].

These compositionality results make use of the finiteness of the transducers' state spaces in order to construct compositions. We shall see in Section 6, however, that compositionality is maintained

even in the case of infinite state spaces. In particular, we show a composition operation

$$\cdot \circ_{\mathsf{D}} \cdot :: \mathit{Trans}_{\mathsf{D}}\ g\ p\ h \to \mathit{Trans}_{\mathsf{D}}\ f\ q\ g \to \mathit{Trans}_{\mathsf{D}}\ f\ (q, p)\ h$$

such that

$$[\![tr_2 \circ_{\mathsf{D}} tr_1]\!]_{\mathsf{D}}\ (q_1, q_2) = [\![tr_2]\!]_{\mathsf{D}}\ q_2 \circ [\![tr_1]\!]_{\mathsf{D}}\ q_1$$

This generalised compositionality also holds for bottom-up tree transducers as well as the more advanced macro tree transducers, which we shall discuss in Section 4.

3. Tree Transducers with Parametric State Space

The generalised notion of top-down tree transducers that we obtain in Haskell is quite expressive due to the availability of arbitrary state spaces. However, when working with them one quickly realises an unexpected restriction in using these state spaces.

3.1 The Problem

Let us reconsider the Haskell DTT from Example 3, which performs substitution. Inspired by this DTT, we would like to define another one that performs inlining of let bindings. This transformation works similarly to the substitution; it only differs in the case of let bindings:

$$\begin{aligned}
&\mathit{trans}_{\mathsf{inline}} :: \mathit{Trans}_{\mathsf{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig})\ \mathit{Sig}\\
&\mathit{trans}_{\mathsf{inline}}\ (m, \mathit{Var}\ v) \quad = \mathbf{case}\ m\,[v]\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\ \mathit{Nothing} \to \mathit{iVar}\ v\\
&\qquad\qquad\qquad\qquad\ \mathit{Just}\ e \quad \to \mathit{toFree}\ e\\
&\mathit{trans}_{\mathsf{inline}}\ (m, \mathit{Let}\ v\ x\ y) = \mathit{Re}\ (m\,[v \mapsto x]\,, y)\\
&\mathit{trans}_{\mathsf{inline}}\ (m, \mathit{Val}\ n) \quad = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathsf{inline}}\ (m, \mathit{Add}\ x\ y) \ = \mathit{Re}\ (m, x)\ `\mathit{iAdd}`\ \mathit{Re}\ (m, y)\\
&\mathit{inline} :: \mu\mathit{Sig} \to \mu\mathit{Sig}\\
&\mathit{inline} = [\![\mathit{trans}_{\mathsf{inline}}]\!]_{\mathsf{D}}\ \emptyset
\end{aligned}$$

In the case of Let, the DTT simply returns the subterm that is in the scope of the let binding, viz. y, and updates the substitution to include the binding $v \mapsto x$. The problem is, however, that this definition does not work as it is not well-typed. The right-hand side $\mathit{Re}\ (m\,[v \mapsto x]\,, y)$ of the offending equation has type $\mathit{Free}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ a, a)$ instead of the required type $\mathit{Free}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig}, a)$. The problem is that the Haskell variable x, which we want to put into the substitution, is of type a.

Let us recall the type of DTTs in Haskell:

$$\mathbf{type}\ \mathit{Trans}_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ (q, f\ a) \to g^*\ (q, a)$$

This type was deliberately designed such that placeholder variables like the abovementioned variable x can only be put into the free monad structure on the right-hand side, guarded by a successor state. In particular, we cannot put such a placeholder variable 'into' a state, which means that it is not possible to store subterms – such as the right-hand side of a let binding – into the state and retrieve it later.

3.2 The Solution

In order to allow placeholder variables to be put into the state we have to use a state space that is parametric, i.e. not a type but a type function of kind $* \to *$:

$$\mathbf{type}\ \mathit{Trans}\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \to g^*\ (q\ a, a)$$

Using this type, the definition of $\mathit{trans}_{\mathsf{inline}}$ is well-typed – with $q = \mathit{Map}\ \mathit{Var}$. However, this type breaks one of the key properties that we have for DTTs, namely that each placeholder variable that occurs on the right-hand side of a transduction rule is guarded by a state, which is represented in Haskell as a pair (q, a). We do not have this property in the above type. In particular, the placeholder variables that might be used to construct a value of type $q\ a$ are not guarded by a state themselves.

We can address this issue by using a recursive type P that makes sure that all placeholder variables are guarded by a state:

$$\mathbf{data}\ P\ q\ a = P\ (q\ (P\ q\ a))\ a$$

We can then use P to define the type for transducers with a parametric state space:

$$\mathbf{type}\ \mathit{Trans}\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \to g^*\ (P\ q\ a)$$

But even this type does not work out right. The problem is now that not only placeholder variables stemming from $f\ a$ have to be guarded by a state but also placeholder variables that have been extracted from the argument of type $q\ \ a$. Again, this problem can be fixed by introducing a second type variable b in order to distinguish the two type of placeholder variables. But at this point it becomes clear that the resulting type is much too complicated.

Instead, we will use the type variable a to encode placeholder variables that need not be guarded by a state and the function type $q\ a \to a$ for those that do need to be guarded by a state:

$$\mathbf{type}\ \mathit{Trans}\ f\ q\ g = \forall\ a\ .\ q\ a \to f\ (q\ a \to a) \to g^*\ a$$

This encoding also allows us to get rid of the type P.

We can further generalise this type: as it stands, we can only put placeholder variables into states, which is overly restrictive. In general we should be able to put in arbitrary terms that may contain placeholder variables. We thus arrive at the following type:

$$\mathbf{type}\ \mathit{Trans}_{\mathsf{M}}\ f\ q\ g = \forall\ a\ .\ q\ a \to f\ (q\ (g^*\ a) \to a) \to g^*\ a$$

Before defining the semantics of this transducer, which will turn out to represent *macro tree transducers*, we shall look at an example implementing the inlining transformation that we started with in this section:

$$\begin{aligned}
&\mathit{trans}_{\mathsf{inline}} :: \mathit{Trans}_{\mathsf{M}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var})\ \mathit{Sig}\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Var}\ v) \quad = \mathbf{case}\ m\,[v]\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\ \mathit{Nothing} \to \mathit{iVar}\ v\\
&\qquad\qquad\qquad\qquad\ \mathit{Just}\ e \quad \to \mathit{Re}\ e\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Let}\ v\ x\ y) = \mathit{Re}\ (y\ (m'\,[v \mapsto \mathit{Re}\ (x\ m')]))\\
&\quad \mathbf{where}\ m' = \mathit{fmap}\ \mathit{Re}\ m\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Val}\ n) \quad = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Add}\ x\ y) \ = \mathit{Re}\ (x\ m')\ `\mathit{iAdd}`\ \mathit{Re}\ (y\ m')\\
&\quad \mathbf{where}\ m' = \mathit{fmap}\ \mathit{Re}\ m
\end{aligned}$$

The first change that we can observe in the above code is that occurrences of the placeholder variables on the right-hand side are assigned a state not by pairing them with a state – (m', x) – but by applying them to a state – $(x\ m')$.

3.3 A More Concise Notation

While we can express the inline transformation with this transducer, it is quite tedious to do so. Since we have to inject the placeholder variables (of type a) from the left-hand side into the term structure of the right-hand side (type $\mathit{Sig}^*\ a$), the code is riddled with applications of Re. However, the encoding of state propagation affords us the opportunity to avoid this explicit plumbing: the only thing we can do with placeholder variables of type a is to inject them into the type $\mathit{Sig}^*\ a$ via Re. We can build this into the recursion scheme, which gives us the following variant of $\mathit{Trans}_{\mathsf{M}}$:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Trans}'_{\mathsf{M}}\ f\ q\ g = \forall\ a\ .\ q\ (g^*\ a) \to\\
&\qquad\qquad\qquad\qquad f\ (q\ (g^*\ a) \to g^*\ a) \to g^*\ a
\end{aligned}$$

We replaced the two "naked" occurrences of the type variable a by $g^*\ a$. The following function translates transducers of this more convenient type into the original $\mathit{Trans}_{\mathsf{M}}$ type by applying Re at the appropriate places:

$$\begin{aligned}
&\Uparrow_{\mathsf{M}} :: (Functor\ f, Functor\ q) \Rightarrow \\
&\quad Trans'_{\mathsf{M}}\ f\ q\ g \to Trans_{\mathsf{M}}\ f\ q\ g \\
&\Uparrow_{\mathsf{M}}\ \ tr\ q\ t = tr\ (fmap\ Re\ q)\ (fmap\ (Re \circ)\ t)
\end{aligned}$$

With this more convenient type, the inline transformation is implemented without syntactic noise:

$$\begin{aligned}
&trans_{\mathsf{inline}} :: Trans'_{\mathsf{M}}\ Sig\ (Map\ \ Var)\ Sig \\
&trans_{\mathsf{inline}}\ m\ (Var\ v) \quad = \mathbf{case}\ m\,[v]\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\quad Nothing \to iVar\ v \\
&\qquad\qquad\qquad\qquad\quad Just\ e \to e \\
&trans_{\mathsf{inline}}\ m\ (Let\ v\ x\ y) = y\ (m\,[v \mapsto x\ m]) \\
&trans_{\mathsf{inline}}\ m\ (Val\ n) \quad = iVal\ n \\
&trans_{\mathsf{inline}}\ m\ (Add\ x\ y) \quad = x\ m\ `iAdd`\ y\ m
\end{aligned}$$

The implementation of $trans_{\mathsf{inline}}$ illustrates the power of the $Trans_{\mathsf{M}}$ recursion scheme in the case for Let: first it passes the current m state unaltered to x, which is then used to add an additional variable assignment to the state m, which is then passed to y. In general, this nesting of recursion can be arbitrarily deep. The type $Trans_{\mathsf{M}}$ encodes this arbitrarily deep nesting of recursion by using the type $q\ (g^*\ a) \to a$ for its input placeholder variables.

3.4 The Semantics of $Trans_{\mathsf{M}}$

We finally come to the semantics of $Trans_{\mathsf{M}}$:

$$\begin{aligned}
&[\![\cdot]\!]_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
&\quad Trans_{\mathsf{M}}\ f\ q\ g \to q\ \mu g \to \mu f \to \mu g \\
&[\![tr]\!]_{\mathsf{M}}\ \ q\ t = run\ t\ q\ \mathbf{where} \\
&\quad run\ (In\ t)\ q = join\ (tr\ q\ (fmap\ run'\ t)) \\
&\quad run'\ t \quad\ \ q = run\ t\ (fmap\ join\ q)
\end{aligned}$$

This definition hardly differs from the definition of the DTT semantics $[\![\cdot]\!]_{\mathsf{D}}$. The only significant difference is the use of the auxiliary function run' in order to apply $fmap\ join$ to the states. The necessity of this can be observed in the type definition of $Trans_{\mathsf{M}}$, where the states of type $q\ a$ are transformed into states of type $q\ (g^*\ a)$ before being assigned to a placeholder variable. In the instantiation of this type that is used in the definition of $[\![\cdot]\!]_{\mathsf{M}}$, this means that $q\ \mu g$ is transformed into $q\ (g^*\ \mu g)$. In order to use these states of type $q\ (g^*\ \mu g)$ recursively, they have to be turned into states of type $q\ \mu g$, which is exactly what $fmap\ join$ does.

With this semantics we can finally define the inlining transformation as follows:

$$\begin{aligned}
&inline :: \mu Sig \to \mu Sig \\
&inline = [\![\Uparrow_{\mathsf{M}}\ \ trans_{\mathsf{inline}}]\!]_{\mathsf{M}}\ \emptyset
\end{aligned}$$

To see the similarity in the semantics of $Trans_{\mathsf{M}}$ and $Trans_{\mathsf{D}}$, let us apply the same style of passing along states to $Trans_{\mathsf{D}}$ as well, i.e. using a function type instead of pairing:

$$\begin{aligned}
&\mathbf{type}\ Trans_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ q \to f\ (q \to a) \to g^*\ a \\
&[\![\cdot]\!]_{\mathsf{D}} :: (Functor\ f, Functor\ g) \Rightarrow \\
&\quad Trans_{\mathsf{D}}\ f\ q\ g \to q \to \mu f \to \mu g \\
&[\![tr]\!]_{\mathsf{D}}\ \ q\ t = run\ t\ q\ \mathbf{where} \\
&\quad run\ (In\ t)\ q = join\ (tr\ q\ (fmap\ run\ t))
\end{aligned}$$

This style of passing states around allows us to avoid the use of explicit Re applications for DTTs as well:

$$\begin{aligned}
&\mathbf{type}\ Trans'_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ q \to f\ (q \to g^*\ a) \to g^*\ a \\
&\Uparrow_{\mathsf{D}} :: Functor\ f \Rightarrow Trans'_{\mathsf{D}}\ f\ q\ g \to Trans_{\mathsf{D}}\ f\ q\ g \\
&\Uparrow_{\mathsf{D}}\ \ tr\ q\ t = tr\ q\ (fmap\ (Re \circ)\ t)
\end{aligned}$$

For example, the definition of the substitution transducer now looks like this:

$$\begin{aligned}
&trans_{\mathsf{subst}} :: Trans'_{\mathsf{D}}\ Sig\ (Map\ \ Var\ \mu Sig)\ Sig \\
&trans_{\mathsf{subst}}\ m\ (Var\ v) \quad = \mathbf{case}\ m\,[v]\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\quad Nothing \to iVar\ v \\
&\qquad\qquad\qquad\qquad\quad Just\ t \to toFree\ t \\
&trans_{\mathsf{subst}}\ m\ (Let\ v\ b\ s) = iLet\ v\ (b\ m)\ (s\ (m \setminus v)) \\
&trans_{\mathsf{subst}}\ m\ (Val\ n) \quad = iVal\ n \\
&trans_{\mathsf{subst}}\ m\ (Add\ x\ y) \quad = x\ m\ `iAdd`\ y\ m \\
&subst :: Map\ \ Var\ \mu Sig \to \mu Sig \to \mu Sig \\
&subst = [\![\Uparrow_{\mathsf{D}}\ \ trans_{\mathsf{subst}}]\!]_{\mathsf{D}}
\end{aligned}$$

In the following section we shall see that the transducer type $Trans_{\mathsf{M}}$ represents a well-studied extension of top-down (and bottom-up) tree transducers, namely macro tree transducers.

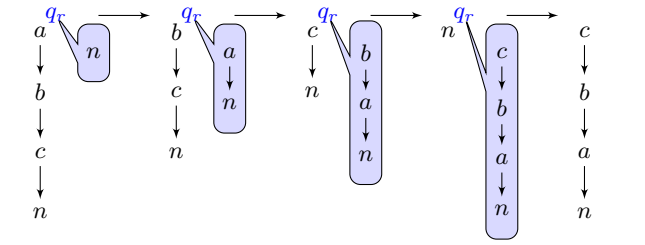


Figure 3. Run of an MTT.

4. Macro Tree Transducers

The idea behind macro tree transducers (MTTs) [11] is to allow passing accumulation parameters along with the state. A simple example of using an accumulation parameter is the implementation of the reversal of list. For simplicity we encode lists as terms over the signature $\mathcal{F} = \{a/1, b/1, c/1, n/0\}$, where a, b and c are the list elements and n is the empty list. For example the list containing the elements a, b and c is represented by the term $a(b(c(n)))$.

The reversal transformation is implemented using a single state q_r with a single accumulation parameter. Accumulation parameters are added to states by simply increasing their arities. In DTTs, states are represented by unary function symbols. For MTTs, we increment the arity for each additional accumulation parameter. In the case of the reversal transformation, q_r has arity two. The elements of the list are put into the accumulator one by one, and once the end of the original list is reached, the accumulator holds the reversal of the list.

$$\begin{aligned}
q_r(a(x), y) &\to q_r(x, a(y)) \\
q_r(b(x), y) &\to q_r(x, b(y)) \\
q_r(c(x), y) &\to q_r(x, c(y)) \\
q_r(n, y) &\to y
\end{aligned}$$

The semantics of MTTs is similar to the semantics of DTTs: simply apply the rules as rewrite rules. For example starting with the term $a(b(c(n)))$, state q_r and accumulator n, we obtain the following run:

$$\begin{aligned}
q_r(a(b(c(n))), n) &\to q_r(b(c(n)), a(n)) \to q_r(c(n), b(a(n))) \\
&\to q_r(n, c(b(a(n)))) \to c(b(a(n))).
\end{aligned}$$

A graphical representation of this run is given in Figure 3. Again we depict the states as annotations rather than nodes in the tree. The difference compared to DTTs is that the each state has a fixed number of memory slots (viz. the accumulation parameters) in which to store trees. In this example, the state q_r has a single memory slot, which is depicted as a blue box in the figure.

As the result of running the transducer on $a(b(c(n)))$, we obtain the reversal $c(b(a(n)))$.

In general, macro tree transducers are defined as follows: A macro tree transducer (MTT) from signature $\mathcal{F}$ to signature $\mathcal{G}$

$$\frac{1 \leq i \leq m}{y_i \in RHS_{n,m}} \qquad \frac{g/k \in \mathcal{G} \quad u_1, \ldots, u_k \in RHS_{n,m}}{g(u_1, \ldots, u_k) \in RHS_{n,m}}$$

$$\frac{1 \leq i \leq n \quad p/(k+1) \in Q \quad u_1, \ldots, u_k \in RHS_{n,m}}{p(x_i, u_1, \ldots, u_k) \in RHS_{n,m}}$$

Figure 4. Inductive definition of $RHS_{n,m}$.

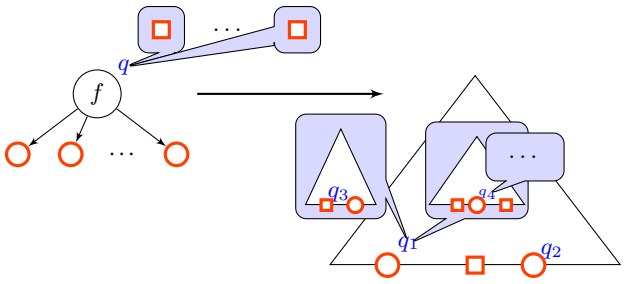


Figure 5. Macro tree transduction rule.

consist of a signature Q of states such that each $q/n \in Q$ has arity $n > 0$ and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n), y_1, \ldots, y_m) \to u \qquad \begin{array}{r}\text{for each } f/n \in \mathcal{F} \\ \text{and } q/(m+1) \in Q\end{array}$$

where u is an element of the set $RHS_{n,m}$, which is inductively defined by the formation rules in Figure 4. In short, the right-hand side may contain accumulation variables y_i, a function symbol application $g(u_1, \ldots, u_k)$ or input variables x_i guarded by a successor state p with appropriate accumulation arguments $u_1, \ldots u_k$.

The shape of the transduction rules of MTTs is depicted in Figure 5. It is instructive to compare this depiction with the depiction of DTT transduction rules in Figure 1. The basic shape of a MTT transduction rule is similar to the DTT transduction rules. The only aspect that changes is that states have more structure – each state has a fixed number of "memory cells" in which accumulation parameters can be stored and retrieved. Accumulation variables are indicated by orange squares. They appear in the state on the left-hand side and can be used in building trees on the right-hand side. The difference compared to the input variables (indicated by orange circles as for DTTs) is that accumulation variables are not guarded by a successor state. Moreover, each state on the right-hand side may also have a number of "memory cells", which must be filled with trees as well. These trees have the same structure again, containing input variables guarded by states as well as accumulation variables. This nesting of trees can be arbitrarily deep, thus allowing an arbitrarily deeply nested recursion.

Similarly to DTT transduction rules, when applying MTT transduction rules the placeholder variables (both the input variables and the accumulation variables) are instantiated with concrete trees.

With the type $Trans_{\mathsf{M}}$ we are able to represent MTTs. As for DTTs, the signatures $\mathcal{F}$ and $\mathcal{G}$ are represented as Haskell functors. In the same way also the state space is represented as a functor.

```
data F a = A a | B a | C a | N
data Q a = Qr a
```

The representation of the input signature is straightforward. Since the original MTT only has one state q, the type Q also has only one constructor. The constructor Qr has one argument as the state q_r in the original MTT has one accumulation parameter.

For the definition of the MTT in Haskell we again assume appropriate smart constructors for the target signature; in this case iA, iB, iC and iN:

```
trans_rev :: Trans'_M F Q F
trans_rev (Qr y) (A x) = x (Qr (iA y))
trans_rev (Qr y) (B x) = x (Qr (iB y))
trans_rev (Qr y) (C x) = x (Qr (iC y))
trans_rev (Qr y) N     = y
```

The above definition is a one to one translation of the transduction rules of the original MTT. We thus obtain the implementation of reverse as follows:

```
reverse :: μF → μF
reverse = ⟦⇑M trans_rev⟧M (Qr iN)
```

5. Annotating Trees

Before discussing the aspect of compositionality of MTTs we give a brief example that illustrates the flexibility that the use of MTTs affords us.

When working with abstract syntax trees (AST), e.g. in a compiler, it is common that the that the AST produced by the parser has addition annotations attached to each node. An example would be annotations that attach to each node the location in a source file from which the node originated. This information can then be used for giving useful error and warning messages.

We can easily extend a functor f to a functor an :&: f that contains annotations of type an:

```
data (an :&: f) a = an :&: (f a)
```

Trees of type $\mu(an :\&: f)$ are like trees of type μf, but additionally each node is annotated with an annotation of type a.

However, many operations on ASTs simply ignore the annotation information as it is not needed. The structure of transducers makes it quite easy to lift them from an arbitrary source signature to a source signature with annotations.

```
ignoreAnn :: (Functor g, Functor q, Functor f) ⇒
  Trans_M f q g → Trans_M (an :&: f) q g
ignoreAnn tr q (_ :&: t) = tr q t
```

In other cases, the annotations should be in principle ignored but nevertheless propagated to the result. For example, it we want to use our inlining transformation from Section 3.3 in a compiler we may want to extend it such that annotations are preserved. We have shown previously [2] that such a lifting can be performed generically for rather simple transformations known as tree homomorphisms[1], which for example occur in the form of *desugaring* transformations. The same idea can also be applied to MTTs. To this end, we need a function that adds a given annotation to every node in a tree:

```
addAnn :: Functor f ⇒ an → f* a → (an :&: f)* a
addAnn an (In t) = In (an :&: fmap (addAnn an) t)
addAnn _ (Re x) = Re x
```

This transformation can in fact be described as a DTTs with a singleton state space. We can then use this function to construct the lifting of MTTs such that they propagate annotations:

```
propAnn :: (Functor g, Functor q, Functor f) ⇒
  Trans_M f q g → Trans_M (an :&: f) q (an :&: g)
propAnn tr q (an :&: t) = addAnn an
                            (tr q (fmap addAnn' t))
  where addAnn' f q' = f (fmap (addAnn an) q')
```

[1] Tree homomorphisms are simply DTTs with a singleton state space

As expected we simply use $addAnn$ to annotate the result of the original MTT tr with the annotation an. However, due to the natrue of MTTs, we have to also propagate the annotaion to the trees that are put *into* accumulation parameters. This task is achieved by the $addAnn'$ function.

Using the thus defined lifting function we can derive the follwing variant of the inlining transformation that propagates annotaions faithfully:

$$inlineAnn :: \mu(an :\&: Sig) \to \mu(an :\&: Sig)$$
$$inlineAnn = [\![propAnn\ (\Uparrow_{\mathsf{M}}\ trans_{\mathsf{inline}})]\!]_{\mathsf{M}}\ \emptyset$$

We close this section with a small example that shows how to generically annotate a tree with unique labels so as to track changes made by subsequent transformations. In particular, we shall annotate each node with a list of type $[Int]$ that describes the path one has to take from the root in order to reach the node. For example, in the tree representing the term $iLet$ `"x"` $(iVal\ 1\ `iAdd`\ iVal\ 2)$ $(iVar$ `"x"` $`iAdd`\ iVar$ `"x"`$)$, the node representing the subterm $iVal\ 2$ has the path $[0, 1]$. We do this using the type class $Traversable$, which for each instance $Traversable\ t$ provides the method

$$mapM :: Monad\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)$$

generalising the $mapM$ function on lists to other functors. Instances of $Traversable$ can be derived mechanically in GHC, in particular for the class of polynomial functors that we consider here. Using this generalised $mapM$ function we can use a state monad to number the arguments of each constructor of a (traversable) functor:

$$number :: Traversable\ f \Rightarrow f\ a \to f\ (Int, a)$$
$$number\ x = fst\ (runState\ (mapM\ run\ x)\ 0)\ \mathbf{where}$$
$$\quad run\ b = \mathbf{do}\ n \leftarrow get; put\ (n+1); return\ (n, b)$$

For example, applied to $Let\ v\ r\ e$, the function $number$ produces $Let\ v\ (0, r)\ (1, e)$.

Equipped with this tool, we can finally define the DTT that performs the desired transformation

$$trans_{\mathsf{path}} :: Traversable\ f \Rightarrow Trans_{\mathsf{D}}\ f\ [Int]\ ([Int] :\&: f)$$
$$trans_{\mathsf{path}}\ q\ t = inFree\ (q :\&: fmap\ (\lambda(n, s) \to s\ (n : q))$$
$$\qquad\qquad (number\ t))$$
$$inFree :: Functor\ f \Rightarrow f\ a \to f^*\ a$$
$$inFree\ t = In\ (fmap\ Re\ t)$$

We thus arrive at the following transformation function, which adds path annotations to any tree:

$$pathAnn :: Traversable\ f \Rightarrow \mu f \to \mu([Int] :\&: f)$$
$$pathAnn = [\![trans_{\mathsf{path}}]\!]_{\mathsf{D}}\ [\,]$$

In the next section, we shall see how to compose tree transducers, which will allow us to combine $trans_{\mathsf{path}}$ with other transducers in order to track the transformations they perform.

6. Composition of Tree Transducers

In this section we demonstrate the compositionality of tree transducers in Haskell. Before we move to composition of MTTs, we have a look at the composition of DTTs.

6.1 Composition of DTTs

One of the first results in the theory on tree transducers was that the sequential composition of two transformations expressed as DTTs is itself expressible as a DTT and this DTT can be effectively constructed from the two original DTTs [31]. The idea behind this construction is quite simple: given two DTTs $\mathcal{A}_1$ (from $\mathcal{F}$ to $\mathcal{G}$) and $\mathcal{A}_2$ (from $\mathcal{G}$ to $\mathcal{H}$) we construct a DTT $\mathcal{A}$ (from $\mathcal{F}$ to $\mathcal{H}$) that performs the composition of $\mathcal{A}_2$ and $\mathcal{A}_1$ by taking all the transduction rules from $\mathcal{A}_1$ and transform their right-hand sides by running the DTT $\mathcal{A}_2$ on them. That is for each rule

$$q_1(f(x_1, \ldots, x_n)) \to u \quad \text{in } \mathcal{A}_1$$

and state p in $\mathcal{A}_2$, we produce the rule

$$(q_1, q_2)(f(x_1, \ldots, x_n)) \to [\![\mathcal{A}_2]\!]\ (q_2, u)$$

Note that the state space of $\mathcal{A}$ is the product $Q_1 \times Q_2$ of the state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$. The additional state information q_2 is used as the initial state for running the transducer $\mathcal{A}_2$ on the right-hand side u. Technically, the transducer $\mathcal{A}_2$ has to be modified slightly since the term u may contain states from $\mathcal{A}$. This is done by adding rules of the form

$$q_2(q_1(x)) \to (q_1, q_2)(x)$$

to $\mathcal{A}_2$, which simply combines the states from the two transducers to get a state in the compound state space $Q_1 \times Q_2$.

The construction in Haskell follows the same idea. In order to simplify the presentation and the subsequent correctness proof we shall reformulate the definition of the DTT semantics $[\![\cdot]\!]_{\mathsf{D}}$ as a fold of an algebra constructed as follows:

$$alg_{\mathsf{D}} :: Functor\ g \Rightarrow Trans_{\mathsf{D}}\ f\ q\ g \to Alg\ f\ (q \to g^*\ a)$$
$$alg_{\mathsf{D}}\ tr\ t\ q = join\ (tr\ q\ t)$$
$$[\![\cdot]\!]_{\mathsf{D}} :: (Functor\ f, Functor\ g) \Rightarrow$$
$$\qquad Trans_{\mathsf{D}}\ f\ q\ g \to q \to \mu f \to \mu g$$
$$[\![tr]\!]_{\mathsf{D}}\ q\ t = fold_\mu\ (alg_{\mathsf{D}}\ tr)\ t\ q$$

The function alg_{D} turns DTT transduction function into the algebra that describes its semantics. The semantics $[\![\cdot]\!]_{\mathsf{D}}$ is then simply the fold of this algebra. Using the same algebra, we can easily generalise the semantics $[\![\cdot]\!]_{\mathsf{D}}$ in a way that corresponds to the addition of rules of the form $q_2(q_1(x)) \to (q_1, q_2)(x)$ as described above:

$$(\!|\cdot|\!)_{\mathsf{D}} :: (Functor\ f, Functor\ g) \Rightarrow$$
$$\quad Trans_{\mathsf{D}}\ f\ q\ g \to q \to f^*\ (q \to a) \to g^*\ a$$
$$(\!|tr|\!)_{\mathsf{D}}\ q\ t = fold_*\ (\lambda a\ q \to Re\ (a\ q))\ (alg_{\mathsf{D}}\ tr)\ t\ q$$

This semantics generalises $[\![\cdot]\!]_{\mathsf{D}}$ in the sense that it works on $f^*\ (q \to a)$ not only μf. In analogy to the automata theoretic construction sketched above, we will use $(\!|\cdot|\!)_{\mathsf{D}}$ in order to run a DTT on the right-hand side of the transduction rules of another DTT. The composition of DTTs is thus constructed as follows:

$$\cdot \circ_{\mathsf{D}} \cdot ::(Functor\ f, Functor\ g, Functor\ h) \Rightarrow$$
$$\quad Trans_{\mathsf{D}}\ g\ q_2\ h \to Trans_{\mathsf{D}}\ f\ q_1\ g \to Trans_{\mathsf{D}}\ f\ (q_1, q_2)\ h$$
$$tr_2 \circ_{\mathsf{D}} tr_1\ (q_1, q_2)\ t = (\!|tr_2|\!)_{\mathsf{D}}\ q_2\ (tr_1\ q_1\ (fmap\ curry\ t))$$

The right-hand side of the first DTT tr_1 is obtained by applying it to its component q_1 of the compound state space as well as the input pattern t. Note that we have to apply currying in order to transform the input pattern for the compound DTT, which is of type $f\ ((q_1, q_2) \to a)$, into the input pattern of the first DTT, which has to be of type $f\ (q_1 \to q_2 \to a)$. After this application, the second DTT tr_2 is run on the thus obtained right-hand side of a rule of the first DTT.

This construction is exactly the same as the automata-theoretic construction as first shown by Rounds [31]. Since the accompanying soundness proof of Rounds does not make use of the finiteness of the state space, it follows that that also our composition construction is sound. That is, we have the following theorem:

Theorem 1 (Soundness of DTT composition)**.** *For all tr_1, tr_2, q_1, q_2, t of appropriate type, we have the following equality:*

$$[\![tr_2]\!]_{\mathsf{D}}\ q_2\ ([\![tr_1]\!]_{\mathsf{D}}\ q_1\ t) = [\![tr_2 \circ_{\mathsf{D}} tr_1]\!]_{\mathsf{D}}\ (q_1, q_2)\ t$$

However, we can give a rather compact direct proof of this soundness theorem without referring the automata-theoretic proof.

Proof of Theorem 1. Let a_1, a_2 and a be the algebras of tr_1, tr_2 and $tr_2 \circ_{\mathsf{D}} tr_1$ according to alg_{D}. Unfolding the definition of $[\![\cdot]\!]_{\mathsf{D}}$, we can reformulate the claimed equality as follows:

$$f\ (fold_\mu\ a_1\ t)\ (q_1, q_2) = fold_\mu\ a\ t\ (q_1, q_2) \tag{1}$$

where f is defined by the equation

$$f\ g\ (q_1, q_2) = fold_\mu\ a_2\ (g\ q_1)\ q_2$$

By the fusion law for folds it suffices to prove the following equality instead:

$$\forall\ x\ .\ fold_\mu\ a_2\ (a_1\ x\ q_1)\ q_2 = a\ (fmap\ f\ x)\ (q_1, q_2) \tag{2}$$

In order to prove this, we take advantage of the observation that $join$ commutes with $fold_*$:

Lemma 1. *Let $e = \lambda z\ q \to Re\ (z\ q)$ and $b = alg_{\mathsf{D}}\ tr$ for some tr. Then the following holds for all x and q:*

$$fold_\mu\ b\ (join\ x)\ q = join\ (fold_*\ e\ b\ (fmap\ (fold_\mu\ b)\ x)\ q)$$

We omit the proof of this lemma here, however it can be found in the associated material on the authors' websites. The proof of (2) follows:

$$\begin{aligned}
& a\ (fmap\ f\ x)\ (q_1, q_2) \\
= \quad & \{\ a = alg_{\mathsf{D}}\ (tr_2 \circ_{\mathsf{D}} tr_1) \text{ and definition of } alg_{\mathsf{D}}\ \} \\
& join\ ((tr_2 \circ_{\mathsf{D}} tr_1)\ (q_1, q_2)\ (fmap\ f\ x)) \\
= \quad & \{\ \text{Definition of } \cdot \circ_{\mathsf{D}} \cdot \text{ and } (\!|\cdot|\!)_{\mathsf{D}}\ \} \\
& join\ (fold_*\ e\ a_2\ (tr_1\ q_1\ (fmap\ (curry \circ f)\ x))\ q_2) \\
= \quad & \{\ curry \circ f = \lambda g\ q_1\ q_2 \to fold_\mu\ a_2\ (g\ q_1)\ q_2\ \} \\
& join\ (fold_*\ e\ a_2\ (tr_1\ q_1\ (fmap\ (fold_\mu\ a_2 \circ)\ x))\ q_2) \\
= \quad & \{\ \text{Parametricity}\ \} \\
& join\ (fold_*\ e\ a_2\ (fmap\ (fold_\mu\ a_2)\ (tr_1\ q_1\ x))\ q_2) \\
= \quad & \{\ \text{Lemma 1}\ \} \\
& fold_\mu\ a_2\ (join\ (tr_1\ q_1\ x))\ q_2 \\
= \quad & \{\ a_1 = alg_{\mathsf{D}}\ tr_1 \text{ and definition of } alg_{\mathsf{D}}\ \} \\
& fold_\mu\ a_2\ (a_1\ x\ q_1)\ q_2
\end{aligned}$$

The equation labelled "Parametricity" follows from the equality

$$\forall\ g\ q\ .\ fmap\ g\ (tr\ q) = tr\ q\ .\ fmap\ (g \circ)$$

which holds for all tr of type $Trans_{\mathsf{D}}\ f\ q\ g$ because of parametricity. □

Note that in the application of the fusion law for folds we make use of the set theoretic semantics. In a CPO semantics, the function f would also need to satisfy a strictness side condition for the fusion law [29]. This strictness condition does in fact not hold for f. However, this should be expected since the closure of DTTs under composition is restricted to *total*, deterministic DTTs. Indeed, Rounds [31] gives a counterexample for the composition with a deterministic DTT that is not total.

6.2 Composition of MTTs

Unlike DTTs, MTTs are in general not closed under composition [11]. That is, a transformation that is expressible as the composition of two MTT-expressible transformations cannot always be expressed as a single MTT. However, for a large class of cases the composition of two MTTs does again yield an MTT. In particular, the composition of an MTT with a DTT (and vice versa) can be expressed as a single MTT, which can be effectively constructed [11].

Similarly, to the case for DTTs described above, we shall reformulate the semantics of MTTs as a fold:

$$\begin{aligned}
& alg_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
& \qquad Trans_{\mathsf{M}}\ f\ q\ g \to Alg\ f\ (q\ (g^*\ a) \to g^*\ a) \\
& alg_{\mathsf{M}}\ tr\ t\ q = join\ (tr\ q\ (fmap\ (\circ fmap\ join)\ t)) \\
& [\![\cdot]\!]_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
& \qquad Trans_{\mathsf{M}}\ f\ q\ g \to q\ \mu g \to \mu f \to \mu g \\
& [\![tr]\!]_{\mathsf{M}}\ q\ t = fold_\mu\ (alg_{\mathsf{M}}\ tr)\ t\ q
\end{aligned}$$

As before we also give a generalised semantics that builds upon the same algebra construction alg_{M}:

$$\begin{aligned}
& (\!|\cdot|\!)_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
& \quad Trans_{\mathsf{M}}\ f\ q\ g \to q\ (g^*\ a) \to f^*\ (q\ (g^*\ a) \to a) \to g^*\ a \\
& (\!|tr|\!)_{\mathsf{M}}\ q\ t = fold_*\ (\lambda a\ q \to Re\ (a\ q))\ (alg_{\mathsf{M}}\ tr)\ t\ q
\end{aligned}$$

We first consider the simpler case of the composition of MTTs with DTTs, viz. a DTT followed by an MTT. To this end we shall use the direct construction given by Kühnemann [26]. This construction follows essentially the same idea as the composition construction for DTTs described in Section 6.1: we take the rules of the first transducer and transform their right-hand sides by running the second transducer on them. The only difference is that since we are dealing with an MTT, the state space of one of the transducer is a functor. Thus we have to replace ordinary pairing with the following type constructor:

$$\mathbf{data}\ (q_1\ :\&:\ q_2)\ a = q_1\ :\&:\ (q_2\ a)$$

We have already seen this construction in Section 5 where it was used for adding annotations to trees.

The definition of the composition operator $\cdot \circ_{\mathsf{MD}} \cdot$ then follows the same pattern as $\cdot \circ_{\mathsf{D}} \cdot$:

$$\begin{aligned}
& \cdot \circ_{\mathsf{MD}} \cdot :: (Functor\ f, Functor\ g, Functor\ h, Functor\ p) \Rightarrow \\
& \quad Trans_{\mathsf{M}}\ g\ p\ h \to Trans_{\mathsf{D}}\ f\ q\ g \to Trans_{\mathsf{M}}\ f\ (q\ :\&:\ p)\ h \\
& tr_2 \circ_{\mathsf{MD}} tr_1\ (q_1\ :\&:\ q_2)\ t = (\!|tr_2|\!)_{\mathsf{M}}\ (fmap\ Re\ q_2) \\
& \qquad\qquad (tr_1\ q_1\ (fmap\ curryF\ t))
\end{aligned}$$

where $curryF$ is a variant of $curry$ defined on the :&: type constructor instead:

$$\begin{aligned}
& curryF :: ((q\ :\&:\ p)\ a \to b) \to q \to p\ a \to b \\
& curryF\ f\ q\ p = f\ (q\ :\&:\ p)
\end{aligned}$$

Since the construction given by $\cdot \circ_{\mathsf{MD}} \cdot$ follows the corresponding automata-theoretic construction and since the accompanying soundness proof [11] does not use the finiteness of the state space, we can conclude the soundness of our construction:

Theorem 2 (Soundness of MTT-DTT composition)**.** *For all tr_1, tr_2, q_1, q_2, t of appropriate type, we have the following equality:*

$$[\![tr_2 \circ_{\mathsf{MD}} tr_1]\!]_{\mathsf{M}}\ (q_1\ :\&:\ q_2)\ t = [\![tr_2]\!]_{\mathsf{M}}\ q_2\ ([\![tr_1]\!]_{\mathsf{D}}\ q_1\ t)$$

Example 4. As a simple example, we can combine the inlining MTT with the DTT that adds annotations:

$$\begin{aligned}
& trans :: Trans_{\mathsf{M}}\ Sig\ ([Int]\ :\&:\ (Map\ Var))\ ([Int]\ :\&:\ Sig) \\
& trans = (propAnn\ (\Uparrow_{\mathsf{M}}\ trans_{\mathsf{inline}})) \circ_{\mathsf{MD}} trans_{\mathsf{path}}
\end{aligned}$$

Here we use $propAnn$ to lift the original inlining MTT to propagate annotations. We then get the following transformation function:

$$\begin{aligned}
& inlineTrack :: \mu Sig \to \mu([Int]\ :\&:\ Sig) \\
& inlineTrack = [\![trans]\!]_{\mathsf{M}}\ ([]\ :\&:\ \emptyset)
\end{aligned}$$

Finally, we consider the composition of an MTT followed by a DTT. This case is slightly more complicated. It is not sufficient to simply run the DTT on the right-hand sides of the rules of

the MTT. In the transduction rules of an MTT, the accumulation variables are used directly on the right-hand side without being guarded by a successor state. The reason for this is the fact that the trees that are stored in the accumulation parameters have already been transformed by the MTT. Hence, in order to achieve the same invariance for the composition of an MTT followed by a DTT, we have to make sure to also run the DTT on the trees in the accumulation parameters.

In the original automata-theoretic construction of Engelfriet and Vogler [11], this nested run of the DTT is achieved by replacing each accumulation parameter of the original MTT with k new ones, where k is the number of states in the DTT. The rules of the MTT are then manipulated such that the i-th copy of an accumulation parameter contains the transformation of the original accumulation parameter by running the DTT on it using the i-th state as a starting state.

This construction makes direct use of the fact that the state space of the DTT is finite as each state with n accumulation parameters in the MTT is replaced by a state with $n * k$ accumulation parameters. However, we can apply the same construction for our generalised transducers with potentially infinite state spaces. The reason for this is that our representation of MTTs does not have the restriction to a finite number of accumulation parameters. Given the state space q_1 of an MTT, we can achieve the construction of producing a copy of each accumulation parameter for each state in the state space q_2 of the DTT with the type $Exp\ q_1\ q_2$ defined as follows:

$$\mathbf{type}\ Exp\ q_1\ q_2\ a = q_1\ (q_2 \to a)$$

The nested function type makes sure that there is a copy of each accumulation parameter for each state in q_2.

We then obtain the state space of the compound transducer by pairing the above exponentiated state space with the state space q_2 of the DTT, which yields the type $q_2\ :\&:\ Exp\ q_1\ q_2$. We combine this construction in a single definition as follows:

$$\mathbf{data}\ (q_1 :\wedge: q_2)\ a = q_1\ (q_2 \to a) :\wedge: q_2$$

We thus arrive at the following construction of the composition of DTTs and MTTs:

$$\begin{aligned}
&\cdot \circ_{\mathsf{DM}} \cdot :: (Functor\ f, Functor\ g, Functor\ h, Functor\ q) \Rightarrow \\
&\quad Trans_{\mathsf{D}}\ g\ p\ h \to Trans_{\mathsf{M}}\ f\ q\ g \to Trans_{\mathsf{M}}\ f\ (q :\wedge: p)\ h \\
&tr_2 \circ_{\mathsf{DM}} tr_1\ (q_1 :\wedge: q_2)\ t = \\
&\quad (\!|tr_2|\!)_{\mathsf{D}}\ q_2\ (tr_1\ (fmap\ (\$)\ q_1)\ (fmap\ (nested\ tr_2)\ t))
\end{aligned}$$

In this case simple currying is not sufficient for preparing the input pattern t. As explained above, we have to run the DTT also on the accumulation parameters. The function $nested$ does exactly that:

$$nested\ tr\ f\ q_1\ q_2 = f\ (fmap\ (\lambda s\ p \to (\!|tr|\!)_{\mathsf{D}}\ p\ s)\ q_1 :\wedge: q_2)$$

To this end the function $nested$ makes use of the fact that we expand the state space using exponentiation. Thus, we can run the DTT on the tree s using the supplied state p as initial state of the run.

Again, we appeal to the automata-theoretic proof of Engelfriet and Vogler [11] to conclude that the above composition operator $\cdot \circ_{\mathsf{DM}} \cdot$ is sound:

Theorem 3 (Soundness of MTT-DTT composition)**.** *For all tr_1, tr_2, q_1, q_2, t of appropriate type, we have the following equality:*

$$[\![tr_2 \circ_{\mathsf{DM}} tr_1]\!]_{\mathsf{M}}\ (q'_1 :\wedge: q_2)\ t = [\![tr_2]\!]_{\mathsf{D}}\ q_2\ [\![tr_1]\!]_{\mathsf{M}}\ q_1\ t$$

where $q'_1 = fmap\ (\lambda s\ q \to [\![tr_2]\!]_{\mathsf{D}}\ q\ s)\ q_1$

In phrasing the soundness theorem for $\cdot \circ_{\mathsf{D}} \cdot$ we have to perform the same manipulation on the states of the MTT as in the definition of $\cdot \circ_{\mathsf{D}} \cdot$ itself: we have to run the DTT on the accumulation parameters, thus transforming q_1 into q'_1 akin to the transformation in $nested$.[2]

Note that the three theorems stating the soundness of the composition of tree transducers can be read as fusion rules. Read from right to left, these rules describe how to fuse two transducer runs into a single one. Using GHC's rewrite rules facility [20], these fusion rules can be implemented as optimisation rules in GHC right away.

7. Macro Tree Transducers with Regular Look-Ahead

MTTs are quite expressive, more expressive than one might consider after a first glance. As illustrated in Section 3 and 4, MTT emerge as a generalisation of DTT. As such, MTTs also retain the top-down flow of state information from DTTs. Naturally, there is also bottom-up version of DTTs, unsurprisingly called *bottom-up tree transducers* (UTTs). The expressive power of DTTs and UTTs is incomparable [9], i.e. there are transformations that can be expressed using one of the two kinds of transducer but not the other. Since MTTs generalise DTTs they can express any transformation expressible as a DTT. Surprisingly, however, MTTs can also express all transformations expressible as a UTTs[3] [11].

The underlying reason is that MTTs are closed under addition of *regular look-ahead* [11]: roughly speaking, an MTT with regular look-ahead (MTTR) is an MTT which has access to additional information that is provided by a second automaton that propagates states bottom-up. Intuitively, the run of such an MTTR can be thought of as a two-phase computation: in the first phase, the bottom-up tree automaton is run, annotating each node of the tree with the state that it computes for it. Afterwards, in the second phase, the MTT is run on this annotated tree and has therefore access to the state information that was provided by the bottom-up automaton in the first phase.

7.1 An Example

To illustrate why such a look-ahead is useful, we reconsider the inlining transformation from Section 3. We want to make this transformation a bit smarter by only performing the inlining if the bound variable occurs exactly once in the scope of the let binding. That means, when transforming a let binding, we have to first compute the number of occurrences of the bound variable in the scope of the let binding. This counting of variable occurrences is a bottom-up computation.

Bottom-up tree automata (UTA), also know as bottom-up tree acceptors, only propagate state information (upwards) without performing a transformation. In Haskell such an automaton is represented by the following type:

$$\mathbf{type}\ State_{\mathsf{U}}\ f\ q = f\ q \to q$$

The alert reader will notice that this is exactly the type for f-algebras with carrier type q. And indeed running a UTA on a term is simply the fold of this algebra:

$$\begin{aligned}
&[\![\cdot]\!]^{\mathsf{S}}_{\mathsf{U}} :: Functor\ f \Rightarrow State_{\mathsf{U}}\ f\ q \to \mu f \to q \\
&[\![st]\!]^{\mathsf{S}}_{\mathsf{U}} = fold_{\mu}\ st
\end{aligned}$$

The UTA that counts free occurrences of variables is then defined as follows:

[2] Technically, the initial state of a tree transducer is part of the transducer itself. Thus, the construction of the initial state $(q'_1 :\wedge: q_2)$ for the compound transducer is actually part of the composition construction.

[3] Both for the case of unrestricted transducers, and for the case of total, deterministic transducers, which we are interested in here.

$$
\begin{aligned}
&St_{\mathsf{numFV}} :: State_{\mathsf{U}}\ Sig\ (Map\ Var\ Int)\\
&St_{\mathsf{numFV}}\ (Add\ x\ y) \;= x \oplus y\\
&St_{\mathsf{numFV}}\ (Val\ \_) \;= \emptyset\\
&St_{\mathsf{numFV}}\ (Let\ v\ x\ y) = x \oplus (y \setminus v)\\
&St_{\mathsf{numFV}}\ (Var\ v) \;= \{v \mapsto 1\}
\end{aligned}
$$

where $\oplus$ adds up the variable counts of two mappings:

$$
\begin{aligned}
&\cdot \oplus \cdot :: Ord\ a \Rightarrow Map\ a\ Int \to Map\ a\ Int \to Map\ a\ Int\\
&x \oplus y = Map.unionWith\ (+)\ x\ y
\end{aligned}
$$

But we do not want to actually run this automaton on its own; we want to combine it with a variant of an MTT that can read the state information provided by a UTA. Let us step back and have a look at the type representing MTTs again:

$$
\begin{aligned}
&\mathbf{type}\ Trans_{\mathsf{M}}\ f\ q\ g = \forall\ a\ .\ q\ a \to\\
&\qquad\qquad f\ (q\ (g^*\ a) \to a) \to g^*\ a
\end{aligned}
$$

The type of MTTs with regular look-ahead (MTTRs) has an additional type variable p that refers to the state space of the UTA that provides the bottom-up state information:

$$
\begin{aligned}
&\mathbf{type}\ Trans^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g = \forall\ a\ .\ q\ a \to\\
&\qquad\qquad f\ (q\ (g^*\ a) \to a, p) \to g^*\ a
\end{aligned}
$$

The type p is added in only one place, viz. paired with the type of input variables. This gives the transduction rules the ability to observe the state of the successor nodes of the current node. Before we consider the semantics, let us look at an example. We shall now implement the smart inline translation that only inlines let bindings whose variable occurs exactly once in its scope:

$$
\begin{aligned}
&trans_{\mathsf{smart}} :: Trans'^{\mathsf{L}}_{\mathsf{M}}\ Sig\ (Map\ Var)\ (Map\ Var\ Int)\ Sig\\
&trans_{\mathsf{smart}}\ m\ (Var\ v) = \mathbf{case}\ m\,[v]\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad Nothing \to iVar\ v\\
&\qquad\qquad\qquad\qquad Just\ e \;\;\to e\\
&trans_{\mathsf{smart}}\ m\ (Let\ v\ (x, \_)\ (y, yvars)) =\\
&\qquad\quad \mathbf{case}\ Map.findWithDefault\ 0\ v\ yvars\ \mathbf{of}\\
&\qquad\quad\ 0 \to y\ m\\
&\qquad\quad\ 1 \to y\ (m\,[v \mapsto x\ m])\\
&\qquad\quad\ \_ \to iLet\ v\ (x\ m)\ (y\ m)\\
&trans_{\mathsf{smart}}\ m\ (Val\ n) \qquad\qquad\quad = iVal\ n\\
&trans_{\mathsf{smart}}\ m\ (Add\ (x, \_)\ (y, \_)) = x\ m\ `iAdd`\ y\ m
\end{aligned}
$$

The transducer coincides with the simpler original $trans_{\mathsf{inline}}$ in all cases except for Let. In the Let case the transducer consults the bottom-up state $yvars$ of the y argument, which contains the numbers of occurrences of free variables in the term represented by y. Depending on how many times the variable v occurs, the transducer simply removes the let binding, performs inlining or performs no transformation at all.

Note that in the above definition, we did not use the type $Trans^{\mathsf{L}}_{\mathsf{M}}$ but instead the type $Trans'^{\mathsf{L}}_{\mathsf{M}}$, which lets us avoid using explicit application of Re by packaging "naked" occurrences of a into $g^*\ a$:

$$
\begin{aligned}
&\mathbf{type}\ Trans'^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g = \forall\ a\ .\ q\ (g^*\ a) \to\\
&\qquad\qquad f\ (q\ (g^*\ a) \to g^*\ a, p) \to g^*\ a
\end{aligned}
$$

As for previous transducer types, also this convenience type can be lifted to its original form by applying Re in the right places:

$$
\begin{aligned}
&\Uparrow^{\mathsf{L}}_{\mathsf{M}} :: (Functor\ q, Functor\ f) \Rightarrow\\
&\qquad Trans'^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g \to Trans^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g\\
&\Uparrow^{\mathsf{L}}_{\mathsf{M}}\ \ tr\ q\ t = tr\ (fmap\ Re\ q)\\
&\qquad\qquad\qquad (fmap\ (\lambda(f, p) \to (Re \circ f, p))\ t)
\end{aligned}
$$

7.2 The Semantics

The easiest way to verbally describe the semantics of MTTRs is via a two-phase execution as alluded to above. At first the UTA is used to annotate the input term with the bottom-up state information, and then the MTTR is run on the annotated tree. However, the implementation in Haskell is best described as a single recursive function:

$$
\begin{aligned}
&[\![\cdot, \cdot]\!]^{\mathsf{L}}_{\mathsf{M}} :: \forall\ g\ f\ q\ p\ .\ (Functor\ g, Functor\ f, Functor\ q) \Rightarrow\\
&\quad State_{\mathsf{U}}\ f\ p \to Trans^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g \to q\ \mu g \to \mu f \to \mu g\\
&[\![st, tr]\!]^{\mathsf{L}}_{\mathsf{M}}\ \ q\ t = fst\ (run\ t)\ q\ \mathbf{where}\\
&\quad run :: \mu f \to (q\ \mu g \to \mu g, p)\\
&\quad run\ (In\ t) = \mathbf{let}\ t' = fmap\ run'\ t\\
&\qquad\qquad\qquad\quad\ p\ = st\ (fmap\ snd\ t')\\
&\qquad\qquad\qquad \mathbf{in}\ (\lambda q \to join\ (tr\ q\ t'), p)\\
&\quad run' :: \mu f \to (q\ (g^*\ \mu g) \to \mu g, p)\\
&\quad run'\ t = \mathbf{let}\ (res, p) = run\ t\\
&\qquad\quad\ \mathbf{in}\ \ (res \circ fmap\ join, p)
\end{aligned}
$$

Here, we also give the types of the auxiliary functions to better illustrate what is going on. The run function performs both the transformation of the MTT and the state propagation of the UTA simultateously. Hence, the result type of the run function is $(q\ \mu g \to \mu g, p)$. The final result is obtained by projecting to the first component of the result of run.

As we have mention in the beginning of this section, MTTs with regular look-ahead can be transformed to ordinary MTTs [11]. This astonishing property of MTTs is, however, mostly of theoretical interest. The MTT that one gets from this transformation basically tries out every possible value of the state that the UTA would provide and then checks during the traversal which of the alternatives was actually the correct one. This happens at every node of the input tree. Thus, there is an exponential increase in the run-time in terms of the size of the state space of the UTA. Moreover, this construction does not work for infinite state spaces such as the state space $Map\ Var\ Int$, which we used in our example of the smart inlining transformation.

A result for MTTRs that we can apply to the Haskell implementation, however, is the compositionality result: Like MTTs, also MTTRs are closed under composition with DTTs from the right and the left.

8. Discussion

8.1 Related Work

The transformations performed by tree transducers are closely related to attribute grammars [30]. In particular, the combination of a top-down and a bottom-up flow of information found in MTTs with regular look-ahead are likewise found in attribute grammars in the form of inherited and synthesised attributes. Viera et al. [37] have shown that attribute grammars can be embedded in Haskell as a library. Attribute grammars can be translated into recursive program schemes [6, 7], which can be expressed as MTTs. Thus, MTTs subsume attribute grammars. Moreover, *attributed tree transducers* [12], a class of tree automata that has been introduced to study the properties of attribute grammars has been shown to be subsumed in expressivity by MTTs [13].

Upwards and downwards accumulations [14–16] generalise the $scanl$ and $scanr$ functions on lists to regular data types. The propagation of state information in a tree automaton follows the same mechanics. As recently pointed out by Gibbons [17], there is also a strong connection between accumulations and attribute grammars: complete attribute evaluation can be expressed as an upwards accumulation followed by a downwards accumulation. The same idea is used in the definition of MTTs with regular look-ahead.

Our work on representing tree transducers in Haskell was influenced by the category theoretic definition of bottom-up tree transducers of Hasuo et al. [18]. For their definition, the authors introduce the idea of representing the placeholder variables in transduction rules as naturality conditions. We use the same idea in our representation, merely replacing naturality with parametric polymorphism. Jürgensen [21, 22] gives a categorical definition of top-down transducers and prove the soundness of their composition.

References

[1] P. Bahr. Modular tree automata. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, volume 7342 of *Lecture Notes in Computer Science*, pages 263–299. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-31113-0_14.

[2] P. Bahr and T. Hvitved. Compositional data types. In *WGP '11*, pages 83–94. ACM, 2011.

[3] P. Bahr and T. Hvitved. Parametric compositional data types. In J. Chapman and P. B. Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–24. Open Publishing Association, 2012. doi: 10.4204/EPTCS.76.3.

[4] P. Bahr and T. Hvitved. `compdata` Haskell library, 2013. URL `http://hackage.haskell.org/package/compdata`. module `Data.Comp.MacroAutomata`.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. Draft.

[6] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I. *Theoretical Computer Science*, 17(2): 163 – 191, 1982. ISSN 0304-3975. doi: 10.1016/0304-3975(82)90003-2.

[7] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes II. *Theoretical Computer Science*, 17(3): 235 – 257, 1982. ISSN 0304-3975. doi: 10.1016/0304-3975(82)90024-X.

[8] J. Endrullis, C. Grabmayer, D. Hendriks, J. Klop, and R. de Vrijer. Proving infinitary normalization. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 64–82. Springer Berlin / Heidelberg, 2009. doi: 10.1007/978-3-642-02444-3_5.

[9] J. Engelfriet. Bottom-up and top-down tree transformations— a comparison. *Mathematical systems theory*, 9(2):198–231, 1975. ISSN 0025-5661. doi: 10.1007/BF01704020.

[10] J. Engelfriet and S. Maneth. The equivalence problem for deterministic MSO tree transducers is decidable. *Information Processing Letters*, 100(5):206 – 212, 2006. ISSN 0020-0190. doi: 10.1016/j.ipl.2006.05.015.

[11] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71 – 146, 1985. ISSN 0022-0000. doi: 10.1016/0022-0000(85)90066-2.

[12] Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.

[13] Z. Fülöp and H. Vogler. A characterization of attributed tree transformations by a subclass of macro tree transducers. *Theory of Computing Systems*, 32(6):649–676, 1999. ISSN 1432-4350. doi: 10.1007/s002240000135. URL `http://dx.doi.org/10.1007/s002240000135`.

[14] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 122–138. Springer Berlin / Heidelberg, 1993. ISBN 978-3-540-56625-0. doi: 10.1007/3-540-56625-2_11.

[15] J. Gibbons. Polytypic downwards accumulations. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 207–233. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64591-7. doi: 10.1007/BFb0054292.

[16] J. Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37(1-3):37–65, 2000. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00022-2. URL `http://www.sciencedirect.com/science/article/pii/S0167642399000222`.

[17] J. Gibbons. Accumulating attributes (for doaitse swierstra, on his retirement). In J. Hage and A. Dijkstra, editors, *Een Lawine van Ontwortelde Bomen: Liber Amicorum voor Doaitse Swierstra*, pages 87–102. 2013.

[18] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical views on computations on trees (extended abstract). In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Automata, Languages and Programming*, volume 4596 of *LNCS*, pages 619–630. Springer, 2007.

[19] H. Hosoya. *Foundations of XML Processing – The Tree-Automata Approach*. Cambridge University Press, 2010.

[20] S. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, page 203, 2001.

[21] C. Jürgensen. *Categorical semantics and composition of tree transducers*. PhD thesis, Technischen Universität Dresden, 2003.

[22] C. Jürgensen and H. Vogler. Syntactic composition of top-down tree transducers is short cut fusion. *Mathematical Structures in Computer Science*, 14(2):215–282, Apr. 2004. ISSN 0960-1295. doi: 10.1017/S0960129503004109.

[23] K. Knight and J. Graehl. An overview of probabilistic tree transducers for natural language processing. In A. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 3406 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2005. doi: 10.1007/978-3-540-30586-6_1.

[24] A. Koprowski and J. Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybernetica*, 19(2):357–392, 2009.

[25] A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and S. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, page 1046. Springer Berlin / Heidelberg, 1998. doi: 10.1007/978-3-540-49382-2_13.

[26] A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp and T. Sato, editors, *Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 114–130. Springer Berlin / Heidelberg, 1999. doi: 10.1007/10705424_8.

[27] A. Kühnemann and J. Voigtländer. Tree transducer composition as deforestation method for functional programs. Technical report, Dresden University of Technology, 2001.

[28] K. Matsuda, K. Inaba, and K. Nakano. Polynomial-time inverse computation for accumulative functions with multiple data traversals. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 5–14, New York, NY, USA, 2012. ACM. doi: 10.1145/2103746.2103752.

[29] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. doi: 10.1007/3540543961_7.

[30] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995. ISSN 0360-0300. doi: 10.1145/210376.197409.

[31] W. C. Rounds. Mappings and grammars on trees. *Mathematical systems theory*, 4(3):257–287, 1970. ISSN 0025-5661. doi: 10.1007/BF01695769.

[32] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Mathematical systems theory*, 27(4):285–346, 1994. ISSN 0025-5661. doi: 10.1007/BF01192143.

[33] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[34] J. W. Thatcher. Transformations and translations from the point of view of generalized finite automata theory. In *Proceedings of the first annual ACM symposium on Theory of computing*, STOC '69, pages

129–142, New York, NY, USA, 1969. ACM. doi: 10.1145/800169.805427.

[35] W. Thomas. Handbook of formal languages, vol. 3. chapter Languages, automata, and logic, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA, 1997. ISBN 3-540-60649-1.

[36] S. Tison. Tree automata and term rewrite systems. In L. Bachmair, editor, *Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pages 27–30. Springer Berlin Heidelberg, 2000. doi: 10.1007/10721975_2.

[37] M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming - ICFP '09*, pages 245–246, New York, New York, USA, 2009. ACM Press. ISBN 9781605583327. doi: 10.1145/1596550.1596586.

[38] J. Voigtländer. Conditions for efficiency improvement by tree transducer composition. In S. Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 57–100. Springer Berlin / Heidelberg, 2002. doi: 10.1007/3-540-45610-4_16.

[39] J. Voigtländer. Formal efficiency analysis for tree transducer composition. *Theory of Computing Systems*, 41(4):619–689, 2007. ISSN 1432-4350. doi: 10.1007/s00224-006-1235-9.

[40] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[41] Z. Ésik. Decidability results concerning tree transducers i. *Acta Cybernetica*, 5:1–20, 1981.

Implicit Parallelism and Iterative Compilation

José Manuel Calderón Trilla and Colin Runciman
Department of Computer Science
University of York
York, United Kingdom
{jmct,colin}@cs.york.ac.uk

June 6th 2013

Abstract

The benefits of writing parallel programs in functional languages has been discussed by several authors [1, 2, 3, 4, 5]. One of the key points is that by using a paradigm that discourages the use of mutable state it becomes easier to ensure that concurrent threads do not interfere with one another. Additionally, the Church-Rosser theorem shows that the path of evaluation through a program does not affect its semantics[1] [1]. Therefore, whenever there is more than one reducible expression, the order in which they are reduced does not matter.

While these facts give the community hope in the design and implementation of systems that automatically exploit parallelism, it has been difficult to achieve good performance in practice [6]. This paper aims to provide a brief look at some of the foundational work done in this area and describe some of the issues that may have prevented the work from achieving the desired results. Then we will outline our current work-in-progress that aims to address some of these issues through the use of iterative compilation.

1 Introduction

The idea of executing functional programs via graph reduction has its roots in the 1970s [7, 8]. This concept planted the seed for research into non-strict semantics and laziness, which would form the foundation for much of the work done on implicit parallelism in lazy functional languages [9, 10]. In the early 1980's there was a growing interest in the design and use of new processing architectures that deviated from the von Neumann style. This was partially motivated by the belief that novel architectures were the only method of producing efficient lazy functional programs (even sequential

[1] This is true as long as the 'chosen' path terminates.

ones), as methods for executing them efficient on standard architectures had not yet been developed [11, 1].

Luckily, work in the mid 1980's showed that graph reduction could be compiled into efficient code for standard architectures, an influential result from this line of work being the *G*-Machine [11]. This did not halt the research into novel architectures, but allowed programmers to write performant code in a lazy functional language without the need for a special purpose machine.

Concurrently with the research into the design of physical and abstract machines for non-strict graph reduction, there was a significant effort in the study of *strictness analysis*. While non-strict semantics provides us with many appealing benefits, it changes the decision to reduce an expression in parallel into a potentially serious one. This due to the fact that unlike strict languages, non-strict languages have the ability to ignore expressions if they are not needed. For example, in a strict language, the expression below would not terminate, while it would be able to compute the correct result in a non-strict language:

```
squareFirst :: Int -> Int -> Int
squareFirst x ⊥ = x * x
```

Having a simple parallelisation heuristic such as 'compute all arguments to functions in parallel' can alter the semantics of a non-strict language. In the case of `squareFirst`, reducing both arguments in parallel would result in a thread that never terminates. Depending on the implementation, this could cause the entire program to run indefinitely [2]. Therefore, care must be taken when deciding what sub-expressions are chosen to be reduced in parallel. Strictness analysis is one method that aims to solve this issue. The goal of strictness analysis is to determine which arguments to a function are *needed* and will therefore be evaluated anyway. For example:

```
if p t f = case p of
             True  -> t
             False -> f
```

In this definition of `if`, the only argument that will always be needed is `p`, therefore we say that `if` is strict in `p`. More formally, a function f with arguments $n_1, n_2 \ldots n_n$ is said to be strict in argument n_m if and only if $f \ldots \bot_m \cdots = \bot$. By identifying the strict arguments to a function, it is possible to reduce those arguments in parallel without changing the semantics of the program.

While research into novel architectures is not common today, we feel that there are important lessons from the successes and failures of that line of

[2] Some implementations do not have a sense of a 'master' thread and will wait for all threads to complete before exiting [4]

research that can be applied to modern parallel implementations and techniques. Combining this with a better understanding of strictness analysis and the ubiquity of multi-core architectures, we feel that it is worthwhile to explore new techniques for implicit parallelism.

Outline of this Paper

Section 2 looks at two important novel architectures that were designed for the execution of parallel lazy functional programs and discusses what can be learned from the research carried out with those designs. Section 3 is a brief overview of some implementations designed for standard architectures. We do not claim to provide a total view of the work done in this area, but instead focus on the lines of research the set the foundation for the abstract machine that we use for our implementation. Section 4 provides the motivation for strictness analysis and the necessity for using non-flat domains when dealing with data structures.

Generally, the rest of the paper is organised into two main parts. Sections 2 3 4 for a review of some of the foundational work in the parallel lazy functional programming, while section 5 outlines our current work which seeks to address some of the weaknesses of earlier techniques.

2 Novel Machines

After Backus's famous Turing Award speech [12] there was a spike of interest in novel architectures for computation. While for many disciplines, the motivation was in order to achieve higher utilisation of parallelism, the functional programming community had additional motivation in that standard architectures were deemed to be inappropriate for lazy evaluation [1, 7]. In this section we will take a look at two of the designs that came out of this line of research: ALICE and GRIP. These two machines show two approaches to the problem and share some characteristics, but it is important to note that these were not the only machines that were designed for parallel graph reduction. Other notable examples were the Manchester Dataflow Machine and SKIM [1].

2.1 ALICE

The first major architecture designed explicitly to reduce applicative programs was designed at Imperial College by Darlington and Reeve [13]. ALICE (Applicative Language Idealised Computing Engine), as it was known, had the ability to perform both strict and non-strict evaluation, which allowed it to be used to test the viability of both approaches. The machine was designed with three main components: *Processing Agents*, *Packet Pools*, and a network for communication. Closures were represented as *packets* which

contained any necessary data and the identification tags of other needed closures. The processing agents carried out reduction by requesting packets from the distributed memory (implemented via multiple Packet Pools) and then returns the resulting packet to the distributed memory.

ALICE's design meant that every closure had the possibility of being reduced in parallel and would therefore be stored in the distributed memory. Unfortunately, it seems that for many expressions the gain of having the expression reduced in parallel was overshadowed by the communication costs in the distributed memory. This issue is known as the *granularity* problem; a parallel task that is too 'fine grained' will not make up for the cost of communicating and managing the task [1]. This problem is not unique to ALICE, in fact, looking into solutions for this problem led to many techniques still in use today [14]. Part of the reason this problem was so detrimental to ALICE was the fact that the overhead for parallel tasks was so high due to the distributed nature of the memory. Utilising a distributed memory without taking into account *where* the data is physically results in high communication overheads. A processing agent that requests closure A may have to wait much longer for A than it would have for B. This leads to situations where the possibility of productive work is lost due to a processing agent requesting a closure that is 'far' away.

There are two important lessons to take away from the research into ALICE's design.

- It is expensive to ignore issues of locality when dealing with parallelism
- Parallelism comes at a cost, some expressions are not worth the cost of the synchronisation overheads

2.2 GRIP

Another machine built specifically for the parallel reduction of functional programs was the Graph Reduction in Parallel (GRIP) machine from the University College of London [15]. GRIP was designed with separate *processing elements* (PE) and *intelligent memory units* (IMU). These unites are combined by a high-speed *future bus*. In order to maximise the use of the future bus, each physical board contained four PEs and one IMU [15]. The physical boards were then connected through the future bus. This creates a two-tier memory architecture. The each PE has a 'local' IMU that provides a high-speed local memory. This local memory can only store part of the program graph. When other parts of the graph are needed, communication to other boards is necessary. The high-speed future bus was specifically chosen in order to minimise the effect of communication costs between boards [15, 1].

2.2.1 Sparking Parallel Tasks

As opposed to reducing all sub-expressions in parallel, GRIP used a *sparking* model in order to express parallelism. In this model, expressions can be annotated as suitable tasks for parallel execution, the tasks are then stored in a global *task pool* from where they are selected and then scheduled onto the PEs [16, 15]. This method was developed as part of the Four-Stroke Reduction Engine described in [14]. The benefit of this model is that only annotated expressions are attempted in parallel allowing for flexibility in the amount of parallelism that is introduced, and the possibility of avoiding the granularity issues seen by ALICE. The sparking model also allows for both explicit and implicit parallelisation, with the parallel annotations being placed by either the programmer or by an implicit strategy.

In experiments, GRIP outperformed many of the earlier attempts at novel architectures [16]. However, even though the use of a high-speed bus minimised communication costs, non-local communication cannot be avoided and increases the overhead significantly [1].

2.3 Discussion of Novel Architectures

The continuous improvement of standard architectures along with the drastic reduction in cost for stock hardware in the 1980's and 1990's created an environment that was very hostile to novel architectures [17]. By the time that a new machine was designed, built, and tested, the increase in speed of stock hardware would allow standard techniques to match or even outperform the dedicated hardware in terms of wall-clock speed. This trend, of improved performance 'for free' from stock hardware manufacturers continued through the 1990's and into the early 2000's. While research into parallel functional programming continued through this period, research into novel architectures and implicit parallelism was less common.

3 Graph Reduction on Standard Architectures

Research into the implementation of non-strict and lazy languages has seen continuous contribution since the 1970's to today [10, 5, 11, 18, 19, 20]. One of the first breakthroughs was a technique that allowed for user defined functions to be compiled to a fixed set of combinators, S, K, and I [21]. This made it easier to implement lazy languages as the compiler writer only had to implement the SKI combinators and primitives on the target platform. A good introduction to this technique can be found in [4]. The shortcoming of the SKI combinator approach was that each individual reduction of a combinator carried out very little work and was therefore not very efficient for large programs [4].

In the mid 1980's Augustsson and Johnsson developed a method to compile user-defined functions directly to assembly code, allowing for more efficient execution of lazy programs [11]. This technique, known as the *G*-Machine, set the ground-work for many of the implementations in use today [22, 18]. The *G*-Machine takes the form of a stack based abstract machine that interprets *G*-Code. Each *G*-Code instruction is able to be easily translated into a target assembly language, making it suitable as an intermediate representation (IR) [11]. Because each user defined function is compiled to its own sequence of assembly instruction, the *G*-Machine does not face the same issue as the earlier SKI-based approach and is able to do more work on each reduction.

Many of the popular Haskell implementations use some form of the *G*-Machine; Hugs, YHC, and NHC utilised variants of the standard *G*-Machine, while GHC (the de-facto standard implementation) uses a modified form of the Spineless Tagless *G*-Machine.

One of the criticisms of the *G*-Machine and one of its successors, the Spineless *G*-Machine, is that the intermediate *G*-Code is difficult to transform and optimize as it is closer to the machine architecture. This was addressed by the development of the Spineless Tagless *G*-Machine, which uses a higher level IR that is more easily manipulated [23].

3.1 Parallelism on Standard Architectures

While there has been work on parallel versions of abstract machines for use on standard architectures, much of the work was concerned with managing distributed memories [24]. When it comes to shared-memory multiprocessors, such as the ones now found in commodity hardware, much of the research dealt with runtime system issues such as garbage collection and the handling of IO threads. These implementations commonly used standard abstract machines [20, 19]. This makes intuitive sense as the semantics of graph reduction presume that the program graph is a global data structure that all 'reducers' are able to manipulate. It is in dealing with the communication of distributed memories that this model breaks down and must be accounted for. Multiprocessors on a shared memory system already have access to shared global data, therefore any necessary communication is carried out via the shared graph.

4 Strictness Analysis

When parallelising a strict language it is possible to attempt every argument to a function in parallel with the function body. Because all of the arguments to a function would be evaluated regardless of whether reduction was occurring in parallel, there is no change to the semantics of a strict program

using this strategy. However, non-strict languages do not share this benefit. Take the following function as an example

Definition:

```
f x y = length x
```

Even though `y` is not needed for the result of this function, `f` will fail to terminate in a strict language when `y` fails to terminate. However, in a non-strict language only the needed arguments`f` can still terminate[3]. This ability, to avoid unnecessary computation, is a fundamental trait of non-strict languages. One consequence of this ability is that the semantics of a lazy language *can* change if unneeded arguments are evaluated! This has important consequences with regard to auto-parallelisation. Take a user defined `or` function:

```
or x y = case x of
            True  -> True
            False -> y
```

The subject of a case expression will always be evaluated, but without actually reducing the subject, it is impossible to know which alternative will be returned by the function. This gives us the following behavior

```
or ⊥ y = ⊥

or x ⊥ = case x of
                  True  -> True
                  False -> ⊥
```

Because we can not be sure as to the value of x at compile time, it would be unsafe to spark `y` to be reduced in parallel, whereas sparking `x` retains the original semantics.

One useful technique for determining which arguments are strict is *abstract interpretation* [4]. In short, this method lifts all expressions in a program into an abstract domain. This process discards information about the program while retaining enough to be a useful static analysis. This is best shown by working through an example.

To begin with, our abstract domain will only deal with two values, *bottom* ($\perp$, represented with 0) and everything else (represented with 1). As mentioned in section 1, a function is said to be strict in an argument if when the argument's value is bottom, the function's result is bottom. We must first redefine our primitives in the abstract domain, for an `if` function, we could define the following

[3]The termination of this function depends on the *spine-strictness* of its first argument. This is a concept we will return to later in this section.

```
if# p t f = p & (t | f)
```

To begin with, the use of `if#` in place of `if` is simply a convention used to differentiate between the two domains. The function `if` is the function used on the domain of normal values, whereas `if#` is used on values in our strictness domain. We have also introduced two new primitives: `&` and `|`. Here we use their standard boolean interpretations. By passing `0` and `1` to `if#` and using our standard truth tables, we can determine which arguments to `if` are strict.

```
if# 0 1 1  = 0 & (1 | 1) = 0
if# 1 0 1  = 1 & (0 | 1) = 1
if# 1 1 0  = 1 & (1 | 0) = 1
```

Here we can clearly see, when the first argument to `if#` is bottom, the result is bottom. This is not true of any other argument, therefore, the function `if` is only strict in its first argument. Because user-defined functions are built up of primitive operations, once all primitives are defined for this new domain we can simply inline the primitive's definition into the user-defined function.

Unfortunately, dealing with recursion is a bit more involved. If a function `f` calls itself or another function, the standard method is to iterate through deeper approximations until a fixed-point is found. For example, if `f` is of the form `f x y = ... f ...` translating this definition into the abstract domain would begin with the simplest approximation `f#0 x y = 0`. This is known as the zeroeth approximation, we then iterate a new approximation using the zeroeth approximation in place of the recursive call. This process continues by having the $\texttt{n}^{\text{th}}$ approximation replace its recursive call with the $\texttt{n-1}^{\text{th}}$ approximation.

```
f#0 x y = 0
f#1 x y = ... f#0 ...
f#2 x y = ... f#1 ...
        ⋮
f#n x y = ... f#(n-1) ...
```

We can stop creating new approximations when we reach a fixed-point on all arguments. Techniques for computing this as efficiently as possible can be found in [25, 26]. However, the only reason to be certain a fixed-point has been found is by iterating through all combinations of argument values at each level of approximation, for sets of functions that are mutually recursive or functions with a high number of arguments this can be expensive,

4.0.1 Dealing with Lists

Take the following function definitions

```
sum     [] = 0
sum (x:xs) = x + sum xs

length     [] = 0
length (x:xs) = 1 + length xs

elem x     [] = False
elem x (y:ys) = case x == y of
                  True  -> True
                  False -> elem x ys
```

Each of these functions takes a list as one of its arguments, but the degree of *neededness* of the list is different for each function. Let us examine each function in turn. It is clear that `sum` not only requires the list to be finite, but also that each element is not bottom, otherwise `sum` cannot terminate. `length` is similar in that it requires the passed list to be finite in length, but does not require that the elements of the list are defined. In fact, for `length`, none of the elements have to be defined for `length` to compute its value. Lastly, `elem` *may* require the list to be finite, but that depends on both the value of the first argument and the values in the list itself. Similarly, `elem` can still return a defined value when some of the elements are undefined, as long as that part of the list is not reached.

This illustrates a very important point when attempting to use strictness analysis on a program. For certain data structures, a flat domain of `0` and `1` is not sufficient. By forcing the strictness of a list to be described in such a manner we force ourselves to be too cautious with lists, only considering them strict in cases like `sum` otherwise we risk changing the semantics of the program.

To address this issue, Wadler developed a four-point domain for lists [27]. This domain described the following levels of strictness:

- 3: A finite list with fully defined elements
- 2: A finite list with the possibility of undefined elements
- 1: Any infinite list, or approximation to one
- 0: Bottom

This provides a more nuanced view of strictness for lists which can be taken advantage of by compilers looking to optimise argument reduction (whether sequential or parallel).

Strictness analysis was seen as a potential way to determine which expressions should be reduced in parallel. If an argument is known to be needed, there is no risk (semantically) in parallel reduction. This exact method was used to achieve promising results in [28]. Because this method does not create tasks for expressions that may not be needed (known as speculative parallelism) it is known as *conservative* parallelisation.

5 Our Current work

5.1 The Compiler

While work on our compiler is not completed, enough progress has been made that we are able to describe the overall design of the system with confidence. Our source language is a subset of the F-Lite language which is also used in the Reduceron research [29]. All static analysis, defunctionalisation and strictness analysis, happens after parsing to an enriched lambda calculus. We then compile to standard *G*-Code which is passed to our runtime system that evaluates the resulting program. The parsing, static analysis, and compilation to *G*-Code is all written in Haskell. The runtime system, which accepts the resulting *G*-Code is written in C. Currently, our compiler is a *quasi-parallel* machine. This is similar to the technique discussed in [1, 30] for the profiling of parallel programs and can be thought of as an upper-bound on the possible performance. We do not see any reason that our compiler cannot be easily converted into a true concurrent implementation. The only downside of such an endeavour is the introduction of non-determinism in the runtime performance.

Our compiler accepts `par` annotations that signal the runtime system to spark the expression. These annotations take the form

```
par a b = b
```

Where `a` becomes a sparked expression within the runtime. The runtime utilises the sparking model discussed in our discussion of the GRIP machine and as introduced by the Four-Stroke Reduction Engine.

We are aiming to begin experimentation with a strictness analysis based on the four-point domain described above. Using the strictness information provided by the analysis we will annotate strict arguments to be reduced in parallel. This has the potential of creating too much parallelism and running into similar problems as ALICE, we are hoping to avoid these through the use of iterative compilation.

5.2 Iterative Compilation

We now have all of the building blocks for what we see as our contribution. We believe that there are several reasons why previous work into implicit

parallelism has not achieved the results that researchers have hoped for. Chief amongst those reasons is that the static placement of parallel annotations is not sufficient for creating well-performing parallel programs.

Imagine that you were writing an explicit parallel program. When writing the source code you may study the structure and then decide where to place your `par` annotations. When the program is compiled and executed you find that the performance was not at the level you had hoped. So you decide not to change anything in your source...

This is how we interpret many of the strategies for implicit parallelisation. Static analysis is performed on the structure of the program only once and then the placement of annotations is not reconsidered. There is one significant exception to this. In 2007 Harris and Singh published their results on a feedback directed implicit parallelism compiler [31]. The results were mostly positive (in that most benchmarks saw an improvement in performance) but were not to the degree that had been desired. Since this research was published we have seen no other attempt in this line of research within the functional programming community.

The work in [31] attempted to use runtime profile data to introduce parallel annotations into the program based on heap allocations. In short, when viewing the parallel execution of a program as a tree, their method seeks to expand the tree based on previous executions of the program. Our goal is to develop a system that begins with a program that perhaps has *too much* parallelism and uses runtime data to prune the execution tree. We have implemented a few mechanisms to make this possible.

5.2.1 Logging

Our implementation collects the following global statistics in order to measure runtime performance.

- Number of reduction cycles
- Number of sparks
- Number of blocked threads
- Number of active threads

These statistics are useful when measuring the overall performance of a parallel program, but tells us very little about the usefulness of the threads themselves.

In order to ensure that the iterative feedback system is able to determine the overall 'health' of a thread, it is important that we collect some statistics pertaining to each individual thread. For this reason we have used a similar system as that outlined in [30]. With the following metrics being recorded *for each thread*:

- Number of reduction cycles
- Number of sparks
- Number of blocked threads
- Which threads have blocked the current thread

This allows us to reason about the productivity of the threads themselves. An ideal thread will perform many reductions, block very few other threads, and be blocked rarely. A 'bad' thread will perform few reductions and be blocked for long periods of time.

5.2.2 Transformation

In order for the iterative feedback to be able to change the parallelization of a program, it must be able to determine which annotation was the source for each thread of execution. The method we have devised is based on the idea that a specific `par` in the source program can be deactivated and therefore no longer create parallel tasks, while maintaining the semantics of the program. The method has two basic steps:

- `par`'s are identified with the *G*-Code and each par is given a unique identifier.
- When a thread creates the heap object representing the call to `par` the runtime system looks up the status of the `par` using its unique identifier. If the `par` is 'on' execution follows as normal. If the `par` is off the thread will not spark the annotated expression.

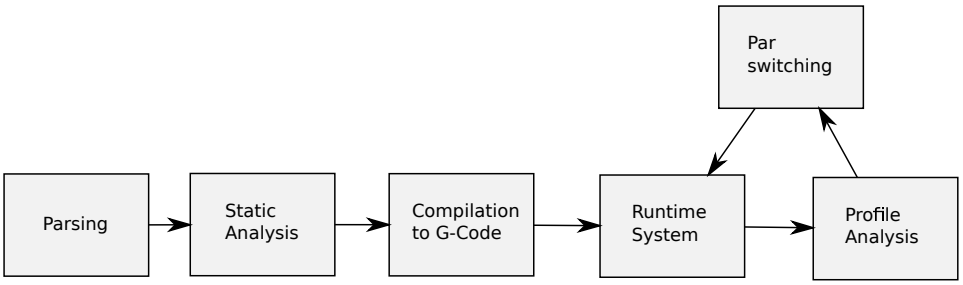


Figure 1: Compiler Pipeline

5.3 Work to be Done

Almost all of the necessary machinery is in place in order to begin experimentation. The main module that is not completed is the strictness analysis,

which we intend to use in order to select the initial `par` placement within a program. This initial placement will then be altered through the activation/deactivation of the `par` sites as described above.

We hope to have this completed soon in order to begin gathering results. We do not expect this system to perform well as is, but instead will use it as a base for exploration into different techniques, one thought is that we will require a more sophisticated strictness analysis and perhaps change the way we deactivate `par` sites by removing them from the program altogether (which would require recompilation but may be worthwhile).

References

[1] K. Hammond, "Parallel functional programming: An introduction," www-fp.dcs.st-and.ac.uk/ kh/papers/pasco94/pasco94.html, 1994.

[2] S. L. Peyon Jones, "Parallel implementations of functional programming languages," *Comput. J.*, vol. 32, no. 2, pp. 175–186, Apr. 1989.

[3] K. Hammond, "Why parallel functional programming matters: Panel statement," in *Reliable Software Technologies - Ada-Europe 2011*, ser. Lecture Notes in Computer Science, A. Romanovsky and T. Vardanega, Eds. Springer Berlin Heidelberg, 2011, vol. 6652, pp. 201–205.

[4] S. L. Peyton Jones and D. R. Lester, *Implementing functional languages*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[5] R. J. M. Hughes, "The Design and Implementation of Programming Languages," Ph.D. dissertation, Programming Research Group, Oxford University, July 1983.

[6] G. Tremblay and G. R. Gao, "The Impact of Laziness on Parallelism and the Limits of Strictness Analysis," in *Proceedings High Performance Functional Computing*. Citeseer, 1995, pp. 119–k133.

[7] D. Turner, "Some History of Functional Programming Languages," in *Symposium on the trends in functional programming 2012*, 2012.

[8] C. P. Wadsworth, "Semantics and Pragmatics of the λ-Calculus," Ph.D. dissertation, Programming Research Group, Oxford University, 1971.

[9] P. Henderson and J. H. Morris Jr, "A Lazy Evaluator," in *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*. ACM, 1976, pp. 95–103.

[10] D. Friedman and D. Wise, "CONS Should Not Evaluate its Arguments," in *Automata, Languages, and Programming*, 1976, pp. 257–281.

[11] L. Augustsson and T. Johnsson, "The chalmers lazy ml-compiler," *Comput. J.*, vol. 32, no. 2, pp. 127–141, Apr. 1989. [Online]. Available: http://dx.doi.org/10.1093/comjnl/32.2.127

[12] J. Backus, "Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[13] P. Harrison and M. Reeve, "The Parallel Graph Reduction Machine, ALICE," in *Graph Reduction*, ser. Lecture Notes in Computer Science, J. Fasel and R. Keller, Eds. Springer Berlin / Heidelberg, 1987, vol. 279, pp. 181–202.

[14] C. Clack and S. Peyton Jones, "The four-stroke reduction engine," in *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, 1986, pp. 220–232.

[15] S. P. Jones, C. Clack, and J. Salkind, "GRIP: A High Performance Architecture for Parallel Graph Reduction," in *Functional Programming Languages and Computer Architecture: Third International Conference (Portland, Oregon)*. Springer Verlag, 1987.

[16] K. Hammond and S. P. Jones, "Some early experiments on the grip parallel reducer," in *IFL'90: International Workshop on the Parallel Implementation of Functional Languages*, 1990.

[17] K. Hammond and G. Michelson, *Research Directions in Parallel Functional Programming*. Springer-Verlag, 2000.

[18] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of Haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

[19] B. Goldberg, "Buckwheat: Graph Reduction on a Shared-Memory Multiprocessor," in *Proceedings of the 1988 ACM conference on LISP and functional programming*, ser. LFP '88. New York, NY, USA: ACM, 1988, pp. 40–51. [Online]. Available: http://doi.acm.org/10.1145/62678.62683

[20] T. Harris, S. Marlow, and S. P. Jones, "Haskell on a Shared-memory Multiprocessor," in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, ser. Haskell '05. New York, NY, USA: ACM, 2005, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/1088348.1088354

[21] D. A. Turner, "A New Implementation Technique for Applicative Languages," *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979.

[22] G. Burn, S. Peyton Jones, and J. Robson, "The Spineless G-Machine," in *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM, 1988, pp. 244–258.

[23] S. Jones, "Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine," *Journal of functional programming*, vol. 2, no. 2, pp. 127–202, 1992.

[24] L. Augustsson and T. Johnsson, "Parallel graph reduction with the (v , g)-machine," in *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, ser. FPCA '89. New York, NY, USA: ACM, 1989, pp. 202–213.

[25] C. Clack and S. L. P. Jones, "Strictness AnalysisA Practical Approach," in *Functional Programming Languages and Computer Architecture*. Springer, 1985, pp. 35–49.

[26] C. Martin and C. Hankin, "Finding fixed points in finite lattices," in *Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 426–445.

[27] P. Wadler, "Strictness analysis on non-flat domains," in *Abstract interpretation of declarative languages*. Ellis Horwood, 1987, pp. 266–275.

[28] G. Hogen, A. Kindler, and R. Loogen, "Automatic Parallelization of Lazy Functional Programs," in *ESOP'92*. Springer, 1992, pp. 254–268.

[29] M. Naylor and C. Runciman, "The reduceron reconfigured," *ACM Sigplan Notices*, vol. 45, no. 9, pp. 75–86, 2010.

[30] N. Charles and C. Runciman, "An Interactive Approach to Profiling Parallel Functional Programs," in *Implementation of Functional Languages*. Springer, 1999, pp. 20–37.

[31] T. Harris and S. Singh, "Feedback Directed Implicit Parallelism," *SIGPLAN Not.*, vol. 42, no. 9, pp. 251–264, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1291220.1291192

Structural Types for Systems of Equations

John Capper Henrik Nilsson

June 11, 2013

Abstract

Characterising a problem in terms of a system of equations is common to many branches of science and engineering. Due to their size, such systems are often described in a modular fashion by composition of individual equation system fragments. Checking the balance between the number of variables (unknowns) and equations is a common approach to early detection of mistakes that might render such a system unsolvable. However, current approaches to modular balance checking have a number of limitations. This paper investigates a more flexible approach that makes it possible to treat equation system fragments as true first-class entities. Furthermore, the approach handles so-called structurally dynamic systems, systems whose behaviour changes discretely and abruptly over time. The central idea is to record balance information in the type of an equation fragment. This information can then be used to determine if individual fragments are well formed, and if composing fragments preserves this property. The type system presented in this paper is developed in the context of Functional Hybrid Modelling (FHM). However, the key ideas are in no way specific to FHM, but should be applicable to any language featuring a notion of modular systems of equations, including systems with first-class components and structural dynamism.

1 Introduction

Systems of equations, also known as simultaneous equations, are abundant in science and engineering. Applications include modelling, simulation, and optimisation, to name but a few. Describing complex problems mathematically, e.g. modelling the engine of a car, often requires a large number of equations; systems consisting of hundreds of thousands of equations are not uncommon. The systems are usually parametrised, describing not just a specific problem instance, but a set of problems. Moreover, it is more of a rule than an exception that no known analytical solution exists, necessitating the use of computers and numerical methods for finding (sufficiently good approximate) solutions. Due to the size of the equation systems, some form of abstraction mechanism that supports a *modular* formulation by composition of individual equation system fragments, typically referred to as system *components*, is often a practical necessity as well.

As an example, consider the full-wave rectifier in Fig. 1. This is not a very big system, consisting of only eight components. Yet, the advantages of being able to derive a system of equations that model this circuit by reusing parametrised models of the individual components should be clear. For instance, a single diode model could be reused four times, for diodes D1–D4. More generally, physically accurate component models can be very involved, making it highly desirable to develop libraries of reusable components.

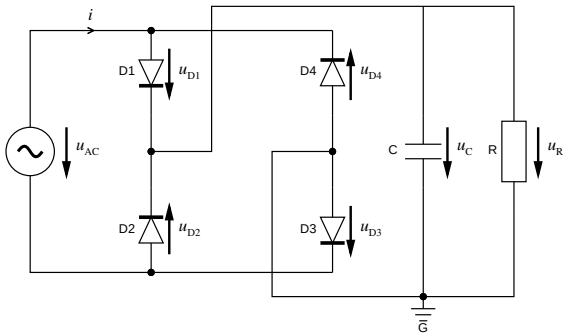


Figure 1: Full-wave rectifier

A number of high-level *equation-based* languages exist across a range of application domains, supporting a modular, parametrised formulation of systems of equations. These languages are supported by an environment that include appropriate approaches for solving the equations given specific values for the parameters, or for solving optimisation or parameter fitting problems. Examples in the area of modelling and simulation of physical systems, languages which could be used to model circuits like the one in Fig. 1, include Modelica [21], VHDL-AMS [16] and Verilog-AMS [1].

In modern, high-level programming languages, *types* play a crucial role. Types aid in creating *safe* programs that conform with their specification. The *power* of a type system can vary greatly, ranging from languages such as C [2], in which types make very few guarantees about correctness, to languages such as Agda [26], where types can be used to specify very precise correctness criteria. In particular, Agda employs dependent types [19] that allow the programmer to not only specify the properties of programs, but also to prove these properties within the language itself.

Equation-based languages are often also typed, with the types playing much the same roles as in conventional programming languages. Additionally, simple invariants related to the structure of the equation systems may be enforced, such as there being equally many equations as variables to solve for, with a view to static detection of structural problems that are likely to render the systems ill-formed and thus unsolvable.

However, there is considerable scope for improving the type systems of current equation-based languages in the latter respect, both in terms of refining the enforced structural invariants, thus allowing more potential problems to be detected early, and to generalise this to a setting where equation system fragments have *first-class* status and where the systems of equations as such may be *structurally dynamic*: evolving over time. Current, main-stream, equation-based languages have quite limited support for structural dynamism, and making these languages more flexible in that respect is an active research area [25, 27, 14, 32]. This work has thus far focused on constructs and mechanisms for expressing and solving structurally dynamic systems, while little attention has been paid to finding suitable structural invariants for this more general setting.

In the following, we seek to address the above points. We develop a type system for modular systems of equations that enforces a refined set of structural invariants while supporting first-class components and structural dynamism. We treat equations in the abstract in our formal development, not assuming any specific application domain. However, we do need a concrete language for expressing modular, structurally

dynamic systems of equations. To that end, we develop a core equation-based language that captures the essence of Functional Hybrid Modelling (FHM) [25, 14]. FHM is a framework for modelling physical systems that can be described by differential equations, such as the full-wave rectifier above. For this reason, most of our examples will also be drawn from physical systems modelling, with the unknowns in the equations representing time-varying entities (functions of time). FHM was chosen as it features component-based modelling, first-class models, and structural dynamism in a relatively minimalistic way. However, we reiterate that the core ideas put forward in this paper are not at all limited to FHM. Our specific contributions are:

1. A novel type system for modular systems of equations supporting first-class components and structural dynamism.
2. A refined set of structural invariants based on classification of equations allowing a larger class of structural anomalies to be prevented compared with existing approaches.
3. A concise small-step semantics for the core of FHM, capturing the subtle behaviour of variables in a modular system of equations.

We have also implemented a type checker for our system capable of inferring types [6]. It is implemented in the dependently typed language Agda [26] primarily to ensure totality and termination of the inference algorithm, but also with a view to facilitate formal verification of aspects of the type system at a later stage.

The remainder of this article is structured as follows. Sections 2 and 3, respectively, introduces modular systems of equations and FHM. Section 4 investigates structural properties of equation systems, setting the stage for Sect. 5 that presents the main technical contribution of this article: the core language, its semantics, the type system, and implementation notes. Section 6 evaluates what has been achieved, in part through a complete worked example. Related work is considered in Sect. 7, while avenues for future work are discussed in Sect. 8. Finally, Section 9 concludes.

2 Modular Systems of Equations

This section gives a detailed introduction to modular systems of equations, covering basic theory, modularity, abstraction over systems, causality, and structural dynamism.

2.1 Preliminaries

A *system of equations* is a set of equations over a set of *variables* or *unknowns*. It has a solution if every variable in the system can be instantiated with a value such that all the equations are simultaneously satisfied. The domain of the variables and signatures for equations is mostly orthogonal to the work presented in this article. However, for reasons of presentation, we will use the domains of reals or time-varying reals unless stated otherwise. The following is an example of a system consisting of two equations and two unknowns:

$$x^2 + y = 0 \quad (1)$$

$$3x = 10 \quad (2)$$

This system can be solved by using (2) to solve for x, substituting the value of x into (1), thus enabling the latter to be used to solve for y. However, consider the following

parametrised version of the system instead. The solvability of the system now depends on the value of the coefficient c: when $c = 0$, no solution exists.

$$x^2 + y = 0 \quad (3)$$
$$cx = 10 \quad (4)$$

Whether or not a system of equations has a solution is an important property. For example, if a system of equations is intended to model a physical system, unsolvability would be indicative of a modelling fault. However, as the trivial example above illustrates, unless all aspects of the system are known, it may not be possible to answer this question, at least not directly. Moreover, depending on the domain, the question is in general undecidable.

Yet, when building systems of equations modularly, it is desirable to catch problems that may ultimately lead to unsolvable systems of equations early; i.e., already at the level of individual equation system fragments or partial compositions of fragments. One property that can be checked modularly is whether the number of equations and unknowns agree. While an equal number of equations and unknowns in itself is neither a necessary nor sufficient condition for solvability, an unequal number is in practice often indicative of problems. Consequently, rules pertaining to the balance between the number of equations and unknowns are adopted in industrial-strength, equation-based languages such as Modelica [21].

We will discuss structural properties in more depth in Sect. 4. In particular, we will identify invariants that are more refined than basic equation and variable balance, thus allowing checking for a larger class of structural problems, while still admitting modular checking in a setting with first-class equation system fragments and structural dynamism. In Sect. 5 we will then proceed to show how these refined invariants can be integrated into a type system.

2.2 Abstraction over Systems of Equations

The equation systems needed to describe real-world problems are usually large and complex. On the other hand, there tends to be a lot of repetitive structure [10], making it beneficial to describe the systems in terms of reusable equation system fragments. For example, consider an electrical circuit comprising resistors, capacitors, and inductors. Each component can be described by a small equation system, and the entire circuit can then be described *modularly* by composition of *instances* of these for specific values of any parameters.

While the exact syntactic details vary between languages, the idea is to encapsulate a set of equations as a component with a well-defined interface. Let us illustrate with an example, temporarily borrowing the syntax of the λ-calculus for the abstraction mechanism:

$$\lambda(x,y) \rightarrow \begin{array}{rcl} x+y+z & = & 0 \\ x-z & = & 1 \end{array}$$

This abstraction is a relation that constrains the possible values of the two *interface variables* x and y according to the encapsulated equations. The variable z is *local* to the abstraction.

Call the above relation *rel*. It can now be used as a building block by *instantiating* it: substituting expressions for the interface variables and renaming local variables as necessary to avoid name clashes. We express this as *relation application*, denoted by

$\diamond$:

$$\begin{aligned} u+v+w &= 10 \\ rel &\diamond (u,v) \\ rel &\diamond (v,w+7) \end{aligned}$$

After unfolding and renaming, often referred to as *flattening* or *elaboration*, the following unstructured (as opposed to modular) set of equations is obtained:

$$\begin{aligned} u+v+w &= 10 \\ u+v+z_1 &= 0 \\ u-z_1 &= 1 \\ v+(w+7)+z_2 &= 0 \\ v-z_2 &= 1 \end{aligned}$$

The relation *rel* contributes 2 equations for each application. Including the top-level equation, the fully elaborated system thus consists of 5 equations in total over 5 unknowns. Note the need to rename the local variable z when unfolding *rel*.

2.3 Causality

A *causal* system is one in which the cause-and-effect relationship between variables is explicit. In other words, the equations are *directed*: the equations are solved in a given order, with known (at that point) variables on one side of the equal sign, and a single unknown on the other (which then becomes a known in subsequent equations). One example is a set of ordinary differential equations (ODEs) in explicit form where the system state at a point in time is considered known, enabling the state derivatives at that point in time to be computed (from which the system state at the "next" point in time can then be approximated). Conversely, an *acausal* system is *undirected* with the equations simply expressing a relation on the variables, without any á priori given inputs and outputs, and thus also without any á priori given strategy for solving the equations. Differential Algebraic Equations (DAEs) [10] are an important example of acausal equations in the domain of modelling and simulation.

As a concrete example, consider Pell's equation [3] over the two unknowns x and y and parametrised by an integer n:

$$x^2 - ny^2 = \pm 1$$

The above equation is acausal: depending on which variable is known in some specific context, the equation can be translated into two different *assignments*:

$$y := \sqrt{(x^2 \pm 1)/n} \qquad\qquad x := \sqrt{\pm 1 + ny^2}$$

The advantage of the acausal formulation is that the equations are more reusable (above, Pell's equation is used in two ways) and more declarative (the equations can be expressed in whatever way is most clear, without undue concerns about how they are going to be solved). These points are crucial advantages in many domains, including modelling and simulation of large-scale physical systems [9]. A number of successful acausal modelling languages have thus been developed, with Modelica [21] being a prominent, state-of-the-art example.

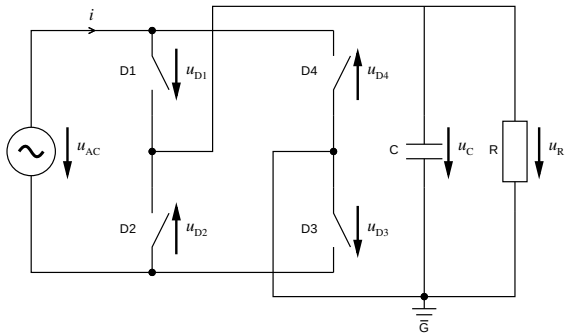


Figure 2: Full-wave rectifier modelled using ideal diodes.

2.4 Structural Dynamism

In a temporal setting, where equations express relations among time-varying entities, the equations themselves may change at various points in time to capture changes in the system configuration. A system of equations that evolve over time is known as *structurally dynamic*.

As an example, consider the model in Fig. 2 of the full-wave rectifier from Fig. 1 [24]. The modeller has chosen an ideal model for the diodes: an electrical switch that is closed (diode conducting) whenever the voltage across it is positive, and open otherwise (diode not conducting). Depending on which switches are open and which are closed, there are up to $2^4 = 16$ structural configurations, each corresponding to a distinct system of equations.

Structurally dynamic systems offer greater expressivity, but also create many problems [25, 27, 14, 32]. Of particular concern in this article is that errors in a system with a large or possibly even unbounded number of configurations may take a very long time to surface if this error only manifests itself when specific system configurations become active. Thus, the larger the class of such errors that can be caught statically by enforcing suitable structural invariants, the better. We consider invariants for structural dynamism in Sect. 4.3.

3 Functional Hybrid Modelling and Hydra

We now turn our attention to a particular approach to acausal, equation-based languages for modelling and simulation of physical systems, Functional Hybrid Modelling (FHM), as this provides a useful, concrete setting for our work. *Hydra* [25, 14] is an example of an FHM language. Its syntax will be adopted until a formal core language is presented in Sect. 5.2.

3.1 Functional Reactive Programming

FHM is inspired by Functional Reactive Programming (FRP) [12, 31], in particular as embodied by *Yampa* [23]. FHM can be viewed as a generalisation of Yampa, hence, we will begin by introducing the two central concepts of Yampa: *signals* and *signal functions*. Conceptually, a signal is a time-varying value: i.e: a function from time to a value of type τ:

$$\begin{aligned} \mathit{Time} &\approx \mathbb{R}^+ \\ \mathit{Signal}\ \tau &\approx \mathit{Time} \rightarrow \tau \end{aligned}$$

Time is represented as $\mathbb{R}^+$, the continuous, non-negative real numbers. The type of the time-varying value is represented by τ. A *signal function* is then simply a function on signals:

$$\mathit{SF}\ \alpha\ \beta \approx \mathit{Signal}\ \alpha \rightarrow \mathit{Signal}\ \beta$$

The $\approx$ symbol is used to emphasise the conceptual nature of the above definitions, which are not used directly in the implementation of Yampa. In particular, signal functions are required to be *temporally causal*: the output of a signal function at time t may only depend on the input signal on the interval $[0,t]$. Despite this, the conceptual definitions are useful for developing an intuition about the semantics of both FRP and FHM.

3.2 Signal Relations

The above definitions of signals and signal functions describe systems that are inherently causal. Signal functions are *directed*, taking an input signal and returning an output signal. FHM generalises signal functions to *signal relations*. Signal relations are acausal: they do not distinguish between inputs and outputs but rather express how signals are related to one another. A signal relation may be viewed as a predicate on a signal:

$$\mathit{SR}\ \tau \approx \mathit{Signal}\ \tau \rightarrow \mathit{Prop}$$

To recover n-ary relations, one can observe the following isomorphism: $\forall\ \alpha\ \beta\ .\ \mathit{Signal}\ \alpha \times \mathit{Signal}\ \beta \simeq \mathit{Signal}\ (\alpha \times \beta)$. Unary signal relations thus suffice for representing n-ary relations. For example, given a binary predicate $(\equiv) : \mathbb{R} \times \mathbb{R} \rightarrow \mathit{Prop}$, the following binary signal relation can be constructed:

$$\begin{aligned} &(\equiv_{sr}) : \mathit{SR}\ (\mathbb{R},\mathbb{R}) \\ &(\equiv_{sr})\ s \approx \forall\ t : \mathit{Time}\ .\ (\equiv)\ (s\ t) \end{aligned}$$

3.3 Hydra

Hydra [25, 14] is a functional hybrid modelling language embedded in the functional programming language Haskell [17]. There are two levels to FHM and thus to Hydra: the *functional level*, concerned with defining ordinary functions operating on time-invariant values, and the *signal level*, concerned with the definition of relations between signals (time-varying values), and, indirectly, the definition of the signals themselves as solutions satisfying the constraints imposed by the signal relations. The definitions at the signal level may freely refer to entities defined at the functional level, but signal level objects are not permitted to escape to the functional level, with the exception of instantaneous values of signals, which may be fed back to the functional level at the time of discrete events. This allows future system configurations to depend on earlier results.

Signal relations are constructed using the **sigrel** primitive, marking the boundary between the two levels:

sigrel *pattern* **where** *equations*

Signal relations are first-class, time-invariant, function-level objects that encapsulate a set of *equations*. These equations range over signal variables introduced by the *pattern*, similar to the abstraction mechanism presented in Sect. 2.2. A pattern is a (possibly nested) tuple (e.g. $((x,y),z)$ constitutes a valid pattern introducing 3 signal variables). We refer to these signal variables as *interface variables*. Signal variables that occur in the set of equations but not in the pattern are referred to as *local variables*. They do not occur anywhere else in the system. Signals are *not* first class entities in the language. There are two basic forms of equations:

atomic equation:	$e_1 = e_2$
signal relation application:	$sr \diamond e_3$

Here, *sr* is a *time-invariant* expression (signal variables must not occur in it) denoting a signal relation, and $\diamond$ denotes signal relation application. To illustrate, consider a component *twoPin* encapsulating equations common to all electrical components with two pins:

$$
\begin{aligned}
&\mathit{type}\ \mathit{Pin} = (\mathbb{R},\mathbb{R})\\
&\mathit{twoPin} : \mathit{SR}\,(\mathit{Pin},\mathit{Pin},\mathit{Voltage})\\
&\mathit{twoPin} = \textbf{sigrel}\,(p,n,u)\ \textbf{where}\\
&\quad p.i + n.i = 0\\
&\quad p.v - n.v = u
\end{aligned}
$$

Pin is a pair of values representing an electrical junction with projections for current and voltage. For clarity, we allow ourselves to use dot-notation for the projections; e.g. $p.i$ for the current through pin p and $p.v$ for the voltage (potential) at pin p. We can now define models of concrete electrical components by adding equations to the basic *twoPin*-model. For example, a model of a resistor:

$$
\begin{aligned}
&\mathit{resistor} : \mathit{Resistance} \to \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\\
&\mathit{resistor}\ r = \textbf{sigrel}\,(p,n)\ \textbf{where}\\
&\quad \textbf{local}\ u\\
&\quad \mathit{twoPin} \diamond (p,n,u)\\
&\quad r * p.i = u
\end{aligned}
$$

The *resistor* component extends *twoPin* by adding an equation that describes the behaviour of a resistor. The component shows how a system can be parametrised by a coefficient, in this example by the parameter r. The syntax **local** has been adopted to make explicit the quantification of the local variable u, and distinguish it from r, which is a time-invariant, functional-level parameter.

From here, we can define models for other two-pin components such as inductors and capacitors in the same way. Note how the *twoPin* signal relation is reused in each case. Here, the keyword **der** indicates the time derivative of a signal:

$$
\begin{aligned}
&\mathit{inductor} : \mathit{Inductance} \to \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\\
&\mathit{inductor}\ i = \textbf{sigrel}\,(p,n)\ \textbf{where}\\
&\quad \textbf{local}\ u
\end{aligned}
$$

$$
\begin{array}{l}
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad l * \mathbf{der}\, p.i \;=\; u \\
\mathit{capacitor} : \mathit{Capacitance} \;\rightarrow\; \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \\
\mathit{capacitor}\, c \;=\; \mathbf{sigrel}\,(p,n)\ \mathbf{where} \\
\quad \mathbf{local}\, u \\
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad c * \mathbf{der}\, u \;=\; p.i
\end{array}
$$

Elaboration of Hydra models proceeds by substitution of variables under signal relation application, resulting in either a flat set of equations, or a λ-expression. To illustrate, consider the modular system of equations in the relation *resistor 220*:

$$
\begin{array}{l}
\mathit{twoPin} \diamond (p,n,u) \\
220 * p.i \;=\; u
\end{array}
$$

A single step of unfolding eliminates the relation application, producing a flat system of 3 equations: two originating from *twoPin*, and a third contributed by *resistor*.

$$
\begin{array}{l}
p.i \;+ n.i \;=\; 0 \\
p.v \;- n.v \;=\; u \\
220 * \;p.i \;=\; u
\end{array}
$$

3.4 Structural Dynamism in Hydra

To express structurally dynamic systems, Hydra employs a switch construct that allows equations to be brought into and removed from a model as needed:

$$
\begin{array}{l}
\mathbf{initially}\,[\,;\mathbf{when}\,\mathit{condition}]\;\Rightarrow \\
\quad \mathit{equations}_1 \\
\mathbf{when}\,\mathit{condition}\;\Rightarrow \\
\quad \mathit{equations}_2 \\
\ldots \\
\mathbf{when}\,\mathit{condition\_n}\;\Rightarrow \\
\quad \mathit{equations}_n
\end{array}
$$

Only the equations from one branch are active at any one point in time. The equations of a branch are switched in whenever the condition guarding the branch *becomes* true, at which point those from the previously active branch are switched out. The keyword **initially** designates the initially active branch. An optional condition allows for the initial branch to be re-activated later. Should more than one switch condition within a switch construct trigger simultaneously, the branches are prioritised syntactically from the top down.

Additional complications arise due to the need to properly *initialise* the new system of equations after a switch. This is a hard problem in general, but it can be addressed at least to some extent by providing separate initialisation and reinitialisation equations [14, 24]. However, we will not consider this further here as this just amounts to additional systems of equations that can be subjected to much the same invariants as the main system.

For a concrete example, consider the following Hydra model of an ideal diode; i.e., essentially a voltage-controlled electrical switch [24]:

$$
\begin{array}{l}
\mathit{icDiode} : \mathit{SR}\,(\mathit{Pin}, \mathit{Pin}) \\
\mathit{icDiode} = \textbf{sigrel}\,(p,n)\ \textbf{where} \\
\quad \textbf{local}\ u \\
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad \textbf{initially}; \textbf{when}\ p.v - n.v > 0 \Rightarrow \\
\quad\quad u = 0 \\
\quad \textbf{when}\ p.i < 0 \Rightarrow \\
\quad\quad p.i = 0
\end{array}
$$

Whenever the voltage across the diode becomes positive, the diode starts conducting, meaning the switch closes resulting in the voltage across the diode becoming zero. Conversely, whenever the current starts to flow backwards through the diode, the diode stops conducting, meaning the switch opens and the current through the diode becomes zero.

4 Structural Properties

4.1 Structural Properties and Solvability

An important question regarding a system of equations is whether it has a solution and, if so, if that solution is unique. In general, one can only answer this question by studying a complete system of equations where all coefficients are known. Unfortunately, this is in direct opposition to the modular approach discussed in Sect. 2 as it would rule out checking components in isolation. Furthermore, as typical application domains, such as physical systems modelling, necessitates that the form of equations is not unduly restricted, one cannot in general hope to construct a decidable type theory capable of determining if an arbitrary modular system of equations has a solution. (For example, a modelling language restricting the systems of equations to be linear would be of very limited use.)

However, there are simple criteria that, while neither necessary nor sufficient for *guaranteeing* solvability, are such that violation of them are likely to be indicative of problems. Indeed, they may even be necessary preconditions for the *specific* approach to solving equations used by a tool. Enforcing that such criteria be met through the static semantics of an equation-based language can thus be useful, and is in fact often done. The following are two commonly used criteria for checking the well-formedness of systems [4, 5, 7, 21, 22]:

1. Balanced system: the number of equations is equal to the number of variables.
2. Structurally non-singular system: there is a bijection between the variables and the equations such that each variable is paired with an equation in which it occurs.

As we are only considering systems of finite size, property 2 implies property 1. Note that these properties are strictly structural: no information beyond which variables occur, and in which equations, is assumed. For illustration, consider the following system:

$$
\begin{aligned}
x + y &= z && (5)\\
x + 3 &= 12 && (6)\\
y^2 + 9 &= z^2 && (7)
\end{aligned}
$$

The bijection $\{x \mapsto 6, y \mapsto 7, z \mapsto 5\}$ between the set of variables $\{x, y, z\}$ and the set of equations $\{5, 6, 7\}$ pairs each variable with an equation in which it occurs. This system is thus both balanced and structurally non-singular. Furthermore, as it happens, it has a solution: $x = 9$, $y = -4$, $z = 5$.

On the other hand, it is easy to construct a system that violates the above criteria, but yet still possesses a solution. Consider:

$$\begin{aligned} x &= 2 \\ x^2 + 1 &= 5 \end{aligned}$$

This system is not even balanced. Yet $x = 2$ is clearly a solution. This shows that the above criteria are not necessary for the existence of a solution, and it is also easy to demonstrate that the criteria are not sufficient either.

Given the above examples, it is reasonable to ask what is it that makes these two criteria useful? The criteria stem from the fact that a *linear* system of equations has a unique solution if and only if the equations are independent and the number of equations and variables agree. If a linear system of equations has more variables than independent equations, it is said to be *underdetermined*. Conversely, if there are more independent equations than variables, it is said to be *overdetermined*. Intuitively, one could interpret each variable as a degree of freedom, and each equation as a constraint that eliminates a degree of freedom; i.e., is used to solve for a variable.

This latter intuition is broadly valid also for general systems of equations. In particular, structural non-singularity, which says that there is an equation that can be used to solve for each variable, is *exactly* what is needed for a number of (symbolic and/or numerical) methods that *attempt* to solve general systems of equations. Thus, if a system is structurally singular, commonly used solution methods will definitely fail. The balance criterion is a coarse approximation of structural non-singularity, essentially assuming that any equation can be used to solve for any variable. However, it is easy to check, and if violated, then that implies that the system is certainly structurally singular. On the other hand, even though neither criterion is necessary for the existence of a solution, insisting that the criteria be met is not overly restrictive in practice. Consequently, both criteria constitute useful static checks that can help find errors early during compilation.

In the following, we will develop criteria along the lines discussed above, but insist that they can be checked modularly, to enable integration into a type system and support first-class equation system fragments, and that they also work in a setting with structural dynamism. We will aim for a better approximation of structural non-singularity than basic balance checking by taking *some* account of which variables occur in which equations, but we will stay well short of attempting a full check for structural non-singularity, as a precise check cannot be done modularly or in a structurally dynamic setting, and as taking individual variable occurrences into account for a more precise approximation leads to a very complicated and expensive system [22]. We will refer to the difference between the number of equations and variables in a system as the *equation-variable balance*. By analogy to the terminology used for linear systems, but regardless of whether the equations are independent or even linear, we will refer to a system where the balance is positive as *overconstrained* and one where the balance is negative as *underconstrained*.

4.2 Criteria for Structurally Well-formed Signal Relations

The crux of the type system discussed in this paper is the insistence that a modular system of equations satisfy certain structural properties. This is enforced by introducing constraints at the type level (Sect. 5). We introduce 5 criteria below, stemming from the setting of FHM, from which such constraints can be generated. It is conceivable that different application domains could require constraints specific to that domain. This is not a problem, provided the constraints are linear inequalities, as the system developed in this article is independent of the criteria used to generate constraints.

In order to formulate structural criteria for well-formedness of signal relations, let us first define a number of terms and quantities pertaining to the different *kinds* of variables and equations. Given a signal relation, the number of interface variables (Sect. 3.3) is denoted by i_Z. The number of local variables, denoted l_Z, is then just the number of variables occurring in the equations minus the number of interface variables. The set of equations in a signal relation can be partitioned into disjoint subsets of interface, local, and mixed equations:

- *interface equation*: only interface variables occur.
- *local equation*: only local variables occur.
- *mixed equation*: both interface and local variable occur.

The number of interface, local, and mixed equations is denoted i_Q, l_Q, and m_Q respectively. Consequently, the total number of equations $a_Q = i_Q + l_Q + m_Q$.

A signal relation is *structurally well-formed* if the following 5 criteria are satisfied:

1. $l_Q + m_Q \geqslant l_Z$: The local variables are not underconstrained.
2. $l_Q \leqslant l_Z$: The local variables are not overconstrained.
3. $i_Q \leqslant i_Z$: The interface variables are not overconstrained.
4. $a_Q - l_Z \leqslant i_Z$: A signal relation must not contribute more equations than there are interface variables (no over-contribution).
5. $l_Q \geqslant 0, m_Q \geqslant 0, and\ i_Q \geqslant 0$: When considering structurally dynamic systems, we will permit negative contributions at intermediate stages, but insist that ultimately, the contribution of each equation kind should be non-negative.

To illustrate, let us return to the *resistor* example from Sect. 3.3. We have $i_Z = 4$ (recall that each *Pin* contains two variables), $l_Z = 1$, $i_Q = 0$, $l_Q = 0$, $m_Q = 3$ (the application of *twoPin* contributes 2 mixed equations), and thus $a_Q = 3$. The following 7 constraints are generated from the 5 criteria: (1) $0 + 3 \geqslant 1$, (2) $0 \leqslant 1$, (3) $0 \leqslant 4$, (4) $3 - 1 \leqslant 4$, and (5) $0 \geqslant 0$, $3 \geqslant 0$, $0 \geqslant 0$. All constraint criteria are satisfied. Hence, *resistor* is structurally well-formed according to the above criteria.

The question remains as to how the above criteria relate to the two criteria discussed in the previous section. The criteria here are stronger than insisting on balance, as a modular form of variable counting can be derived using criteria (4) and (5) alone. However, the constraints are weaker than insisting on a bijection between equations and variables: the constraints would need to consider the incidence matrices of equations and variables to determine if a bijection exists, as investigated by Nilsson [22]. However, by taking some account of which variables occur in which equations through the partitioning into interface, local, and mixed equations, we have achieved a better

approximation to checking for structural non-singularity than basic balance checking, while retaining a modular formulation that, as we will see in the next section, can be extended to account for structural dynamism.

4.3 Well-formedness and Structural Dynamism

Recall that a structurally dynamic system of equations is one where the equations are allowed to vary over time (Sect. 2.4). As FHM permits structurally dynamic systems (Sect. 3.4), we need to consider how to generalise the notion of structural well-formedness to work in a structurally dynamic setting. The nature of structural dynamism in FHM means that a very large, possibly even unbounded, number of system configurations are possible. Thus, we cannot hope to enumerate the configurations and check each one. Rather, we need to reconcile the structural properties of the branches of the switch blocks (the variable parts of an FHM system) - without losing too much information - into structural properties that hold at all times for each switch block as a whole, and then use this reconciled information to ascertain the well-formedness of the entire system.

There are a number of ways to compare the structure of different switch branches. One approach might be to insist that each branch have an identical structure: every branch consists of the same number of each kind of equation. Let us call this the *strong* approach for the purpose of this discussion. However, this approach is very restrictive. To understand why, consider a switch with two branches: the first branch consists of an interface equation and a local equation, the second branch consists of two mixed equations. These branches clearly have a very different structure, but are arguably interchangeable: both branches can be used to solve for one interface variable and one local variable.

An obvious alternative is to discard the equation kind information altogether and require only that each branch of a switch block contribute the same number of equations. Let us call this the *weak* approach. Clearly, the previous example now checks under this scheme as both branches contribute 2 equations. However, this approach is arguably too permissive: there are equation systems that contribute the same number of equations but are not structurally compatible. Indeed, this was the very reason to introduce equation kinds in the first place.

Instead, we adopt reconciliation constraints that enforce a stronger notion of structural compatibility than simple equation-variable balance, without requiring the branches of a switch block to be structurally identical. We refer to this as the *fair* approach. The constraints are defined over an n-branch switch block, containing n sets of equations $q_1 \ldots q_n$, where q_k consists of l_k local equations, m_k mixed equations, and i_k interface equations. The variables l, m, and i are fresh variables denoting the local, mixed, and interface contribution of the reconciled block as a whole. The constraints are parametrised on k, and the reconciliation constraints for a switch block are obtained by instantiating them for each branch:

6. $l \geqslant l_k \geqslant 0$: The reconciled system contributes at least as many local equations as the systems being reconciled. All local contributions must be positive.

7. $i \geqslant i_k \geqslant 0$: The reconciled system contributes at least as many interface equations as the systems being reconciled. All interface contributions must be positive.

8. $m \leqslant m_k - (l - l_k) - (i - i_k)$: The reconciled system may use mixed equations (from inside or outside the switch block) to compensate for any deficit in the required number of interface or local equations. This may result in m being negative, requiring the enclosing context of the switch block to contribute additional mixed equations.

9. $l + m + i = l_k + m_k + i_k$: The reconciled system contributes the same number of equations as each branch. Thus, each branch must have the same contribution.

The driving intuition is that we must find and associate some specific, *time-invariant* number of local variables and interface variables with each switch block such that the block, regardless of which branch is active, can provide that many equations to solve for interface and local variables respectively. The reason is that we then can rely on the block to *always* contribute equations to that end, meaning we effectively can view the block as a static equation system fragment with that specific contribution. Note that these two numbers must be at least as high as the maximal number of local equations and interface equations respectively over all branches. Otherwise some branches will contribute more local or interface equations than can be used. A subtlety is that the number of *mixed* equations contributed by a switch block is allowed to be negative. This just means that the switch block may need to "borrow" some mixed equations from the enclosing context in order to make up for a deficit of the number of local or interface equations in some branches.

To demonstrate, consider the following contrived example $dynamism_1$:

$$
\begin{array}{l}
dynamism_1 : SR\,(\mathbb{R},\mathbb{R}) \to SR\,\mathbb{R} \\
dynamism_1\ sr = \textbf{sigrel}\ x\ \textbf{where} \\
\quad \textbf{local}\ u \\
\quad \textbf{initially} \\
\quad\quad f\ u = 0 \\
\quad\quad g\ x = 0 \\
\quad \textbf{when}\ u < 0 \Rightarrow \\
\quad\quad sr \diamond (x,u)
\end{array}
$$

The relation contains a switch block with two branches: the **initially** branch consists of 1 local equation and 1 interface equation, while the **when** branch consists of n mixed equations, where n is the contribution of the relation sr. The switch block would be rejected under the strong approach, as the structure of the two branches is not identical.

However, under the fair approach, the block is reconcilable. Applying the rules to each branch of the switch results in 8 constraints that must be satisfied: $l \geqslant 1 \geqslant 0, l \geqslant 0 \geqslant 0, i \geqslant 0 \geqslant 0, i \geqslant 1 \geqslant 0, m \leqslant 0 - (l-1) - (i-1), m \leqslant n - (l-0) - (i-0)$, $l + m + i = 2$, and $l + m + i = n$. Through simplification, we can verify that they are satisfiable with $l = 1, m = 0, i = 1$, and $n = 2$.

For another example, consider $dynamism_2$ below. The switch block provides an interface equation in one branch and a local equation in the other. These branches are thus not immediately reconcilable. However, by considering the mixed equation in the enclosing context, it is possible for the entire relation to be balanced, regardless of which branch is active:

$$
\begin{array}{l}
dynamism_2 : SR\,\mathbb{R} \\
dynamism_2 = \textbf{sigrel}\ x\ \textbf{where} \\
\quad \textbf{local}\ u
\end{array}
$$

```
initially
  f x
when x > 0 ⇒
  g u
h x u
```

Applying the fair approach results in the following constraints: $l \geqslant 0 \geqslant 0, l \geqslant 1 \geqslant 0$, $i \geqslant 1 \geqslant 0, i \geqslant 0 \geqslant 0, m \leqslant 0-(l-0)-(i-1), m \leqslant 0-(l-1)-(i-0)$, $l+m+i = 1, l+m+i = 1$. Simplifying the constraints yields a solution at $l = 1$, $i = 1$, $m = -1$. Thus, the switch block contributes (or in this instance *requires*) -1 mixed equations. The interpretation of the above is that the switch block may be reconciled provided that it appears in a context containing at least 1 mixed equation.

Finally, consider the example $dynamism_3$ where the weak approach is too permissive, but, by contrast, the fair approach correctly rules out the switch block as irreconcilable:

```
dynamism₃ : SR ℝ
dynamism₃ = sigrel x where
  local u v
  initially
    u     = v
    f u v = 0
  when u + v < 0 ⇒
    g x = 0
    x   = u
```

The **initially** branch consists of 2 local equations, whereas the **when** branch consists of 1 interface equation and 1 mixed equation. Clearly, with only a single mixed equation, it should not be possible to account for the 2 local equations demanded by the reconciled relation. Indeed, running the criteria over the above relation results in the constraints $l \geqslant 2$, $i \geqslant 1$, and $l+m+i = 2$, implying that $m \leqslant -1$. However, there are no additional mixed equations in the enclosing context, and criterion 5 insists that m must be non-negative when checking the body of a signal relation. Hence, $dynamism_3$ is rightly rejected.

5 A Structural Type System

This section constitutes the main technical contribution of the article, presenting and formalising a type system for checking structural properties of equation-based languages. The type system is developed for a modest language of equations embedded into the simply-typed λ-calculus. Such an embedding reflects the two-tiered approach also used by FHM. The type system includes polymorphic types, similar to those found in a Hindley-Milner-style polymorphic system. However, here, polymorphic types are used to describe systems that are somehow *flexible* in their equation structure. Fig. 3 gives the notational conventions used throughout the remainder of this section.

5.1 Key Ideas

The fundamental idea behind the type system is to refine the type of a signal relation by including structural information. Specifically, a *balance* is associated with each signal

Description	Symbol	Description	Symbol
t	λ-terms	τ	functional monotypes
v	λ-values	σ	functional polytypes
x, y, z	λ-variables	ν	equation types
Γ	contexts	μ	constraint equation types
q	equation terms	k	kinds
qv	equation values	n, m, o	balance variables
sw	switch blocks	C, D, E	constraints
sv	switch values	c, d	constraint expressions
u, v, w	local signal variables		

Figure 3: Notational Conventions

relation type to indicate the number of equations that a signal relation *contributes* when used as a component of a larger system of equations.

FHM is a language for higher-order modelling, permitting equation systems to appear as parameters. As a result, the structural information required to compute an exact contribution is not necessarily known à priori. Hence signal relation types are generally parametric in their balance, through a *balance variable*, allowing for polymorphic signal relations that contribute a varying number of equations depending on the usage context.

However, the contribution of a signal relation (represented by a balance variable) is subject to *balance constraints* (simply *constraints* from now on) dictated by the structural criteria from Sect. 4. The context in which a signal relation is applied is used to generate the constraints, which must be satisfiable for a type to be valid. The constraints restrict the contribution of a component to a contiguous interval. Constraints may involve the contributions of several components, and are thus not directly associated with a single signal relation in general, as will become clear in the following.

To illustrate, consider the refined type for *resistor* from Sect. 3.3. We adopt a syntax similar to Haskell's for type class constraints to express constraints on balance variables:

$$\mathit{resistor} : (n = 2) \Rightarrow \mathit{Resistance} \rightarrow \mathit{SR}\,(\mathit{Pin}, \mathit{Pin})\,n$$

Here, the balance variable n is constrained to the value *2*. This can be verified by first flattening the signal relation applications to obtain a set of *3* equations over *5* variables (note that each *Pin* contains two variables), then removing one equation which must be used to solve for the local variable u, giving a net contribution of two equations.

Note that a representation of expressions containing integers and linear inequalities has been introduced at the type level. This extension may appear to be a restricted form of dependent types [19]. However, these type level representations, whilst determined by the structure of terms, are not value-level terms themselves. Hence, we do not consider our system to be dependently typed.

At this point, it is useful to consider the meaning of constrained types. Intuitively, $t : C \Rightarrow \tau$ means that if it is possible to find a valuation for all balance variables such that the constraints in C are satisfied, then τ is a valid type for the term t. As an example, consider the contrived type:

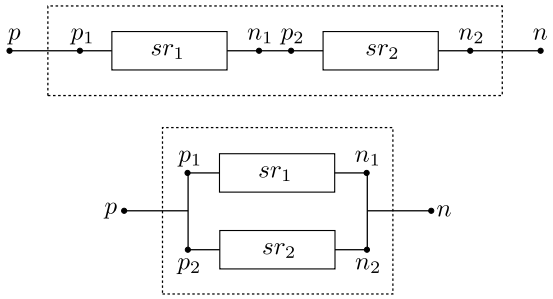


Figure 4: Serial (top) and parallel (bottom) composition of two-pinned circuit components.

$$(2 \leqslant m \leqslant 4, 3 \leqslant n \leqslant 5) \Rightarrow SR\, m \rightarrow SR\, n$$

This can be viewed as the type of a signal relation parametrised on a signal relation. The following types are all examples of valid instances of the above:

$$\begin{array}{ll} (m = 2, n = 3) & \Rightarrow SR\, m \rightarrow SR\, n \\ (m = 4, n = 4) & \Rightarrow SR\, m \rightarrow SR\, n \\ (m = 3, 4 \leqslant n \leqslant 5) & \Rightarrow SR\, m \rightarrow SR\, n \end{array}$$

An appropriate way to think of a signal relation parametrised on signal relations is that it is required to *accept* relations with any specific balances as long as the associated constraints are satisfied, and that once applied to relations with appropriate balances is capable of *contributing* a number of equations within bounds defined by the remaining constraints.

The power and utility of the type refinements can be seen in the examples given below. The examples define higher-order combinators over two-pinned circuit components, giving rise to refined types that are parametric in the resultant contribution. The combinators define serial and parallel composition of two-pinned circuit components. The visual representation in Fig. 4 shows the topological structure that is captured by the two combinators. In both cases, 4 new pins, p_1, p_2, n_1, and n_2 are created *internally* to wire together the components, where the naming conventions p and n refer to the positive and negative pins of a component, respectively. In Hydra, the internal pins correspond to new local variables. The equations in *parallel* and *serial* are then just applications of the subcomponents along with Kirchhoff's first and second laws for electrical circuits [18]:

$$\begin{array}{l} parallel : SR\,(Pin, Pin) \rightarrow SR\,(Pin, Pin) \rightarrow SR\,(Pin, Pin) \\ parallel\, sr_1\, sr_2 = \\ \quad \textbf{sigrel}\,(p, n)\ \textbf{where} \\ \quad\quad \textbf{local}\ p_1\ p_2\ n_1\ n_2 \\ \quad\quad sr_1 \diamond (p_1, n_1) \\ \quad\quad sr_2 \diamond (p_2, n_2) \\ \quad\quad p.i + p_1.i + p_2.i = 0 \\ \quad\quad n.i + n_1.i + n_2.i = 0 \\ \quad\quad p.v = p_1.v \\ \quad\quad n.v = n_1.v \end{array}$$

$$p_1.v = p_2.v$$
$$n_1.v = n_2.v$$

$$\begin{array}{l}
serial : SR\,(Pin,Pin) \rightarrow SR\,(Pin,Pin) \rightarrow SR\,(Pin,Pin) \\
serial\,sr_1\,sr_2 = \\
\quad \textbf{sigrel}\,(p,n)\ \textbf{where} \\
\quad\quad \textbf{local}\ p_1\ p_2\ n_1\ n_2 \\
\quad\quad sr_1 \diamond (p_1,n_1) \\
\quad\quad sr_2 \diamond (p_2,n_2) \\
\quad\quad -p.i + p_1.i = 0 \\
\quad\quad n_1.i\ + p_2.i = 0 \\
\quad\quad n_2.i\ - n.i\ = 0 \\
\quad\quad p.v\ = p_1.v \\
\quad\quad n_1.v = p_2.v \\
\quad\quad n_2.v = n.v
\end{array}$$

Upon applying the constraint criteria from Sect. 4 to compute the refined types, it transpires that parallel and serial share the same constraints, and thus, the same refined type. This give us some reassurance that the type system is imposing sensible constraints: the two circuits are connected in different ways, with different equations describing the composition, but in both cases we arrive at the same "composition constraints":

$$\begin{array}{l}
parallel, serial : (o = n + m - 2, 0 \leqslant o \leqslant 4) \Rightarrow \\
\quad SR\,(Pin,Pin)\,n \rightarrow SR\,(Pin,Pin)\,m \rightarrow SR\,(Pin,Pin)\,o
\end{array}$$

Note that in both cases there are 8 local variables (as each pin constitutes two variables) and 6 atomic equations that can be used to solve for them, meaning that the two signal relation applications at least have to contribute two more equations.

The constraints are also parametric in n, m, and o. Hence, they may be safely instantiated to any set of values satisfying these constraints, for example: $\{n \mapsto 1, m \mapsto 2\}$ or $\{n \mapsto 3, m \mapsto 3\}$. For further reassurance, consider using *parallel* to compose two resistors. The refined type of the composition admits the same constraints as the resistor component in isolation, i.e. the composition of two resistors is, as expected, just another resistor:

$$\begin{array}{l}
parRes : (n = 2) \Rightarrow SR\,(Pin,Pin)\,n \\
parRes = parallel\,(resistor\ 1000)\,(resistor\ 2200)
\end{array}$$

The goal of a type checker is not merely to accept well-typed programs, but also to reject certain ill-formed programs as ill-typed. The definition *broken* given below is such a program. The program is flawed in that there is no relation to which it can safely be applied. The relation *sr* must contribute at least *3* equations for the local variables $\{u,v,w,x\}$ (the forth being accounted for by the local atomic equation), but must not exceed a contribution of 2 equations as dictated by the second application. Consequently, our type system detects this by attempting to generate inconsistent constraints. Note that this program would happily be accepted by the unrefined type system.

$$\begin{array}{l}
broken : (\ldots, n \leqslant 2, n \geqslant 3, \ldots) \Rightarrow SR\ Pin\ n \rightarrow SR\ Pin\ m \\
broken\ sr = \textbf{sigrel}\,(a,b)\ \textbf{where} \\
\quad \textbf{local}\ u\ v\ w\ x
\end{array}$$

$sr \diamond (u + v, w + x)$
$sr \diamond (a, b)$
$v + x = 0$

5.2 An FHM Core Language

One of the primary goals of this article is to formalise the intuition of the type system discussed earlier in this section. A prerequisite to formalising such a system is to make precise the object language of study. Figures 5–8 define such a language, an FHM *core language*, which shall be used as the basis for metatheoretical study of our refined type system throughout the remainder of the article.

However, to allow us to focus on the structural aspects of the type system, the core language has been simplified compared with what would be needed for actual modelling. Thus, the core language deviates from the description of FHM and Hydra given in Sect. 3 in a number of ways. Most notable is the exclusion of signal-level constructs and signal types. Whilst this exclusion might seem like a major departure from Hydra, the refined type system is only concerned with the *kind* of equations and signal relation applications that arise in an equation system, information that can easily be determined prior to type checking. The core language is also based upon the simply-typed λ-calculus rather than Haskell. However, as we shall see, the type system also shares many similarities with Hindley-Milner-style polymorphic calculi [20].

The *types* in the core language are grouped into three categories: monotypes (τ), polytypes (σ), and equation types (μ, ν), see Fig. 5. The monotypes describe the simple function spaces and signal relation types, which are monomorphic in any balance variables. This small set of monotypes would be far richer in a real implementation, but is kept minimal for the sake of formalisation. Dispensing with signal-level values also means that signal relation types need now only be parametrised on a balance (rather than signal-level type and balance), eliminating the need to represent signal types in the core language at all.

Polytypes allow balance variables occurring in monotypes to be *bound*. This is very similar to the notion of binding of type variables found in Hindley-Milner-style type systems. The definition of polytypes is also a convenient place to introduce constraints on monotypes.

Equations in Hydra are essentially untyped, requiring only that their components be well-typed where appropriate. However, in the core language it is necessary to introduce simple equation types (ν) for the sake of constraint tracking. Equation types indicate the number of local, mixed, and interface equations that a compound equation (i.e., system of equations) contains. A numeric literal is not sufficient to describe the contribution, as an equation may contain elements whose contribution cannot be determined statically, for example the application of a signal relation introduced by a λ-abstraction. Thus, it is important to recognise that these expressions can only be approximations of the actual contribution. The category μ associates an equation type with constraints that may be inherited from any signal relations being applied within the compound equation.

The syntax of constraints is given in Fig. 6. Constraint expressions (c) are essentially $\langle \mathbb{Z}, + \rangle$: the group of integers closed under addition. This makes it easy to normalise and compare expressions, which is essential for determining type equality. A minimal language of constraints (C) provides inequality and conjunction. Equality

$\sigma ::=$	polytype:
$\forall\, n\,.\,\sigma$	quantified type
$C \Rightarrow \tau$	constrained type
$\tau ::=$	monotype:
$\tau_1 \rightarrow \tau_2$	function space
$SR\ n$	signal relation
$\mu ::=$	equation type:
$C \Rightarrow \nu$	constrained equation
$\nu ::=$	simple equation:
$Eq\ c_1\ c_2\ c_3$	equation

Figure 5: Types.

constraints $c_1 = c_2$ are translated to $c_1 \leqslant c_2, c_2 \leqslant c_1$, while the shorthand notation $c_1 \leqslant c_2 \leqslant c_3$ is expanded to $c_1 \leqslant c_2, c_2 \leqslant c_3$.

Type equality for monotypes is syntactic. Equality of polytypes is up to α-renaming of bound balance variables and equality of any constraints. Two constrained monotypes are equal when their constraints agree on the intervals of each balance variable. See section 5.6 for a discussion on our present approach to checking constraint equality.

$\Gamma ::=$	context:
$\bullet$	empty
$\Gamma, x : \sigma$	extension
$C ::=$	constraint:
ε	empty constraint
$c_1 \leqslant c_2$	expression inequality
c_1, c_2	constraint conjunction
$c ::=$	constraint expression:
n	balance variable
zero	zero
succ c	successor
$c_1 + c_2$	addition
$-\,c$	negation

Figure 6: Contexts and constraints.

Finally, the syntax of core terms and values are given in Fig. 7 and Fig. 8, respectively. As the simply-typed λ-calculus has been chosen as a template for the core language, the functional-level terms (t) contain the expected λ-terms: variables, function abstraction, and function application. Let bindings are introduced to allow us to *generalise* balance variables, Hindley-Milner style, allowing for polymorphic balance. To keep the system focused on balance aspects, no other polymorphism is supported. It would be straightforward to add support for other forms of polymorphism.

The abstraction over the signal-level is evident in the **sigrel** construct where the pattern introducing the bound variables has been replaced by two natural numbers, i and l, giving the number of interface variables and local variables in scope, respectively. While we are no longer concerned with the binding of signal-level variables, it is still necessary to keep track of the *number* of interface and local variables for the purpose of generating and checking constraints. Local variables are thus accounted for explicitly in the core calculus. Additionally, the number of local variables will increase in a non-trivial way during evaluation. Care is taken in the semantics to correctly account for this non-standard reduction behaviour. Rather than being eliminated due to substitution, local variables are instead propagated up and aggregated in the top level signal relation.

The most important simplification compared with Hydra can be seen in the productions for equations (q). Instead of classifying equations according to what signal variables occur in them as part of the type system, equations are classified directly by labelling them with a *kind*: **local**, **mixed**, or **interface**. This simplification makes our presentation of the type system substantially clearer, without compromising on any of the fundamental concepts of FHM as the labelling can easily be carried out prior to type checking.

Equations may also take the form of a *switch block* (*sw*). The syntax of switch blocks defines a non-empty list of equations, with the initially active branch tagged by the keyword **initially**. In Hydra (Sect. 3.4), each **when**-branch carries a signal expression that provides the condition for activating the branch. As the core language is not concerned with signal-level expressions, this condition is omitted from the syntax. Figure 9 illustrates how Hydra is mapped into the core lanuage.

$t ::=$	functional term:
x	λ-bound variable
$t_1\ t_2$	application
$\lambda\ x\ .\ t_1$	abstraction
let $x\ =\ t_1$ **in** t_2	let binding
sigrel $i\ l$ **where** q	signal relation
$sw ::=$	switch block:
initially q	initial branch
sw **when** q	conditional branch
$q ::=$	equation term:
atomic k	atomic equation
$t \diamond k$	sig. rel. application
$q_1 \wedge q_2$	pairing
sw	switch block
$k ::=$	equation kind:
local	local equation
mixed	mixed equation
interface	interface equation

Figure 7: Terms.

The next section gives a small-step reduction semantics for the core language. This

$v ::=$	functional value:
$\quad \lambda\ x\ .\ t$	abstraction
$\quad$ **sigrel** $i\ l$ **where** qv	signal relation
$qv ::=$	equation value:
$\quad$ **atomic** k	atomic equation
$\quad qv_1 \wedge qv_2$	pairing
$\quad sv$	switch block
$sv ::=$	switch value:
$\quad$ **initially** qv	initial branch
$\quad sv$ **when** q	conditional branch

Figure 8: Values.

semantics depends on the definition of values, which describe terms that have been reduced as far as is desirable. The body of an abstraction value is a term, meaning that reduction is not performed under binders. This is desirable, as it means open values (values containing free variables) need not be considered. Furthermore, equation values need not contain a production for signal relation applications, as all relation applications will be eliminated from a closed term. Finally, switch values only insist that the **initially** branch be an equation value, as this is the only active branch at the start of simulation. Should a branch condition fire, causing that branch to be switched and the the previously active branch to be switched out, this new branch will be treated as initial, and evaluated before simulation resumes.

5.3 Semantics

Meaning is ascribed to the core language via a small-step reduction semantics [29] given in Fig. 10. The semantics consist of two relations that describe the valid individual reductions for terms and equations. The reflexive transitive closure of these relations can then be taken as the sequences of valid reductions from terms to values.

The relation $t_1 \longrightarrow t_2$ gives meaning to terms, stating that the term t_1 reduces to the term t_2 in one step. The relation $q_1 \xrightarrow{l} q_2$ relates three objects, stating that the equation q_1 reduces to the equation q_2 in one step, introducing l new local variables in the process. If the semantics were to account for bound signal variables individually, such a reduction would have the form $\Delta \vdash q_1 \longrightarrow \Delta, \Sigma \vdash q_2$, describing the reduction of an equation q_1 in the local variable context Δ, to the equation q_2 in the local variable context Δ extended by a set of new local variables Σ.

There are no surprises regarding the reduction rules for λ-terms, with S-APP1, S-APP2, S-APPABS, S-LET, and S-LETV describing the normal rules for a call-by-value λ-calculus with **let**-expressions. A signal relation may undergo a step of reduction by reducing the equation that it contains (S-SIGREL); the current set of local variables (l_1) must also be extended with any new local variables that result from this equation reduction (l_2).

The left branch of an equation pair is reduced first, with the right branch being reduced only after a value is found for the left, as dictated by the rules S-PAIR1 and S-PAIR2. This ordering is arbitrary, but necessary for a deterministic semantics.

```
parallel sr1 sr2 =
  sigrel (p,n) where
    local p1 n1 p2 n2
    sr1 ◇ (p1,n1)
    sr2 ◇ (p2,n2)
    p.i + p1.i + p2.i = 0
    n.i + n1.i + n2.i = 0
    p.v = p1.v
    n.v = n1.v
    p1.v = p2.v
    n1.v = n2.v
```

(a) Hydra: parallel composition

```
parallel =
  λ sr1 . λ sr2 . sigrel 4 8 where
    sr1 ◇ local ∧
    sr2 ◇ local ∧
    atomic mixed ∧
    atomic mixed ∧
    atomic mixed ∧
    atomic mixed ∧
    atomic local ∧
    atomic local
```

(b) Core: parallel composition

```
icDiode = sigrel (p,n) where
  local u
  twoPin ◇ (p,n,u)
  initially; when p.v − n.v > 0 ⇒
    u = 0
  when p.i < 0 ⇒
    p.i = 0
```

(c) Hydra: ideal diode

```
icDiode = sigrel 4 1 where
  twoPin ◇ mixed ∧
  initially
    atomic local
  when
    atomic local
  when
    atomic interface
```

(d) Core: ideal diode

Figure 9: Comparison of Hydra and Core.

A deterministic reduction strategy for $\diamond$ is achieved by first reducing the term being applied. This does not bring any new local variables into scope (S-RAPP). Once the left-hand side of $\diamond$ has been reduced to a signal relation value, the application is reduced to the equation values contained within the relation (S-RAPPABS). If the signal level were not treated in the abstract, the signal expression appearing to the right of $\diamond$ would need to be substituted into the body of these equations. However, the casting of equation kinds ($\downarrow$ below) reflects aspects of this substitution. Finally, the new local variables brought into scope by this reduction step are the local variables declared within the signal relation value.

Note in the rule S-RAPPABS how the equations from within the applied signal relation are transformed by the function $\downarrow$ to bring them into the enclosing context. This transformation is included to keep the notion of equation kinds in line with the kinds expected by the enclosing relation. As discussed previously, the constrained types are necessarily approximations of the actual structure of the flattened system of equations. As reduction proceeds, new information about the flattened structure becomes available, specifically, information regarding the precise kinds of each equation contained with a signal relation abstraction.

$$\begin{array}{ll} \textbf{atomic local} & \downarrow k = \textbf{atomic local} \\ \textbf{atomic mixed} & \downarrow k = \textbf{atomic mixed} \\ \textbf{atomic interface} & \downarrow k = \textbf{atomic}\ k \end{array}$$

$$\frac{t_1 \longrightarrow t_2}{t_1\ t_3 \longrightarrow t_2\ t_3}\ \text{(S-APP1)} \qquad \frac{t_1 \longrightarrow t_2}{v\ t_1 \longrightarrow v\ t_2}\ \text{(S-APP2)}$$

$$\frac{}{(\lambda\, x\, .\, t)\ v \longrightarrow [x \mapsto v]\, t}\ \text{(S-APPABS)}$$

$$\frac{q_1 \xrightarrow{l_2} q_2}{\textbf{sigrel}\ i\ l_1\ \textbf{where}\ q_1 \longrightarrow \textbf{sigrel}\ i\ (l_1 + l_2)\ \textbf{where}\ q_2}\ \text{(S-SIGREL)}$$

$$\frac{t_1 \longrightarrow t_2}{\textbf{let}\ x = t_1\ \textbf{in}\ t_3 \longrightarrow \textbf{let}\ x = t_2\ \textbf{in}\ t_3}\ \text{(S-LET)}$$

$$\frac{}{\textbf{let}\ x = v\ \textbf{in}\ t \longrightarrow [x \mapsto v]\, t}\ \text{(S-LETV)} \qquad \frac{t_1 \longrightarrow t_2}{t_1 \diamond k \xrightarrow{0} t_2 \diamond k}\ \text{(S-RAPP)}$$

$$\frac{}{(\textbf{sigrel}\ i\ l\ \textbf{where}\ qv) \diamond k \xrightarrow{l} qv \downarrow k}\ \text{(S-RAPPABS)}$$

$$\frac{q_1 \xrightarrow{l} q_3}{q_1 \wedge q_2 \xrightarrow{l} q_3 \wedge q_2}\ \text{(S-PAIR1)} \qquad \frac{q_1 \xrightarrow{l} q_2}{qv \wedge q_1 \xrightarrow{l} qv \wedge q_2}\ \text{(S-PAIR2)}$$

$$\frac{q_1 \xrightarrow{l} q_2}{\textbf{initially}\ q_1 \xrightarrow{l} \textbf{initially}\ q_2}\ \text{(S-INITIAL)}$$

$$\frac{sw_1 \xrightarrow{l} sw_2}{sw_1\ \textbf{when}\ q \xrightarrow{l} sw_2\ \textbf{when}\ q}\ \text{(S-WHEN)}$$

Figure 10: Small step semantics.

$$\begin{array}{ll} qv_1 \wedge qv_2 & \downarrow k = (qv_1 \downarrow k) \wedge (qv_2 \downarrow k) \\ \textbf{initially}\ qv & \downarrow k = \textbf{initially}\ (qv \downarrow k) \\ sv\ \textbf{when}\ q & \downarrow k = (sv \downarrow k)\ \textbf{when}\ q \end{array}$$

The function $\downarrow$ is thus *not* an abstraction of substitution of signal-level variables from the enclosing context into the equations of the applied signal relation, except that local equations must remain local because they are not affected by substitution (as no interface variables occur in them) and because they are not included in the balance of the applied signal relation. Mixed equations also remain **mixed** to retain an optimistic view that they still might be used to solve for any variable, including interface variables in a larger enclosing context, or **local** variables from the initial context in which the equation was defined.

The **initially** branch of a switch is the only branch that will be active at the start of simulation. Thus, the semantics of **initially** and **when** are only concerned with reducing the equations within this initial branch.

5.4 Typing Rules

The type system is defined through three relations (Fig. 11). We have opted for a declarative presentation: in particular, our approach describes polymorphic aspects of the system using non-deterministic *generalisation* and *instantiation* rules in the spirit of the original paper by Milner [20]. The algorithm used to compute types in our system is omitted here as it is essentially the standard algorithm: see Sect. 5.6 for a brief discussion, or the supporting implementation for details [6]. Alternatively, we could have taken the approach found in Pierce [28], making type reconstruction constraints appear explicitly in the rules. However, we felt that this obfuscated the presentation by hiding the important details of the type refinements. The nature of the type system also requires a number of simple type-level computations to be performed for which we define ancillary functions.

The relation for checking λ-terms is denoted by $\Gamma \vdash t : \sigma$, stating that a term t has the type scheme σ in the typing context Γ. Similarly, the relation $\Gamma \vdash_q q : \mu$ states that an equation q has the constrained equation type μ in the context Γ. Equations require the functional typing context Γ as terms may appear in equations (to the left of $\diamond$).

The rules for variables (T-VAR), application (T-APP), abstraction (T-ABS), and let (T-LET) are the normal rules of the simply-typed λ-calculus with additional plumbing to handle the accumulation of constraints. Type checking signal relations (T-SIGREL) requires slightly more attention, using the type of the equations contained within the relation to generate a new set of constraints against the *fresh* balance variable n. The *free* function is responsible for ensuring that the new balance variable n does not already appear free in the context.

The rules for typing equations suggest a simple algorithm for traversing a tree of equations and accumulating the number of local, mixed, and interface equations that a compound equation is capable of *contributing*, whilst simultaneously aggregating constraints from applied signal relations. The strategy can be seen in the rule for relation application (T-RELAPP): given a relation of type *SR n* (i.e. a signal relation contributing n equations), create an equation type contributing n equations of kind k using the *kind* helper function.

Typing switch blocks is a matter of typing the equation fragments at each branch, and then reconciling these types using the approach outlined earlier in Sect. 4.3. Rather than considering all the branches at once, the typing rules provide a syntax-directed approach that reconciles each **when** branch with the remaining branches of the switch.

The instantiation (T-INST) and generalisation rules (T-GEN) allow types to be instantiated or generalised with respect to their balance variables, respectively. Instantiation insists that the new type be more specific according to the ordering relation $\sqsubseteq$. Generalisation allows a free balance variable to be bound, provided that it does not appear free in the environment. Note the strong correspondence between our notion of balance variables and the notion of type variables found in other Hindley-Milner-style systems.

The final matter that requires attention are the various ancillary functions used in the typing rules. The function *free* is left abstract, but it just returns the set of free variables of a context. The (pseudo) predicate *fresh* is also left abstract: it simply enforces that new variables are picked to prevent unintended interference with variables already in use. The operator $\oplus$ is used in T-PAIR to aggregate the contributions of two simple equation types:

$$(Eq\ c_1\ c_2\ c_3) \oplus (Eq\ d_1\ d_2\ d_3) = Eq\ (c_1 + d_1)\ (c_2 + d_2)\ (c_3 + d_3)$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \text{(T-VAR)} \qquad \frac{\Gamma \vdash t_2 : D \Rightarrow \tau_1 \quad \Gamma \vdash t_1 : C \Rightarrow \tau_1 \to \tau_2}{\Gamma \vdash t_1\, t_2 : C,D \Rightarrow \tau_2} \quad \text{(T-APP)}$$

$$\frac{\Gamma, x : \varepsilon \Rightarrow \tau_1 \vdash t : C \Rightarrow \tau_2}{\Gamma \vdash \lambda\, x\, .\, t : C \Rightarrow \tau_1 \to \tau_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : C \Rightarrow \tau_1 \quad \Gamma, x : C \Rightarrow \tau_1 \vdash t_2 : D \Rightarrow \tau_2}{\Gamma \vdash \textbf{let}\, x = t_1\, \textbf{in}\, t_2 : D \Rightarrow \tau_2} \quad \text{(T-LET)}$$

$$\frac{}{\Gamma \vdash_q \textbf{atomic}\, k : \varepsilon \Rightarrow \mathit{kind}\,(k,l)} \quad \text{(T-ATOMIC)}$$

$$\frac{\Gamma \vdash t : C \Rightarrow \mathit{SR}\, n}{\Gamma \vdash_q t \diamond k : C \Rightarrow \mathit{kind}\,(k,n)} \quad \text{(T-RELAPP)}$$

$$\frac{\Gamma \vdash_q q_1 : C \Rightarrow \nu_1 \quad \Gamma \vdash_q q_2 : D \Rightarrow \nu_2}{\Gamma \vdash_q q_1 \wedge q_2 : C,D \Rightarrow \nu_1 \oplus \nu_2} \quad \text{(T-PAIR)}$$

$$\frac{C = \mathit{cons}\,(\nu,n,i,l) \quad \Gamma \vdash_q q : D \Rightarrow \nu \qquad \mathit{fresh}\,(n)}{\Gamma \vdash \textbf{sigrel}\, i\, l\, \textbf{where}\, q : C,D \Rightarrow \mathit{SR}\, n} \quad \text{(T-SIGREL)}$$

$$\frac{D = \mathit{cons}_{sw}\,(\mathit{Eq}\, l\, m\, i, \nu) \quad \Gamma \vdash_q q : C \Rightarrow \nu \qquad \mathit{fresh}\,(l,m,i)}{\Gamma \vdash_{sw} \textbf{initially}\, q : C,D \Rightarrow \mathit{Eq}\, l\, m\, i} \quad \text{(T-INITIAL)}$$

$$\frac{\Gamma \vdash_q q : D \Rightarrow \nu_2 \quad \Gamma \vdash_{sw} sw : C \Rightarrow \nu_1 \quad E = \mathit{cons}_{sw}\,(\nu_1,\nu_2)}{\Gamma \vdash_{sw} sw\, \textbf{when}\, q : C,D,E \Rightarrow \nu_1} \quad \text{(T-WHEN)}$$

$$\frac{\Gamma \vdash x : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash x : \sigma_2} \quad \text{(T-INST)} \qquad \frac{\Gamma \vdash t : \sigma \qquad n \notin \mathit{free}\,(\Gamma)}{\Gamma \vdash t : \forall\, n\, .\, \sigma} \quad \text{(T-GEN)}$$

Figure 11: Typing rules.

The *cons* and cons_{sw} functions are responsible for generating constraints, as discussed in Sect 4. The signature of *cons* requires an equation type ν, a fresh balance variable n, and the number of interface variables i and local variables l. The cons_{sw} function requires two equation types ν_1 and ν_2 to be reconciled:

$$
\begin{aligned}
&cons\,(Eq\ c_I\ c_M\ c_L, n, i, l) \ = \\
&\quad n \ = \ c_I + c_M + c_L - l, \\
&\quad n \leqslant i, \\
&\quad c_I \leqslant i, \\
&\quad c_L \leqslant l \leqslant c_L + c_M, \\
&\quad c_I \geqslant 0, c_M \geqslant 0, c_L \geqslant 0
\end{aligned}
\qquad
\begin{aligned}
&cons_{sw}\,(Eq\ l\ m\ i, Eq\ l_k\ m_k\ i_k) \ = \\
&\quad l \geqslant l_k \geqslant 0, \\
&\quad i \geqslant i_k \geqslant 0, \\
&\quad m \leqslant m_k - (l - l_k) - (i - i_k), \\
&\quad l + m + i \ = \ l_k + m_k + i_k
\end{aligned}
$$

The *kind* function provides a convenient method for constructing equation types with a given contribution and of a particular kind:

$$
\begin{aligned}
kind\,(\textbf{local}, \quad c) &= Eq\ 0\ 0\ c \\
kind\,(\textbf{mixed}, \quad c) &= Eq\ 0\ c\ 0 \\
kind\,(\textbf{interface}, c) &= Eq\ c\ 0\ 0
\end{aligned}
$$

Finally, we define the $\sqsubseteq$ predicate with the rule given below. The rule ensures that no free variables occuring in the monotype become bound by a quantifier, but existing quantifiers may be replaced by new types, including types that introduce new balance variables:

$$
\frac{\tau_2 \ = \ [\alpha_i \ \mapsto \ \tau_i]\ \tau_1 \qquad fresh\,(\beta_i)}{\forall\, \alpha_1 \ldots \forall\, \alpha_n \,.\, \tau_1 \ \sqsubseteq \ \forall\, \beta_1 \ldots \forall\, \beta_m \,.\, \tau_2}
$$

5.5 Metatheoretical Properties

The soundness of a type system is often specified via two properties: progress and preservation (also referred to as subject reduction). A type system has progress if, for every closed, well-typed term t, either t is a value or else there is some t' for which $t \longrightarrow t'$. A type system has subject reduction if evalution of an expression preserves the type of that expression [28].

In our setting, preservation of refined types does not hold. While this might come as a surprise, and may seem indicative of underlying problems, it is actually entirely unsurprising given our objectives. As has been discussed throughout this article, the type refinements we propose are only an approximation for detecting anomalies that would definitely lead to a structurally singular system of equations. Hence, in Hydra, there are modular systems of equations where the fact that they are structural singular only become apparent once they have been completely flattened. In other words, during the process of evaluation, a Hydra program may not necessarily remain structurally well-formed (Sect. 4.1) according to our refinements (but it will remain well-typed). This is expected: the main point is that the refined type system enables many mistakes to be caught early.

The lack of subject reduction is due entirely to the type refinements. Therefore, we shall show that progress and preservation does hold for the unrefined system. Such a proof — which is a new, albeit minor, contribution of this article — gives us some reassurance that the foundations of our system are sound.

Rather than define a new set of typing rules we instead choose to work with *erased* types; the additions made to the type system by our refinements are essentially ignored. Specifically, we erase constraint sets, balance variables on signal relations (*SR*), and constraint expressions on equations (*Eq*).

Progress can be define formally as follows: given t where $\vdash \ t \ : \ \tau$, then either t is a value, or $\exists\, t' \,.\, t \ \longrightarrow \ t'$. The proof proceeds by induction on the typing derivations

(see Fig. 11). We will omit proofs where they do not differ from the standard (and well known) approach to proving soundness of the simply-typed λ-calculus [28].

1. T-ATOMIC: Trivial, atomic equations are values.
2. T-RELAPP: By the induction hypothesis, either there exists a t' such that $t \longrightarrow t'$, or t is a value. In the first case, the rule S-RAPP simply applies. If t is a value, then by canonicity is must be a **sigrel**, in which case S-RAPPABS applies.
3. T-PAIR: By the induction hypothesis, both q_1 and q_2 may be either values or may take a step of evaluation. If q_1 can take a step, then S-PAIR1 applies. If q_1 is a value, and q_2 may take a step, then S-PAIR2 applies. If both q_1 and q_2 are values, then the entire expression is a value.
4. T-SIGREL: If the hypothesis q may take a step then the rule S-SIGREL applies. If q is a value, then the expression as a whole is a value.
5. T-INITIAL: As above, S-INITIAL applies when q is a value.
6. T-WHEN: As above again, S-WHEN applies when sw is a value. □

Preservation is defined formally as: given t, such that $\Gamma \vdash t : \tau$, and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$. This proof proceeds by induction on the reduction step $t \longrightarrow t'$. As before, preservation proofs about the λ-calculus are omitted where they do not differ.

1. S-SIGREL: The induction hypothesis states that $\Gamma \vdash_q q_1 : \nu$, and $\Gamma \vdash_q q_2 : \nu$. By inversion of the rule T-SIGREL, we can deduce that ν is an equation type, and by extensions must be Eq. Thus, using T-SIGREL again directly on the left- and right-hand side, we can deduce the type SR for both.
2. S-RAPP: The induction hypothesis gives us $\Gamma \vdash t_1 : \tau$, and $\Gamma \vdash t_2 : \tau$. By inversion of T-RELAPP, $\tau = SR$, and hence by application again of T-RELAPP, both sides agree on the overall type Eq.
3. S-RAPPABS: Once again, we can decompose the hypothesis (**sigrel** $i\ l$ **where** qv) $\diamond$ k by inversion, resulting in $qv : \nu$, and thus $qv : Eq$. The type Eq for the left-hand side follows from the rules T-SIGREL and T-RELAPP successively.
4. S-PAIR1: Inversion of T-PAIR along with the induction hypothesis give a straightforward proof of preservation.
5. S-PAIR2: As above, commuted.
6. S-INITIAL: Follows from the induction hypothesis derived from $q_1 \longrightarrow q_2$ and inversion of T-INITIAL.
7. S-WHEN: Follows from the induction hypothesis derived from $sw_1 \longrightarrow sw_2$ and inversion of T-WHEN. □

The above correctness properties are not the only opportunity to validate our type system, not least because they do not consider the type refinements developed in this article. However, the most practically relevant such properties can only be stated in a setting of a language that does not abstract away from which variables occur in which equations; i.e., a language like Hydra, but unlike our core language. As we have not formalised Hydra or its semantics in this article, we do not pursue this further here, but we will sketch the kind of correctness properties we expect to hold, and why, in future work (Sect. 8).

5.6 Implementation

We have implemented a prototype type checker for the type system described in this section. It is implemented in the dependently-typed programming language Agda [26], thus ensuring the totality and termination of the checker. The checker can infer refined types without the need for any type annotations, meaning that the inference algorithm is also total in this sense. The implemented inference algorithm operates in much the same way as *Algorithm $\mathscr{W}$* with regards to generating type schemes. Specifically, we make no contributions to type inference and instead refer the interested reader to the supporting implementation for details [6].

As discussed in section 5.2, deciding type equality in part requires deciding the equality of constraints. Our present implementation does this by using the *Fourier-Motzkin Quantifier Elimination* algorithm [30], which determines a contiguous interval for each occurring balance variable. This allows the equality of two constrained monotypes to be checked by checking that the constraints agree on the intervals of each balance variable. Fourier-Motzkin's algorithm operates on linear systems of inequalities. Another alternative might have been Collin's Quantifier Elimination [11]. However, Collin's algorithm targets polynomial systems of inequalities which means that it may have higher time complexity that Fourier-Motzkin's algorithm. Thus, Fourier-Motzkin's algorithm is the better choice for us.

Fourier-Motzkin elimination has worst case exponential time complexity in the number of balance variables. However, as shown by Pugh [30], the modified variant that searches for integer solutions is capable of solving most common problem sets in low-order polynomial time. Furthermore, systems typically involve only a handful of balance variables, making it feasible to check most cases where complexity is exponential in the number of variables.

6 Evaluation

We have carried out our development in the context of an abstract version of an FHM-like, acausal modelling and simulation language, leaving out most aspects that were not directly relevant to our specific purposes. We did this partly to keep things simple and allow ourselves to focus on the core issues, and partly, as explained in the introduction, because the ideas underpinning our type system could be useful for any language with a notion of modular systems of equations.

However, this begs the question how we can evaluate what we have achieved insofar as we at this point are not in a position to carry out any large usability studies. In this section, we attempt to address that question in two ways. First, we position our work relative to other work based on exploiting structural properties of systems of equations for which there is independent evidence of usability. Second, we provide a fairly substantial case study that covers all aspects of the language, including structural dynamism.

6.1 Structural Properties in the Wild

Based on years of practical experience, a notion of balance checking was considered to be sufficiently useful to be incorporated into version 3.0 of the Modelica standard [21] in 2007. See Sect. 7.1 for a discussion of how Modelica compares to the work described in this paper from the perspective of variable and equation balance. Here we just point

out that our system checks more fine-grained structural properties than Modelica as we distinguish between different kinds of equations. This means our system is capable of catching a strictly larger set of errors, and thus is no less useful than the system presently used in Modelica. A concrete example is given towards the end of the case study in the next section. Additionally our type-based approach scales to first-class equation fragments and structurally dynamic systems of equations, features that may be commonplace in the next generation of acausal modelling languages [32].

The work by Bunus & Fritzson [5], discussed in Sec. 7.3, lies at the other end of the spectrum in terms of precision. Because they work on systems of equations after flattening, Bunus et al. are able to perform a global analysis, which is much more detailed than our type system, or Modelica's balance checking, is capable of. For example, Bunus & Fritzon show how their approach can identify specific equations as likely being the cause of a problem, and even prioritize among a number of ways to address a problem. In essence, the key difference is that Bunus & Fritzson do an analysis at the granularity of individual variable occurrences, while we approximate this by considering occurrences of variables only at the granularity of two different variable kinds: local and interface variables.

While Bunus and Fritzson's approach does not support checking of components in isolation, and is thus not a feasible starting point for a *type system* for modular equations, their approach does demonstrate the practical utility of taking more fine-grained structural properties into account than just the variable-equation balance.

In summary, in terms of "error finding power", the type system presented in this paper is somewhere between what currently is used in Modelica and the approach investigated by Bunus and Fritzson, both of which empirically are useful for finding problems. Yet, our type-based approach offer distinct advantages over both.

6.2 Case Study: Half-Wave Rectifier

To demonstrate the practical applications of the type system developed in this article, we now present a case study. At this point, the reader may want to first review the examples that were presented in Sect. 5.1. These demonstrated our type system at work, including how it can catch certain mistakes. However, the examples were small and in some cases also artificial. In contrast, this case study concerns a complete model of a half-wave rectifier composed of a number of electrical components including, in particular, a diode: see Fig. 12. We are going to model the diode as an ideal component (initially closed), resulting in a structurally dynamic model. The model, borrowed from a paper on FHM [24] and originally adapted from Cellier's and Kofman's book *Continuous System Simulation* [10, pp. 439-443], raises particular simulation challenges as the in-line inductor causes the causality to change when the model switches between the two different structural configurations (the ideal diode is open or closed).

Besides the diode, the half-wave rectifier includes a voltage source, an inductor, two resistors, a capacitor, and a ground reference. The implementation of some of these components, such as the resistor, can be found earlier in the paper. However, for convenience the definition of each of these components is given below along with their refined types (with trivially satisfied constraints omitted) and a brief justification for assigning each type.

First of all, recall the definition of *twoPin*, the abstraction that captures the common aspects of electrical components with two pins:

$$twoPin : (n = 2) \Rightarrow SR\,(Pin, Pin, Voltage)\,n$$

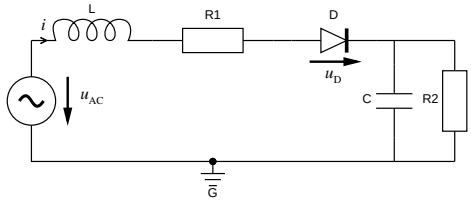


Figure 12: Half-wave rectifier with in-line inductor.

$$
\begin{aligned}
&\mathit{twoPin} \;=\; \textbf{sigrel}\;(p,n,u)\;\textbf{where}\\
&\quad p.i + n.i \;=\; 0\\
&\quad p.v - n.v \;=\; u
\end{aligned}
$$

There are manifestly two equations and no local variables to solve for, so the net contribution is two equations.

The alternating current voltage source is defined as follows, with the amplitude and frequency given by the parameters v and f, respectively:

$$
\begin{aligned}
&\mathit{vSourceAC} : (n \;=\; 2) \;\Rightarrow\; \mathit{Voltage} \;\rightarrow\; \mathit{Frequency} \;\rightarrow\; \mathit{SR}\;(\mathit{Pin},\mathit{Pin})\;n\\
&\mathit{vSourceAC}\;v\,f \;=\; \textbf{sigrel}\;(p,n)\;\textbf{where}\\
&\quad \textbf{local}\;u\\
&\quad u \;=\; v * \mathit{sin}\;(2 * \pi * f * \mathit{time})
\end{aligned}
$$

Applying the constraint criteria to the voltage source component gives an overall contribution of two equations. This contribution, along with the contributions of several of the components to follow, is easily justified: an application of *twoPin* contributes two equations, while the atomic equation is reserved for solving the local variable u. Applying the typing rules and then simplifying constraints yields the same result.

The resistor, inductor, and capacitor are defined as follows:

$$
\begin{aligned}
&\mathit{resistor} : (n \;=\; 2) \;\Rightarrow\; \mathit{Resistance} \;\rightarrow\; \mathit{SR}\;(\mathit{Pin},\mathit{Pin})\;n\\
&\mathit{resistor}\;r \;=\; \textbf{sigrel}\;(p,n)\;\textbf{where}\\
&\quad \textbf{local}\;u\\
&\quad \mathit{twoPin} \diamond (p,n,u)\\
&\quad r * p.i \;=\; u\\
&\mathit{inductor} : (n \;=\; 2) \;\Rightarrow\; \mathit{Inductance} \;\rightarrow\; \mathit{SR}\;(\mathit{Pin},\mathit{Pin})\;n\\
&\mathit{inductor}\;i \;=\; \textbf{sigrel}\;(p,n)\;\textbf{where}\\
&\quad \textbf{local}\;u\\
&\quad \mathit{twoPin} \diamond (p,n,u)\\
&\quad l * \mathit{der}\;p.i \;=\; u\\
&\mathit{capacitor} : (n \;=\; 2) \;\Rightarrow\; \mathit{Capacitance} \;\rightarrow\; \mathit{SR}\;(\mathit{Pin},\mathit{Pin})\;n\\
&\mathit{capacitor}\;c \;=\; \textbf{sigrel}\;(p,n)\;\textbf{where}\\
&\quad \textbf{local}\;u\\
&\quad \mathit{twoPin} \diamond (p,n,u)\\
&\quad c * \mathit{der}\;u \;=\; p.i
\end{aligned}
$$

Like the voltage source, the relations that result from *resistor*, *capacitor*, and *inductor* (after the application of any functional parameters) each contribute two equations for the reasons given above. After all, from the perspective of our type system, the sets

of equations that constitute each component are essentially the same: an application of *twoPin* to a set of mixed variables, and an atomic equation. The only difference is that the atomic equation in these cases is mixed.

The *ground* component, unlike previous components, is connected via only a single pin:

$$
\begin{array}{l}
\mathit{ground} : (n = 1) \Rightarrow \mathit{SR}\ \mathit{Pin} \\
\mathit{ground} = \textbf{sigrel}\ p\ \textbf{where} \\
\quad p.v = 0
\end{array}
$$

Its purpose it to set a reference voltage level. Thus, the component is very simple: it contains only a single equation and introduces no new local variables. Hence, our intuition would dictate that the ground component contributes one equation as there are no local variables. This is in agreement with the type assigned by our type system.

The final, and most involved component in the circuit is the initially closed, ideal diode:

$$
\begin{array}{l}
\mathit{icDiode} : (n = 2) \Rightarrow \mathit{SR}\ (\mathit{Pin}, \mathit{Pin})\ n \\
\mathit{icDiode} = \textbf{sigrel}\ (p,n)\ \textbf{where} \\
\quad\ \ \textbf{local}\ u \\
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad \textbf{initially}; \textbf{when}\ p.v - n.v > 0 \Rightarrow \\
\quad\quad u = 0 \\
\quad \textbf{when}\ p.i < 0 \Rightarrow \\
\quad\quad p.i = 0
\end{array}
$$

The diode is a particularly interesting example as the type of equations contributed is dependent upon the current structural configuration: initially, the switch block defines a local equation, whereas the second branch defines an interface equation. This conflict is resolved thanks to the *fair* policy (Sect. 4.3) employed when generating constraints for structurally dynamic code. The two branches of the switch block are reconciled by demanding that a mixed equation is present in the enclosing context. In other words, the switch block contributes 1 interface equation, 1 local equation, and -1 mixed equation. Thus, the net contribution of the diode is two: 2 mixed equations from the application of twoPin and 1 equation from the switch block, 1 of which must be used to solve for the local variable. At this point, it is worth noting that the *strong* approach would be too restrictive: the contributions from the different branches are clearly not identical.

The complete half-wave rectifier can now be described as follows:

$$
\begin{array}{l}
\mathit{halfWaveRectifier} : (n = 0) \Rightarrow \mathit{SR}\ ()\ n \\
\mathit{halfWaveRectifier} = \textbf{sigrel}\ ()\ \textbf{where} \\
\quad \textbf{local}\ \mathit{lp}\ \mathit{ln}\ \mathit{rp}_1\ \mathit{rn}_1\ \mathit{rp}_2\ \mathit{rn}_2 \\
\quad \textbf{local}\ \mathit{dp}\ \mathit{dn}\ \mathit{cp}\ \mathit{cn}\ \mathit{acp}\ \mathit{acn}\ \mathit{gp} \\
\quad \mathit{resistor}\ 1.0 \qquad\quad \diamond (\mathit{rp}_2, \mathit{rn}_2) \\
\quad \mathit{icDiode} \qquad\quad \diamond (\mathit{dp}, \mathit{dn}) \\
\quad \mathit{capacitor}\ 0.0 \qquad \diamond (\mathit{cp}, \mathit{cn}) \\
\quad \mathit{vSourceAC}\ 1.0\ 1.0 \diamond (\mathit{acp}, \mathit{acn}) \\
\quad \mathit{ground} \qquad\quad \diamond \mathit{gp} \\
\quad \textbf{connect}\ \mathit{acp}\ \mathit{lp} \\
\quad \textbf{connect}\ \mathit{ln}\ \mathit{rp}_1 \\
\quad \textbf{connect}\ \mathit{rn}_1\ \mathit{dp}
\end{array}
$$

connect *dn cp* rp_2
connect *acn cn* rn_2 *gp*

The rectifier is a non-trivial example, consisting of seven subcomponents, many of which have subcomponents of their own. However, if one follows the typing rules, it is a straightforward matter to construct the appropriate type. There are total of 26 local variables (recall that each pin contains two variables) and by no coincidence, the body of the relation contains a total contribution of 26 equations. Note that the **connect** keyword is used as a shorthand for Kirchhoff's circuit laws, where **connect** $p_1 \ldots p_x$ desugars to x atomic equations: a sum-to-zero equation and $x - 1$ voltage equalities.

The type system does not merely guarantee that the model is balanced, it strengthens the claim by imposing additional constraints that are also satisfied. For example, suppose the programmer made an error in the implementation of diode: instead of applying *twoPin* to a mixed set of variables (i.e. $twoPin \diamond (p,n,u)$), the application was instead made to a set of interface variables (i.e. $twoPin \diamond (p,n,0)$). In a setting with a fair number of both interface and local variables it is entirely plausible that such an error might go unnoticed. This mistake would mean that there are no mixed equations to satisfy the -1 mixed equation requirement of the switch block. Interestingly, if one were to only count variables and equations, without any notion of equation kinds (see related work, Sect. 7.1 and Sect, 7.2), the aforementioned error would not be detected. Furthermore, in our system this error would be detected early while type checking *icDiode*, and not only once the full model has been assembled.

7 Related Work

Equation-based modelling is a broad and varied topic. In this section, the most relevant work relating to static checking of structural properties is reviewed and compared with our own.

7.1 Modelica

Modelica is an industrial-strength, equation-based language for acausal modelling of hybrid systems. The language design draws heavily from concepts in object-oriented programming with notions like classes and inheritance used to structure the models. As of version 3.0 of the Modelica specification [21, pp. 43–48, p. 270] models are required to be locally balanced. A model is locally balanced if it locally declares or inherits the same number of variables and equations. No attempt is made to classify equations depending on whether the variables occurring in them are local or not. Moreover, the language specification only requires checking of the local balance once specific values of parameters are known. The number of variables and equations may depend on the constants through conditional selection among blocks of equations and array sizes. The possibility of checking that a model is locally balanced for *all* possible values of the parameters is left as a "quality-of-implementation" issue.

Compared to our approach, Modelica is quite restrictive: there are good reasons for why certain components need to be locally *un*balanced, and then used as building blocks of larger systems that ultimately will be balanced. For this reason, Modelica allows components to be marked as *partial*, thereby disabling balance checking (in isolation) for those components. Modelica also lacks a notion of true first-class models: there are methods for parametrising models on other models, but these do not approach

the generality of FHM. However, this does mean that checking balances late, once parameters are fully known, suffices in the case of Modelica. Furthermore, because Modelica does not classify equations depending on which variables occur in them, the class of structural properties checked by Modelica is smaller than that covered by our type system (see Sect. 4).

7.2 Broman, Nyström & Fritzson

Broman et al. [4] developed a more flexible approach to modular balance checking than the approach described by the current Modelica specification [21] (to which it is a precursor). Most notably, models are not required to be locally balanced provided that the fully assembled system is balanced. The type system, dubbed Structural Constraint Delta (C_Δ), is developed for a subset of Modelica called *Featherweight Modelica*.

The core idea behind C_Δ is to refine the notion of type equality such that two models are equal only if they are equal under the Modelica interpretation (see [21]) and have the same variable-equation balance. This idea is extended to a subtyping relationship where $S <: C$ holds only when S is a Modelica subtype of C, and S and C have the same variable-equation balance. This refinement is motivated by the principle of safe substitution; in this instance, stating that it is only safe to replace one class by another if the replacement preserves the global balance of a system.

The refined notion of type equality is realised by annotating the type of a class with the difference, C_Δ, between the total number of defined equations and variables. The annotation is a concrete value as Featherweight Modelica classes are not first-class entities: the information required to compute the annotation is always manifest in the structure of the object being analysed. Hence, the C_Δ may always be computed in a bottom-up fashion.

By contrast, the type system discussed in Sect. 5 lifts a number of restrictions inherent to C_Δ. Our approach permits first-class models. Hence, we do not rely on manifest type information as the structure of a model may be partially or even completely unknown. Furthermore, parameterised models are parameteric in their balance; a model may be instantiated with different values for its parameters, resulting in distinct balances for each usage of the model within the same context.

As with Modelica, the approach taken by Broman is strictly balance oriented. In contrast, our system captures some structural properties beyond simple balance. For example, signal relations are valid only when they do not over- or under-constrain their local variables.

To our knowledge, the idea of incorporating balance checking into the type system of a non-causal modelling language was suggested independently by Nilsson et al. [25, 22] and Broman et al., with the latter giving the first detailed account of such an approach.

7.3 Bunus & Fritzson

Bunus & Fritzson [5] describe a static analysis technique for pinpointing problems with modular systems of equations developed in equation-based languages such as Modelica. The primary motivation for their work is to develop effective debugging techniques for equation systems. They are concerned with structural properties, as we are, but, allowing systems to be flattened before analysis grants them the capacity to perform a much more fine-grained localisation of problems. In essence, viewing the flattened system as a bipartite graph (the nodes being the equations on the one hand and the

occurring variables on the other), they attempt to put the equations in a one to one correspondence with variables occurring in them by performing a Dulmage-Mendelsohn canonical decomposition. This will partition the system into a *well-constrained* part (a one to one correspondence is possible), an *over-constrained* part (too many equations), and an *under-constrained* part (too many variables). If the latter two parts are empty, the system as a whole is structurally well-constrained.

The main contribution of the work is the localisation and reporting of program errors in a method consistent with the programmers perception of the system. An efficient technique for annotating equations for future analysis is also outlined. The methods discussed are robust, even in the face of program optimisations that may change the intermediate structure of the modular system of equations. Bunus & Fritzson implemented a prototype of their tool, attached to the MathModelica simulation environment, and evaluated the usability of their system in that setting. A case study is presented in their paper.

Because the methods outlined are intended to be used after a modularly constructed system has been flattened, the methods are in many ways complimentary to the type system presented in this article. The methods could even be performed during simulation, making them potentially very useful for analysis of iteratively-staged, structurally-dynamic systems [15]. In any case, the work by Bunus & Fritzson illustrates the benefits from going beyond basic balance checking for finding problems with systems of equations. Some of those benefits are also realised by our system thanks to the classification of equations into different kinds depending on the variables that occur in them; i.e., an approximation of individual variable occurrences.

7.4 Furic

Furic [13] proposes a novel approach for model composition for Modelica with improved guarantees of compositionality. A notion of variable and equation balance is central to this. Like in the approach adopted by Modelica, no classification of equations is made depending on whether occurring variables are local or not. Furic's balance checking algorithm works on a *physical connection graph* describing the structure of an assembled system. Its present formulation is thus not modular. However, Furic suggests that the additional syntactic information that the proposed approach makes available could form a basis for a type system for enhanced static checking and separate compilation. Interestingly, Furic's approach supports a much more flexible notion of structural dynamism than Modelica does at present. However, this hinges on either pre-enumerating all configuration for checking purposes, or running the checking algorithm at each structural change during simulation.

Despite being quite different from our type-based approach, Furic's work underscores the practical importance of enforcing constraints on the variable and equation balance for modularly constructed systems of equations. Moreover, his approach to composition offers a number of advantages over Modelica's, and it would be interesting to see if it can be recast into a type-based approach, and maybe even adapted to the FHM setting.

7.5 Nilsson

The work by Nilsson [22], which is a precursor to this work, outlines an approach to static checking that makes stronger guarantees about the structure of equations and variables beyond that of simple balance. In many cases, Nilsson's *structural types* are

able to rule out systems with structural singularities that would otherwise be accepted under a simple balance checking approach. As with the system developed in this paper (Sect. 5), Nilsson develops his approach for the FHM framework.

The incidence matrix of a system of equations represents the occurrences of variables in equations. By approximating incidence matrices in the types of signal relations and equations, Nilsson approaches the capabilities of Bunus and Fritzson's technique [5], while retaining the capability of checking fragments in isolation. Partitioning equations into classes depending on whether the occurring variables are local or interface or both is central to Nilsson's approach and led to the notion of equation kinds in this paper.

Nilsson's work is a preliminary investigation into structural types. It does not consider first-class models, and it is not clear that it would be possible to generalise the method to a first-class setting while retaining the precision of the types. Structurally dynamic systems are considered, but only briefly. The time complexity of the given algorithm to compute structural types is also a concern as it relies on partitioning the set of mixed equations in all possible ways. Moreover, the the flip-side of the precision of the types is that they may be hard to understand and cumbersome to use in practice. Suitable methods by which to communicate type errors to the programmer would also have to be investigated, although the paper does suggest that the work by Bunus & Fritzson could provide a good starting point. By contrast, the type system presented here does handle first-class models, but is not able to detect as many structural problems. Additionally, this paper also considers structural dynamic systems in depth.

7.6 Capper & Nilsson

The key ideas in this article were first presented by Capper et al. [7], which contained a preliminary investigation into capturing structural properties of equation systems using constrained types. Since the initial investigation, the type system has been improved and extended in a number of ways, as described in this paper.

The core language has seen major improvements: eliminating unnecessary noise from the language has lead to improvements in the presentation of the semantics and type system. Moreover, the semantics now give an accurate account of variables in a modular systems of equations, which has already shown to be useful for work in progress by Capper et al. based upon early work for a denotational model of FHM [8].

An important extension featured in this article is the handling of structurally dynamic systems. In particular, we outline constraints that define a *fair* policy (Sect. 4.3) for reconciling the branches of a switch, allowing structural properties of the branches to be verified. Furthermore, the existing constraint criteria for signal relations have been refined.

8 Future Work

There are a number of avenues of potential future work stemming from the system developed in this article. In this section, these avenues are briefly explored, including discussion about the utility, complexity, and importance of each extension.

The early discussion of structural dynamism (Sect. 2.4) raised the issue of (re)-initialisation. Specifically, automatic initialisation of equation systems is in general a hard

problem, and thus, we chose not to consider these aspects when designing the constraints for the refined type system. Currently, Hydra allows the modeller to express initialisation logic explicitly using (re)-initialisation equations. Whilst this approach remains the most practical solution, it would be desirable to capture structural properties of initialisation equations in the refined type system. For example, such equations might be considered as a new *kind* of equation, for which new structural invariants could be enforced.

A related problem is that of redundant equations, the utility of which underpins work by Nilsson et al. [24] on simulating ideal diodes by exploiting structural dynamism. The crux of the paper relies on introducing redundant equations — equations that do not specify new constraints when added to a system — intentionally creating an *unbalanced* system of equations. Such a system would be rejected by our refined type system (indeed, this is the point of the refined types), where instead it would be preferable to allow the modeller to express such intentions (i.e., that an equation is redundant and is only present to ensure that a particular technique for solving the equations will succeed). One solution might be to allow the modeller to mark equations as *weak* or *dependent*, leaving it up to the type system to decided whether said *weak* equations should be considered when generating constraints.

Another important consideration is the usability of the type system. From the perspective of translating a model into a program, full type inference means the modeller need not be concerned with annotating (or even understanding) the constraints at work in the background. However, it is then unclear how best to communicate type errors resulting from unsatisfiable constraints to the modeller. While simple examples might result in obvious structural invariants being violated, desugaring of higher-level syntactic features may cause equations system to become unrecognisable to the modeller. In such instances, the work by Bunus et al. [5] may prove useful in tracking the surface-level meaning of programs through syntactic transformations, allowing errors to be communicated in a more meaningful way.

In Sect. 4 we regard the constraint criteria as domain agnostic: the criteria are applicable regardless of the chosen domain. It would be useful to consider generating constraints for specific domains, where more information about the structure of equations means that stronger structural invariants can be expressed.

Finally, as mentioned in Sect. 5.5, an important undertaking would be to prove metatheoretical properties that related directly to the refinements of the type system. However, as stated in the aforementioned section, to state interesting properties beyond simple preservation of unrefined types, one would likely need to consider a system with *concrete* signal relation reduction (i.e. reduction that does not abstract away from variable occurances).

We would be particularly interested to show that if given $t : C \Rightarrow SR\,()\,n$ and also $\neg$ *satisfiable* (C), then there exists a structural configuration (i.e., a particular choice of switch branches), such that t elaborates to a structurally singular system of equations. In other words, if a complete modular system of equations is well-typed but ill-formed, then, at least for one possible configuration, it really is a bad system, thus justly ruling it out. Intuitively, this follows directly from the criteria of Sect. 4. Any one of these criteria is unsatisfiable only when it is clear that there isn't going to be a way to pair each equation with a variable even when grossly overapproximating which variables occur in which equation. During elaboration, the exact set of variables occurring in each equation is gradually going to become manifest. But this set is necessarily a subset of the overapproximation of occurrences on which the criteria is defined. Thus, if pairing was not possible *before* elaboration, it is certainly not going to be possible

after elaboration, where the number of choices of which equation to pair with which variable is not going to be greater than before (but likely much smaller).

9 Summary and Conclusions

This article presents a novel and powerful approach to detecting structural problems in modular systems of equations. Components can be analysed in isolation, rather than requiring assembly into a complete system of equations, thus allowing over- and underconstrained systems to be detected early, aiding in error localisation. In particular, we advance the current state-of-the-art by presenting a type system that is capable of handling first-class, structurally dynamic models. Furthermore, the type system is able to detect more structural properties than existing type systems by considering the *kinds* of equations that occurr in a modular system of equations. We also put forth a concise small-step semantics for FHM that considers the non-trivial impact of evaluation on local variables.

Finally, it is worth remarking that the principles of the developed type system are in no way specific to FHM and should be applicable to modular equation systems in general.

References

[1] Accellera Organization: Verilog-AMS language reference manual — analog & mixed-signal extensions to Verilog HDL version 2.3.1 (2009)

[2] Aho, A.V.: The C Programming Language (1988)

[3] Barbeau, E.J.: Pell's Equation, Problem Books in Mathematics. Springer-Verlag (2003)

[4] Broman, D., Nyström, K., Fritzson, P.: Determining over- and under-constrained systems of equations using structural constraint delta. In: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 151–160. ACM, Portland, Oregon, USA (2006)

[5] Bunus, P., Fritzson, P.: A debugging scheme for declarative equation based modeling languages. In: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), *Lecture Notes in Computer Science*, vol. 2257, pp. 280–298. Springer-Verlag, OR, USA (2002)

[6] Capper, J.J.: Source code repository. `www.cs.nott.ac.uk/~jjc`

[7] Capper, J.J., Nilsson, H.: Static balance checking for first-class modular systems of equations. In: Proceedings of the 11th Symposium on Trends in Functional Programming. Oklahoma, USA (2010)

[8] Capper, J.J., Nilsson, H.: Towards a formal semantics for structurally dynamic non-causal modelling languages. In: Types in Language Design and Implementation. Philadelphia, Pennsylvania, USA (2012)

[9] Cellier, F.E.: Object-oriented modelling: Means for dealing with system complexity. In: Proceedings of the 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands, pp. 53–64 (1996)

[10] Cellier, F.E., Kofman, E.: Continuous System Simulation. Springer-Verlag (2006)

[11] Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Proceedings Second GI Conference on Automata Theory and Formal Languages, *Lecture Notes in Computer Science*, vol. 33, pp. 134–183. Springer-verlag (1975)

[12] Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of ICFP'97: International Conference on Functional Programming, pp. 163–173 (1997)

[13] Furic, S.: Enforcing model composability in Modelica. In: F. Casella (ed.) Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009, *Linköping Electronic Conference Proceedings*, vol. 43, pp. 868–879. Linköping University Electronic Press (2009)

[14] Giorgidze, G., Nilsson, H.: Higher-order non-causal modelling and simulation of structurally dynamic systems. In: F. Casella (ed.) Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009, *Linköping Electronic Conference Proceedings*, vol. 43, pp. 208–218. Linköping University Electronic Press (2009)

[15] Giorgidze, G., Nilsson, H.: Mixed-level embedding and JIT compilation for an iteratively staged DSL. In: J. Mariño (ed.) Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010), *Lecture Notes in Computer Science*, vol. 6559, pp. 48–65. Springer-Verlag (2011)

[16] IEEE Std 1076.1-2007: IEEE Standard VHDL Analog and Mixed-Signal Extensions. IEEE Press (2007)

[17] Jones, S.P., et al.: Haskell 98 – A non-strict, purely functional language. `http://www.haskell.org/onlinereport` (1999)

[18] Kirchhoff's circuit laws. Wikipedia. Visited Oct. 2012

[19] McKinna, J., Altenkirch, T., McBride, C.: Why Dependent Types Matter. ACM SIGPLAN Notices **41**(1)

[20] Milner, R.: A theory of type polymorphism in programming. JCSS: Journal of Computer and System Sciences **17** (1978)

[21] Modelica Association: Modelica — A Unified Object-Oriented Language for Systems Modelling; Language Specification Version 3.3 (2012). URL `http://www.modelica.org`

[22] Nilsson, H.: Type-based structural analysis for modular systems of equations. In: P. Fritzson, F. Cellier, D. Broman (eds.) Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, no. 29 in Linköping Electronic Conference Proceedings, pp. 71–81. Linköping University Electronic Press, Paphos, Cyprus (2008)

[23] Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02), pp. 51–64. ACM Press, Pittsburgh, Pennsylvania, USA (2002)

[24] Nilsson, H., Giorgidze, G.: Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes. In: Proceedings of the 7th EUROSIM Congress on Modelling and Simulation. Czech Technical University Publishing House, Prague, Czech Republic (2010)

[25] Nilsson, H., Peterson, J., Hudak, P.: Functional hybrid modeling. In: Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages, *Lecture Notes in Computer Science*, vol. 2562, pp. 376–390. Springer-Verlag, New Orleans, Lousiana, USA (2003)

[26] Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Tech. rep., Chalmers University of Technology (2007)

[27] Nytsch-Geusen, C., Ernst, T., Nordwig, A., Schwarz, P., Schneider, P., Vetter, M., Wittwer, C., Nouidui, T., Holm, A., Leopold, J., Schmidt, G., Mattes, A., Doll, U.: MOSILAB: Development of a Modelica-based generic simulation tool supporting model structural dynamics. In: Proceedings of the 4th International Modelica Conference, pp. 527–535. Hamburg, Germany (2005)

[28] Pierce, B.: Types and Programming Languages. The MIT Press (2002)

[29] Plotkin, G.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark (1981)

[30] Pugh, W.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In: Supercomputing 91 (1991)

[31] Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation, pp. 242–252 (2000)

[32] Zimmer, D.: Equation-based modeling of variable-structure systems. Ph.D. thesis, Swiss Federal Institute of Technology, Zürich (2010)

Inheritance and Overloading in Agda

Paolo Capriotti

3 June 2013

Abstract

One of the challenges of the formalization of mathematics in a proof assistant is defining things in such a way that the syntax resembles the usual informal mathematical notation as much as possible.

I present a collection of techniques that make it possible to obtain a reasonably compact notation in an Agda implementation of basic algebra and category theory.

Although the solution is not completely satisfactory by itself, it shows how the current feature set of a language like Agda could be enhanced in order to solve the problem completely, and that the extensions needed would be minimal.

Introduction

Informal mathematical notation is full of abuses and clever syntactical conventions. We write $\circ$ for the composition in any category, we apply things like functors and natural transformations to their arguments, even though they are not strictly functions, and we use all the notation and results for monoids when we are talking about groups.

When formalizing mathematics in a proof checker like Agda, however, we need to be a little more careful. As in a given scope, there can only be one definition for a given name of symbol, we need to employ some cleverness if we want to replicate the flexibility that the informal notation allows.

The crudest solution is to just use different names for different things. Although this doesn't sound like a bad principle at first, it becomes unweildy pretty quickly. We would need different symbols for operations on natural numbers, integers, rationals. For every new category or group that we define, we would need to come up with names for its operations.

Modules

Fortunately, Agda comes with a remarkably powerful module system. By dividing groups of related definitions into modules, we can then reuse the same names for different definitions. Since every record is also a module, we can, for

example, define a record `Category` with all the structure that defines what a category is:

```
record Category : Set₁ where
  field
    obj : Set
    hom : obj → obj → Set
    id : (x : obj) → hom x x
    _∘_ : {x y z : obj} → hom y z → hom x y → hom x z

    - laws, etc...
```

and then just open it when we need it:

```
open Category C
id-id : (x : obj) → id x ∘ id x ≡ id x
```

Unfortunately, sometimes we need to deal with more than one category at a time. For example, to define the notion of functor, we need at least two. We cannot simply open the module twice, of course, so we are forced to choose: either we qualify everything explicitly, or we use the renaming feature of Agda, and pick new names for every definition that would appear twice.

Explicit qualification is really awkward and makes for completely unreadable code. For example:

```
(f : Category.hom C x y) → Category._∘_ C f (Category.id C x) ≡ f
```

is the type of the left identity law for a category. This is not really a viable solution.

So the only choice we have left is to rename duplicate definitions. Here is how a definition of `Functor` would look like:

```
record Functor (C D : Category) : Set where
  open Category C renaming
    ( obj to objC -
    ; hom to homC )
    - etc...
  open Category D renaming
    ( obj to objD -
    ; hom to homD )
    - etc...
  field
    apply : objC → objD
    map : {x y : objC} → homC x y → homD (apply x) (apply y)
    - etc...
```

This is not too bad, and in fact some existing category theory libraries for Agda, like for example [1] do indeed use this approach.

Unfortunately, this is still very far from a satisfactory overloading mechanism, as it requires enormous amounts of boilerplate code on every usage of overloaded definitions, it's error prone, and still much more noisy than the corresponding informal notation, even ignoring the renaming boilerplate.

Instance arguments

Since version 2.3.0, Agda provides a feature which is specifically designed to make real overloading of names possible: *instance arguments* ([4]).

Instance arguments are similar to implicit arguments: when an argument of a function is marked as "instance", it doesn't need to be given, and is going to be automatically inferred at the function call site.

The strategy for inference of instance arguments, however, differs from the one used for implicit arguments. While the latter is based on unification, the former searches for possible candidates *in scope*, and succeeds if it finds exactly one.

Syntactically, instance arguments are enclosed in double curly braces.

With instance arguments in hand, we can now solve the overloading problem more effectively. If we mark the `Category` argument of each field of our record above as an instance argument, we can open the record only once, and Agda will automatically fill in the correct category from the scope, without us having to qualify it explicitly.

There is even a special syntax for that:

```
open Category {{ ... }} hiding (obj)
open Category using (obj)
```

will simultaneously open the `Category` module and mark the `Category` argument of each of the fields as an instance argument. We refrain from using instance arguments for the `obj` field, as without the `Category` argument, it's likely to be often ambiguous, and `obj C` is an acceptable notation already.

Now we can write, for example:

```
id-id : (x : obj C)(y : obj D) → id x ∘ id x ≡ id x
```

where the object `y` of the category `D` doesn't play any role, but it's included to show that this also works when multiple categories are in scope.

Implementing hierarchies

Using instance arguments directly as in the previous section is a good enough solution for simple cases, but in practice, things are a bit more involved.

In particular, in a formalization of algebra or category theory, concepts are organised into a hierarchy: sets, monoids, groups, abelian groups, rings, etc. are not completely independent entities, but each is, in some sense, a subtype of the previous one.

Agda doesn't have built-in support for subtyping, but we can encode the first three levels, for example, with something like:

```
record IsMonoid (X : Set) : Set where
  field
    unit : X
    _*_ : X → X → X
    - laws...

Monoid : Set₁
Monoid = Σ Set IsMonoid

mon-carrier : Monoid → Set
mon-carrier = proj₁

open IsMonoid {{ ... }} public

record IsGroup (M : Monoid) : Set where
  private
    X = mon-carrier M
    is-mon = proj₂ M   - this must be in scope for
                       - the following to type-check
  field
    inv : X → X
    left-inv : (x : X) → inv x * x ≡ unit

Group : Set₁
Group = Σ Monoid IsGroup

grp-carrier : Group → Set
grp-carrier G = mon-carrier (proj₁ G)

open IsGroup {{ ... }} public
```

We could inline the definition of `IsMonoid` into the `Monoid` record (and similarly `IsGroup` into `Group`), but keeping them separate has many benefits, as I will show later.

Enabler interfaces

Whenever records containing overloaded definitions (which I will refer to as *instance records*) are organised into a hierarchy, they are usually not readily available in a scope, so they need to be explicitly extracted, as in the definition of `IsGroup` above.

The situation is even worse than that, when multiple levels are involved. For example, to use the full structure of a group `G` we need to write something like:

```
is-grp = proj₂ G
is-mon = proj₂ (proj₁ G)
```

and in general, we need a number of statements equal to how many levels deep we need to dig into the hierarchy to find all the definitions that we need.

One solution is to package all these statements into a single module per type, which I refer to as the *enabler* for that type:

```
module mon-enabler (M : Monoid) where
  mon-instance = proj₂ M

module grp-enabler (G : Group) where
  open mon-enabler (proj₁ G) public
  grp-instance = proj₂ G
```

This still requires some boilerplate, but now it is almost entirely on the definition side. The client code can just open the top-level enabler and immediately gain access to the full interface, including definitions in super-types.

Furthermore, the code size of a single enabler is now constant, rather than linear, because we can define new enablers "recursively" over previously defined ones, for example:

```
record IsCommutative (M : Monoid) : Set where
  open mon-enabler M
  field
    comm : (x y : mon-carrier M) → x * y ≡ y * x

CommMonoid : Set₁
CommMonoid = Σ Monoid IsCommutative

AbGroup : Set₁
AbGroup = Σ Group (λ G → IsCommutative (proj₁ G))

module cmon-enabler (M : CommMonoid) where
  open mon-enabler (proj₁ M) public
  cmon-instance = proj₂ M
```

```
module abg-enabler (A : AbGroup) where
  open grp-enabler (proj₁ A) public
  abg-instance = proj₂ A
```

Incidentally, this code also shows why it's beneficial to separate the definition for a new concept X into `IsX` and `X` proper: we can easily reuse the `IsX` records to create parallel hierarchies with minimal code duplication.

Coercions and static methods

Although definitions contained in all the instance records of an inheritance chain can now be accessed very easily just by opening the appropriate enabler module, there is still no way to create *new* definitions in such a way that they can be shared by all super-types.

If we prove a theorem about monoids, like for example:

```
left-right-unit  : {M : Monoid}(x : mon-carrier M)
                 → let open mon-enabler M
                 in unit * x ≡ x * unit
```

we might want to apply it to groups, abelian groups, etc.

So we define all possible *coercions* to `Monoid`:

```
mon-is-mon : Monoid → Monoid
mon-is-mon M = M   - trivial coercion

grp-is-mon : Group → Monoid
grp-is-mon = proj₁

cmon-is-mon : CommMonoid → Monoid
cmon-is-mon = proj₁

abg-is-mon : AbGroup → Monoid
abg-is-mon A = proj₁ (proj₁ A)
```

then, using instance methods again, we can easily define functions that work for any subtype of `Monoid`:

```
module monoid-static
  {Source : Set₁}
  {{ c : Source → Monoid }}
  (source : Source)
  where
    private M = c source
    open mon-enabler M
```

```
    carrier : Set
    carrier = mon-carrier M

    left-right-unit : (x : carrier) → unit * x ≡ x * unit
    - etc...


open monoid-static public
```

I call such definitions *static methods*, because they behave similarly to static methods in object oriented languages. By contrast, we refer to definitions appearing in some instance record as *instance methods*.

We can even turn enabler modules into static methods:

```
module as-monoid
  {Source : Set₁}
  {{ c : Source → Monoid }}
  (source : Source)
  where
    private M = c source
    mon-instance = proj₂ M
```

Now we can enable `Monoid` methods for any superclass of it:

```
module example (A : AbGroup) where
  open as-monoid A
  - we can use _*_ and unit for A here
```

Unfortunately, coercions to all super-types need to be defined manually for each type in the hierarchy. There does not seem to be way to alleviate this problem with instance arguments, as the instance search is limited to the current scope, and cannot combine instances in any way.

Potentially, a code generator or some kind of macro system could be used to generate coercions automatically. The transitive closure of coercions from `Σ X isY` to `X` could be generated this way, and any other desired coercion could be added manually.

Implementation

The `agda-base` library ([2]) contains an implementation of the inheritance and overloading patterns described in this paper.

The library code employs some extra tricks, like wrapping instance records and coercions into specialized data types.

The data type for instance records is called `Styled` and has a phantom parameter `style`, which can be used to implement alternative notations for instance methods. For example, besides the usual monoid enabler for the `default` style, one can define a secondary enabler for monoids, exposing an instance record with an `additive` style parameter.

Therefore, the user can select the notation to use by opening the corresponding enabler, without having to define additional monoid instances, or perform any renaming of definitions.

Unfortunately, the implementation presents some performance issues during type-checking, probably related to the interaction between instance search and unification of universe levels, as some of the instance definitions (like the one for composition) contain a large number of level meta-variables.

Conclusion and future work

I showed how Agda's instance records can be used to implement hierarchies of data types with overloaded methods.

Although I was mainly focused on the formalization of hierarchies of algebraic or categorical structures, the techniques exemplified here should be also applicable to the domains where object oriented design is usually employed.

The solution is not completely satisfactory, as it requires relatively large amounts of boilerplate code, and it makes type-checking quite slow when combined with universe polymorphism.

However, boilerplate code is limited to data type definitions, whereas the client code looks clean and very close to informal mathematical notation. Furthermore, it is relatively easy to see how the boilerplate generation could be automated or integrated in the language.

I have only dealt with Agda, here, but similar considerations should be true for other implementations of type theory.

The Coq proof assistant, for example, provides built-in support for coercions ([3]), and provides a *type class* construct ([5]), which subsumes instance records and enablers. It is likely that those features would make it possible to achieve comparable (or even better) expressiveness in Coq with minimal amounts of boilerplate.

However, Coq's features feel much less minimalistic. In particular, the instance search employs not very well-specified heuristics, and thus is not as predictable as Agda's.

I feel the ideas presented in this paper show there is a sweet spot in the design space of instance search and implicit coercion features that has not yet been implemented, and that it lies not very far from the current capabilities of the Agda system.

Investigating in more detail and possibly implementing this sweet spot is the subject of future work.

References

[1] `https://github.com/copumpkin/categories`.

[2] `https://github.com/pcapriotti/agda-base`.

[3] `http://coq.inria.fr/distrib/current/refman/Reference-Manual021.html#Coercions-full`.

[4] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In *ICFP*, pages 143–155, 2011.

[5] Matthieu Sozeau and Oury Nicolas. First-Class Type Classes.

A Slow Road to Fast Code
sequencing optimisation passes using Monte-Carlo tree search (work in progress)

Glyn Faulkner *
Department of Computer Science, The University of York, York, YO10 5GH, UK
glyn.faulkner@york.ac.uk

Colin Runciman
Department of Computer Science, The University of York, York, YO10 5GH, UK
colin.runciman@york.ac.uk

Abstract

This paper describes an experimental iterative compiler, CARBOLIC, which is an attempt to apply Monte-Carlo tree search (MCTS) to the task of finding effective sequences of optimisation passes.

The preliminary results indicate that CARBOLIC can be used to optimise a benchmark program for speed, but that the gains achieved are very small for the time taken. We discuss the reasons for these results, potential improvements to the compiler, and MCTS's suitability for determining compiler optimisation sequences.

1 Introduction

Code optimisation can be viewed as a search task. Somewhere in a vast and irregular optimisation 'landscape' we hope to find an executable which embodies some desirable combination of characteristics. The most familiar characteristics are high execution speed and small executable size, but there are many things we might wish to optimise for besides those routinely offered by the authors of traditional compilers.

One way of achieving optimisation is the application of a sequence of discrete optimisation 'passes' to the abstract syntax tree or some intermediate representation of the program. Each pass transforms the program in a way that is expected to improve its performance or make the program more amenable to transformation by subsequent passes.

*Glyn Faulkner's research is supported by an EPSRC studentship

The number of possible optimisation passes available to us may be large. They may interact with each other and with the complexities of modern computer architectures in hard-to-predict ways. The effectiveness of an optimisation sequence may vary with the particular program being compiled [1] and the data that program operates on [2]. There is no single best sequence of passes applicable to all software in all circumstances.

1.1 Overview

Section 2 gives a brief overview of the field of iterative compilation. Section 3 explains Monte-Carlo tree search. Section 4 describes the design and operation of the CARBOLIC compiler and some preliminary results obtained using the prototype. Section 5 summarises our preliminary conclusions – mainly various needs for further work.

2 Iterative compilation

The traditional approach to compiler optimisation sequencing is to determine, during development of the compiler, some fixed sequences of optimisation passes which typically result in execution speed gains, or binary size reduction, across a broad range of source programs on the target platform. These sequences are then accessed by the end-user of the compiler using simple command-line flags (e.g. `-O3` to optimise for speed and `-Os` to optimise for code size).

Iterative compilation, in contrast, is an experimental approach which attempts to determine a sequence of optimisation passes tailored to a specific program by executing it. An iterative compiler repeatedly compiles the program and measures its performance with different optimisations, finally selecting the best performing sequence.

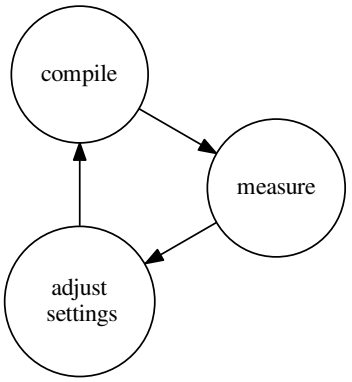


Figure 1: The iterative compilation cycle.

One of the strengths of iterative compilation is its generality. It requires very little up-front knowledge about the target platform and it can be used to optimise

for non-standard performance characteristics. Examples include optimisation for power consumption [3] and optimisation for best performance without exceeding a given maximum code size [4]. If you can measure it, an iterative compiler can try to optimise for it!

In practice, iterative compilation has been used for a variety of purposes. Unsurprisingly, an early application was in the selection of general-purpose optimisation sequences for traditional compilers. Researchers at Intel used this approach to choose a sequence of optimisation passes for an experimental IA-64 compiler[5]. The experimentally determined sequence outperformed an optimisation sequence previously proposed by Intel's human compiler experts across a suite of benchmarks.

Iterative compilation has also found a niche in the embedded systems arena, where the high cost of repeated compilations can be offset against cheaper hardware required to run the highly-tuned software [6].

An important variable in iterative compilation is how the change in compilation rules is determined. Research by Kisuki et al. [7] has shown that near-optimal performance can be achieved using even a trivial random-sampling strategy, if sufficiently many trials are undertaken. But a good search strategy can significantly reduce the number of trials required to approach optimality.

Genetic algorithms have been used by some researchers to good effect [8]–[11].

CARBOLIC takes the novel approach of using Monte-Carlo tree search, because it has shown potential as a general-purpose search strategy with good efficiency on problems involving large and complex search spaces [12].

3 Monte-Carlo tree search

Monte-Carlo tree search (henceforth MCTS) is a search technique originally developed for computer game AI [13]. Its introduction lead to a significant leap in the quality of computer Go players. Since then it has proved to be a valuable technique in a variety of artificial intelligence tasks [12].

Here is how it works in the context of two-player games. For all the possible 'next moves' leading to a new position, we play a series of simulated 'games' consisting of random moves from that position. Each simulation runs until the game is either won or lost, and the initial move that yields the highest number of winning simulations is selected. The rationale is that each move takes you into a portion of the search space which contains the highest number of favourable outcomes.

MCTS has also been successfully applied to solitaire games, including Morpion Solitaire, an NP-hard graph-colouring puzzle where the goal is to maximise the number of moves before a game-ending position is reached [14], and SameGame, where the object is to clear a grid of coloured balls by removing contiguous groups of the same colour [15].

This use in single-player games is significant, as selection of optimisation sequences can be viewed as a solitaire game where the objective is to minimise a "score" (i.e. the execution time).

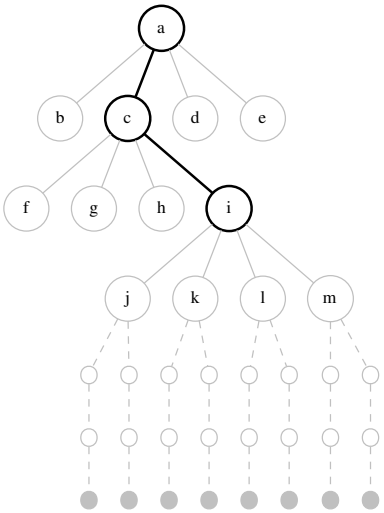


Figure 2: Monte-Carlo tree search. Dotted edges indicate simulations to determine which edge to follow from node i.

For the problem of code optimisation, a "move" is the selection of the next pass in the sequence. Like a peg solitaire game the "winning" condition is that the simulations yield the lowest score after a number of optimisations.

What makes MCTS so powerful is that it combines the coverage of a tree-search with the generality and relatively low-cost of statistical sampling. It requires very little domain knowledge in order to work, but the addition of such knowledge can improve performance by reducing the size of the search space.

This generality and lack of requirement for a detailed model of the "game" is similar to the platform-agnosticity of iterative compilation.

4 Introducing CARBOLIC

CARBOLIC (*Contrived Acronym for a Relatively Basic Optimising LLVM-based Iterative Compiler*) is a prototype iterative optimisation framework written in Ocaml and using LLVM as the source and target of the optimisation passes.

LLVM [16] was selected for the back-end for the following reasons:

1. it has a modular design
2. it features a large library of optimisation passes
3. the optimiser is accessible via the `opt` command-line utility for easy prototyping, and has the *pass-manager* library API for fine-grained control
4. it is in widespread use as a back-end to compilers for multiple languages, including C and Haskell

Like GHC's LLVM back-end, the current version of CARBOLIC uses LLVM's `opt` and `llc` command-line tools when operating on LLVM files. When manipulating C and Haskell it can also call `clang` and `ghc`. It is intended that a future version will interact with LLVM's pass-manager directly, allowing different passes to be applied to different functions in the same source file.

The primary reasons for selection of Ocaml as the source language for CARBOLIC were the mature Ocaml bindings included with the LLVM library, and a bias on behalf of the authors toward declarative programming!

4.1 Operation

CARBOLIC reads a library `L` of available passes from a configuration file, then repeatedly compiles and optimises the source file as follows over `N` iterations, building a sequence `s` of optimisations. `n` is the current length of `s`.

1. for each pass `p` in library `L`:
   (a) run fifty simulations, each resulting in an executable optimised with a sequence of `N` passes. `s ++ [p] ++ r` where `r` is a sequence of `N - (n + 1)` random passes.
   (b) run the resulting executables three times each, taking an average of their *system* + *user* time using standard POSIX system calls, as in the Unix `time` command.
   (c) calculate the average of the fifty simulation times to reach a single score for the pass being tested.
2. append the pass that produced the lowest score to `s`
3. if `n` now equals `N` then exit, otherwise repeat from step 1

4.2 Preliminary Results

The following results were obtained using Hutton's *Countdown* program [17], modified to run non-interactively, as a benchmark. Compilation was done using GHC 7.6.3 and LLVM 3.0. All experiments were run on a quad-core Intel i7 clocked at 3.4GHz, with 8GB of RAM.

To restrict the search space somewhat, CARBOLIC's optimisation library contained only a subset of the hundred or so optimisations available in LLVM. These were the 47 optimisation passes included in the `opt` command's `-O3` optimisation level, as they were expected to be well suited to optimising for speed. The number of iterations was set to ten.

Two experiments were run. In each case the aim is to improve the performance of LLVM code generated by GHC, by applying a sequence of further LLVM optimisation passes.

1. GHC's internal optimisations were disabled. Equivalent to
   `ghc -O0 -fllvm -fforce-recomp $LLVM_PASSES countdown.hs`

2. GHC's `-O1` optimisations were enabled. Equivalent to
   `ghc -O1 -fllvm -fforce-recomp $LLVM_PASSES countdown.hs`

For each of these two experiments CARBOLIC's total run-time was approximately eight hours, during which over 20,000 executables were produced.

Table 1 and Figure 3 show the changes in execution speed for both experiments, from the un-optimised binary to the final sequence of ten optimisation passes. Both show a greatest improvement of approximately 10% – after four passes in experiment 1 and after eight passes in experiment 2.

Pass no.	Experiment 1	Experiment 2
0	0.283	0.141
1	0.283	0.131
2	0.269	0.130
3	0.267	0.130
4	**0.251**	0.129
5	0.267	0.129
6	0.263	0.130
7	0.261	0.129
8	0.259	**0.125**
9	0.272	0.129
10	0.263	0.129

Table 1: Execution time in seconds after selection of each pass. Bold values indicate the best performance.

In both experiments an executable with better performance characteristics than any of those listed above was seen during a simulation. In experiment 1 this occurred during selection of the second pass and ran in 0.248 seconds. In experiment 2 the highest-performing simulation was seen during selection of the sixth pass and ran in 0.120 seconds.

Tables 2 and 3 give details of the derived optimisation sequences for four interesting executables.

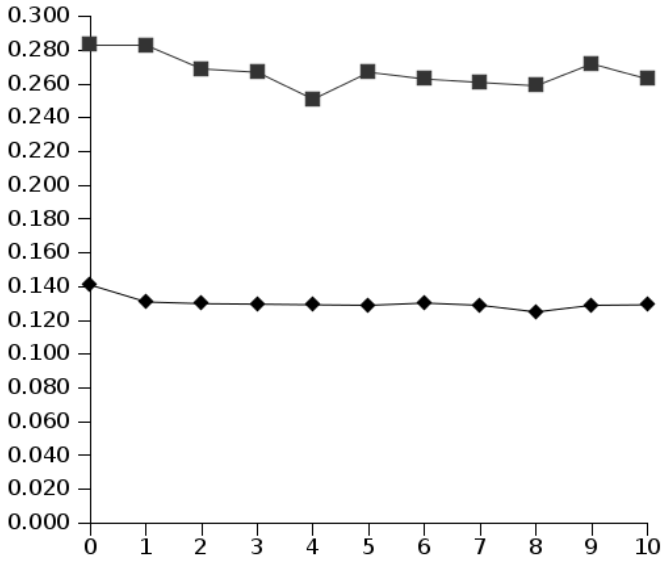


Figure 3: Data from Table 1 plotted to show change in execution speed after selection of each pass. Squares are experiment 1 and diamonds experiment 2.

Executable	Time	Passes
Unoptimised	0.283s	-
10 passes	0.263s	**-scalarrepl-ssa -simplifycfg -ipsccp -inline** -simplify-libcalls -jump-threading -targetlibinfo -scalar-evolution -licm -basiccg
Fastest MCTS	0.251s	**-scalarrepl-ssa -simplifycfg -ipsccp -inline**
Fastest Simulation	0.248s	**-scalarrepl-ssa** -loop-deletion -loop-deletion -loop-simplify -scalarrepl-ssa -dse -tailcallelim -ipsccp -gvn -simplifycfg -dse

Table 2: Some executables produced during experiment 1, with sequences of optimisation passes. Passes used to obtain the fastest result are highlighted in bold.

Metric	Time	Passes
Unoptimised	0.141s	-
10 passes	0.129s	-scalarrepl-ssa -inline -globalopt -prune-eh -lower-expect -jump-threading -scalarrepl-ssa -reassociate** -globalopt -lazy-value-info
Fastest MCTS	0.125s	**-scalarrepl-ssa -inline -globalopt -prune-eh -lower-expect -jump-threading -scalarrepl-ssa -reassociate**
Fastest Simulation	0.120s	**-scalarrepl-ssa -inline -globalopt -prune-eh -lower-expect** -memcpyopt -dse -functionattrs -memdep -no-aa -scalar-evolution

Table 3: Some executables produced during experiment 2, with sequences of optimisation passes. Passes used to obtain the fastest result are highlighted in bold.

5 Discussion

CARBOLIC is a prototype, with a naïve implementation of MCTS. The performance gains are modest. In both of the experiments a small trend toward faster-running binaries was seen. A longer-running benchmark might give clearer results, but with current compile times exceeding eight hours this would rapidly become prohibitive.

The number of optimisation passes was arbitrarily set to ten. The number of simulations to run per node was arbitrarily set to fifty. This gives us a total of just over 20,000 compilations. With 47 optimisation passes available there are 5.26×10^{16} possible optimisation sequences of length ten, including those where the same pass is applied multiple times. Our total coverage was therefore 4×10^{-11} of a percent! In the absence of heuristic knowledge about pass-sequencing, arguably the number of trials was several orders of magnitude too small.

Given that at each stage we know the exact size of the search space, we can use tried and tested methods from statistics to determine the number of simulations we need to run in order to reach a certain confidence interval that we have seen a representative cross-section of the search space.

MCTS is also sensitive to how the scores are calculated for the simulation stage. Choosing the pass for which random continuations give the lowest average may be too simplistic. It has been suggested in at least one study that simple score-averaging across simulations can result in a tendency to make mediocre moves in certain games [18].

6 Future work

There is much work still to be done before it will be possible to use CARBOLIC to determine whether MCTS has applications in the field of iterative compilation.

- we should use intelligent scaling of sample-size based on the size and desired coverage of the search space
- we need to investigate the applicability of more advanced selection criteria used in the MCTS literature
- we can add heuristic knowledge about likely effects of combining passes
- it would benfit us to gather empirical data across multiple runs of CARBOLIC on whether certain optimisation sub-sequences often lead to high-performing executables
- we would like to add support for arbitrary fitness criteria, to allow optimising for non-standard characteristics
- we might wish to integrate CARBOLIC more closely with LLVM pass-manager

References

[1] K.D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *The Journal of Supercomputing*, vol. 23, 2001, pp. 7–22.

[2] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the impact of input data sets on program behavior and its applications," *Journal of Instruction-Level Parallelism*, vol. 5, 2003, pp. 1–33.

[3] S.V. Gheorghita, H. Corporaal, and T. Basten, "Iterative compilation for energy reduction," *Journal of Embedded Computing*, vol. 1, 2005, pp. 509–520.

[4] P. Van Der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis, "Using iterative compilation for managing software pipeline-unrolling trade-offs," *Proc. SCOPES99*, 1999.

[5] K. Chow and Y. Wu, "Feedback-directed selection and characterization of compiler optimizations," *Proc. 2nd Workshop on Feedback Directed Optimization*, ACM, 1999.

[6] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, E. Rohou, and others, "Iterative compilation in a non-linear optimisation space," *Workshop on Profile and Feedback-Directed Compilation*, ACM, 1998.

[7] T. Kisuki, P. Knijnenburg, M. O'Boyle, F. Bodin, and H. Wijshoff, "A feasibility study in iterative compilation," *High Performance Computing*, Springer, 1999, pp. 121–132.

[8] K.D. Cooper, P.J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," *ACM SIGPLAN Notices*, ACM, 1999, pp. 1–9.

[9] M. Stephenson, U.M. O'Reilly, M. Martin, and S. Amarasinghe, "Genetic programming applied to compiler heuristic optimization," *Genetic Programming*, 2003, pp. 231–280.

[10] Y. Che and Z. Wang, "A lightweight iterative compilation approach for optimization parameter selection," *First International Multi-Symposiums on Computer and Computational Sciences, 2006. IMSCCS'06.*, IEEE, 2006, pp. 318–325.

[11] A.P. Nisbet, "GAPS: Iterative feedback directed parallelisation using genetic algorithms," *Workshop on Profile and Feedback-Directed Compilation,(Paris, France)*, 1998 organization.

[12] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, 2012, pp. 1–43.

[13] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo tree search: A new framework for game ai," *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008, pp. 216–217.

[14] T. Cazenave, "Reflexive Monte-Carlo search," *Proc. Comput. Games Workshop*, 2007.

[15] M.P. Schadd, M.H. Winands, H.J. Van Den Herik, G.M.-B. Chaslot, and J.W. Uiterwijk, "Single-player Monte-Carlo tree search," *Computers and Games*, Springer, 2008, pp. 1–12.

[16] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[17] G. Hutton, "Functional Pearl: the countdown problem," *Journal of Functional Programming*, vol. 12, 2002, pp. 609–616.

[18] Y. Björnsson and H. Finnsson, "Cadiaplayer: A simulation-based general game player," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, 2009, pp. 4–15.

Towards a framework for the implementation and verification of translations between argumentation models

(Extended away day version)

Bas van Gijzel

June 9, 2013

Abstract

In the last two decades the general interest in abstract argumentation as well as structured argumentation has surged. There has been a plethora of new argumentation models, from general frameworks to more domain specific ones. It has been shown that many of these models can be translated to Dung's abstract argumentation frameworks. Considering the amount effort put into the optimisation of Dung's AF's, one would expect dozens of these translations to be implemented and running to make use of these efficient algorithms. However, this is not the case at present. By providing a tutorial implementation of Dung's frameworks in Haskell, and formalising this implementation in a theorem prover, we aim to provide a solid base for the implementation and verification of other argumentation models, and very importantly formal translations between models.

1 Introduction

In the last two decades the general interest in abstract argumentation as well as structured argumentation has surged. There has been a plethora of new argumentation models, from general frameworks [16, 2, 5] to more domain specific ones [13, 12]. It has been shown that many of these models can be translated to Dung's framework [16, 11, 10, 3, 15]. Considering the amount effort put into the optimisation of Dung's AF's [6], one would expect dozens of these translations to be implemented and running to make use of these efficient algorithms. However, this is not the case at present.

There are a number of possible reasons:

- Most implementations of argumentation models are not publicly available (or the website is no longer available) and thus coupling of implementations is not that easily done.

- Translations can be notoriously complex, both in implementation and in verification. As a good example, consider the translation of Carneades to ASPIC$^+$ [11, 10], or the translation of abstract dialectical frameworks [3] to Dung. Both proofs are at least a page long, and are hard to verify even by experts in the field.

This paper makes an initial push to solving this problem. We provide and discuss all the Haskell programming code of an implementation of Dung's argumentation frameworks, containing a decent amount of the standard definitions, making this paper both documentation and implementation. Similar to our previous work [9] we will provide our implementation as a public library[1]. Our choice of programming language, Haskell, is motivated by this previous attempt, where we managed to implement an argumentation model intuively enough to be easily readable by an argumentation theorist with no previous knowledge of Haskell. After discussing the Haskell implementation, we discuss and provide (in the Appendix) a formalisation of this implementation in a theorem prover. Although we just give a sketch of how the actual translation of Carneades to Dung and its derived properties/proofs can be done we already provide the following contributions:

- an intuitive implementation of Dung's AFs in Haskell, staying faithful to the actual mathemical definitions;
- to our knowledge, the first formalisation of an argumentation model in a theorem prover;
- a general methodology for implementing translations between frameworks and proving properties between them;
- an effort towards the first formalisation of a translation between argumentation models.

The paper is structured as follows. In Section 2 we give an introduction to Dung's abstract argumentation frameworks, each time providing implementations of the definitions in functional programming language, Haskell. In Section 3 we discuss our formalisation of this implementation in a theorem prover, Agda, and what we can gain from this. In Section 4 we provide an introduction to Carneades, again give corresponding Haskell definitions and end with a sketch on how to encode useful properties between Carneades and Dung using insights from Section 3. We conclude in Section 5 with a discussion of what we have learnt from this initial study and how we can take this further.

[1] See `http://hackage.haskell.org/package/Dung` (to be uploaded soon).

2 An implementation of AFs in Haskell

The abstract argument system, or argumentation framework (AF) as introduced by Dung [8] is a very simple, but general model that is able to capture various contemporary approaches to non-monotonic reasoning. It has also been the translation target for many modern structured argumentation models [16, 11, 10, 3, 15] that have been introduced later in the literature. In this section we will give a significant part of the standard definitions of Dung's AFs, including an algorithm for the grounded semantics and show how these definitions can be almost immediately translated into (a slightly stylised version) of the functional programming language, Haskell[2]. The purpose of this section is to show how a functional programming language such as Haskell can be used to quickly implement a prototype of an argumentation model and to set up the possibility for proving properties of this implementation. This section can also serve as a tutorial like introduction to the implementation of AFs up to grounded semantics.

An abstract argumentation framework consists of abstract arguments and a binary relation representing attack.

Definition 2.1. Abstract argumentation framework An *abstract argumentation framework* is a tuple $\langle Args, Def \rangle$, such that $Args$ is a set of arguments and $Def \subseteq Args \times Args$ is a defeat relation on the arguments in $Args$.

It is important to note that the type of argument is left abstract in the definition above, allowing for possible instantiations of argument by other frameworks such as ASPIC$^+$ [16]. Several of the following definitions however, will need to have some notion of equality on arguments to be able to make the needed comparisons. For now, we use *String*s to just label arguments.

```
data DungAF arg = AF [arg] [(arg, arg)]
  deriving (Show)
type AbsArg = String
```

Note that we use lists instead of sets to allow for an easier presentation.

Example 2.2. An example (abstract) argumentation framework containing three arguments where the argument C *reinstates* the argument A by defeating its defeating argument B is captured by $AF_1 = \langle \{a, b, c\}, \{(a, b), (b, c)\} \rangle$.

$$A \longrightarrow B \longrightarrow C$$

And in Haskell:

[2] The source code of this Section, see `http://www.cs.nott.ac.uk/~bmv/Code/dunginhaskell_away_day.lhs`, is written in literate Haskell and can immediately be run by a standard Haskell compiler.

$a, b, c :: AbsArg$
$a =$ `"A"`
$b =$ `"B"`
$c =$ `"C"`
$AF_1 :: DungAF\ AbsArg$
$AF_1 = AF\ [a, b, c]\ [(a, b), (b, c)]$

We now quickly give a few standard definitions for AFs such as the acceptability of arguments and admissibility of sets. We will use an arbitrary but fixed argumentation framework $AF = \langle Args, defeats \rangle$.

Definition 2.3 (Conflict-free)**.** A set $S \subseteq Args$ of arguments attacks an argument $A \in Args$ iff there exists a $B \in S$ such that $(B, A) \in Def$.

Definition 2.4 (Conflict-free)**.** A set $S \subseteq Args$ of arguments is called *conflict-free* iff there is no A, B in S such that $(A, B) \in Def$.

Definition 2.5 (Acceptability)**.** An argument $A \in Args$ is acceptable with respect to a set S of arguments, or alternatively S *defends* A, iff for all arguments $B \in S$: if $(B, A) \in Def$ then there is a $C \in S$ for which $(C, B) \in Def$.

The semantics Dung defined for an argumentation framework can be defined by using the *characteristic function* of an AF.

Definition 2.6 (Characteristic function)**.** The *characteristic function* of AF, $F_{AF} : 2^{Args} \to 2^{Args}$, is a function, such that, given a conflict-free set of arguments S, $F_{AF}(S) = \{A \mid A \text{ is acceptable w.r.t. to } S\}$.

A conflict-free set of arguments is said to be *admissible* if it is a defendable position.

Definition 2.7 (Admissibility)**.** A conflict-free set of arguments S is admissible iff every argument A in S is acceptable with respect to S, i.e. $S \subseteq F_{AF}(S)$.

Dung defined the semantics of the argumentation frameworks by using the concept of *extensions*. An extension is always a subset of $Args$, and can intuitively be seen as a set of arguments that are acceptable when taken together. We will just discuss the grounded extension, but for completeness we give the four standard semantics defined by Dung.

Definition 2.8 (Extensions)**.** Given a conflict-free set of arguments S, argumentation framework AF and if the domain of F is ordered with respect to set inclusion then:

- S is a *complete extension* iff $S = F_{AF}(S)$.
- S is a *grounded extension* iff it is the least fixed point of F_{AF}.
- S is a *preferred extension* iff it is a maximal fixed point of F_{AF}.
- S is a *stable extension* iff it is a preferred extension defeating all arguments in $Args \backslash S$.

Alternatively, the grounded and preferred extensions can respectively be characterised as the smallest and a maximal complete extension.

In our Haskell implementation of these definitions we resort to a few standard library functions: *null* is a function takes a list as an argument and returns a *True* if it is empty and *False* otherwise, *and* is the equivalent to $\bigwedge$ on lists, while *or* is the equivalent to $\bigvee$ on lists.

```
setAttacks :: Eq arg ⇒ DungAF arg → [arg] →
              arg → Bool
setAttacks (AF _ def) args arg
   = or [b ≡ arg | (a, b) ← def, a ∈ args]

conflictFree :: Eq arg ⇒ DungAF arg → [arg] → Bool
conflictFree (AF _ def) args
   = null [(a, b) | (a, b) ← def, a ∈ args, b ∈ args]

acceptable :: Eq arg ⇒ DungAF arg → arg →
              [arg] → Bool
acceptable af@(AF _ def) a args
   = and [setAttacks af args b | (b, a') ← def, a ≡ a']

f :: Eq arg ⇒ DungAF arg → [arg] → [arg]
f af@(AF args' _) args
   = [a | a ← args', acceptable af a args]

f_AF1 :: [AbsArg] → [AbsArg]
f_AF1 = f AF_1

admissible :: Eq arg ⇒ DungAF arg → [arg] → Bool
admissible af args = args ⊆ f af args

groundedF :: Eq arg ⇒ ([arg] → [arg]) → [arg]
groundedF f = groundedF' f []
  where groundedF' f args
              | f args ≡ args = args
              | otherwise     = groundedF' f (f args)
```

Then as expected:

```
groundedF f_AF1
> ["A", "C"]
```

Note that by the required *Eq arg* ⇒, Haskell forces us to see that we need an equality on arguments to be able implement these functions.

Given an argumentation framework, we can determine which arguments are justified by applying an argumentation semantics. However in contrast to the succise extension based approach, we will take the *labelling-based* approach to grounded semantics. The labelling based approach is more commonly used in actual implementation and to correctly formalise Dung's semantics we would not need a formalisation of fixpoints if we take the labelling based approach. In the below algorithm, for correctness, the "if then" in the ∃ has been changed

to "and". Although the fix here is obvious, it does make a case in point for formalisation of more complex mathematics such as a translation between argumentation models.

Algorithm 2.9. Algorithm for grounded labelling (Algorithm 6.1 of [14])

1. $\mathcal{L}_0 = (\emptyset, \emptyset, \emptyset)$
2. **repeat**
3. $\text{in}(\mathcal{L}_{i+1}) = \text{in}(\mathcal{L}_i) \cup \{x \mid x \text{ is not labelled in } \mathcal{L}_i,$
   $\forall y : \text{if } y \mathcal{R} x \text{ then } y \in \text{out}(\mathcal{L}_i)\}$
4. $\text{out}(\mathcal{L}_{i+1}) = \text{out}(\mathcal{L}_i) \cup \{x \mid x \text{ is not labelled in } \mathcal{L}_i,$
   $\exists y : y \mathcal{R} x \text{ and } y \in \text{in}(\mathcal{L}_{i+1})\}$
5. **until** $\mathcal{L}_{i+1} = \mathcal{L}_i$
6. $\mathcal{L}_\mathcal{G} = (\text{in}(\mathcal{L}_i), \text{out}(\mathcal{L}_i), \mathcal{A} - (\text{in}(\mathcal{L}_i) \cup \text{out}(\mathcal{L}_i)))$

The Haskell equivalent to a labelling:

```
data Status = In | Out | Undecided
  deriving (Eq, Show)
```

For our Haskell implementation, we will first translate the two conditions for x containing quantifiers in line 3 and 4.

```
  -- if all attackers are Out
unattacked :: Eq arg ⇒ [arg] →
              DungAF arg → arg → Bool
unattacked outs (AF _ def) arg =
  let attackers = [a | (a, b) ← def, arg ≡ b]
  in null (attackers \\ outs)


  -- if there exists an attacker that is In
attacked :: Eq arg ⇒ [arg] →
            DungAF arg → arg → Bool
attacked ins (AF _ def) arg =
  let attackers = [a | (a, b) ← def, arg ≡ b]
  in ¬ (null (attackers `intersect` ins))
```

We split the implementation in two parts. A function for the grounded labelling which can immediately be applied to an AF, and a function actually implementing the algorithm, which has an additional two arguments that accumulate the *In*s and *Out*s.

```
grounded :: Eq arg ⇒ DungAF arg → [(arg, Status)]
grounded af@(AF args _) = grounded' [] [] args af


grounded' :: Eq a ⇒ [a] → [a] →
             [a] → DungAF a  → [(a, Status)]
```

```
grounded' ins outs [] _
=       map (λx → (x, In)) ins
  ++    map (λx → (x, Out)) outs
grounded' ins outs args af =
  let newIns  = filter (unattacked outs af) args
      newOuts = filter (attacked ins af)    args
  in if null (newIns ++ newOuts)
     then map (λx → (x, In)) ins
          ++ map (λx → (x, Out)) outs
          ++ map (λx → (x, Undecided)) args
     else grounded' (ins ++ newIns)
                    (outs ++ newOuts)
                    (args \\ (newIns ++ newOuts))
                    af
```

Then as expected:

```
grounded AF1
> [("A", In), ("C", In), ("B", Out)]
```

Finally, the grounded extension can be defined by returning only those arguments that are In from the grounded labelling.

```
groundedExt :: Eq arg ⇒ DungAF arg → [arg]
groundedExt af = [arg | (arg, In) ← grounded af]
```

3 Formalising the implementation in a theorem prover

Now that we have been able to construct an implementation of Dung's AFs, we can formalise this implementation into a theorem prover of our choosing. Agda is also functional in nature and very close to Haskell making it an obvious choice. Agda is a programming language and a theorem prover at the same time. Types with accompanying implementations (functions), correspond to theorems with accompanying proofs through the Curry-Howard correspondence, or the proofs-as-programs interpretation. This means that if we write an implementation/proof of grounded semantics in Agda we already gain a few nice results for free. All functions that are implemented are guaranteed to be terminating, which means that because we successfully implemented the grounded semantics, we immediately know that our algorithm is terminating on all (finite) inputs and because Agda will always give back a labelling, we also have proven that the grounded extension always exists! The correctness of these proofs are automatically checked by the Agda type checker and thus the correctness of the proofs only depends on the core implementation of Agda.

The complete Agda code will not be the topic of the away day talk and will therefore not be included in the body of the paper. However, for completeness I have included the relevant proofs in the Appendix[3].

4 Integrating other frameworks

The previous section talked about the formalisation of some basic definitions of AFs in Agda. However, given this initial effort, there is a lot space for interesting future work. In this section I will give definitions and again corresponding implementations in Haskell of the Carneades argumentation model [13, 12], an argumentation model designed to capture standards and burdens of proof. This is based on previous work in [9]. After that, I will sketch which useful properties of a possible translation between Carneades and Dung's AFs we might want to prove, given that we again have a corresponding implementation (of Carneades) in the same theorem prover.

We will start with the definition of argument in Carneades.

Definition 4.1. Carneades' Arguments Let $\mathcal{L}$ be a propositional language. An *argument* is a tuple $\langle P, E, c\rangle$ where $P \subset \mathcal{L}$ are its *premises*, $E \subset \mathcal{L}$ with $P \cap E = \emptyset$ are its *exceptions* and $c \in \mathcal{L}$ is its *conclusion*. For simplicity, all members of $\mathcal{L}$ must be literals, i.e. either an atomic proposition or a negated atomic proposition. An argument is said to be *pro* its conclusion c (which may be a negative atomic proposition) and *con* the negation of c.

In Carneades all logical formulae are literals in propositional logic; i.e., all propositions are either positive or negative atoms. Taking atoms to be strings suffice in the following, and propositional literals can then be formed by pairing this atom with a Boolean to denote whether it is negated or not:

$$\textbf{type}\ \mathit{PropLiteral} = (\mathit{Bool}, \mathit{String})$$

The negation for a literal p, written $\overline{p}$ is then given as follows:

$$\begin{aligned}&\mathit{negate} :: \mathit{PropLiteral} \to \mathit{PropLiteral}\\ &\mathit{negate}\ (b, x) = (\neg\ b, x)\end{aligned}$$

We chose to realise an *argument* as a datatype (to allow a manual equality instance) containing a tuple of two lists of propositions, its *premises* and its *exceptions*, and a proposition that denotes the *conclusion*:

$$\begin{aligned}\textbf{data}\ \mathit{Argument} = \mathit{Arg}\ ([\mathit{PropLiteral}],\\ [\mathit{PropLiteral}], \mathit{PropLiteral})\end{aligned}$$

Arguments are considered equal if their premises, exceptions and conclusion are equal; thus arguments are identified by their logical content. The equality instance for *Argument* (omitted for brevity) takes this into account by comparing the lists as sets.

[3] For the complete Agda code see: `http://www.cs.nott.ac.uk/~bmv/Code/AF2.agda`

A set of arguments determines how propositions depend on each other. Carneades requires that there are no cycles among these dependencies. Following Brewka and Gordon [4], we use a dependency graph to determine acyclicity of a set of arguments.

Definition 4.2. Acyclic set of arguments A set of *arguments* is *acyclic* iff its corresponding dependency graph is acyclic. The corresponding dependency graph has a node for every literal appearing in the set of arguments. A node p has a link to node q whenever p depends on q in the sense that there is an argument pro or con p that has q or $\overline{q}$ in its set of premises or exceptions.

Our implementation of this acyclic set will be considered abstract for simplicity, instead just providing functions to retrieve all arguments and all used propositions in the *ArgSet*.

$$\begin{aligned}&\textbf{type } \mathit{ArgSet} = \ldots\\ &\mathit{getAllArgs} :: \mathit{ArgSet} \to [\mathit{Argument}]\\ &\mathit{getProps} \quad :: \mathit{ArgSet} \to [\mathit{PropLiteral}]\end{aligned}$$

The main structure of the argumentation model is called a Carneades Argument Evaluation Structure (CAES):

Definition 4.3. Carneades Argument Evaluation Structure (CAES) A *Carneades Argument Evaluation Structure* (CAES) is a triple $\langle arguments, audience, standard\rangle$, where *arguments* is an acyclic set of arguments, *audience* is an audience (see below), and *standard* is a total function mapping each proposition to to its specific proof standard.

The transliteration into Haskell is almost immediate:

$$\begin{aligned}&\textbf{newtype } \mathit{CAES} = \mathit{CAES}\ (\mathit{ArgSet}, \mathit{Audience},\\ &\qquad\qquad\qquad\qquad \mathit{PropStandard})\end{aligned}$$

We will skip most of the definitions of audience and proof standards. The only directly relevant part for the evaluation are the *assumptions* (of an audience) which is simply a consistent set of literals assumed to be acceptable, similar to axioms. In Haskell this is $[\mathit{PropLiteral}]$.

Two concepts central to the evaluation of a CAES are *applicability of arguments*, which arguments should be taken into account, and *acceptability of propositions*, which conclusions can be reached under the relevant proof standards, given the beliefs of a specific audience.

Definition 4.4. Applicability of arguments Given a set of arguments and a set of assumptions (in an audience) in a CAES C, then an argument $a = \langle P, E, c\rangle$ is *applicable* iff

- $p \in P$ implies p is an assumption or [$\overline{p}$ is not an assumption and p is acceptable in C] and

- $e \in E$ implies e is not an assumption and [$\overline{e}$ is an assumption or e is not acceptable in C].

Definition 4.5. Acceptability of propositions Given a CAES C, a proposition p is *acceptable* in C iff $(s\ p\ C)$ is *true*, where s is the proof standard for p.

Note that these two definitions in general are mutually dependent because acceptability depends on proof standards, and most sensible proof standards depend on the applicability of arguments. This is the reason that Carneades restricts the set of arguments to be acyclic.

```
applicable :: Argument → CAES → Bool
applicable (Arg (prems, excns, _))
           caes@(CAES (_, (assumptions, _), _))
  = and ([(p ∈ assumptions) ∨
          (p `acceptable` caes) | p ← prems]
           ++
          [(e ∈ assumptions) ↓
           (e `acceptable` caes) | e ← excns ])
    where
      x ↓ y = ¬ (x ∨ y)
acceptable :: PropLiteral → CAES → Bool
acceptable c caes@(CAES (_, _, standard))
  = c `s` caes
  where s = standard c
```

One of the shortcomings in the discussed version of Carneades is that it is not able to handle cycles. The translation given in [11, 10] attempts to alleviate this problem, by providing a translation to ASPIC$^+$ which is known to generate AFs [16] and then providing semantics to cycle-containing structures by delegating those to Dung's semantics.

Then given that we already have some Haskell data type for ASPIC$^+$ arguments:

```
data ASPICPlusArgument = ...
```

We can define a Dung AF instantiated with concrete ASPIC$^+$ arguments.

```
type ASPICAF = DungAF ASPICPlusArgument
```

Given this, we have the necessary ingredients to sketch how a translation function and the properties we would like to prove look like. As a first step we assume we have already defined a translation function, and also have access to two functions that are able to retrieve the corresponding Dung argument to a Carneades *Argument* and *PropLiteral*. The type signatures are given below:

```
translate :: CAES → DungAF ASPICPlusArgument
arg2dung :: DungAF ASPICPlusArgument →
```

$$Argument \rightarrow ASPICPlusArgument$$
$$prop2dung :: DungAF\ ASPICPlusArgument \rightarrow PropLiteral \rightarrow ASPICPlusArgument$$

Given a translation function, we can talk about the properties we would need to be able to convince ourselves that the translation is actually correct. To do so, we would want to prove properties that are commonly expected of a translation functions in argumentation theory, namely that arguments and propositions that were acceptable/unacceptable in the original model, after translation to the other model, are identifiable and will still be acceptable/unacceptable. These conditions are commonly called correspondence properties.

For the translation function here, we can refer to existing definitions of the correspondence of applicability of arguments and acceptability of propositions (Theorem 4.10 of [11]).

Theorem 4.1. *Let C be a CAES, $\langle arguments, audience, standard\rangle$, $\mathcal{L}_{CAES}$ the propositional language used and let the argumentation framework corresponding to C be AF. Then the following holds:*

1. *An argument $a \in arguments$ is applicable in C iff there is an argument contained in the complete extension of AF with the corresponding conclusion arg_a.*
2. *A propositional literal $c \in \mathcal{L}_{CAES}$ is acceptable in C or $c \in assumptions$ iff there is an argument contained in the complete extension of AF with the corresponding conclusion c.*

Informally, the properties state that every argument and proposition in a CAES, after translation, will have a corresponding argument and keep the same acceptability status. I will now sketch the implementation of these properties in Haskell. If the translation function is a correct implementation, the Haskell implementation of the correspondence properties should always return *True*, however to constitute an actual (mechanised) proof we would need to convert the translation and the implementation of the correspondence properties in Haskell to a theorem prover like Agda.

```
corApp :: CAES → Bool
corApp caes@(CAES (argset, (assumptions, _), _)) =
  let translatedCAES = translate caes
      applicableArgs  = filter (`applicable` caes)
                               (getAllArgs argset)
      corDungArgs     = map (arg2dung translatedCAES)
                            applicableArgs
  in corDungArgs ≡ groundedExt translatedCAES
corAcc :: CAES → Bool
corAcc caes@(CAES (argset, (assumptions, _), _)) =
  let translatedCAES = translate caes
```

$$
\begin{aligned}
&\mathit{acceptableProps} &&= \mathit{filter}\ (\text{`}\mathit{acceptable}\text{`}\ \mathit{caes}) \\
&&&\qquad (\mathit{getProps}\ \mathit{argset}) \\
&\mathit{corDungArgs} &&= \mathit{map}\ (\mathit{prop2dung}\ \mathit{translatedCAES}) \\
&&&\qquad \mathit{acceptableProps} \\
&\textbf{in}\ \mathit{corDungArgs} \equiv \mathit{groundedExt}\ \mathit{translatedCAES}
\end{aligned}
$$

5 Conclusions

In this paper we have discussed a significant part of Dung's argumentation frameworks and shown that Haskell can provide a short and intuitive implementation, while keeping true to the original mathematical definitions. We have discussed the merits of formalising such an implementation in Agda, showing that is feasible to formalise an implementation of an argumentation model into a theorem prover. Finally, we gave a description of a translation function and its required properties, showing how we can make proper use of the formalisations. Such a formalisation would give us the means to translate between argumentation models in a verified manner.

The initial results are encouraging, despite that we haven't formalised an actual translation yet. The relatively successful formalisation of Dung's argumentation frameworks suggest that the implementation and formalisation of a translation is not far off. It is important to note that our approach is not necessarily meant to give the final implementation of a model. The intended use of this approach is for quick prototyping/testing of argumentation models, followed by an implementation and verification of a translation between models, delegating the actual evaluation of arguments to an optimised implementation.

Instead of translating between argumentation models, we can also choose to translate to a specific format, such as a file format or a general format such as the Argument Interchange Format [7, 17]. Especially the recent work on giving a logical specification to the AIF [1] would be a good application for a theorem prover.

References

[1] F. Bex, S. Modgil, H. Prakken, and C. Reed. On logical specifications of the Argument Interchange Format. *Journal of Logic and Computation*, 2012.

[2] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentation-theoretic framework for default reasoning. *Artificial Intelligence*, 93:63–101, 1997.

[3] G. Brewka, P. E. Dunne, and S. Woltran. Relating the semantics of abstract dialectical frameworks and standard AFs. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 780–785, 2011.

[4] G. Brewka and T. F. Gordon. Carneades and abstract dialectical frameworks: A reconstruction. In M. Giacomin and G. R. Simari, editors, *Computational Models of Argument. Proceedings of COMMA 2010*, pages 3–12, Amsterdam etc, 2010. IOS Press 2010.

[5] G. Brewka and S. Woltran. Abstract dialectical frameworks. In *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 102–111. AAAI Press, 2010.

[6] G. Charwat, W. Dvorák, S. A. Gaggl, J. P. Wallner, and S. Woltran. Implementing abstract argumentation - a survey. Technical Report DBAI-TR-2013-82, 2013.

[7] C. Chesñevar, J. McGinnis, S. Modgil, I. Rahwan, C. Reed, G. Simari, M. South, G. Vreeswijk, and S. Willmott. Towards an argument interchange format. *The Knowledge Engineering Review*, 21(4):293–316, 2006.

[8] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.

[9] B. van Gijzel and H. Nilsson. Haskell gets argumentative (in production). In *Proceedings of the Symposium on Trends in Functional Programming (TFP 2012)*, pages ??–??, St Andrews, UK, 2012. LNCS.

[10] B. van Gijzel and H. Prakken. Relating Carneades with abstract argumentation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 1113–1119, 2011.

[11] B. van Gijzel and H. Prakken. Relating Carneades with abstract argumentation via the ASPIC$^+$ framework for structured argumentation. *Argument & Computation*, 3(1):21–47, 2012.

[12] T. F. Gordon, H. Prakken, and D. Walton. The Carneades model of argument and burden of proof. *Artificial Intelligence*, 171(10-15):875–896, 2007.

[13] T. F. Gordon and D. Walton. Proof burdens and standards. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 239–258. Springer US, 2009.

[14] S. Modgil and M. Caminada. Proof theories and algorithms for abstract argumentation frameworks. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer US, 2009.

[15] S. Modgil and H. Prakken. A general account of argumentation with preferences. *Artificial Intelligence*, 2012.

[16] H. Prakken. An abstract framework for argumentation with structured arguments. *Argument & Computation*, 1:93–124, 2010.

[17] I. Rahwan and C. Reed. The argument interchange format. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 383–402. Springer US, 2009.

A Complete Agda implementation

We give our Agda code in the following pages, supplied with documentation. The Agda code is structured similarly to the Haskell code, but with a few main differences. Agda does not (yet) have syntax for list comprehensions, which means list comprehension need to be desugared. We need a few simple lemmas to prove some obvious mathematical facts, but these can all be handled by an automatic solver in the Agda library. To simplify the proof, instead of filtering out all *attacked* and *unattacked* at once, we filter out one or none at the time, corresponding to the *foundV* and *notFoundV* constructors of the *Find* datatype. Finally, the implementation uses *Vector*s (lists with a fixed size), so that the size of the list is always known at compile time. This is useful because to prove that the *grounded′* function terminates, we need to prove that the length of the arguments that still need to be labelled is structurally decreasing.

```
data DungAF (A : Set) : Set where
  AF : List A → List (A × A) → DungAF A

AbsArg = String

a : AbsArg
a = "A"

b : AbsArg
b = "B"

c : AbsArg
c = "C"

  -- an AF such that: A → B → C
AF₁ : DungAF AbsArg
AF₁ = AF (a :: b :: c :: []) ((a , b) :: (b , c) :: [])

  -- an AF such that: A ↔ B
exampleAF2 : DungAF AbsArg
exampleAF2 = AF (a :: b :: []) ((a , b) :: (b , a) :: [])

  -- The Agda equivalent to a labelling:
data Status : Set where
In  : Status
Out : Status
Undecided : Status


  -- For the Agda equivalent of unattacked and attacked we need the Agda equivalents
  -- of the Haskell intersect and (\\) which are intersectBy and deleteFirstBy,
  -- where instead of writing Eq a ⇒ we need to explicitly supply an equality function.
  -- proj₁ and proj₂ respectively take the first and second element of a pair.

  -- Given an equality function, a list of arguments (from a certain AF) that are already considered,
  -- compute whether an argument (arg) is attacked.
unattacked : { A : Set } → (A → A → Bool) → List A → DungAF A → A → Bool
unattacked _≡_ outs (AF _ def) arg =
```

```
  null
    (deleteFirstsBy _≡_
      (List.map proj₁ (filter ((λ x → x ≡ arg) ∘ proj₂) def)) outs)
  -- Given an equality function, a list of arguments (from a certain AF) that are already considered In,
  -- compute whether an argument (arg) is attacked.
attacked : {A : Set} → (A → A → Bool) → List A → DungAF A → A → Bool
attacked _≡_ ins (AF _ def) arg =
  ¬ (null
    (intersectBy _≡_
      (List.map proj₁ (filter ((λ x → x ≡ arg) ∘ proj₂) def)) ins))
  -- False as the empty (or impossible) Type
data False : Set where
  -- True as the Unit Type (type with only one value)
  -- (using a record allows Agda to automatically infer the only allowed value)
record True : Set where
  -- going from a decidable predicate to a Type
isTrue : Bool → Set
isTrue true  = True
isTrue false = False


  -- going from a decidable predicate to a Type
isFalse : Bool → Set
isFalse true  = False
isFalse false = True
trueIsTrue : {x : Bool} → x ≡ true → isTrue x
trueIsTrue refl = _
falseIsFalse : {x : Bool} → x ≡ false → isFalse x
falseIsFalse refl = _
  -- from decidable predicate to predicate on types
satisfies : {A : Set} → (A → Bool) → A → Set
satisfies p x = isTrue (p x)


  -- A simple lemma to transform to transform isFalse to isTrue
lemma : {x : Bool} → isFalse x → isTrue (¬ x)
lemma {true} ()
lemma {false} prf = prf


infixr 30 _:allV:_
data AllV {A : Set} (P : A → Set) : {n : ℕ} → Vec A n → Set where
allV []      : AllV P []
_:allV:_ : {x : A} {n : ℕ} {xs : Vec A n} → P x → AllV P xs → AllV P (x :: xs)
data FindV {A : Set} (p : A → Bool) : {n : ℕ} → Vec A n → Set where
  foundV    : {k : ℕ} {m : ℕ} (xs : Vec A k) (y : A) → satisfies p y → (ys : Vec A m) → FindV p (xs ++ y :: ys)
  notfoundV : {n : ℕ}           {xs : Vec A n}       → AllV (satisfies (¬ ∘ p)) xs      → FindV p xs


findV : {A : Set} {n : ℕ} (p : A → Bool) (xs : Vec A n) → FindV
findV p []       = notfoundV allV []
findV p (x :: xs) with p x | inspect p x
... | true  | [prf] = foundV [] x (trueIsTrue prf) xs
... | false | _     with findV p xs
findV p (x :: _)  | false | [prf] | foundV xs y py ys = foundV (x :: xs) y py ys
findV p (x :: xs) | false | [prf] | notFoundV npxs   = notfoundV (lemma (falseIsFalse prf) :allV: npxs)
```

```
  -- Simple arithmetic lemmas automatically solved by the RingSolver
  -- (This is done by giving a syntactical representation of the theorem
  -- and letting the RingSolver rewrite this. This is succesful since
  -- I was able to use refl)
lemma2 : {m n k l : ℕ} → (suc (m + n + (k + l))) → (m + n + (k + suc l))
lemma2 {m} {n} {k} {l} = solve 4
  (λ m' n' k' l' →
    con 1 :+ (m' :+ n' :+ (k' :+ l')) :=
    m' :+ n' :+ (k' :+ (con 1 :+ l')))
  refl m n k l

lemma3 : {m n k l : ℕ} → (m + suc n + (k + l)) → (m + n + (k + suc l))
lemma3 {m} {n} {k} {l} = solve 4
           (λ m' n' k' l' →
             m' :+ (con 1 :+ n') :+ (k' :+ l') :=
             m' :+ n' :+ (k' :+ (con 1 :+ l')))
           refl m n k l

lemma4 : {a k l : ℕ} → a ≡ k + suc l → a ≡ suc (k + l)
lemma4 {a} {k} {l} p = trans p
  (solve 2 (λ k' l' → k' :+ (con 1 :+ l') := con 1 :+ (k' :+ l'))
           refl k l)

lemma5 : {a k l : ℕ} → suc a ≡ k + suc l → a ≡ k + l
lemma5 {a} {k} {l} p = cong pred
  (lemma4 {suc a} {k} {l} p)

  -- length of Vectors
length : {A : Set} {n : ℕ} → Vec A n → ℕ
length {_} {n} _ = n


  -- grounded' helper function for grounded defined below (using Vectors)
  -- grounded' takes 3 Vectors. The current ins and outs (starting empty),
  -- the arguments to process (args), a Dung AF (af)
  -- a predicate on the arguments allowing for comparison (_≡_)
  -- and a proof that there is a number equal to the length of args (o)
grounded' : {A : Set} → {m n o : ℕ} → (Σ ℕ λ k → k ≡ o) → (A → A → Bool)
                    → Vec A m → Vec A n → Vec A o → DungAF A → Vec (A × Status) (m + n + o)
  -- Base case:
  -- We have no more arguments to process.
grounded' _ _ ins outs [] _ = (map (λ x → (x , In)) ins ++ map (λ x → (x , Out)) outs) ++ []
  -- Inductive cases:
  -- Otherwise, we can possibly find an unattacked/attacked argument
grounded' _ _≡_ ins outs args af with findV (unattacked _≡_ (toList outs) af) args |
                                     findV (attacked   _≡_ (toList ins)  af) args
  -- Two impossible cases (() means that there is no valid constructor):
  -- The length of args is zero, while we did manage to find an unattacked/attacked element
  -- Thus we use lemma4 to rewrite o so we can match on suc using lemma4 (and suc _ ≢ zero)
grounded' {o = .(k + suc l)} (zero , p) _≡_ _ _ _ af | foundV {k} {l} _ _ _ _
                                                     | _ with lemma4 {zero} {k} {l} p
... | ()
grounded' {o = .(k + suc l)} (zero , p) _≡_ _ _ _ af | notFoundV _
                                                     | foundV {k} {l} _ _ _ _ with lemma4 {zero} {k} {l} p
... | ()
  -- Two cases:
  -- We have found an unattacked/attacked element.
  -- The Vector we try to return is of the "wrong" length, so we need to rewrite it using the basic lemma3
  -- and substitute this value in the Vector constructur
```

```
  -- Similarly we need to rewrite the proof of the length of o, using lemma5
grounded′ {_} {m} {n} {o = .(k + suc l)} (suc a , p) _≡_ ins outs
          .(xs ++ y :: ys) af | foundV {k} {l} xs y _ ys | _                = subst (Vec _) (lemma2 {m} {n} {k} {l})
            (grounded′ (a , lemma5 {a} {k} {l} p) _≡_ (y :: ins) outs (xs ++ ys) af)
grounded′ {_} {m} {n} {o = .(k + suc l)} (suc a , p) _≡_ ins outs
          .(xs ++ y :: ys) af | notFoundV _ | foundV {k} {l} xs y _ ys = subst (Vec _) (lemma3 {m} {n} {k} {l})
            (grounded′ (a , lemma5 {a} {k} {l} p) _≡_ ins (y :: outs) (xs ++ ys) af)
  -- Final case (fixpoint):
  -- We haven't found any unattacked/attacked element and thus are done.
grounded′ _ _ ins outs args _ | notFoundV _ | notFoundV _ = (map (λ x → (x , In)) ins
                                                           ++ map (λ x → (x , Out)) outs)
                                                           ++ map (λ x → (x , Undecided)) args


  -- The actual grounded labelling function.
  -- This function calls grounded' with the correct Vectors and lengths.
grounded : {A : Set} → (A → A → Bool) → DungAF A → List (A × Status)
grounded _≡_ (AF args def) = toList
  (grounded′ ((length (fromList args)) , refl)
             _≡_ [] [] (fromList args) (AF args def))
testGrounded1 : List (AbsArg × Status)
testGrounded1 = grounded String._==_ exampleAF1
  -- ("C", In) :: ("A", In) :: ("B", Out) :: []
testGrounded2 : List (AbsArg × Status)
testGrounded2 = grounded String._==_ exampleAF2
  -- ("A", Undecided) :: ("B", Undecided) :: []


  -- Defining the grounded extension using the grounded labelling is trivial,
  -- we just need to keep all the arguments with an In label
groundedExt : {A : Set} → (A → A → Bool) → DungAF A → List A
groundedExt _≡_ (AF args def) = List.map proj₁ ((filter ((_≡_ In) ∘ proj₂) (grounded _≡_ (AF args def))))
```

The Under-Performing Unfold

A new approach to optimising corecursive programs

Jennifer Hackett Graham Hutton
University of Nottingham
{jph,gmh}@cs.nott.ac.uk

Mauro Jaskelioff
Universidad Nacional de Rosario, Argentina
CIFASIS–CONICET, Argentina
jaskelioff@cifasis-conicet.gov.ar

Abstract

This paper presents a new approach to optimising corecursive programs by factorisation. In particular, we focus on programs written using the corecursion operator unfold. We use and expand upon the proof techniques of guarded coinduction and unfold fusion, capturing a pattern of generalising coinductive hypotheses by means of abstraction and representation functions. The pattern we observe is simple, has not been observed before, and is widely applicable. We develop a general program factorisation theorem from this pattern, demonstrating its utility with a range of practical examples.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Theory, Proof Methods, Optimisation

Keywords fusion, factorisation, coinduction, unfolds

1. Introduction

When writing programs that produce data structures it is often natural to use the technique of *corecursion* [8, 20], in which subterms of the result are produced by recursive calls. This is particularly useful in lazy languages such as Haskell, as it allows us to process parts of the result without producing the rest of it. In this way, we can write programs that deal with large or infinite data structures and trust that the memory requirements remain reasonable.

However, while this technique may allow us to save on the number of recursive calls by producing data lazily, the *time cost* of each call or the *space cost* of the arguments can still be a problem. For this reason, it is necessary to examine various approaches to reducing these costs. A commonly-used approach is *program fusion* [4, 11, 31], where separate stages of a computation are combined into one to avoid producing intermediate data. For this paper, our focus will be on the opposite technique of *program factorisation* [7], where a computation is split into separate parts. In particular, we consider the problem of splitting a computation into the combination of a more efficient *worker* program that uses a different representation of data and a *wrapper* program that effects the necessary change of representation [10].

The primary contribution of this paper is a general factorisation theorem for programs in the form of an *unfold* [9], a common pattern of corecursive programming. By exploiting the categorical principle of duality, we adapt the theory of *worker-wrapper factorisation for folds*, which was developed initially by Hutton, Jaskelioff and Gill [15] and subsequently extended by Sculthorpe and Hutton [24]. We discuss the practical considerations of the dual theory, which differ from those of the original theory and avoid the need for strictness side conditions. In addition, we revise the theory to further improve its utility. As we shall see, the resulting theory for unfolds captures a common pattern of coinductive reasoning that is simple, has not been observed before, and is widely applicable. We demonstrate the application of the new theory with a range of practical examples of varying complexity.

The development of our new theory is first motivated using a small programming example, which we then generalise using category theory. We use only simple concepts for this generalisation, and only a minimal amount of categorical knowledge is required. Readers without a background in category theory should not be deterred. The primary application area for our theory is functional languages such as Haskell, however the use of categorical concepts means that our theory is more widely applicable.

2. Coinductive Types and Proofs

Haskell programmers will be familiar with recursive type definitions, where a type is defined in terms of itself. For example, the type of natural numbers can be defined as follows:

$$\mathbf{data}\ \mathbb{N} = \mathit{Zero} \mid \mathit{Succ}\ \mathbb{N}$$

This definition states that an element of the type $\mathbb{N}$ is either Zero or $\mathit{Succ}\ n$ for some n that is also an element of the type $\mathbb{N}$. More formally, recursive type definitions can be given meaning as fixed points of type-level equations. For example, we can read the above as defining $\mathbb{N}$ to be a fixed point of the equation $X = 1 + X$, where 1 is the unit type and $+$ is the disjoint sum of types.

Assuming that such a fixed point equation has at least one solution, there are two we will typically be interested in. First of all, there is the *least* fixed point, which is given by the smallest type that is closed under the constructors. This is known as an *inductive* type. In a set-theoretic context, the inductive interpretation of our definition of $\mathbb{N}$ is simply the set of natural numbers, with constructors Zero and Succ having the usual behaviour.

Alternatively, there is the *greatest* fixed point, which is given by the largest type that supports deconstruction of values by pattern matching. This is known as a *coinductive* type [12]. In a set-theoretic context, the coinductive interpretation of our definition of $\mathbb{N}$ is the set of naturals augmented with an infinite value ∞ that is the solution to the equation $\infty = \mathit{Succ}\ \infty$.

In general, coinductively-defined sets have infinite elements, while inductively-defined sets do not. In the setting of Haskell, however, types correspond to *(pointed) complete partial orders*

(CPOs) rather than sets, where there is no distinction between inductive and coinductive type definitions as the two notions coincide [6]. For the purposes of this article we will use Haskell syntax as a metalanguage for programming in both set-theoretic and CPO contexts. It will therefore be necessary to distinguish between inductive and coinductive definitions, which we do so by using the keywords **data** and **codata** respectively.

For example, we could define an inductive type of lists and a coinductive type of streams as follows, using the constructor $(:)$ in both definitions for consistency with Haskell usage:

$$\begin{aligned}&\mathbf{data}\quad [a] &&= a : [a] \mid []\\ &\mathbf{codata}\ \mathit{Stream}\ a &&= a : \mathit{Stream}\ a\end{aligned}$$

If types are sets, the first definition gives finite lists while the second gives infinite streams. If types are CPOs, the first definition gives both finite and infinite lists, while the second gives infinite streams. Also note that in the context of sets, if streams were defined using **data** rather than **codata** the resulting type would be empty.

2.1 Coinduction

To reason about elements of inductive types one can use the technique of induction. Likewise, to reason about elements of coinductive types one can use the dual technique of *coinduction*. To formalise this precisely involves the notion of *bisimulation* [8, 12]. In this section we shall give an informal presentation of *guarded* coinduction [3, 28], a special case that avoids the need for such machinery. This form of coinduction is closely related to the unique fixed point principle developed by Hinze [14].

To prove an equality $lhs = rhs$ between expressions of the same coinductive type using guarded coinduction, we simply attempt the proof in the usual way using equational reasoning. However, we may also make use of the *coinductive hypothesis*, which allows us to substitute *lhs* for *rhs* (or vice-versa) provided that we only do so immediately underneath a constructor of the coinductive type. We say that such a use of the coinductive hypothesis is *guarded*. For example, if we define the following functions that produce values of type $\mathit{Stream}\ \mathbb{N}$

$$\begin{aligned}&\mathit{from}\ n &&= n : \mathit{from}\ (n+1)\\ &\mathit{skips}\ n &&= n : \mathit{skips}\ (n+2)\\ &\mathit{double}\ (n : ns) &&= n * 2 : \mathit{double}\ ns\end{aligned}$$

then we can show that $\mathit{skips}\ (n*2) = \mathit{double}\ (\mathit{from}\ n)$ for any natural number n using guarded coinduction:

$$\begin{aligned}&\mathit{skips}\ (n*2)\\ =\quad &\{\ \text{definition of } \mathit{skips}\ \}\\ &n*2 : \mathit{skips}\ ((n*2)+2)\\ =\quad &\{\ \text{arithmetic}\ \}\\ &n*2 : \mathit{skips}\ ((n+1)*2)\\ =\quad &\{\ \text{coinductive hypothesis}\ \}\\ &n*2 : \mathit{double}\ (\mathit{from}\ (n+1))\\ =\quad &\{\ \text{definition of } \mathit{double}\ \}\\ &\mathit{double}\ (n : \mathit{from}\ (n+1))\\ =\quad &\{\ \text{definition of } \mathit{from}\ \}\\ &\mathit{double}\ (\mathit{from}\ n)\end{aligned}$$

Despite the apparent circularity in using an instance of our desired result in the third step of the proof, the proof is guarded because the use of the coinductive hypothesis only occurs directly below the $(:)$ constructor. Therefore the reasoning is valid.

3. Example: Tabulating a Function

Consider the problem of *tabulating* a function $f :: \mathbb{N} \to a$ by applying it to every natural number in turn and forming a stream from the results. We would like to define a function that performs this task, specified informally as follows:

$$\begin{aligned}&\mathit{tabulate} :: (\mathbb{N} \to a) \to \mathit{Stream}\ a\\ &\mathit{tabulate}\ f = [f\ 0, f\ 1, f\ 2, f\ 3, \ldots]\end{aligned}$$

The following definition satisfies this specification:

$$\mathit{tabulate}\ f = f\ 0 : \mathit{tabulate}\ (f \circ (+1))$$

However, this definition is inefficient, as with each recursive call the function argument becomes more costly to apply, as shown in the following expansion of the definition:

$$\mathit{tabulate}\ f = [f\ 0, (f \circ (+1))\ 0, (f \circ (+1) \circ (+1))\ 0, \ldots]$$

The problem is that the natural number is recomputed from scratch each time by repeated application of $(+1)$ to 0. If we were to save the result and re-use it in future steps, we could avoid repeating work. The idea can be implemented by defining a new function that takes the current value as an additional argument:

$$\begin{aligned}&\mathit{tabulate}' :: (\mathbb{N} \to a, \mathbb{N}) \to \mathit{Stream}\ a\\ &\mathit{tabulate}'\ (f, n) = f\ n : \mathit{tabulate}'\ (f, n+1)\end{aligned}$$

The correctness of the more efficient implementation for tabulation can be captured by the following equation

$$\mathit{tabulate}\ f = \mathit{tabulate}'\ (f, 0)$$

which can written in point-free form as

$$\mathit{tabulate} = \mathit{tabulate}' \circ (\lambda f \to (f, 0))$$

This equation can be viewed as a *program factorisation*, in which $\mathit{tabulate}$ is factored into the composition of $\mathit{tabulate}'$ and the function $\lambda f \to (f, 0)$. This latter function effects a *change of data representation* from the old argument type $\mathbb{N} \to a$ to the new argument type $(\mathbb{N} \to a, \mathbb{N})$. In order to try to prove the above equation, we proceed by guarded coinduction:

$$\begin{aligned}&\mathit{tabulate}\ f\\ =\quad &\{\ \text{definition of } \mathit{tabulate}\ \}\\ &f\ 0 : \mathit{tabulate}\ (f \circ (+1))\\ =\quad &\{\ \text{coinduction hypothesis}\ \}\\ &f\ 0 : \mathit{tabulate}'\ (f \circ (+1), 0)\\ =\quad &\{\ \text{assumption}\ \}\\ &f\ 0 : \mathit{tabulate}'\ (f, 1)\\ =\quad &\{\ \text{definition of } \mathit{tabulate}'\ \}\\ &\mathit{tabulate}'\ (f, 0)\end{aligned}$$

To complete the proof, the assumption used in the third step $\mathit{tabulate}'\ (f \circ (+1), 0) = \mathit{tabulate}'\ (f, 1)$ must be verified. We could attempt to prove this as follows:

$$\begin{aligned}&\mathit{tabulate}'\ (f \circ (+1), 0)\\ =\quad &\{\ \text{definition of } \mathit{tabulate}'\ \}\\ &(f \circ (+1))\ 0 : \mathit{tabulate}'\ (f \circ (+1), 1)\\ =\quad &\{\ \text{composition, arithmetic}\ \}\\ &f\ 1 : \mathit{tabulate}'\ (f \circ (+1), 1)\\ =\quad &\{\ \text{assumption}\ \}\\ &f\ 1 : \mathit{tabulate}'\ (f, 2)\\ =\quad &\{\ \text{definition of } \mathit{tabulate}'\ \}\\ &\mathit{tabulate}'\ (f, 1)\end{aligned}$$

Once again, however, the proof relies on an assumption that needs to be verified. We could continue like this *ad infinitum* without ever actually completing the proof! We can avoid this problem by *generalising* our correctness property to

$$\mathit{tabulate}\ (f \circ (+n)) = \mathit{tabulate}'\ (f, n)$$

which in the case of $n = 0$ simplifies to the original equation. The proof of the generalised property is now a straightforward application of guarded coinduction, with no assumptions required:

$$\begin{array}{ll} & \mathit{tabulate}\ (f \circ (+n)) \\ = & \quad \{\ \text{definition of } \mathit{tabulate}\ \} \\ & (f \circ (+n))\ 0\ :\ \mathit{tabulate}\ (f \circ (+n) \circ (+1)) \\ = & \quad \{\ \text{simplification}\ \} \\ & f\ n\ :\ \mathit{tabulate}\ (f \circ (+(n+1))) \\ = & \quad \{\ \text{coinduction hypothesis}\ \} \\ & f\ n\ :\ \mathit{tabulate}'\ (f, n+1) \\ = & \quad \{\ \text{definition of } \mathit{tabulate}'\ \} \\ & \mathit{tabulate}'\ (f, n) \end{array}$$

It is often necessary to generalise coinductive hypotheses in this way for proofs by guarded coinduction, just as it is often necessary to generalise inductive hypotheses for proofs by induction.

4. Abstracting

The above example is an instance of a general pattern of optimisation that is simple yet powerful. We abstract from this example to the general case in two steps, firstly by generalising on the underlying datatypes involved and secondly by generalising on the pattern of corecursive definition that is used.

4.1 Abstracting on Datatypes

In the tabulation example, we replaced the original function of type $(\mathbb{N} \to a) \to \mathit{Stream}\ a$ with a more efficient function of type $(\mathbb{N} \to a, \mathbb{N}) \to \mathit{Stream}\ a$, changing the type of the argument. Essentially, we used a "larger" type as a representation of a "smaller" type. We can generalise this idea to any two types where one serves as a representation of the other.

Suppose we have two types, a and b, with conversion functions $\mathit{abs} :: b \to a$ and $\mathit{rep} :: a \to b$ such that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_a$. We can think of b as a larger type that faithfully represents the elements of the smaller type a. Now suppose that we are given a function $\mathit{old} :: a \to c$, together with a more efficient version $\mathit{new} :: b \to c$ that acts on the larger type b. Then the correctness of the more efficient version can be captured by the equation

$$\mathit{old} = \mathit{new} \circ \mathit{rep}$$

However, using the assumption that $\mathit{abs} \circ \mathit{rep} = \mathit{id}_a$ we can strengthen this property by the following calculation:

$$\begin{array}{ll} & \mathit{old} = \mathit{new} \circ \mathit{rep} \\ \Leftrightarrow & \quad \{\ \mathit{abs} \circ \mathit{rep} = \mathit{id}_a\ \} \\ & \mathit{old} \circ \mathit{abs} \circ \mathit{rep} = \mathit{new} \circ \mathit{rep} \\ \Leftarrow & \quad \{\ \text{cancelling } \mathit{rep} \text{ on both sides}\ \} \\ & \mathit{old} \circ \mathit{abs} = \mathit{new} \end{array}$$

In summary, if we wish to show that function new is correct, it suffices to show that $\mathit{old} \circ \mathit{abs} = \mathit{new}$. This stronger correctness property may be easier to prove than the original version.

We now apply the above idea to our earlier example. In this case, the appropriate abs and rep functions are:

$$\begin{array}{l} \mathit{abs} :: (\mathbb{N} \to a, \mathbb{N}) \to (\mathbb{N} \to a) \\ \mathit{abs}\ (f, n) = f \circ (+n) \\ \mathit{rep} :: (\mathbb{N} \to a) \to (\mathbb{N} \to a, \mathbb{N}) \\ \mathit{rep}\ f = (f, 0) \end{array}$$

The required relationship $\mathit{abs} \circ \mathit{rep} = \mathit{id}$ follows immediately from the fact that 0 is the identity for addition. The above calculation can therefore be specialised to our example as follows:

$$\begin{array}{ll} & \forall\, f\ .\ \mathit{tabulate}\ f = \mathit{tabulate}'\ (f, 0) \\ \Leftrightarrow & \quad \{\ \text{definition of } \mathit{rep}\ \} \\ & \forall\, f\ .\ \mathit{tabulate}\ f = \mathit{tabulate}'\ (\mathit{rep}\ f) \\ \Leftrightarrow & \quad \{\ \text{composition, extensionality}\ \} \\ & \mathit{tabulate} = \mathit{tabulate}' \circ \mathit{rep} \\ \Leftrightarrow & \quad \{\ \mathit{abs} \circ \mathit{rep} = \mathit{id}\ \} \\ & \mathit{tabulate} \circ \mathit{abs} \circ \mathit{rep} = \mathit{tabulate}' \circ \mathit{rep} \\ \Leftarrow & \quad \{\ \text{cancelling } \mathit{rep} \text{ on both sides}\ \} \\ & \mathit{tabulate} \circ \mathit{abs} = \mathit{tabulate}' \\ \Leftrightarrow & \quad \{\ \text{composition, extensionality}\ \} \\ & \forall\, f, n\ .\ \mathit{tabulate}\ (\mathit{abs}\ (f, n)) = \mathit{tabulate}'\ (f, n) \\ \Leftrightarrow & \quad \{\ \text{definition of } \mathit{abs}\ \} \\ & \forall\, f, n\ .\ \mathit{tabulate}\ (f \circ (+n)) = \mathit{tabulate}'\ (f, n) \end{array}$$

The final equation is precisely the generalised correctness property from the previous section, but has now been obtained from an abstract framework that is generic in the underlying datatypes.

4.2 Abstracting on Corecursion Pattern

For the next step in the generalisation, we make some assumptions about the corecursive structure of the functions that we are dealing with. In particular, we assume that they are instances of a specific pattern of corecursive definition called *unfold* [9, 18].

4.2.1 Unfold for Streams

An unfold is a function that produces an element of a coinductive type as its result, producing all subterms of the result using recursive calls. This pattern can be abstracted into an operator, which we define in the case of streams as follows:

$$\begin{array}{l} \mathrm{unfold} :: (a \to b) \to (a \to a) \to a \to \mathit{Stream}\ b \\ \mathrm{unfold}\ h\ t\ x = h\ x\ :\ \mathrm{unfold}\ h\ t\ (t\ x) \end{array}$$

The function $\mathrm{unfold}\ \ h\ \ t$ produces a stream from a seed value x by using the function h to produce the head of the stream from the seed, and applying the function t to produce a new seed that is used to generate the tail of the stream in the same manner. For efficiency we could choose to combine the h and t functions into a single function of type $a \to (b, a)$, allowing for increase sharing between the two computations. However, we present a 'tuple-free' version because it leads to simpler equational reasoning.

By providing suitable definitions for h and t, it is straightforward to redefine the functions $\mathit{tabulate}$ and $\mathit{tabulate}'$:

$$\begin{array}{lll} \mathit{tabulate} & = \mathrm{unfold}\ h\ t & \\ & \quad \mathbf{where}\ h\ \ f = f\ 0 & \\ & \qquad\ \ t\ \ f = f \circ (+1) & \\ \mathit{tabulate}' & = \mathrm{unfold}\ h'\ t' & \\ & \quad \mathbf{where}\ h'\ (f, n) = f\ n & \\ & \qquad\ \ t'\ \ (f, n) = (f, n+1) & \end{array}$$

In this way, unfold allows us to factor out the basic steps in the computations. A similar unfold operator can be defined for any coinductive type. For example, for infinite binary trees

$$\mathbf{codata}\ \mathit{Tree}\ a = \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

the following definition for $\mathrm{unfold}\ \ l\ \ n\ \ r$ produces a tree from a seed value by using l and r to produce new seeds for the left and right subtrees, and n to produce the node value:

$$\begin{array}{l} \mathrm{unfold} :: (a \to a) \to (a \to b) \to (a \to a) \to a \to \mathit{Tree}\ b \\ \mathrm{unfold}\ l\ n\ r\ x = \mathit{Node}\ (\mathrm{unfold}\ l\ n\ r\ (l\ x)) \\ \qquad\qquad\qquad\qquad\quad (n\ x) \\ \qquad\qquad\qquad\qquad\quad (\mathrm{unfold}\ l\ n\ r\ (r\ x)) \end{array}$$

Once again, the l, n and r functions could be combined.

4.2.2 Unfold Fusion

The unfold operator for any type has an associated *fusion* law [18], which provides sufficient conditions for when the composition of

an unfold with another function can be expressed as a single unfold. In the case of streams, the law is as follows:

Theorem 1 (Unfold Fusion for Streams)**.** *Given*

$$h :: a \to c \quad h' :: b \to c \quad g :: b \to a$$
$$t :: a \to a \quad t' :: b \to b$$

we have the following implication:

$$\begin{array}{ll} & \text{unfold } h\ t \circ g = \text{unfold } h'\ t' \\ \Leftarrow & \\ & h' = h \circ g \ \wedge \ g \circ t' = t \circ g \end{array}$$

The proof is a simple application of guarded coinduction:

$$\begin{array}{ll} & \text{unfold } h'\ t'\ x \\ = & \quad \{ \text{ definition of unfold } \} \\ & h'\ x\ :\ \text{unfold } h'\ t'\ (t'\ x) \\ = & \quad \{ \text{ coinduction hypothesis } \} \\ & h'\ x\ :\ \text{unfold } h\ t\ (g\ (t'\ x)) \\ = & \quad \{ \text{ first assumption: } h' = h \circ g \ \} \\ & h\ (g\ x)\ :\ \text{unfold } h\ t\ (g\ (t'\ x)) \\ = & \quad \{ \text{ second assumption: } g \circ t' = t \circ g \ \} \\ & h\ (g\ x)\ :\ \text{unfold } h\ t\ (t\ (g\ x)) \\ = & \quad \{ \text{ definition of unfold } \} \\ & \text{unfold } h\ t\ (g\ x) \end{array}$$

The fusion law provides sufficient conditions for when our strengthened correctness property $old \circ abs = new$ holds. Assuming that old and new can both be expressed as unfolds, then:

$$\begin{array}{ll} & old \circ abs = new \\ \Leftrightarrow & \quad \{\ old = \text{unfold } h\ t,\ new = \text{unfold } h'\ t'\ \} \\ & \text{unfold } h\ t \circ abs = \text{unfold } h'\ t' \\ \Leftarrow & \quad \{ \text{ fusion } \} \\ & h' = h \circ abs \ \wedge \ abs \circ t' = t \circ abs \end{array}$$

4.3 Unfold Factorisation for Streams

Combining the two ideas of abstracting on the types and abstracting on the corecursion pattern, we obtain a general theorem for factorising functions defined using unfold for streams.

Theorem 2 (Unfold Factorisation for Streams)**.** *Given*

$$abs :: b \to a \quad h :: a \to c \quad h' :: b \to c$$
$$rep :: a \to b \quad t :: a \to a \quad t' :: b \to b$$

satisfying the assumptions

$$\begin{array}{ll} abs \circ rep & = id_a \\ h' & = h \circ abs \\ abs \circ t' & = t \circ abs \end{array}$$

we have the factorisation

$$\text{unfold } h\ t = \text{unfold } h'\ t' \circ rep$$

Using this result, we can split a function unfold $h\ \ t$ *into the composition of a* worker *function* unfold $h'\ t'$ *that uses a different representation of data and a* wrapper *function* rep *that effects the necessary change of data representation.*

We now apply this to the $tabulate$ example. As we have already shown that $abs \circ rep = id$, it is only necessary to verify the remaining two assumptions. We start with the first assumption:

$$\begin{array}{ll} & h\ (abs\ (f, n)) \\ = & \quad \{ \text{ definition of } abs \ \} \\ & h\ (f \circ (+n)) \\ = & \quad \{ \text{ definition of } h \ \} \\ & (f \circ (+n))\ 0 \\ = & \quad \{ \text{ simplification } \} \\ & f\ n \\ = & \quad \{ \text{ definition of } h' \ \} \\ & h'\ (f, n) \end{array}$$

Now, the second assumption:

$$\begin{array}{ll} & t\ (abs\ (f, n)) \\ = & \quad \{ \text{ definition of } abs \ \} \\ & t\ (f \circ (+n)) \\ = & \quad \{ \text{ definition of } t \ \} \\ & f \circ (+n) \circ (+1) \\ = & \quad \{ \text{ simplification } \} \\ & f \circ (+(n+1)) \\ = & \quad \{ \text{ definition of } abs \ \} \\ & abs\ (f, n+1) \\ = & \quad \{ \text{ definition of t' } \} \\ & abs\ (t'\ (f, n)) \end{array}$$

In conclusion, by generalising from our tabulation example we have derived a framework for factorising corecursive functions that are defined using the unfold operator for streams.

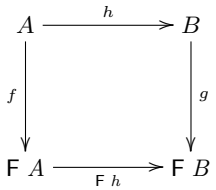
5. Categorifying

To recap, we have combined the idea of a change of data representation with an application of fusion to produce a factorisation theorem for stream unfolds. This theorem covers cases not covered by fusion alone. For example, attempting to prove the $tabulate$ example correct simply using fusion fails in precisely the same way that our attempted proof using coinduction failed.

However, so far we have only concerned ourselves with the coinductive type of streams. If we wish to apply this technique to other coinductive types, it would seem that we must define unfold and prove its fusion law for every such type we intend to use. Thankfully this is not the case, as category theory provides a convenient generic approach to modelling coinductive types and their unfold operators using the notion of *final coalgebras* [18].

5.1 Final Coalgebras

Suppose that we fix a category $\mathcal{C}$ and a functor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ on this category. Then an F-*coalgebra* is a pair (A, f) consisting of an object A along with an arrow $f : A \to \mathsf{F}\ A$. We often omit the object A as it is implicit in the type of f. A *homomorphism* between coalgebras $f : A \to \mathsf{F}\ A$ and $g : B \to \mathsf{F}\ B$ is an arrow $h : A \to B$ such that $\mathsf{F}\ h \circ f = g \circ h$. This property is captured by the following commutative diagram:



Intuitively, a coalgebra $f : A \to \mathsf{F}\ A$ can be thought of as giving a *behaviour* to elements of A, where the possible behaviours are specified by the functor F. For example, if we define $\mathsf{F}\ X = 1 + X$ on the category **Set** of sets and total functions, then a coalgebra $f : A \to 1 + A$ is the transition function of a state machine in which each element of A is either a terminating state or has a single successor. In turn, a homomorphism corresponds to a behaviour-preserving mapping, in the sense that if we first apply the homomorphism h and then the target behaviour captured by g, we

obtain the same result as if we apply the source behaviour captured by f and then apply h to the components of the result.

A *final* coalgebra, denoted $(\nu\mathsf{F}, out)$, is an F-coalgebra to which any other coalgebra has a unique homomorphism. If a final coalgebra exists, it is unique up to isomorphism. Given a coalgebra $f : A \to \mathsf{F}\ A$, the unique homomorphism from f to the final coalgebra out is denoted $\mathrm{unfold}\ f :: A \to \nu\mathsf{F}$. This *uniqueness property* can be captured by the following equivalence:

$$h = \mathrm{unfold}\ f \quad \Leftrightarrow \quad \mathsf{F}\ h \circ f = out \circ h$$

We also have a fusion rule for unfold:

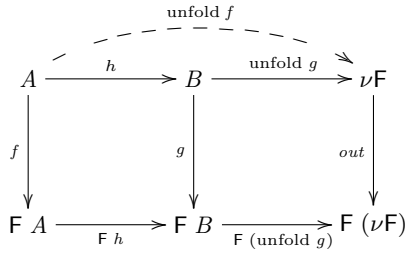
Theorem 3 (Unfold Fusion for Final Coalgebras)**.** *Given*

$$f : A \to \mathsf{F}\ A \qquad g : B \to \mathsf{F}\ B \qquad h : A \to B$$

we have the following implication:

$$\begin{array}{l} \quad \mathrm{unfold}\ g \circ h = \mathrm{unfold}\ f \\ \Leftarrow \\ \quad \mathsf{F}\ h \circ f = g \circ h \end{array}$$

The proof of this theorem can be conveniently captured by the following commutative diagram:



The left square commutes by assumption while the right square commutes because unfold g is a coalgebra homomorphism. Therefore, the outer rectangle commutes, meaning that unfold $g \circ h$ is a homomorphism from f to out. Finally, because homomorphisms to the final coalgebra out are unique and unfold f is also such a homomorphism, the result unfold $g \circ h =$ unfold f holds.

We illustrate the above concepts with a concrete example. Consider the functor $\mathsf{F}\ X = \mathbb{N} \times X$ on the category $\mathbf{Set}$. This functor has a final coalgebra $(Stream\ \mathbb{N}, \langle head, tail\rangle)$, consisting of the set $Stream\ \mathbb{N}$ of streams of natural numbers together with the function $\langle head, tail\rangle : Stream\ \mathbb{N} \to \mathbb{N} \times Stream$ that combines the stream destructors $head : Stream\ \mathbb{N} \to \mathbb{N}$ and $tail : Stream\ \mathbb{N} \to Stream\ \mathbb{N}$. Given any set A and functions $h : A \to \mathbb{N}$ and $t : A \to A$, the function $\mathrm{unfold}\langle h, t\rangle : A \to Stream\ \mathbb{N}$ is uniquely defined by the two equations

$$\begin{array}{l} head \circ \mathrm{unfold}\langle h, t\rangle = h \\ tail\ \circ \mathrm{unfold}\langle h, t\rangle = \mathrm{unfold}\langle h, t\rangle \circ t \end{array}$$

which are equivalent to the more familiar definition of unfold using using the stream constructor $(:)$ presented earlier:

$$\mathrm{unfold}\ h\ t\ x = h\ x\ :\ \mathrm{unfold}\ h\ t\ (t\ x)$$

We also note that the earlier fusion law for streams is simply a special case of the more general fusion law where $\mathsf{F}\ X = \mathbb{N} \times X$. The fusion precondition simplifies as follows

$$\begin{array}{ll} & \mathsf{F}\ g \circ \langle h', t'\rangle = \langle h, t\rangle \circ g \\ \Leftrightarrow & \quad \{\ \text{definition of F, products}\ \} \\ & \langle h', g \circ t'\rangle = \langle h \circ g, t \circ g\rangle \\ \Leftrightarrow & \quad \{\ \text{separating components}\ \} \\ & h' = h \circ g\ \ \wedge\ \ g' \circ t = t \circ g \end{array}$$

and the postcondition is clearly equivalent. All of the above also holds for $Stream\ A$ for an arbitrary set A.

It is now straightforward to generalise our earlier unfold factorisation theorem from streams to final coalgebras. Combining the general unfold fusion law with the same type abstraction idea from before, we obtain the following theorem.

Theorem 4 (General Unfold Factorisation)**.** *Given*

$$\begin{array}{ll} abs : B \to A & f : A \to \mathsf{F}\ A \\ rep : A \to B & g : B \to \mathsf{F}\ B \end{array}$$

satisfying the assumptions

$$\begin{array}{l} abs \circ rep = id_A \\ \mathsf{F}\ abs \circ g = f \circ abs \end{array}$$

we have the factorisation

$$\mathrm{unfold}\ f = \mathrm{unfold}\ g \circ rep$$

that splits the original corecursive program unfold f *into the composition of a worker* unfold g *and a wrapper* rep*.*

5.2 Exploiting Duality

Our results up to this point have been generic with respect to the choice of a category $\mathcal{C}$. This is helpful, because not only are the results general, they are also subject to *duality*.

In category theory, the principle of duality states that if a property holds of all categories, then the *dual* of that property must also hold of all categories. By applying this duality principle to our general unfold factorisation theorem, we obtain the following factorisation theorem for *folds*, the categorical dual of unfolds:

Theorem 5. *Given*

$$\begin{array}{ll} abs : A \to B & f : \mathsf{F}\ A \to A \\ rep : B \to A & g : \mathsf{F}\ B \to B \end{array}$$

satisfying the assumptions

$$\begin{array}{l} rep \circ abs = id_A \\ g \circ \mathsf{F}\ abs = abs \circ f \end{array}$$

we have the factorisation

$$\mathrm{fold}\ f = rep \circ \mathrm{fold}\ g$$

that splits the original recursive program fold f *into the composition of a wrapper* rep *and a worker* fold g*.*

Note that now rep is required to be a left-inverse of abs, rather than the other way around. If we swap their names to reflect this new situation, we see that this is a special case of the following general result, due to Sculthorpe and Hutton [24]:

Theorem 6 (Worker-Wrapper Factorisation for Initial Algebras)**.**

Given

$$\begin{array}{ll} abs : B \to A & f : \mathsf{F}\ A \to A \\ rep : A \to B & g : \mathsf{F}\ B \to B \end{array}$$

satisfying one of the assumptions

$$\begin{array}{ll} (A)\ abs \circ rep & = id_A \\ (B)\ abs \circ rep \circ f & = f \\ (C)\ \mathrm{fold}\ (abs \circ rep \circ f) & = \mathrm{fold}\ f \end{array}$$

and one of the conditions

$$\begin{array}{ll} (1)\ g = rep \circ f \circ \mathsf{F}\ abs & (1\beta)\ \mathrm{fold}\ g = \mathrm{fold}\ (rep \circ f \circ \mathsf{F}\ abs) \\ (2)\ g \circ \mathsf{F}\ rep = rep \circ f & (2\beta)\ \mathrm{fold}\ g = rep \circ \mathrm{fold}\ f \\ (3)\ f \circ \mathsf{F}\ abs = abs \circ g & \end{array}$$

we have the factorisation

$$\text{fold } f = abs \circ \text{fold } g$$

If we now apply duality in turn to this theorem, we obtain an even more general version of our unfold factorisation theorem:

Theorem 7 (Worker-Wrapper Factorisation for Final Coalgebras)**.**

Given

$$\begin{array}{ll} abs : B \to A & f : A \to \mathsf{F}\ A \\ rep : A \to B & g : B \to \mathsf{F}\ B \end{array}$$

satisfying one of the assumptions

$$\begin{array}{ll} (A)\ abs \circ rep & = id_A \\ (B)\ f \circ abs \circ rep & = f \\ (C)\ \text{unfold } (f \circ abs \circ rep) & = \text{unfold } f \end{array}$$

and one of the conditions

$$\begin{array}{l} \textit{(1)}\ g = \mathsf{F}\ rep \circ f \circ abs \\ \textit{(2)}\ \mathsf{F}\ abs \circ g = f \circ abs \\ \textit{(3)}\ \mathsf{F}\ rep \circ f = g \circ rep \end{array}$$

$$\begin{array}{l} \textit{(1}\beta\textit{)}\ \text{unfold } g = \text{unfold } (\mathsf{F}\ rep \circ f \circ abs) \\ \textit{(2}\beta\textit{)}\ \text{unfold } g = \text{unfold } f \circ abs \end{array}$$

we have the factorisation

$$\text{unfold } f = \text{unfold } g \circ rep$$

At this point it would be reasonable to ask why we did not simply present the dualised theorem straight away. This is indeed possible, but we do not feel it would be a good approach. In particular, our systematic development that starts from a concrete example and then applies steps of abstraction, generalisation and dualisation provides both motivation and explanation for the theorem.

We now turn our attention to interpreting Theorem 7. First of all, assumptions (B) and (C) are simply generalised versions of our original assumption (A), in the sense that (A) $\Rightarrow$ (B) $\Rightarrow$ (C). Secondly, conditions (1) and (3) are alternatives to the original condition (2), providing a degree of flexibility for the user of the theorem to select the most convenient. In general these three conditions are unrelated, but any of them is sufficient to ensure that the theorem holds. Finally, the β conditions in the second group arise as weaker versions of the corresponding conditions in the first, i.e. (1) $\Rightarrow$ (1β) and (2) $\Rightarrow$ (2β), and given assumption (C) the two β conditions are equivalent. We omit proofs as they are dual to those in [24].

While the new assumptions and conditions are dual to those of the original theorem, they differ significantly in terms of their practical considerations, which we shall discuss now. Firstly, assumption (B) in the theorem for fold, i.e. $abs \circ rep \circ f = f$, can be interpreted as "for any x in the range of f, $abs\ (rep\ x) = x$". This can therefore be proven by reasoning only about such x. When applying the theorem for unfold, this kind of simple reasoning for assumption (B) is not possible, as f is now applied last rather than first and hence cannot be factored out of the proof.

Secondly, condition (2) in the fold case, i.e. $rep \circ f = g \circ \mathsf{F}\ rep$, allowed g to depend on a precondition set up by rep. If such a precondition is desired for the unfold case, condition (3) must be used. This has important implications for use of this theorem as a basis for optimisation, as we will often derive g based on a specification given by one of the conditions.

Finally, we note that proving (C), (1β) or (2β) for the fold case usually requires induction. To prove the corresponding properties for the unfold case will usually require the less widely-understood technique of coinduction. These properties may therefore turn out to be less useful in the unfold case for practical purposes, despite only requiring a technique of comparable complexity. If we want to use this theory to avoid coinduction altogether, assumption (C) and the β conditions are not applicable. Section 5.3 offers a way around this problem in the case of (C).

5.3 Refining Assumption (C)

As it stands, assumption (C) is expressed as an equality between two corecursive programs defined using unfold, and hence may be non-trivial to prove. However, we can derive an equivalent assumption that may be easier to prove in practice:

$$\begin{array}{ll} & \text{unfold } f = \text{unfold } (f \circ abs \circ rep) \\ \Leftrightarrow & \quad \{ \text{ uniqueness property of unfold } (f \circ abs \circ rep) \ \} \\ & out \circ \text{unfold } f = \mathsf{F}\ (\text{unfold } f) \circ f \circ abs \circ rep \\ \Leftrightarrow & \quad \{ \text{ unfold } f \text{ is a homomorphism } \} \\ & out \circ \text{unfold } f = out \circ \text{unfold } f \circ abs \circ rep \\ \Leftrightarrow & \quad \{ \ out \text{ is an isomorphism } \} \\ & \text{unfold } f = \text{unfold } f \circ abs \circ rep \end{array}$$

We denote this equivalent version of assumption (C) as (C'). As this new assumption concerns only the conversions abs and rep along with the original program unfold f, it may be provable simply from the original program's correctness properties.

Assumption (C') also offers a simpler proof of Theorem 7 than one obtains by dualising the proof in [24]. We start from this assumption and use the fact that in this context, conditons (1), (2) and (1β) all imply (2β):

$$\begin{array}{ll} & \text{unfold } f = \text{unfold } f \circ abs \circ rep \\ \Rightarrow & \quad \{ \ (2\beta)\text{: unfold } g = \text{unfold } f \circ abs \ \} \\ & \text{unfold } f = \text{unfold } g \circ rep \end{array}$$

The proof in the case of condition (3) remains the same as previously. We conclude by noting that the implication (B) $\Rightarrow$ (C') is not as obvious as the original implication (B) $\Rightarrow$ (C). Altering the theory in this manner thus "moves work" from proving the main result to proving the relationships between the conditions.

5.4 Applying the Theory in Haskell

The category that is usually used to model Haskell types and functions is $\mathbf{CPO}$, the category of (pointed) complete partial orders and continuous functions. While a fold operator can be defined in this category, its uniqueness property carries a *strictness* side condition [18]. As a result, the worker-wrapper theory for folds in $\mathbf{CPO}$ requires a strictness condition of its own [24]. However, in the case of unfold in $\mathbf{CPO}$ the uniqueness property holds with no side conditions [18] so our theorem can be freely used to reason about Haskell programs without such concerns.

6. Examples

We now present a collection of worked examples, demonstrating how our new factorisation theorem may be applied. Firstly, we revisit the tabulation example, and show that it is a simple application of the theory. Secondly, we consider the problem of cycling a list, where we reduce the time cost by delaying expensive operations and performing them in a batch. We believe that this will be a common use of our theory. Thirdly, we consider the problem of taking the initial segment of a list, which allows us to demonstrate how sometimes different choices of condition can lead to the same result. Finally, we consider the problem of flattening a tree. In all cases the proofs are largely mechanical and the main inspiration necessary is in the choice of a new data representation.

With the exception of the initial segment example, all of the following examples occur in the context of the category $\mathbf{Set}$ of sets

and total functions. Working in **Set** results in simpler reasoning as we do not need to consider issues of partiality.

6.1 Example: Tabulating a Function

We can instantiate our theory as shown above to give a proof of the correctness of our tabulate example. The proof uses assumption (A) and condition (2). Therefore we see that this example is a simple application of the worker-wrapper machinery.

6.2 Example: Cycling a List

The function $cycle$ takes a non-empty finite list and produces the stream consisting of repetitions of that list. For example:

$$cycle\ [1,2,3] = [1,2,3,1,2,3,1,2,3,\ldots$$

One possible definition for $cycle$ is as follows, in which we write $[a]^+$ for the type of non-empty lists of type a:

$$\begin{aligned}&cycle :: [a]^+ \to Stream\ a\\ &cycle\ (x : xs) = x : cycle\ (xs +\!\!+ [x])\end{aligned}$$

However, this definition is inefficient, as the append operator $+\!\!+$ takes linear time in the length of the input list. Recalling that $Stream\ a$ is the final coalgebra of the functor $\mathsf{F}\ X = a \times X$, we can rewrite $cycle$ as an unfold:

$$\begin{aligned}&cycle = \text{unfold}\ h\ t\\ &\quad \textbf{where}\ h\ xs = head\ xs\\ &\qquad\quad\ \ t\ xs = tail\ xs +\!\!+ [head\ xs]\end{aligned}$$

The idea we shall apply to improve the performance of $cycle$ is to combine several $+\!\!+$ operations into one, thus reducing the average cost. To achieve this, we create a new representation where the original list of type $[a]^+$ is augmented with a (possibly empty) list of elements that have been added to the end. We keep this second list in reverse order so that appending a single element is a constant-time operation. The rep and abs functions are as follows:

$$\begin{aligned}&rep :: [a]^+ \to ([a]^+, [a])\\ &rep\ xs = (xs, [\,])\end{aligned}$$

$$\begin{aligned}&abs :: ([a]^+, [a]) \to [a]^+\\ &abs\ (xs, ys) = xs +\!\!+ reverse\ ys\end{aligned}$$

Given these definitions it is easy to verify assumption (A):

$$\begin{aligned}&\ \ abs\ (rep\ xs)\\ &=\quad \{\ \text{definition of } rep\ \}\\ &\ \ abs\ (xs, [\,])\\ &=\quad \{\ \text{definition of } abs\ \}\\ &\ \ xs +\!\!+ reverse\ [\,]\\ &=\quad \{\ \text{definition of } reverse\ \}\\ &\ \ xs +\!\!+ [\,]\\ &=\quad \{\ [\,] \text{ is unit of } +\!\!+\ \}\\ &\ \ xs\end{aligned}$$

For this example we take condition (2), i.e. $\mathsf{F}\ \ abs \circ g = f \circ abs$, as our specification of g, once again specialising to the two conditions $h \circ abs = h'$ and $t \circ abs = abs \circ t'$. From this we can calculate h' and t' separately. First we calculate h':

$$\begin{aligned}&\ \ h'\ (xs, ys)\\ &=\quad \{\ \text{specification}\ \}\\ &\ \ h\ (abs\ (xs, ys))\\ &=\quad \{\ \text{definition of } abs\ \}\\ &\ \ h\ (xs +\!\!+ reverse\ ys))\\ &=\quad \{\ \text{definition of } h\ \}\\ &\ \ head\ (xs +\!\!+ reverse\ ys)\\ &=\quad \{\ xs \text{ is nonempty}\ \}\\ &\ \ head\ xs\end{aligned}$$

Now we calculate a definition for t'. Starting from the specification $abs \circ t' = t \circ abs$, we calculate as follows:

$$\begin{aligned}&\ \ t\ (abs\ (xs, ys))\\ &=\quad \{\ \text{definition of } abs\ \}\\ &\ \ t\ (xs +\!\!+ reverse\ ys))\\ &=\quad \{\ \text{case analysis}\ \}\\ &\ \ \textbf{case}\ xs\ \textbf{of}\\ &\qquad [x] \qquad\ \to t\ ([x] \qquad +\!\!+ reverse\ ys)\\ &\qquad (x : xs') \to t\ ((x : xs') +\!\!+ reverse\ ys)\\ &=\quad \{\ \text{definition of } +\!\!+\ \}\\ &\ \ \textbf{case}\ xs\ \textbf{of}\\ &\qquad [x] \qquad\ \to t\ (x : ([\,]\ +\!\!+ reverse\ ys))\\ &\qquad (x : xs') \to t\ (x : (xs' +\!\!+ reverse\ ys))\\ &=\quad \{\ \text{definition of } t\ \}\\ &\ \ \textbf{case}\ xs\ \textbf{of}\\ &\qquad [x] \qquad\ \to [\,]\ +\!\!+ reverse\ ys +\!\!+ [x]\\ &\qquad (x : xs') \to xs' +\!\!+ reverse\ ys +\!\!+ [x]\\ &=\quad \{\ \text{definition of } reverse, +\!\!+\ \}\\ &\ \ \textbf{case}\ xs\ \textbf{of}\\ &\qquad [x] \qquad\ \to \qquad reverse\ (x : ys)\\ &\qquad (x : xs') \to xs' +\!\!+ reverse\ (x : ys)\\ &=\quad \{\ \text{definition of } abs\ \}\\ &\ \ \textbf{case}\ xs\ \textbf{of}\\ &\qquad [x] \qquad\ \to abs\ (reverse\ (x : ys), \qquad [\,])\\ &\qquad (x : xs') \to abs\ (xs' \qquad\qquad\quad , x : ys)\\ &=\quad \{\ \text{pulling } abs \text{ out of cases}\ \}\\ &\ \ abs\ (\textbf{case}\ xs\ \textbf{of}\\ &\qquad [x] \qquad\ \to (reverse\ (x : ys) \quad , \qquad [\,])\\ &\qquad (x : xs') \to (xs' \qquad\qquad\quad , x : ys))\end{aligned}$$

Hence, t' can be defined as follows:

$$\begin{aligned}&t'\ ([x], ys) \quad = (reverse\ (x : ys), [\,])\\ &t'\ (x : xs, ys) = (xs, x : ys)\end{aligned}$$

In conclusion, by applying our worker-wrapper theorem, we have calculated a factorised version of $cycle$

$$\begin{aligned}&cycle = \text{unfold}\ h'\ t' \circ rep\\ &\quad \textbf{where}\ h'\ (xs, ys) \quad = head\ xs\\ &\qquad\quad\ \ t'\ ([x], ys) \quad = (reverse\ (x : ys), [\,])\\ &\qquad\quad\ \ t'\ (x : xs, ys) = (xs, x : ys)\end{aligned}$$

which can be written directly as

$$\begin{aligned}&cycle = cycle' \circ rep\\ &\quad \textbf{where}\\ &\qquad cycle'\ ([x], ys) \quad = x : cycle'\ (reverse\ (x : ys), [\,])\\ &\qquad cycle'\ (x : xs, ys) = x : cycle'\ (xs, x : ys)\end{aligned}$$

This version only performs a $reverse$ operation once for every cycle of the input list, so the average cost to produce a single element is now constant. We believe that this kind of optimisation — in which costly operations are delayed and combined into a single operation — will be a common use of our theory.

6.3 Example: Initial Segment of a List

This example is particularly interesting as it does not require the function being optimised to be explicitly written as an unfold at any point. As the function in question is partial, the relevant category in this case is **CPO** rather than **Set**.

The function $init$ takes a list and returns the list consisting of all the elements of the original list except the last one:

$$\begin{aligned}&init :: [a] \to [a]\\ &init\ [\,] \qquad = \bot\end{aligned}$$

$$
\begin{aligned}
&init\ [x] &&= [\,]\\
&init\ (x\ :\ xs) &&= x\ :\ init\ xs
\end{aligned}
$$

Here, $\bot$ represents the failure of the function to produce a result. (In Haskell we would not need to give this first case, but we make it explicit here for the purposes of reasoning.)

Each call of $init$ checks to see if the argument is empty. However, the argument of the recursive call can never be empty, as if it were then the second case would have been matched rather than the third. We would therefore like to perform this check only once. We can use unfold worker-wrapper to achieve this, by essentially using a "de-consed" list as our representation:

$$
\begin{aligned}
&rep :: [a] \to (a, [a])\\
&rep\ [\,] &&= \bot\\
&rep\ (x\ :\ xs) &&= (x, xs)
\end{aligned}
$$

$$
\begin{aligned}
&abs :: (a, [a]) \to [a]\\
&abs\ (x, xs) &&= x\ :\ xs
\end{aligned}
$$

In this case, assumption (A) fails:

$$
\begin{aligned}
&\quad abs\ (rep\ [\,])\\
&= \quad \{\ \text{definition of } rep\ \}\\
&\quad abs\ \bot\\
&= \quad \{\ abs \text{ is strict}\ \}\\
&\quad \bot\\
&\neq [\,]
\end{aligned}
$$

If we were to rewrite $init$ as an unfold, we could then prove (B). However, we instead avoid this by using the alternative assumption (C'). This expands to $init \circ abs \circ rep = init$, which we prove by case analysis on the argument. For the empty list:

$$
\begin{aligned}
&\quad init\ (abs\ (rep\ [\,]))\\
&= \quad \{\ \text{definition of } rep\ \}\\
&\quad init\ (abs\ \bot)\\
&= \quad \{\ abs \text{ is strict}\ \}\\
&\quad init\ \bot\\
&= \quad \{\ init \text{ is strict}\ \}\\
&\quad \bot\\
&= \quad \{\ \text{definition of } init\ \}\\
&\quad init\ [\,]
\end{aligned}
$$

For the undefined value $\bot$:

$$
\begin{aligned}
&\quad init\ (abs\ (rep\ \bot)\\
&= \quad \{\ init, abs \text{ and } rep \text{ are all strict}\ \}\\
&\quad \bot
\end{aligned}
$$

Otherwise:

$$
\begin{aligned}
&\quad init\ (abs\ (rep\ (x\ :\ xs)))\\
&= \quad \{\ \text{definition of } rep\ \}\\
&\quad init\ (abs\ (x, xs))\\
&= \quad \{\ \text{definition of } abs\ \}\\
&\quad init\ (x\ :\ xs)
\end{aligned}
$$

Using Condition (2β)

Firstly, we demonstrate a derivivation of a new worker function that avoids writing $init$ explicitly in terms of unfold. The only condition that permits this is (2β), which expands to $init' = init \circ abs$. We can calculate the definition of $init'$ simply by applying this specification to (x, xs):

$$
\begin{aligned}
&\quad init'\ (x, xs)\\
&= \quad \{\ \text{specification of } init'\ \}\\
&\quad init\ (abs\ (x, xs))\\
&= \quad \{\ \text{definition of } abs\ \}\\
&\quad init\ (x\ :\ xs)\\
&= \quad \{\ \text{definition of } init\ \}\\
&\quad \mathbf{case}\ (x\ :\ xs)\ \mathbf{of}\\
&\qquad [\,] \to \bot\\
&\qquad [y] \to [\,]\\
&\qquad (y\ :\ ys) \to y\ :\ init\ ys\\
&= \quad \{\ \text{removing redundant case}\ \}\\
&\quad \mathbf{case}\ (x\ :\ xs)\ \mathbf{of}\\
&\qquad [y] \to [\,]\\
&\qquad (y\ :\ ys) \to y\ :\ init\ ys\\
&= \quad \{\ \text{factoring out } x\ \}\\
&\quad \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \to [\,]\\
&\qquad ys \to x\ :\ init\ ys\\
&= \quad \{\ ys \text{ is nonempty}\ \}\\
&\quad \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \to [\,]\\
&\qquad (y\ :\ ys') \to x\ :\ init\ (y\ :\ ys')\\
&= \quad \{\ \text{definition of } abs\ \}\\
&\quad \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \to [\,]\\
&\qquad (y\ :\ ys') \to x\ :\ init\ (abs\ (y, ys'))\\
&= \quad \{\ \text{specification of } init'\ \}\\
&\quad \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \to [\,]\\
&\qquad (y\ :\ ys') \to x\ :\ init'\ (y, ys')
\end{aligned}
$$

As we used the specification of $init'$ in its own derivation, the above derivation only guarantees partial correctness. We should take a moment to convince ourselves that the resulting definition does actually satisfy the specification. In this case, both sides are clearly total, so there is no problem. In conclusion, we have derived an alternative definition for the function $init$

$$
\begin{aligned}
&init = init' \circ rep\\
&\quad \mathbf{where}\\
&\qquad rep\ \ [\,] &&= \bot\\
&\qquad rep\ \ (x\ :\ xs) &&= (x, xs)\\
&\qquad init'\ (x, [\,]) &&= [\,]\\
&\qquad init'\ (x, y\ :\ ys) &&= x\ :\ init'\ (y, ys)
\end{aligned}
$$

that only performs the check for the empty list once.

Note that we derived the more efficient version of $init$ using worker-wrapper factorisation for unfolds without ever writing the program in question as an unfold. This is a compelling argument for the flexibility of the theory, but we should also note that because of our choice of assumption and condition none of the extra structure of unfolds is needed, as the equality chain

$$
\begin{aligned}
&\quad init' \circ rep\\
&= \quad \{\ (2\beta)\ \}\\
&\quad init \circ abs \circ rep\\
&= \quad \{\ \text{(C')}\ \}\\
&\quad init
\end{aligned}
$$

holds regardless of whether $init$ and $init'$ are unfolds. In a sense, because we have chosen the weakest properties, the theory gives us less. This suggests that we should generally prefer to use stronger properties if possible. The use of partially-correct reasoning is also unsatisfactory, as it requires us to appeal to totality.

Using Condition (1)

If we write $init$ explicitly as an unfold, this example can also use condition (1). The unfold for (possibly finite) lists is:

$$
\begin{array}{l}
\text{unfold} :: (a \to \mathit{Maybe}\ (b, a)) \to a \to [b] \\
\text{unfold}\ f\ x = \textbf{case}\ f\ x\ \textbf{of} \\
\quad \mathit{Nothing} \quad \to [\,] \\
\quad \mathit{Just}\ (b, x') \to b\ :\ \text{unfold}\ f\ x'
\end{array}
$$

Using this, we can define init as follows:

$$
\begin{array}{l}
\mathit{init} :: [a] \to [a] \\
\mathit{init} = \text{unfold}\ f \\
\quad \textbf{where} \\
\quad\quad f :: [a] \to \mathit{Maybe}\ (a, [a]) \\
\quad\quad f\ [\,] \quad\quad = \bot \\
\quad\quad f\ [x] \quad\quad = \mathit{Nothing} \\
\quad\quad f\ (x\ :\ xs) = \mathit{Just}\ (x, xs)
\end{array}
$$

In order to construct a function g such that $\mathit{init} = \text{unfold}\ g \circ \mathit{rep}$, where rep is defined as before, we use worker-wrapper factorisation. Condition (1) gives us the explicit definition $g = \mathsf{F}\ \mathit{rep} \circ f \circ \mathit{abs}$. Instantiating this for our particular F, we have:

$$
\begin{array}{l}
g\ x = \textbf{case}\ f\ (\mathit{abs}\ x)\ \textbf{of} \\
\quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
\quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x')
\end{array}
$$

We attempt to simplify this, noting that the type of x is $(a, [a])$. We calculate g separately for the input $(a, [\,])$

$$
\begin{array}{ll}
 & g\ (a, [\,]) \\
= & \quad \{\ \text{definition of } g\ \} \\
 & \textbf{case}\ f\ (\mathit{abs}\ (a, [\,]))\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ \text{definition of } \mathit{abs}\ \} \\
 & \textbf{case}\ f\ [a]\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ f\ [a] = \mathit{Nothing}, \text{cases}\ \} \\
 & \mathit{Nothing}
\end{array}
$$

and for (a, as), where as is non-empty:

$$
\begin{array}{ll}
 & g\ (a, as) \\
= & \quad \{\ \text{definition of } g\ \} \\
 & \textbf{case}\ f\ (\mathit{abs}\ (a, as))\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ \text{definition of } \mathit{abs}\ \} \\
 & \textbf{case}\ f\ (a\ :\ as)\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ f\ (a\ :\ as) = \mathit{Just}\ (a, as)\ \text{when } as\ \text{nonempty, cases}\ \} \\
 & \mathit{Just}\ (a, \mathit{rep}\ as) \\
= & \quad \{\ as\ \text{is nonempty, let } as = a'\ :\ as'\ \} \\
 & \mathit{Just}\ (a, \mathit{rep}\ (a'\ :\ as')) \\
= & \quad \{\ \text{definition of } \mathit{rep}\ \} \\
 & \mathit{Just}\ (a, (a', as'))
\end{array}
$$

We thus obtain the following definition of g:

$$
\begin{array}{l}
g\ (a, [\,]) \quad\quad = \mathit{Nothing} \\
g\ (a, a'\ :\ as) = \mathit{Just}\ (a, (a', as))
\end{array}
$$

Because we are working in **CPO**, we must also consider the behaviour of the function g on the input $(a, \bot)$:

$$
\begin{array}{ll}
 & g\ (a, \bot) \\
= & \quad \{\ \text{specification of } g\ \} \\
 & \textbf{case}\ f\ (\mathit{abs}\ (a, \bot))\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ \text{definition of } \mathit{abs}\ \} \\
 & \textbf{case}\ f\ (a\ :\ \bot)\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ f\ \text{cannot pattern match on } a\ :\ \bot\ \} \\
 & \textbf{case}\ \bot\ \textbf{of} \\
 & \quad \mathit{Nothing} \quad \to \mathit{Nothing} \\
 & \quad \mathit{Just}\ (y, x') \to \mathit{Just}\ (y, \mathit{rep}\ x') \\
= & \quad \{\ \text{case exhaustion}\ \} \\
 & \bot
\end{array}
$$

However, because g is defined by pattern matching on the second component of the tuple, the equation $g\ (a, \bot) = \bot$ clearly holds. Therefore, the above definition of g satisfies worker-wrapper condition (1) and so the factorisation

$$\mathit{init} = \text{unfold}\ g \circ \mathit{rep}$$

is correct. Note that unfold g is precisely the init' function that we derived above, now defined as an unfold.

While we had to write init explicitly as an unfold to perform this derivation, the calculation of the improved program was more straightforward than before, and largely mechanical.

Remarks

This above example shows that the same optimised function can sometimes be obtained using different approaches. It is worth noting that this particular optimisation is also an instance of call-pattern specialisation [23], as implemented in the Glasgow Haskell Compiler. However, neither one of these approaches subsumes the other, it simply happens in this case that they coincide.

6.4 Example: Flattening a Tree

Our final example concerns the left-to-right traversal of a binary tree. The naïve way to implement such a traversal is as follows, which corresponds to expressing the function as a fold:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\ a = \mathit{Null} \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
\\
\mathit{flatten} :: \mathit{Tree}\ a \to [a] \\
\mathit{flatten}\ \mathit{Null} \quad\quad\quad\quad = [\,] \\
\mathit{flatten}\ (\mathit{Fork}\ t1\ x\ t2) = \mathit{flatten}\ t1 +\!\!+ [x] +\!\!+ \mathit{flatten}\ t2
\end{array}
$$

However, this approach takes time quadratic in the number of nodes. Alternatively, we can write the function as an unfold, removing the leftmost element of the tree each time:

$$
\begin{array}{l}
\mathit{flatten} :: \mathit{Tree}\ a \to [a] \\
\mathit{flatten} = \text{unfold}\ \mathit{removemin} \\
\quad \textbf{where}\ \mathit{removemin}\ \mathit{Null} = \mathit{Nothing} \\
\quad\quad\quad\quad \mathit{removemin}\ (\mathit{Fork}\ t1\ x\ t2) = \\
\quad\quad\quad\quad\quad \textbf{case}\ \mathit{removemin}\ t1\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad \mathit{Nothing} \quad\quad \to \mathit{Just}\ (x, t2) \\
\quad\quad\quad\quad\quad\quad \mathit{Just}\ (y, t1') \to \mathit{Just}\ (y, \mathit{Fork}\ t1'\ x\ t2)
\end{array}
$$

This approach takes time proportional to $n * l$, where n is the number of nodes and l is the leftwards depth of the tree, i.e. the depth of the deepest node counting only left branches.

However, we can use our worker-wrapper theory to improve this further, exploiting the isomorphism between lists of rose trees (trees with an arbitrary number of children for each node) and binary trees. First we define the type of rose trees:

$$\textbf{data}\ \mathit{RoseTree}\ a = \mathit{RoseTree}\ a\ [\mathit{RoseTree}\ a]$$

Now we define a version of flatten for lists of rose trees, in which the list acts as a priority queue of the elements in the original tree,

so that at each stage we remove the root of the first tree in the queue, and push all its children onto the front of the queue:

```
flatten' :: [RoseTree a] → [a]
flatten' = unfold g
  where g [] = Nothing
        g (RoseTree x ts1 : ts2) = Just (x, ts1 ++ ts2)
```

We define *rep* and *abs* functions to convert between this priority queue representation and the original binary tree representation.

```
rep :: Tree a → [RoseTree a]
rep = reverse ∘ listify
  where listify Null = []
        listify (Fork t1 x t2) =
          RoseTree x (rep t2) : listify t1

abs :: [RoseTree a] → Tree a
abs = delistify ∘ reverse
  where delistify [] = Null
        delistify (RoseTree x ts1 : ts2) =
          Fork (delistify ts2) x (abs ts1)
```

Essentially, *rep* pulls apart a tree along its left branch into a list, while *abs* puts the tree back together. The use of *reverse* is necessary to ensure that the leftmost node is at the head of the list.

The result is an alternate definition $flatten = flatten' \circ rep$ that has comparable performance on balanced trees, but much better performance on trees with long left branches.

We now verify the correctness of this new definition using our worker-wrapper theorem. Firstly, we show that assumption (A) holds, i.e. $abs \circ rep = id$. To do this we note that because *reverse* is self-inverse, $abs \circ rep = delistify \circ listify$. We prove $delistify \circ listify = id$ by induction on trees. First, the base case:

```
  delistify (listify Null)
=   { definition of listify }
  delistify []
=   { definition of delistify }
  Null
```

Then the inductive case:

```
  delistify (listify (Fork t1 x t2))
=   { definition of listify }
  delistify (RoseTree x (rep t2) : listify t1)
=   { definition of delistify }
  Fork (delistify (listify t1)) x (abs (rep t2))
=   { abs ∘ rep = delistify ∘ listify }
  Fork (delistify (listify t1)) x (delistify (listify t2))
=   { inductive hypothesis }
  Fork t1 x t2
```

Now we must prove that g satisfies one of the worker-wrapper specifications. We choose condition (2) to verify, in this case:

```
removemin (abs ts) =
  case g ts of
    Nothing      → Nothing
    Just (x, ts') → Just (x, abs ts')
```

We verify this equation by induction on the length of the priority queue. For the base case when the queue is empty, we have:

```
  removemin (abs [])
=   { definition of abs }
  removemin Null
=   { definition of removemin }
  Nothing
=   { g [] = Nothing }
  case g [] of
    Nothing      → Nothing
    Just (x, ts') → Just (x, abs ts')
```

For the inductive case, rather than $RoseTree\ x\ ts1\ :\ ts2$ we use $ts1 \mathbin{+\!\!+} [RoseTree\ x\ ts2]$, as *abs* reverses its argument:

```
  removemin (abs (ts1 ++ [RoseTree x ts2]))
=   { definition of abs }
  removemin (delistify
    (RoseTree x ts2 : reverse ts1))
=   { definition of delistify }
  removemin (Fork (delistify (reverse ts1))
                  x
                  (abs ts2))
=   { abs = delistify ∘ reverse }
  removemin (Fork (abs ts1) x (abs ts2))
=   { definition of removemin }
  case removemin (abs ts1) of
    Nothing      → Just (x, abs ts2)
    Just (y, t1') →
      Just (y, Fork t1' x (abs ts2))
=   { inductive hypothesis }
  case
    case (g ts1) of
      Nothing      → Nothing
      Just (y, ts1') → Just (y, abs ts1')
  of
    Nothing      → Just (x, abs ts2)
    Just (y, t1') →
      Just (y, Fork t1' x (abs ts2))
=   { case of case, pattern matching }
  case g ts1 of
    Nothing      → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, Fork (abs ts1') x (abs ts2))
=   { definition of abs }
  case g ts1 of
    Nothing      → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, Fork (delistify (reverse ts1'))
                    x
                    (abs ts2))
=   { definition of abs }
  case g ts1 of
    Nothing      → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, abs (ts1' ++ [RoseTree x ts2]))
```

We must prove that this last expression is equal to

```
case g (ts1 ++ [RoseTree x ts2]) of
  Nothing      → Nothing
  Just (y, ts') → Just (y, abs ts')
```

which we do by case analysis on ts1. When $ts1 = []$, we have:

```
  case g [] of
    Nothing      → Just (x, abs ts2)
    Just (y, ts1') →
      Just (y, abs (ts1' ++ [RoseTree x ts2]))
=   { definition of g, case }
  Just (x, abs ts2)
=   { case }
  case Just (x, ts2) of
```

$$
\begin{array}{ll}
& \quad Nothing \quad \to Nothing \\
& \quad Just\ (y, ts') \ \to Just\ (y, abs\ ts') \\
= & \quad \{ \text{ definition of } g \ \} \\
& \textbf{case}\ g\ [RoseTree\ x\ ts2]\ \textbf{of} \\
& \quad Nothing \quad \to Nothing \\
& \quad Just\ (y, ts') \ \to Just\ (y, abs\ ts') \\
= & \quad \{ \ [] \text{ identity of } +\!\!+ \ \} \\
& \textbf{case}\ g\ ([] +\!\!+ [RoseTree\ x\ ts2])\ \textbf{of} \\
& \quad Nothing \quad \to Nothing \\
& \quad Just\ (y, ts') \ \to Just\ (y, abs\ ts')
\end{array}
$$

In turn, when $ts1 = RoseTree\ z\ ts3\ :\ ts4$:

$$
\begin{array}{ll}
& \textbf{case}\ g\ (RoseTree\ z\ ts3\ :\ ts4)\ \textbf{of} \\
& \quad Nothing \quad \to Just\ (x, abs\ ts2) \\
& \quad Just\ (y, ts1') \to \\
& \qquad Just\ (y, abs\ (ts1' +\!\!+ [RoseTree\ x\ ts2])) \\
= & \quad \{ \text{ definition of } g, \text{ case } \} \\
& Just\ (z, abs\ (ts3 +\!\!+ ts4 +\!\!+ [RoseTree\ x\ ts2])) \\
= & \quad \{ \text{ case } \} \\
& \textbf{case}\ Just\ (z, ts3 +\!\!+ ts4 +\!\!+ [RoseTree\ x\ ts2])\ \textbf{of} \\
& \quad Nothing \quad \to Nothing \\
& \quad Just\ (y, ts') \ \to Just\ (y, abs\ ts') \\
= & \quad \{ \text{ definition of } g \ \} \\
& \textbf{case}\ g\ (RoseTree\ z\ ts3\ :\ (ts4 +\!\!+ [RoseTree\ x\ ts2]))\ \textbf{of} \\
& \quad Nothing \quad \to Nothing \\
& \quad Just\ (y, ts') \ \to Just\ (y, abs\ ts')
\end{array}
$$

Therefore, condition (2) and assumption (A) are satisfied, and hence the following worker-wrapper factorisation is valid:

$$flatten = flatten' \circ rep$$

We conclude by noting that while this result can be obtained from fusion alone, the necessary proof is very involved, requiring a lemma about the relationship between $removemin$ and abs. The worker-wrapper proof, while long, is mechanical. For comparison, the fusion-based proof is available on the web at `http://www.cs.nott.ac.uk/~jph/flatten_fusion.pdf`.

7. Related Work

We have divided the related work into four categories. The first two relate to the history of the unfold operator in programming languages and category theory respectively. The third relates to the use of fusion in program optimisation, while the fourth relates to applications of program factorisation.

7.1 Unfold in Programming Languages

The use of unfold in programming is a lot more recent than that of fold. While fold-like operations trace their history back to APL [16], the earliest unfold-like mechanism appears to be in Miranda list comprehensions [27], which have a special ". ." syntax that can be used to express unfold-like computations without the need for explicit recursion. However, in Miranda there was no dedicated unfold operator such as the one in Haskell, which became part of the standard library in Haskell 98 [22].

The unfold operator seems to first appear in a recognisable form in 1988 in *Introduction to Functional Programming* by Bird and Wadler [1], where it is defined in terms of map, $takeWhile$ and $iterate$. No direct recursive definition is given, and it only appears on a single page. Meijer, Fokkinga and Paterson noted in 1991 that the unfold operator from [1] was categorically dual to fold [18].

In 1998, Gibbons and Jones published the paper *The Under-Appreciated Unfold* [9], which gave the inspiration for the title of this paper. The paper argued that unfold was an underutilised programming tool, and justified this by presenting algorithms for breadth-first traversal using both fold and unfold, arguing that the unfold-based algorithms were clearer.

7.2 Unfold in Category Theory

It seems that categorical unfolds (also known as anamorphisms) were largely developed in parallel to unfold in programming languages. The earliest mention of categorical unfold appears to be in Hagino's 1987 PhD thesis *A Categorical Programming Language* [13] and subsequently in Malcolm's 1990 thesis *Algebraic Data Types and Program Transformation* [17]. Interestingly, neither of these make any linguistic distinction between folds and unfolds, using the same terminology to refer to both. It seems that this distinction was not made until Meijer et al.'s 1991 paper *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* [18], which concerned folds and unfolds in **CPO** and essentially unified the programming language work with the categorical work.

7.3 Program Fusion

In functional programming, many successful program optimisations are based upon fusion, in which separate parts of a program are fused together to eliminate intermediate data structures. Fusion was first introduced by Wadler in 1990 by the name of *deforestation* [31]. Since then, it has been widely explored, especially in regards to specific recursion patterns.

A particularly successful example is $foldr/build$ fusion, in which list-producers are defined in terms of a function $build$ while list-consumers are defined in terms of $foldr$. Introduced by Gill, Launchbury and Peyton Jones [11], this pattern has been the subject of much research [2, 26, 30] and is implemented in GHC.

A more recent innovation by Coutts, Leshchinsky and Stewart is *stream fusion*, a way to optimise list-processing functions by changing the representation of lists [4]. This approach makes essential use of an unfold-like operator, and is related to worker-wrapper factorisation as it involves a change of intermediate data type. However, stream fusion utilises the recursive structure of the underlying data, whereas our approach does not.

7.4 Program Factorisation

Compared to fusion-based techniques, program factorisation or "fission" seems far less well-explored. Gibbons' 2006 paper, *Fission for Program Comprehension* [7], presents an application of this idea which differs from our work in two ways. Firstly, Gibbons does not concern himself with program optimisation; rather, his intended use is understanding an already-written program by breaking it down into the combination of separate parts that are easier to comprehend. Secondly, while Gibbons' approach is based upon applying fusion in reverse, the proof of our approach to program factorisation actually involves a forward application of fusion.

8. Conclusion

In this paper, we have presented a novel approach to optimising programs written in the form of an unfold, showing how a useful approach comes from the commonly-used idea of generalising a coinductive hypothesis. We have provided a general factorisation theorem that can be used either to guide the derivation of an optimised program or to prove such a program correct, resulting in a technique that we believe has wide applicability. We demonstrated the utility of our technique with a collection of examples.

8.1 Further Work

We have only considered programs in the form of an unfold, but there are other corecursive patterns that can be considered. One example is *apomorphisms* [29], which capture the idea of *primitive corecursion*, allowing construction of the result to short-cut by

producing the remainder of the result in a single step. The apomorphism operator for streams can be defined as follows:

$$\begin{aligned} apo\ h\ t\ x = h\ x\ :\ &\mathbf{case}\ t\ x\ \mathbf{of} \\ &\quad Left\ x' \ \ \rightarrow apo\ h\ t\ x' \\ &\quad Right\ xs \rightarrow xs \end{aligned}$$

Apomorphisms have their own fusion law, so it seems likely that they would have a useful worker-wrapper factorisation theorem.

Another possible direction is to adapt our method to deal with circular definitions. For example, we can write a circular definition of the infinite stream of Fibonacci numbers:

$$fibs = 0\ :\ 1\ :\ zipWith\ (+)\ fibs\ (tail\ fibs)$$

Coinductive techniques can be used to reason about circular definitions like this one, but whether a factorisation theorem analogous to ours exists for such definitions remains to be seen.

We could also consider extending this work to monadic and comonadic unfolds. Monadic unfolds [21] are of particular interest; consider the monadic unfold operator for streams:

$$\begin{aligned} \mathrm{unfoldM} :: Monad\ m \Rightarrow &(a \rightarrow m\ (b, a)) \rightarrow \\ &a \rightarrow m\ (Stream\ b) \\ \mathrm{unfoldM}\ f\ x = \mathbf{do}\ &(b, x') \leftarrow f\ x \\ &bs \leftarrow \mathrm{unfoldM}\ f\ x' \\ &return\ (b\ :\ bs) \end{aligned}$$

Any monad with a strict bind operator will fail to produce a result, instead simply bottoming out. Intuitively, the problem is that an infinite number of effects must be applied before the final result can be produced. Thus we see that there is a fundamental difference between ordinary and monadic unfolds that raises interesting questions concerning the worker-wrapper theory.

The theory we have presented is concerned with correctness. In order to reason about efficiency gains, we also need an operational theory, for which purposes we are currently exploring the use of improvement theory [19]. Finally, while we have noted that the proofs are often mechanical, we have yet to consider how our new theory may be *mechanised*. A team at the University of Kansas is currently working on the implementation of various worker-wrapper theories as an extension to the Glasgow Haskell Compiler [5, 25], with promising initial results.

Acknowledgments

The authors would like to thank Richard Bird for assistance with tracing the history of the unfold operator.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. 1988.

[2] O. Chitil. Type Inference Builds a Short Cut to Deforestation. In *ICFP '99*. ACM, 1999.

[3] T. Coquand. Infinite Objects in Type Theory. In *TYPES '93*, volume 806 of *Lecture Notes in Computer Science*. Springer, 1993.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *ICFP '07*. ACM, 2007.

[5] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Haskell Symposium (Haskell '12)*. ACM, 2012.

[6] P. J. Freyd. Remarks on Algebraically Compact Categories. In *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1992.

[7] J. Gibbons. Fission for Program Comprehension. In *MPC '06*, volume 4014 of *Lecture Notes in Computer Science*. Springer, 2006.

[8] J. Gibbons and G. Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4), April-May 2005.

[9] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *ICFP '98*. ACM, 1998.

[10] A. Gill and G. Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2), Mar. 2009.

[11] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *FPCA '93*. Springer, 1993.

[12] A. D. Gordon. A Tutorial on Co-induction and Functional Programming. In *In Glasgow Functional Programming Workshop*. Springer, 1994.

[13] T. Hagino. *A Categorical Programming Language*. PhD thesis, Department of Computer Science, University of Edinburgh, 1987.

[14] R. Hinze. Functional Pearl: Streams and Unique Fixed Points. In *ICFP '08*, New York, NY, USA, 2008.

[15] G. Hutton, M. Jaskelioff, and A. Gill. Factorising Folds for Faster Functions. *Journal of Functional Programming Special Issue on Generic Programming*, 20(3&4), June 2010.

[16] K. E. Iverson. *A Programming Language*. Wiley, 1962.

[17] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, 1990.

[18] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA '91*, volume 523 of *Lecture Notes in Computer Science*. Springer, 1991.

[19] A. Moran and D. Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *POPL '99*, pages 43–56. ACM, 1999.

[20] L. S. Moss and N. Danner. On the Foundations of Corecursion. *Logic Journal of the IGPL*, 5(2), 1997.

[21] A. Pardo. Monadic Corecursion — Definition, Fusion Laws, and Applications. *Electr. Notes Theor. Comput. Sci.*, 11, 1998.

[22] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[23] S. Peyton Jones. Call-Pattern Specialisation for Haskell Programs. In *ICFP '07*. ACM, 2007.

[24] N. Sculthorpe and G. Hutton. Work It, Wrap It, Fix It, Fold It. Submitted to the Journal of Functional Programming, 2013.

[25] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. In *Draft Proceedings of Implementation and Application of Functional Languages (IFL '12)*, 2012.

[26] A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. In *FPCA '95*. Springer, 1995.

[27] D. A. Turner. *Miranda System Manual*. Research Software Ltd., Canterbury, England, 1989. Available online at http://miranda.org.uk/.

[28] D. A. Turner. Elementary Strong Functional Programming. In *FPLE '95*, volume 1022 of *Lecture Notes in Computer Science*. Springer, 1995.

[29] T. Uustalu and V. Vene. Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically. *Informatica, Lith. Acad. Sci.*, 10 (1), 1999.

[30] J. Voigtländer. Proving Correctness via Free Theorems: The Case of the destroy/build-Rule. In *PEPM '08*. ACM, 2008.

[31] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.*, 73(2), 1990.

Work It, Wrap It, Fix It, Fold It

NEIL SCULTHORPE
University of Kansas, USA

GRAHAM HUTTON
University of Nottingham, UK

Abstract

The worker/wrapper transformation is a general-purpose technique for refactoring recursive programs to improve their performance. The two previous approaches to formalising the technique were based upon different recursion operators and different correctness conditions. In this article we show how these two approaches can be generalised in a uniform manner by combining their correctness conditions, how the theory can be extended with new conditions that are necessary (in addition to sufficient) to ensure the correctness of the worker/wrapper technique, and explore the benefits that result. All the proofs have been mechanically verified using the Agda system.

1 Introduction

A fundamental objective in computer science is the development of programs that are clear, efficient and correct. However, these aims are often in conflict. In particular, programs that are written for clarity may not be efficient, while programs that are written for efficiency may be difficult to comprehend and contain subtle bugs. The goal of *program transformation* is to resolve these tensions by systematically rewriting programs to improve their efficiency, without compromising their correctness.

The focus of this article is the *worker/wrapper transformation*, a transformation technique for improving the performance of recursive programs by improving the choice of data structures used. The basic idea is simple and general: given a recursive program of some type A, we aim to factorise it into a more efficient *worker* program of some other type B, together with a *wrapper* function of type $B \to A$ that allows the new worker to be used in the same context as the original program.

Special cases of the worker/wrapper transformation have been used for many years, particularly in the implementation of optimising compilers. For example, the technique has played a key role in the Glasgow Haskell Compiler (GHC) since its inception more than twenty years ago, to replace the use of boxed data structures by more efficient unboxed data structures when safe to do so (Peyton Jones & Launchbury, 1991). However, it is only recently — in two articles that lay the foundations for the present work (Gill & Hutton, 2009; Hutton *et al*., 2010) — that the worker/wrapper transformation has been formalised, and considered as a general approach to program optimisation.

The original formalisation (2009) was based upon a least-fixed-point semantics of recursive programs. Within this setting the worker/wrapper transformation was explained and

formalised, proved correct, and a range of different programming applications presented. Three key benefits of the technique were also identified:

- It provides a general and systematic approach to transforming a recursive program of one type into an equivalent program of another type.
- It is straightforward to understand and apply, requiring only basic equational reasoning techniques, and often avoiding the need for induction.
- It allows many optimisation techniques that at first sight may seem to be unrelated to be captured within a single unified framework.

Using fixed points allowed the worker/wrapper transformation to be formalised, but did not take advantage of the additional structure that is present in many recursive programs. To this end, a more structured approach (2010) was then developed based upon initial-algebra semantics, a categorical approach to recursion that is widely used in program optimisation (Bird & de Moor, 1997). More specifically, a worker/wrapper theory was developed for programs defined using *fold* operators, which encapsulate a common pattern of recursive programming. In practice, using fold operators results in simpler transformations than the approach based upon fixed points. Moreover, it also admitted the first formal proof of correctness of a new technique for optimising monadic programs.

While the two previous articles were nominally about the same technique, they were quite different in their categorical foundations and correctness conditions. The first was founded upon least fixed points in the category **CPO** of complete partial orders and continuous functions, and identified a hierarchy of conditions on the conversion functions between the original and worker types that are sufficient to ensure correctness. In contrast, the second was founded upon initial algebras in an arbitrary category $\mathbb{C}$, and identified a lattice of sufficient correctness conditions on the original and worker algebras. This raises the question of whether it is possible to combine or unify the two different approaches. The purpose of this new article is to show how this can be achieved, and to explore the benefits that result. More precisely, the article makes the following contributions:

- We show how the least-fixed-point and initial-algebra approaches to the worker/wrapper transformation can be generalised in a uniform manner by combining the different sets of correctness conditions from the two approaches.
- We extend the theory with new conditions that are necessary (in addition to sufficient) to guarantee the correctness of the worker/wrapper technique, thereby ensuring that the theory is as widely applicable as possible.
- We develop a specialised worker/wrapper theory for fold operators in the category **CPO** with fewer strictness conditions than we obtain by instantiating the general theory for initial algebras, by instantiating the general theory for least fixed points.

The article is aimed at readers who are familiar with the basics of least-fixed-point semantics (Schmidt, 1986), initial-algebra semantics (Bird & de Moor, 1997), and the worker/wrapper transformation (Gill & Hutton, 2009; Hutton *et al*., 2010), but all necessary concepts and results are reviewed. An extended version of the article that includes a series of worked examples and all the proofs is available from the authors' web pages, along with a mechanical verification of the proofs in Agda.

2 Least-Fixed-Point Semantics

The original formalisation of the worker/wrapper transformation was based on a least-fixed-point semantics of recursion, in a domain-theoretic setting in which programs are continuous functions on complete partial orders. In this section we review some of the basic definitions and properties from this approach to program semantics, and introduce our notation. For further details, see for example (Schmidt, 1986).

A *complete partial order* (cpo) is a set with a partial-ordering $\sqsubseteq$, a least element $\bot$, and limits $\bigsqcup$ (least upper bounds) of all non-empty chains. In turn, a function f between cpos is *continuous* if it is monotonic and preserves the limit structure. If it also preserves the least element, i.e. $f\ \bot = \bot$, the function is *strict*. A *fixed point* of a function f is a value x for which $f\ x = x$. Kleene's well-known fixed-point theorem states that any continuous function f on a cpo has a least fixed point, denoted by $\mathit{fix}\, f$.

The basic proof technique for least fixed points is *fixed-point induction* (Winskel, 1993). Suppose that f is a continuous function on a cpo and that P is a *chain-complete* predicate on the same cpo, i.e. whenever the predicate holds for all elements in a non-empty chain then it also holds for the limit of the chain. Then fixed-point induction states that if the predicate holds for the least element of the cpo (the base case) and is preserved by the function f (the inductive case), then it also holds for $\mathit{fix}\, f$:

Lemma 2.1 (*Fixed-Point Induction*)
If P is chain-complete, then:

$$P\ \bot\ \wedge\ (\forall\, x.\ P\, x\ \Rightarrow\ P\ (f\ x))\ \ \Rightarrow\ \ P\ (\mathit{fix}\, f)$$

Fixed-point induction can be used to verify the well-known *fixed-point fusion* property (Meijer *et al*., 1991), which states that the application of a function to a *fix* can be re-expressed as a single *fix*, provided that the function is strict and satisfies a simple commutativity condition with respect to the *fix* arguments:

Lemma 2.2 (*Fixed-Point Fusion*)

$$f\ \circ\ g\ =\ h\ \circ\ f\ \wedge\ \mathrm{strict}\, f\ \ \Rightarrow\ \ f\ (\mathit{fix}\, g)\ =\ \mathit{fix}\, h$$

Finally, a key property of *fix* that was used in the original domain-theoretic formalisation of the worker/wrapper transformation is the *rolling rule* (Backhouse, 2002), which allows the first argument of a composition to be pulled outside a *fix*, resulting in the composition swapping the order of its arguments, or 'rolling over':

Lemma 2.3 (*Rolling Rule*)

$$\mathit{fix}\ (f\ \circ\ g)\ =\ f\ (\mathit{fix}\ (g\ \circ\ f))$$

3 Worker/Wrapper for Least Fixed Points

Within the domain-theoretic setting of the previous section, consider a recursive program defined as the least fixed point of a function $f : A \to A$ on some type A. Now consider a more efficient program that performs the same task, defined by first taking the least fixed

point of a function $g : B \to B$ on some other type B, and then converting the resulting value back to the original type by applying a function $abs : B \to A$. The equivalence between these two programs is captured by the following equation:

$$fix\, f \;=\; abs\, (fix\, g)$$

We call $fix\, f$ the original program, $fix\, g$ the *worker* program, abs the *wrapper* function, and the equation itself the *worker/wrapper factorisation* for least fixed points. We now turn our attention to identifying conditions to ensure that it holds.

3.1 Assumptions and Conditions

First, we require an additional conversion function $rep : A \to B$ from the original type to the new type. This function is not required to be an inverse of abs, but we do require one of the following *worker/wrapper assumptions* to hold:

(A) $abs \circ rep \;=\; id_A$

(B) $abs \circ rep \circ f \;=\; f$

(C) $fix\, (abs \circ rep \circ f) \;=\; fix\, f$

These assumptions form a hierarchy, with (A) $\Rightarrow$ (B) $\Rightarrow$ (C). Assumption (A) is the strongest and usually the easiest to verify, and states that abs is a left inverse of rep, which in the terminology of data representation means that the *abstract* type A can be faithfully represented by the *concrete* type B, hence the choice of the names abs and rep for the conversion functions between the two types. For some applications, however, assumption (A) may not be true in general, but only for values produced by the body function f of the original program, as captured by the weaker assumption (B), or we may also need to take the recursive context into account, as captured by (C).

Additionally, we require one of the following *worker/wrapper conditions* that relate the body functions f and g of the original and worker programs:

(1) $g \;=\; rep \circ f \circ abs$ $\qquad$ (1β) $fix\, g \;=\; fix\, (rep \circ f \circ abs)$

(2) $rep \circ f \;=\; g \circ rep \;\wedge$ strict rep $\qquad$ (2β) $fix\, g \;=\; rep\, (fix\, f)$

(3) $abs \circ g \;=\; f \circ abs$

In general, there is no relationship between the conditions in the first column, i.e. none implies any of the others, while the β conditions in the second column arise as weaker versions of the corresponding conditions in the first. The implications (1) $\Rightarrow$ (1β) and (2) $\Rightarrow$ (2β) follow immediately using extensionality and fixed-point fusion respectively, which in the latter case accounts for the strictness side condition in (2). We will return to the issue of strictness in Section 3.2. Furthermore, given assumption (C), it is straightforward to show that conditions (1β) and (2β) are in fact equivalent. Nonetheless, it is still useful to consider both conditions, as in some practical situations one may be simpler to use than the other. Note that attempting to weaken condition (3) in a similar manner gives $fix\, f \;=\; abs\, (fix\, g)$, which there is no merit in considering as this is precisely the worker/wrapper factorisation result that we wish to establish.

Given functions

$f : A \to A$
$g : B \to B$

for some types A and B, and conversion functions

$rep : A \to B$
$abs : B \to A$

then we have a set of *worker/wrapper assumptions*

(A) $abs \circ rep = id_A$
(B) $abs \circ rep \circ f = f$
(C) $fix\,(abs \circ rep \circ f) = fix\,f$

and a set of *worker/wrapper conditions*

(1) $g = rep \circ f \circ abs$
(2) $rep \circ f = g \circ rep \wedge \text{strict}\ rep$
(3) $abs \circ g = f \circ abs$

(1β) $fix\,g = fix\,(rep \circ f \circ abs)$
(2β) $fix\,g = rep\,(fix\,f)$

Provided that any of the assumptions hold and any of the conditions hold, then *worker/wrapper factorisation* is valid:

$fix\,f = abs\,(fix\,g)$

Furthermore, if any of the assumptions hold, and any of the conditions except (3) hold, then *worker/wrapper fusion* is valid:

$rep\,(abs\,(fix\,g)) = fix\,g$

Figure 1: Worker/wrapper transformation for least fixed points.

In terms of how the worker/wrapper conditions are used in practice, for some applications the worker program *fix g* will already be given, and our aim then is to *verify* that one of the conditions is satisfied. In such cases, we use the condition that admits the simplest verification, which is often one of the stronger conditions (1), (2) or (3) that do not involve the use of *fix*. For other applications, our aim will be to *construct* the worker program. In such cases, conditions (1), (1β) or (2β) provide explicit but inefficient definitions for the worker program in terms of the body function f of the original program, which we then attempt to make more efficient using program-fusion techniques. This was the approach that was taken by Gill & Hutton (2009). However, as shown by Hutton *et al*. (2010), in some cases it is preferable to use conditions (2) or (3), which provide a *specification* for the body function g of the worker, rather than a definition.

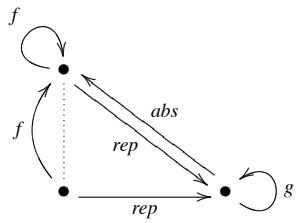
3.2 Worker/Wrapper Factorisation

We can now state the main result of this section: provided that any of the worker/wrapper assumptions hold, and any of the worker/wrapper conditions hold, then worker/wrapper factorisation is valid, as summarised in Figure 1. To prove this result it suffices to consider assumption (C) and conditions (1β) and (3) in turn, as (A), (B), (1) and (2) are already

covered by their weaker versions, and (2β) is equivalent to (1β) in the presence of (C). For condition (1β), factorisation is verified by the following simple calculation:

$$
\begin{array}{ll}
 & \mathit{fix}\, f \\
= & \qquad \{\ (\mathrm{C})\ \} \\
 & \mathit{fix}\, (\mathit{abs} \circ \mathit{rep} \circ f) \\
= & \qquad \{\ \text{rolling rule}\ \} \\
 & \mathit{abs}\, (\mathit{fix}\, (\mathit{rep} \circ f \circ \mathit{abs})) \\
= & \qquad \{\ (1\beta)\ \} \\
 & \mathit{abs}\, (\mathit{fix}\, g)
\end{array}
$$

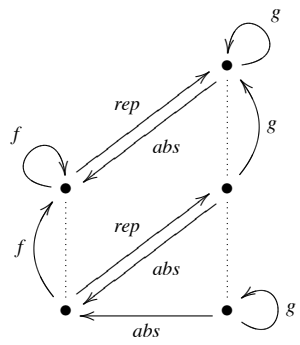
For condition (3), at first glance it may appear that we don't need assumption (C) at all, as condition (3) on its own is sufficient to verify the result by fusion. But the use of fusion requires that *abs* is strict. However, using assumption (C) and fixed-point induction, we can prove the factorisation result without this extra strictness condition; see the extended version of this article for the details. But perhaps *abs* being strict is implied by the assumptions and conditions? In fact, given assumption (A), this is indeed the case. However, for the weaker assumption (B), *abs* is not necessarily strict. The simplest counterexample is shown in the following diagram, in which the bullets are elements, dotted lines are orderings ($x \sqsubseteq y$) and solid lines are mappings ($x \mapsto y$):



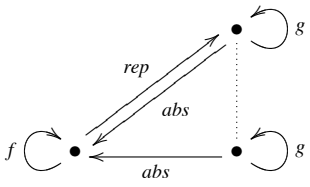
In particular, this example satisfies assumption (B), condition (3), and worker/wrapper factorisation, but *abs* is non-strict. Because (B) implies (C), the same counterexample also shows that the strictness of *abs* is not implied by (C) and (3). It is interesting to note that in the past condition (3) was regarded as being uninteresting because it just corresponds to the use of fusion (Hutton *et al.*, 2010). But in the context of *fix* this requires that *abs* is strict. However, as we have now seen, in the case of (B) and (C) this requirement can be dropped. Hence, worker/wrapper factorisation for condition (3) is applicable in some situations where fusion is not, i.e. when *abs* is non-strict.

Recall that showing (2) $\Rightarrow$ (2β) using fixed-point fusion requires that *rep* is strict. It is natural to ask if we can drop strictness from (2) by proving worker/wrapper factorisation in another way, as we did above with condition (3). The answer is no, and we verify this by exhibiting a non-strict *rep* that satisfies $\mathit{rep} \circ f = g \circ \mathit{rep}$ and assumption (A), but for

which worker/wrapper factorisation does not hold. The simplest example is as follows:



Because (A) $\Rightarrow$ (B) $\Rightarrow$ (C), the same counterexample shows that $rep \circ f = g \circ rep$ on its own is also insufficient for assumptions (B) and (C). However, while the addition of strictness is sufficient to ensure worker/wrapper factorisation, it is not *necessary*, which can be verified by exhibiting a non-strict *rep* that satisfies $rep \circ f = g \circ rep$, assumption (A), and worker/wrapper factorisation. The simplest such example is shown below. As before, this example also verifies that strictness is not necessary for (B) and (C).



3.3 Worker/Wrapper Fusion

When applying worker/wrapper factorisation, it is often desirable to fuse together instances of the conversion functions *rep* and *abs* to eliminate the overhead of repeatedly converting between the new and original types (Gill & Hutton, 2009). In general, it is not the case that $rep \circ abs$ can be fused to give id_B. However, provided that any of the assumptions (A), (B) or (C) hold, and any of the conditions except (3) hold, then the following *worker/wrapper-fusion* property is valid, as summarised in Figure 1:

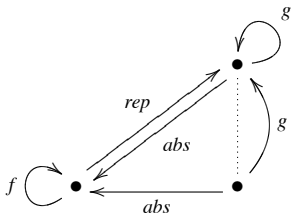
$$rep\ (abs\ (fix\ g)) = fix\ g$$

In a similar manner to Section 3.2, for the purposes of proving this result it suffices to consider assumption (C) and condition (2β):

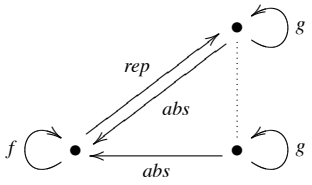
$$\begin{array}{ll} & rep\ (abs\ (fix\ g)) \\ = & \quad \{\ \text{worker/wrapper factorisation, (C) and } (2\beta)\ \} \\ & rep\ (fix\ f) \\ = & \quad \{\ (2\beta)\ \} \\ & fix\ g \end{array}$$

As with worker/wrapper factorisation, we confirm that strictness of *rep* is sufficient but not necessary in the case of condition (2), by exhibiting a non-strict *rep* that satisfies

$rep \circ f = g \circ rep$, assumption (A), and worker/wrapper fusion:



Finally, in the case of condition (3), the example below shows that (3) and (A) are not sufficient to ensure worker/wrapper fusion. Even if we also assume that *rep* is strict, *abs* is strict, or that both conversion functions are strict, this is still insufficient to ensure that worker/wrapper fusion holds in general for condition (3).



3.4 Relationship to Previous Work

The worker/wrapper results for *fix* presented in this section generalise those in (Gill & Hutton, 2009). The key difference is that the original article only considered worker/wrapper factorisation for condition (1β), although it wasn't identified as an explicit condition but rather inlined in the statement of the theorem itself, whereas we have shown that the result is also valid for (1), (2), (2β) and (3). Moreover, worker/wrapper fusion was only established for assumption (A) and condition (1β), whereas we have shown that any of the assumptions (A), (B) or (C) and any of the conditions (1), (1β), (2) or (2β) are sufficient. We also exhibited a counterexample to show that (3) is not a sufficient condition for worker/wrapper fusion under any of the assumptions.

We conclude by noting that in the context of assumption (C), the equivalent conditions (1β) and (2β) are not just sufficient to ensure that worker/wrapper factorisation and fusion hold, but are in fact necessary too. In particular, given these two properties, we can then verify that condition (2β) holds by the following simple calculation:

$$
\begin{array}{ll}
 & \mathit{fix}\ g \\
= & \qquad \{ \text{ worker/wrapper fusion } \} \\
 & \mathit{rep}\ (\mathit{abs}\ (\mathit{fix}\ g)) \\
= & \qquad \{ \text{ worker/wrapper factorisation } \} \\
 & \mathit{rep}\ (\mathit{fix}\ f)
\end{array}
$$

Hence, whereas previous work only identified conditions that are sufficient to ensure that worker/wrapper factorisation and fusion are valid, we now have conditions that are both necessary *and* sufficient. For practical applications however, we will normally use the simplest assumption and condition that is applicable, which is often assumption (A), and one of the conditions (1), (2) or (3) that do not involve the use of *fix*.

4 Initial-Algebra Semantics

We now turn our attention to the other previous formalisation of the worker/wrapper transformation, which was based upon an initial-algebra semantics of recursion in a categorical setting in which programs are defined using fold operators. In this section we review the basic definitions and properties from this approach to program semantics, and introduce our notation. For further details, see for example Bird & de Moor (1997).

Suppose that we fix a category $\mathbb{C}$ and a functor $F : \mathbb{C} \to \mathbb{C}$ on this category. Then an *F-algebra* is a pair (A,f) comprising an object A and an arrow $f : F\ A \to A$. In turn, an F-homomorphism from one such algebra (A,f) to another (B,g) is an arrow $h : A \to B$ such that $h \circ f = g \circ F\ h$. Algebras and homomorphisms themselves form a category, with composition and identities inherited from the original category $\mathbb{C}$. An *initial algebra* is an initial object in this new category, and we write $(\mu F, in)$ for an initial F-algebra, and $fold\ f$ for the unique homomorphism from this initial algebra to any other algebra (A,f). Moreover, the arrow $in : F\ \mu F \to \mu F$ has an inverse $out : \mu F \to F\ \mu F$, which establishes an isomorphism $F\ \mu F \cong \mu F$. The above definition for $fold\ f$ can also be expressed as the following equivalence, known as the *universal property* of *fold*:

Lemma 4.1 (*Universal Property of Fold*)

$$h = fold\ f \ \Leftrightarrow\ h \circ in = f \circ F\ h$$

The $\Rightarrow$ direction of this equivalence states that $fold\ f$ is a homomorphism from the initial algebra $(\mu F, in)$ to another algebra (A,f), while the $\Leftarrow$ direction states that any other such homomorphism h must be equal to $fold\ f$. Taken as a whole, the universal property expresses in an equational manner that $fold\ f$ is the unique homomorphism from $(\mu F, in)$ to (A,f), and forms the basic proof technique for the fold operator. For example, the universal property can be used to verify the corresponding versions of fixed-point fusion (Lemma 2.2) and the rolling rule (Lemma 2.3) for initial algebras:

Lemma 4.2 (*Fold Fusion*)

$$h \circ f = g \circ F\ h \ \Rightarrow\ h \circ fold\ f = fold\ g$$

Lemma 4.3 (*Rolling Rule*)

$$fold\ (f \circ g) = f \circ fold\ (g \circ F\ f)$$

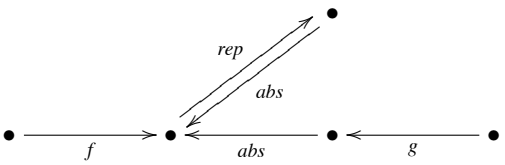
5 Worker/Wrapper for Initial Algebras

Within the category-theoretic setting of the previous section, consider a recursive program defined as the fold of an algebra $f : F\ A \to A$ for some object A. Now consider a more efficient program that performs the same task, defined by first folding an algebra $g : F\ B \to B$ on some other object B, and then converting the resulting value back to the original object type by composing with an arrow $abs : B \to A$. The equivalence between these two programs is captured by the following equation:

$$fold\ f = abs \circ fold\ g$$

In a similar manner to least fixed points, we call *fold f* the original program, *fold g* the *worker* program, *abs* the *wrapper* arrow, and the equation itself the *worker/wrapper factorisation* for initial algebras. The properties that we use to validate the factorisation equation are similar to those that we identified for least fixed points, and are summarised in Figure 2. As previously, the assumptions form a heirarchy (A) $\Rightarrow$ (B) $\Rightarrow$ (C), the conditions (1β) and (2β) are weaker versions of (1) and (2) and are equivalent given assumption (C), and in general there is no relationship between conditions (1), (2) and (3). As we are working in an arbitrary category the notion of strictness is not defined, and hence there is no requirement that *rep* be strict for (2); we will return to this point in Section 6.

Worker/wrapper fusion can also be formulated for initial algebras, as shown in Figure 2. Moreover, the counterexample from Section 3.3 that shows that fusion is not in general valid for condition (3) for least fixed points can readily be adapted to the case of initial algebras. Specifically, if we define a constant functor $F : \mathbf{SET} \to \mathbf{SET}$ on the category of sets and total functions by $F\,X = \mathbf{1}$ and $F\,f = id_{\mathbf{1}}$, where $\mathbf{1}$ is any singleton set, then the following definitions satisfy (3) and (A) but not worker/wrapper fusion:



The worker/wrapper results for *fold* presented in this section generalise those in (Hutton *et al.*, 2010). The key difference is that the original article only considered worker/wrapper factorisation for assumption (A) and conditions (1), (2) and (3) (in which context (1) is stronger than the other two conditions), whereas we have shown that the result is also valid for the weaker assumptions (B) and (C) (in which context (1), (2) and (3) are in general unrelated) and the weaker conditions (1β) and (2β). Moreover, worker/wrapper fusion was essentially only established for assumption (A) and condition (1), whereas we have shown that any of the assumptions (A), (B) or (C) and any of the conditions (1), (1β), (2) or (2β) are sufficient. We also showed that (3) is not sufficient for worker/wrapper fusion under any of the assumptions. Finally, we note that as with least fixed points, in the context of assumption (C) the equivalent conditions (1β) and (2β) are both necessary and sufficient to ensure worker/wrapper factorisation and fusion for initial algebras.

6 From Least Fixed Points to Initial Algebras

In Section 5 we developed the worker/wrapper theory for initial algebras. Given that the results were formulated and proved for an arbitrary category $\mathbb{C}$, we would expect them to hold in the category **CPO** of cpos and continuous functions that was used in the least-fixed-point approach. This is indeed the case, with one complicating factor: when **CPO** is the base category, the universal property has a strictness side condition. Hence, when our general worker/wrapper theory for initial algebras is applied in the category **CPO**, some of the results require additional strictness conditions. In this section we review these extra conditions, and show how most of them can be eliminated by instantiating our general worker/wrapper theory for least fixed points.

Given algebras

$$f : F\,A \to A$$
$$g : F\,B \to B$$

for some functor F, and conversion arrows

$$rep : A \to B$$
$$abs : B \to A$$

then we have a set of *worker/wrapper assumptions*

(A) $abs \circ rep = id_A$
(B) $abs \circ rep \circ f = f$
(C) $fold\,(abs \circ rep \circ f) = fold\,f$

and a set of *worker/wrapper conditions*

(1) $g = rep \circ f \circ F\,abs$ (1β) $fold\,g = fold\,(rep \circ f \circ F\,abs)$
(2) $rep \circ f = g \circ F\,rep$ (2β) $fold\,g = rep \circ fold\,f$
(3) $abs \circ g = f \circ F\,abs$

Provided that any of the assumptions hold and any of the conditions hold, then *worker/wrapper factorisation* is valid:

$$fold\,f = abs \circ fold\,g$$

Furthermore, if any of the assumptions hold, and any of the conditions except (3) hold, then *worker/wrapper fusion* is valid:

$$rep \circ abs \circ fold\,g = fold\,g$$

Figure 2: Worker/wrapper transformation for initial algebras.

6.1 Strictness

Recall that the basic proof technique for the fold operator is its universal property. In the category **CPO**, this property has a strictness side condition (Meijer *et al.*, 1991):

Lemma 6.1 (*Universal Property of Fold in* **CPO**)
If h is strict, then:

$$h = fold\,f \;\Leftrightarrow\; h \circ in = f \circ F\,h$$

The universal property of fold, together with derived properties such as fusion and the rolling rule, form the basis of our proofs of worker/wrapper factorisation and fusion for initial algebras in Section 5. Tracking the impact of the extra strictness condition above on these results is straightforward but tedious, so we omit the details here (which are available in the online Agda proofs) and just present the results: for conditions (1), (1β), (2) and (2β), both factorisation and fusion require that f, *rep* and *abs* are strict, while for (3), factorisation requires that g and *abs* are strict.

In summary, instantiating the worker/wrapper results for initial algebras to the category **CPO** is straightforward, but deriving the results in this manner introduces many strictness side conditions that may limit their applicability. Some of these conditions could be avoided by using more liberal versions of derived properties such as fold fusion and the

rolling rule that are proved from first principles rather than being derived from the universal property. However, it turns out that most of the strictness conditions can be avoided using our worker/wrapper theory for least fixed points.

6.2 From Fix to Fold

As noted earlier, the generalised worker/wrapper results for initial algebras are very similar to those for least fixed points. Indeed, unifying the results in this manner is one of primary contributions of this article. This suggests that it may be possible to derive one set of results from the other. In this section we show how the initial-algebra results in **CPO** can be derived from those for least fixed points, by exploiting the fact that in this context *fold* can be defined in terms of *fix* (Meijer *et al*., 1991):

Lemma 6.2 (*Definition of Fold using Fix in* **CPO**)

$$fold\, f \;=\; fix\, (\lambda\, h \;\to\; f \circ F\, h \circ out)$$

Suppose that we are given two algebras $f : F\, A \to A$ and $g : F\, B \to B$, and two conversion functions $rep : A \to B$ and $abs : B \to A$. We now seek to use the general worker/wrapper results for the fix operator to derive assumptions and conditions under which worker/wrapper factorisation holds for the fold operator. That is,

$$fold\, f \;=\; abs \circ fold\, g$$

First, we define functions f' and g' such that $fold\, f = fix\, f'$ and $fold\, g = fix\, g'$:

$$\begin{array}{ll} f' \;:\; (\mu F \to A) \to (\mu F \to A) & \qquad g' \;:\; (\mu F \to B) \to (\mu F \to B) \\ f' \;=\; \lambda\, h \to f \circ F\, h \circ out & \qquad g' \;=\; \lambda\, h \to g \circ F\, h \circ out \end{array}$$

Then we define conversion functions between the types for $fold\, f$ and $fold\, g$:

$$\begin{array}{ll} rep' \;:\; (\mu F \to A) \to (\mu F \to B) & \qquad abs' \;:\; (\mu F \to B) \to (\mu F \to A) \\ rep'\, h \;=\; rep \circ h & \qquad abs'\, h \;=\; abs \circ h \end{array}$$

Using these definitions, the worker/wrapper equation $fold\, f = abs \circ fold\, g$ in terms of *fold* is equivalent to the following equation in terms of *fix*:

$$fix\, f' \;=\; abs'\, (fix\, g')$$

This equation has the form of worker/wrapper factorisation for *fix*, and is hence valid provided one of the assumptions and one of the conditions from Figure 1 are satisfied for f', g', rep' and abs'. By expanding definitions, it is now straightforward to simplify each of these assumptions and conditions in terms of the original functions f, g, *rep* and *abs*; see the extended version of the article for the details. A similar procedure can be applied to worker/wrapper fusion. The end result is a worker/wrapper theory for initial algebras in **CPO** that has the same form as Figure 2, except that condition (2) requires that *rep* is strict. Compared to the derivation in Section 6.1, this new approach eliminates all but one strictness requirement, and hence the resulting theory is more generally applicable.

One might ask if we can also drop strictness from condition (2), but the answer is no. In order to verify this, let us take $Id : \mathbf{CPO} \to \mathbf{CPO}$ as the identity functor, for which it can

be shown by fixed-point induction that $fold\ f\ \bot = fix\ f$. Now consider the counterexample from Section 3.2 that shows that strictness cannot be dropped from (2) in the theory for *fix*. This example satisfies (A) and $rep \circ f = g \circ Id\ rep$, but not worker/wrapper factorisation $fold\ f = abs \circ fold\ g$. In particular, if we assume factorisation is valid we could apply both sides to $\bot$ to obtain $fold\ f\ \bot = abs\ (fold\ g\ \bot)$, which by the above result is equivalent to $fix\ f = abs\ (fix\ g)$, which does not hold for this example as shown in Section 3.2. Hence, by contradiction, $fold\ f = abs \circ fold\ g$ is invalid.

7 Related Work

A historical review of the worker/wrapper transformation and related work was given in Gill & Hutton (2009), so we direct the reader to that article rather than repeating the details here. More recently, Gammie (2011) has observed that the manner in which the worker/wrapper-fusion rule was used in the original article may lead to the introduction of non-termination. However, this is a well-known consequence of the fold/unfold approach to program transformation (Burstall & Darlington, 1977; Tullsen, 2002), which in general only preserves partial correctness, rather than being a problem with the fusion rule itself, which is correct as it stands. Alternative, but less expressive, transformation frameworks that guarantee total correctness have been proposed, such as the use of expression procedures (Scherlis, 1980; Sands, 1995). Gammie's solution was to add the requirement that *rep* be strict to the worker/wrapper-fusion rule, which holds for the relevant examples in the original article. However, we have not added this requirement in the present article, as this would unnecessarily weaken the fusion rule without overcoming the underlying issue with fold/unfold transformation. Gammie also pointed out that the stream memoisation example in (Gill & Hutton, 2009) incorrectly claims that assumption (A) holds, but we note that the example as a whole is still correct as the weaker assumption (B) does hold.

In this article we have focused on developing the theory of the worker/wrapper transformation, with the aim of making it as widely applicable as possible. Meanwhile, a team at the University of Kansas is putting the technique into mechanised practice as part of the HERMIT project (Farmer *et al*., 2012). In particular, they are developing a general purpose system for optimising Haskell programs that allows programmers to write *active source* (Sittampalam & de Moor, 2003) that includes sufficient annotations to permit the Glasgow Haskell Compiler (GHC) to apply custom transformations automatically. The worker/wrapper transformation was the first high-level technique encoded in the system, and it then proved relatively straightforward to mechanise a selection of new and existing worker/wrapper examples (Sculthorpe *et al*., 2013). This corresponds with our experience of applying the transformation by hand, namely that once the initial decision regarding the desired change in type is made, applying the transformation is largely a matter of routine manipulation, and hence is ripe for mechanical assistance. Working with the automated system has also revealed that other, more specialised, transformation techniques can be cast as instances of worker/wrapper, and consequently that using the worker/wrapper infrastructure can simplify mechanising those transformations (Sculthorpe *et al*., 2013). Once the system is fully developed, it will permit the worker/wrapper transformation to be applied to larger and more sophisticated examples than is currently possible by hand.

8 Conclusions and Future Work

The original worker/wrapper article (Gill & Hutton, 2009) formalised the basic technique using least fixed points. The follow-up article (Hutton *et al*., 2010) showed how it can be improved by exploiting the structure of initial algebras. This article showed how the two approaches can be generalised in a uniform manner by combining their different sets of correctness conditions. Moreover, we showed how the new theories can be further generalised with conditions that are both necessary and sufficient to ensure the correctness of the underlying transformations. All the proofs have been mechanically checked using the Agda proof assistant, and are available online.

It is interesting to recount how the conditions (1β) and (2β) were developed. Initially we focused on combining assumptions (A), (B) and (C) from the first article with conditions (1), (2) and (3) from the second. However, the resulting theory was still not powerful enough to handle some examples we intuitively felt should fit within the framework. It was only when we looked again at the proofs for worker/wrapper factorisation and fusion that we realised that conditions (1) and (2) could be further weakened, resulting in conditions (1β) and (2β), and proofs that they are equivalent and maximally general.

In terms of further work, practical applications of the worker/wrapper technique are being driven forward by the HERMIT project at the University of Kansas, as described in Section 7. On the foundational side, it would be interesting to exploit additional forms of structure to further extend the generality and applicability of the technique, for example by considering other recursion operators such as unfold (Gibbons & Jones, 1998) and hylomorphisms (Meijer *et al*., 1991), framing the technique using more general categorical constructions such as limits and colimits (MacLane, 1971), together with more sophisticated notions of computation such as monadic (Wadler, 1992), comonadic (Uustalu & Vene, 2008) and applicative (McBride & Paterson, 2008) programs.

Acknowledgements

The first author was supported by NSF award number 1117569. We would like to thank Nicolas Frisby and Andy Gill for useful discussions on practical applications of this work, and Jennifer Hacket for the strictness counterexample in Section 6.

References

Backhouse, Roland. (2002). Galois Connections and Fixed Point Calculus. *Pages 89–150 of: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer.

Bird, Richard, & de Moor, Oege. (1997). *Algebra of Programming*. Prentice Hall.

Burstall, Rod. M., & Darlington, John. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, **24**(1), 44–67.

Farmer, Andrew, Gill, Andy, Komp, Ed, & Sculthorpe, Neil. (2012). The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. *Pages 1–12 of: Haskell Symposium*. ACM.

Gammie, Peter. (2011). Strict Unwraps Make Worker/Wrapper Fusion Totally Correct. *Journal of Functional Programming*, **21**(2), 209–213.

Gibbons, Jeremy, & Jones, Geraint. (1998). The Under-Appreciated Unfold. *Pages 273–279 of: International Conference on Functional Programming*.

Gill, Andy, & Hutton, Graham. (2009). The Worker/Wrapper Transformation. *Journal of Functional Programming*, **19**(2), 227–251.

Hutton, Graham, Jaskelioff, Mauro, & Gill, Andy. (2010). Factorising Folds for Faster Functions. *Journal of Functional Programming*, **20**(3&4), 353–373.

MacLane, Saunders. (1971). *Categories for the Working Mathematician*. Graduate Texts in Mathematics, no. 5. Springer-Verlag.

McBride, Conor, & Paterson, Ross. (2008). Applicative Programming With Effects. *Journal of Functional Programming*, **18**(1), 1–13.

Meijer, Erik, Fokkinga, Maarten M., & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *Pages 124–144 of: Functional Programming Languages and Computer Architecture*. Springer.

Peyton Jones, Simon, & Launchbury, John. (1991). Unboxed Values as First Class Citizens in a Non-Strict Functional Language. *Pages 636–666 of: Functional Programming Languages and Computer Architecture*. Springer.

Sands, David. (1995). Higher-Order Expression Procedures. *Pages 178–189 of: Partial Evaluation and Semantics-Based Program Manipulation*. ACM.

Scherlis, William Louis. (1980). *Expression Procedures and Program Derivation*. Ph.D. thesis, Stanford University.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon.

Sculthorpe, Neil, Farmer, Andrew, & Gill, Andy. (2013). The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. *Implementation and Application of Functional Languages 2012*. Springer.

Sittampalam, Ganesh, & de Moor, Oege. (2003). Mechanising Fusion. *Pages 79–103 of:* Gibbons, Jeremy, & de Moor, Oege (eds), *The Fun of Programming*. Palgrave.

Tullsen, Mark. (2002). *PATH, A Program Transformation System for Haskell*. Ph.D. thesis, Yale University.

Uustalu, Tarmo, & Vene, Varmo. (2008). Comonadic Notions of Computation. *Pages 263–284 of: Coalgebraic Methods in Computer Science*. Elsevier.

Wadler, Philip. (1992). The Essence of Functional Programming. *Pages 1–14 of: Principles of Programming Languages*. ACM.

Winskel, Glynn. (1993). *The Formal Semantics of Programming Languages — An Introduction*. Foundation of Computing. MIT.

Notes on the Kan semisimplicial types model
Work in progress!

Ambrus Kaposi

June 2013

1 Motivation

To make dependently-typed functional programming convenient, the underlying type theory should have nice features such as extensionality and transport of structures along equivalences. Homotopy type theory [14] is such a candidate, however it is not yet clear how to implement it as a programming language. A large step towards implementation would be providing a constructive model of homotopy type theory. Such a model could be implemented in a type theory with an existing implementation such as Martin-Löf type theory.

2 Models of type theory in type theory

A type theory can be viewed as a formal inference system which allows the derivation of certain kinds of judgements. These kinds are given together with derivation rules that say which judgements can be derived from which other judgements. We give an example of such a type theory with 8 kinds of judgements and several inference rules in section 3.

A semantics for an inference system is a collection of interpretations each specifying the set of true judgements.

A model of a type theory is a sound semantics of the deduction system corresponding to the type theory. This can be made precise by the notion of generalized algebraic theory [3]. A notion of model specific to Martin-Löf type theory (MLTT) is categories with families ([7], [9]) which defines what a model is in the language of category theory. Using this definition, one only needs to check whether a particular semantics is an instance of a category with families, and if yes, this ensures that the semantics is sound. We will we use an informal definition of model specific to dependent type theories but more general than categories with families since we are also interested in models of type theories which do not have all the rules of MLTT.

We are interested in modelling a type theory in MLTT which is a rich and powerful language for mathematics and also has nice proof-theoretic properties: every term in the empty context can be reduced to a normal form [12]. And there are practical implementations that actually compute these normal forms such as Agda [13]. The advantage of using such a language to define a model is that if the definitional equality of the object theory is modelled by definitional equality in MLTT, we get canonicity for the modelled theory for free [8].

A model of a type theory in MLTT is:

1. For each kind of judgement, a type (family) representing the judgement: if the type has an element, it means that the judgement is true, eg. the context validity judgement $\Gamma \vdash$ is represented by $\Gamma : Con$ where Con is a type.

2. For each derivation rule, a function from the representations of the premises of the derivation rule to the representation of the conclusion of the derivation rule.

Question: are there other requirements? See [11] and [9].

Because each derivation rule is validated by a function, the semantics is indeed sound. If additional functions can be defined representing additional derivation rules, these additional derivation rules are also justified by the model.

Some examples of models of variants of MLTT are given in the following table. The object theory is the theory being modelled and metatheory is the theory in which the implementation is done.

object theory	**model**	**metatheory**
MLTT	see [4]	MLTT (def. equality is interpreted by a separate type and normalisation is proved separately)
MLTT with functional extensionality and UIP	observational type theory (OTT) [1]	MLTT with a proof-irrelevant Prop universe
MLTT (without some definitional rules for projections) with functional extensionality and quotient types	setoid model by Hofmann [8]	MLTT
weak MLTT with a univalent hProp universe	setoid model by Coquand [5]	MLTT
MLTT with a univalent hProp and hSet universe	groupoid model	ZFC?
MLTT with univalence for all universes	Voevodsky's Kan simplicial set model [10]	ZFC (uses non-constructive properties, see [6])
MLTT	Semisimplicial set model [2]	MLTT
weak MLTT with a univalent hSet universe	truncated Kan semisimplicial set model [2]	MLTT
weak MLTT with univalence for all universes	Kan semisimplicial set model [2]	constructive parts of ZFC

If the definitional equality of the object theory is not modelled by the definitional equality of the metatheory, one needs to prove canonicity separately as in the case of [4]. Extensional type theory can also be modelled in MLTT using the above definition of model, however in this case, the equality reflection

rule does not let us to use the definitional equality of the metatheory as the interpretation for the definitional equality of the object theory.

In the next section, we list the rules of Martin-Löf type theory together with the additional rules we are interested to be satisfied by a model: rules for homotopy type theory.

3 Rules of type theory

Below we give the rules of Martin-Löf type theory with explicit substitution. We follow the presentation of [2].

3.1 Kinds of judgements

We have ten kinds of judgements:

$\Gamma \vdash$	Γ is a valid context
$\sigma : \Delta \to \Gamma$	σ is a substitution from context Δ to Γ
$\Gamma \vdash A$	A is a type in context Γ
$\Gamma \vdash F : (A)\mathsf{Type}$	F is a type family indexed over A in context Γ
$\Gamma \vdash t : A$	t is a term of type A in context Γ

The next five kinds of judgements express definitional equality for the above constructs: contexts, substitutions, types, type families and terms.

$\vdash \Gamma \equiv \Delta$	contexts Γ and Δ are definitionally equal
$\sigma \equiv \delta$	substitutions σ and δ are definitionally equal
$\Gamma \vdash A \equiv B$	types A and B in context Γ are definitionally equal
$\Gamma \vdash F \equiv G : (A)\mathsf{Type}$	F is a type family indexed over A in context Γ
$\Gamma \vdash t \equiv r : A$	terms t and r of type A in context Γ are definitionally equal

If it is clear from the context, we will omit the context and type, just write equations like $t \equiv r$.

In an implementation of type theory, definitional equality expresses reduction behaviour. An implementation defines what a normal form of an expression (a substitution, context, type or term) is. It also provides an algorithm the input of which is an expression (in the empty context for types and terms) and the output of which is a definitionally equal expression in normal form. This algorithm should terminate for all inputs.

3.2 Context formation rules

A context is a dependent list of types.

$$\frac{}{() \vdash} \qquad \frac{\Gamma \vdash \quad \Gamma \vdash A}{\Gamma.A \vdash}$$

3.3 Rules for substitution construction

A substitution $\Delta \to \Gamma$ can be thought of as a list of terms each having their free variables in Δ and having types Γ (which is a list of types).

$$\frac{\Gamma \vdash}{1 : \Gamma \to \Gamma} \qquad \frac{\sigma : \Delta \to \Gamma \quad \delta : \Theta \to \Delta}{\sigma\delta : \Theta \to \Gamma}$$

$$\frac{\sigma : \Delta \to \Gamma \quad \Gamma \vdash A \quad \Delta \vdash u : A\sigma}{(\sigma, u) : \Delta \to \Gamma.A} \qquad \frac{\Gamma \vdash A}{\mathsf{p} : \Gamma.A \to \Gamma}$$

Types, type families and terms can be substituted. The notation for substitution is just post-position.

$$\frac{\Gamma \vdash A \quad \sigma : \Delta \to \Gamma}{\Delta \vdash A\sigma} \qquad \frac{\Gamma \vdash t : A \quad \sigma\Delta \to \Gamma}{\Delta \vdash t\sigma : A\sigma} \qquad \frac{\Gamma \vdash F : (A)\mathsf{Type} \quad \sigma : \Delta \to \Gamma}{\Delta \vdash F\sigma : (A\sigma)\mathsf{Type}}$$

3.4 Type family formation rule

We use a separate kind of judgement for type families and type level application instead of direct substitution on the type level. This notation is somewhat closer to the Agda notation where type level abstraction and application have the same notation as on the term level.

$$\frac{\Gamma \vdash A \quad \Gamma.A \vdash B}{\Gamma \vdash \lambda B : (A)\mathsf{Type}}$$

3.5 Type formation rules

The first type formation rule is type level application. It can be also thought of as a type level elimination rule.

$$\frac{\Gamma \vdash F : (A)\mathsf{Type} \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{app}(F, a)} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash F : (A)\mathsf{Type}}{\Gamma \vdash \mathsf{Fun}\ A\ F}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash F : (A)\mathsf{Type}}{\Gamma \vdash \mathsf{Sum}\ A\ F}$$

3.6 Term introduction rules

The variable introduction rule is given by a projection to the last element of the context. Earlier elements can be given by applying the p substitution (multiple times, if necessary).

$$\frac{\Gamma \vdash A}{\Gamma.A \vdash \mathsf{q} : A\mathsf{p}}$$

Introduction rules for product and sum:

$$\frac{\Gamma.A \vdash b : \mathsf{app}(F\mathsf{p}, \mathsf{q})}{\Gamma \vdash \lambda b : \mathsf{Fun}\ A\ F} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : \mathsf{app}(F, a)}{\Gamma \vdash (a, b) : \mathsf{Sum}\ A\ F}$$

3.7 Term elimination rules

$$\frac{\Gamma \vdash w : \mathsf{Fun}\ A\ F \quad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}(w, u) : \mathsf{app}(F, u)} \qquad \frac{\Gamma \vdash c : \mathsf{Sum}\ A\ F}{\Gamma \vdash \mathsf{p}c : A} \qquad \frac{\Gamma \vdash c : \mathsf{Sum}\ A\ F}{\Gamma \vdash \mathsf{q}c : app(F, \mathsf{p}c)}$$

3.8 Rules for definitional equality

The following rules are typed, we express them in an untyped form for better readability. Rules expressing that contexts together with substitutions form a category:

$$1\sigma \equiv \sigma = \sigma 1 \quad (\sigma\delta)\nu \equiv \sigma(\delta\nu)$$

The following rules express the relation between substitution and variables:

$$(\sigma, u)\delta \equiv (\sigma\delta, u\delta) \quad \mathsf{p}(\sigma, u) \equiv \sigma \quad \mathsf{q}(\sigma, u) \equiv u \quad 1 = (\mathsf{p}, \mathsf{q})$$

Rules for applying substitutions on types, type families and terms:

$$(A\sigma)\delta \equiv A(\sigma\delta) \quad A1 \equiv A \quad (F\sigma)\delta \equiv F(\sigma\delta) \quad F1 \equiv F$$
$$(a\sigma)\delta \equiv a(\sigma\delta) \quad a1 \equiv a$$

Substitution rules:

$$\mathsf{app}(F, u)\sigma \equiv \mathsf{app}(F\sigma, u\sigma) \quad \mathsf{app}(w, u)\sigma \equiv \mathsf{app}(w\sigma, u\sigma)$$
$$(\mathsf{Fun}\ A\ F)\sigma \equiv \mathsf{Fun}(A\sigma)(F\sigma) \quad (\mathsf{Sum}\ A\ F)\sigma \equiv \mathsf{Sum}(A\sigma)(F\sigma)$$

Computation rules:

$$\mathsf{app}((\lambda B)\sigma, u) \equiv B(\sigma, u) \quad \mathsf{app}((\lambda b)\sigma, u) \equiv b(\sigma, u)$$
$$\mathsf{p}(a, b) \equiv a \quad \mathsf{q}(a, b) \equiv b$$

3.9 Weak type theory

In a weak type theory, the following rule is excluded:

$$(\lambda t)\sigma \equiv \lambda(t(\sigma\mathsf{p}, \mathsf{q}))$$

This expresses substitution under lambda or the ξ rule. The usual formulation (without explicit substitution) is as follows:

$$\frac{t \equiv s}{\lambda x.t \equiv \lambda x.s}$$

The first publication of type theory [12] did not have this rule and Martin-Löf argued against this rule by stating that the informal notion of definitional equality as used in mathematics (the formal counterpart of which is called convertibility, but usually we call both just definitional equality) has exactly the following properties [11]:

1. A definiens (formal: contractum, right hand side) is definitionally equal to it's definiendum (formal: redex, left hand side).
2. Preservation under substitution, i.e. if we substitute two definitionally equal expressions for a variable in a third expression, the results should be definitionally equal. Formally: if $a \equiv b$, then $u(1, a) \equiv u(1, b)$ in our notation.
3. It is an equivalence relation (reflexive, symmetric and transitive).

These properties correspond to the idea of unfolding definitions: we define a left hand side by a right hand side and each time we encounter the left hand side we just replace it with the right hand side – this is how most implementations of type theory work and also how mathematicians use the mostly implicit notion of definitional equality. Furthermore, the weaker notion of definitional equality makes the models of type theory (as defined by Martin-Löf in [11]) more well-behaved.

Martin-Löf also argues that the η-rules for functions and products are should not be valid according to this reasoning.

3.10 Rules for equality

We would like to validate the following rules of propositional equality (type formation, introduction):

$$\frac{\Gamma \vdash A \quad \Gamma \vdash a : A \quad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Eq}_A\, a\, u} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}\, a : \mathsf{Eq}_A\, a\, a}$$

Also the elimination rule J (usually called subst or transport):

$$\frac{\Gamma \vdash e : \mathsf{Eq}_A a u \quad \Gamma \vdash F : (A)\mathsf{Type} \quad \Gamma \vdash p : \mathsf{app}(F, a)}{\Gamma \vdash \mathsf{J}\, e\, p : \mathsf{app}(F, u)}$$

The usual dependent eliminator can be derived from this if we have that the type $\mathsf{Sum}\ A\ (\lambda \mathsf{Eq}_{A\mathsf{p}}\ a\ q)$ is contractible.

The corresponding substitution rules:

$$(\mathsf{Eq}_A\ a\ u)\sigma \equiv \mathsf{Eq}_{A\sigma}\ (a\sigma)\ (u\sigma) \qquad ((\mathsf{J}\, e\, p)\sigma \equiv \mathsf{J}\ (e\sigma)\ (p\sigma))$$

The computation rule is $\mathsf{J}\,(\mathsf{refl} a)\, p \equiv p$. Sometimes this rule is only validated up to propositional equality, that is the computation rule becomes $\mathsf{Eq}_{Eq_A\ a\ a}(\mathsf{J}(\mathsf{refl} a) p) p$. This is the case in the Kan semisimplicial set model described below.

3.11 Rules of homotopy type theory

We are interested in models which validate additional rules such as that of extensionality (up to propositional equality):

$$\frac{\Gamma \vdash p : \mathsf{Fun}\ A\ (\lambda \mathsf{Eq}_{\mathsf{app}(F\mathsf{p},\mathsf{q})}\ \mathsf{app}(f\mathsf{p},\mathsf{q})\ \mathsf{app}(g\mathsf{p},\mathsf{q}))}{\Gamma \vdash \mathsf{ext}\ p : \mathsf{Eq}_{\mathsf{Fun}\ A\ F}\ f\ g}$$

With the substitution rule $(\mathsf{ext}\ u)\sigma \equiv \mathsf{ext}\ (u\sigma)$.

We would like to have Voevodsky's univalence axiom as a rule which can be expressed by saying that if we have a function between two sets and a proof that this function is an isomorphism, then we can convert this into a proof that the two sets are equal. In this case we can use any structure on one of the sets and transport it to the other set.

Additional rules which are not yet clear how to formalise giving higher inductive types and truncations should be also considered. Higher inductive types provide quotient types as a special case.

4 Semisimplicial set model

The category Δ of linear posets contains as elements sets $[n] = \{0, 1, ..., n\}$ and as morphisms strictly increasing maps.

A simplicial set is a presheaf $\Delta^{op} \to \mathsf{Set}$.

Let Δ_+ be the category with the same objects as Δ but only with the injective functions as arrows. Every non-identity injective function is a composition of so-called face maps $\epsilon^i : [n] \to [n+1]$ which is an injection that skips the ith element.

A semisimplicial set is a presheaf $\Delta_+^{op} \to \mathsf{Set}$, that is, a functor from Δ_+^{op} to Set.

By unfolding this definition we get the following structure:

- For the object part of the functor we take sets $X[0]$, $X[1]$ etc. as the images of objects [0], [1] etc.
- For the morphism part of the functor it is enough to give the images of the face maps (and identities) and generate the images of compositions of injections by the composition of images. Hence, we take functions $d_i : X[n] \to X[n-1] (i = 0, ...n)$ as the image of each non-identity injection and the identity function as the image of the identity injection. The functor laws are satisfied.

We can define the same structure in type theory saying that $X[0]$ is a type and $X[1]$ is a type family indexed over two elements of $X[0]$ and the images of the two facemaps are just the projection functions on the type:

$$
\begin{aligned}
&X\epsilon^0 : X[1]\, x_0\, x_1 \to X[0] \\
&X\epsilon^0\_ \;=\; x_0 \\
&X\epsilon^1 : X[1]\, x_0\, x_1 \to X[0] \\
&X\epsilon^1\_ \;=\; x_1
\end{aligned}
$$

Similarily for higher levels. The first few levels are defined as follows (curly brackets mean implicit arguments):

$$
\begin{aligned}
X[0] \quad &: \mathsf{Type} \\
X[1] \quad &: X[0] \to X[0] \to \mathsf{Type} \\
X[2] \quad &: \{x_0\, x_1\, x_2 : X[0]\} \to X[1]\, x_0\, x_1 \to X[1]\, x_0\, x_2 \to X[1]\, x_1\, x_2 \to \mathsf{Type} \\
X[3] \quad &: \{x_0\, x_1\, x_2\, x_3 : X[0]\} \\
&\quad \{x_{01} : X[1]\, x_0\, x_1\}\{x_{02} : X[1]\, x_0\, x_2\}\{x_{03} : X[1]\, x_0\, x_3\} \\
&\quad \{x_{12} : X[1]\, x_1\, x_2\}\{x_{13} : X[1]\, x_1\, x_3\}\{x_{23} : X[1]\, x_2\, x_3\} \\
&\quad \to X[2]\, x_{01}\, x_{02}\, x_{12} \to X[2]\, x_{01}\, x_{03}\, x_{13} \\
&\quad \to X[2]\, x_{02}\, x_{03}\, x_{23} \to X[2]\, x_{12}\, x_{13}\, x_{23} \to \mathsf{Type}
\end{aligned}
$$

The above series hasn't been yet formalised in type theory. The geometric interpretation of the series is as follows: $X[0]$ is the set of points (0-dimensional simplices), $X[1]$ is the set of directed edges (1-dimensional simplices) parameterised by two points, that is an edge $x_{01} : X[1]\, x_0\, x_1$ has starting point x_0 and end point x_1. An element $x_{012} : X[1]\, x_{01}\, x_{02}\, x_{12}$ is a triangle (2-dimensional simplex) with sides x_{01}, x_{02} and x_{12}. The next level simplices are tetrahedrons. The indices should be always increasing. The direction of the edges of the triangle is fixed by the indices so that they are increasing. The face maps give the faces of a simplex. The faces of an egde are two points, the faces of a triangle are three edges, the faces of a tetrahedron are triangles etc.

The semisimplicial set model gives interpretations of judgements in terms of the above defined semisimplicial sets, eg. contexts are interpreted as semisimplicial sets, substitutions are mappings between semisimplicial sets etc [2]. This model validates all rules of MLTT including the ξ-rule.

5 Kan semisimplicial set model

A Kan semisimimplicial set is a semisimplicial set with additional operations for Kan completions. A Kan completion operation for level n, given n faces of level $n-1$ forming a "horn" (a simplex of level n with one face omitted), gives a face of level $n-1$ which was omitted in the horn.

For level n this means $n+1$ completion operators which give face which is missing from the horn, and $n+1$ filler operators which say that the horn completed with the face given by the former completion is indeed an n-simplex.

We define Kan completions in a heterogeneous way in type theory as follows. On level 1, if we have a point in A_i we get a point in the other set.

$$
\begin{aligned}
&A_0, A_1 : \mathsf{Type}\\
&A_0 \leftrightarrow A_1 :\equiv (comp_0^1 : A_0 \to A_1) \times (comp_1^1 : A_1 \to A_0)
\end{aligned}
$$

On level 2, if we have 2 lines between three points, we get a third line completing the triangle.

$$
\begin{aligned}
&A_i : \mathsf{Type}\\
&R_{ij} : A_i \to A_j \to \mathsf{Type}\\
&R_{01} \leftrightarrow R_{02} \leftrightarrow R_{12} :\equiv \{a_i : A_i\}(comp_0^2 : R_{01}\, a_0\, a_1 \to R_{02}\, a_0\, a_2 \to R_{12}\, a_1\, a_2)\\
&\qquad\qquad \times \{a_i : A_i\}(comp_1^2 : R_{01}\, a_0\, a_1 \to R_{12}\, a_1\, a_2 \to R_{02}\, a_0\, a_2)\\
&\qquad\qquad \times \{a_i : A_i\}(comp_2^2 : R_{02}\, a_0\, a_2 \to R_{12}\, a_1\, a_2 \to R_{01}\, a_0\, a_1)
\end{aligned}
$$

Level 3 completion, given 4 points, 6 lines, 3 triangles, produces the fourth triangle completing the tetrahedron. And so on for higher levels.

The Kan filler operations for the first few levels are given below. First level:

$$
\begin{aligned}
&A_0, A_1 : \mathsf{Type}\\
&R : A_0 \to A_1 \to \mathsf{Type}\\
&Coh(R, comp^1) :\equiv (Comp_0^1 : (x : A_0) \to R\, x\, (comp_0^1 x))\\
&\qquad\qquad \times (Comp_1^1 : (x : A_1) \to R\, (comp_1^1 x)\, x)
\end{aligned}
$$

Second level:

$$
\begin{aligned}
&A_i : \mathsf{Type}\\
&R_{ij} : A_i \to A_j \to \mathsf{Type}\\
&T : \{a_i : A_i\} \to R_{ij}\, a_i\, a_j \to R_{ik}\, a_i\, a_k \to R_{jk}\, a_j\, a_k \to \mathsf{Type}\\
&Coh(T, comp^2) :\equiv\\
&\quad (Comp_0^2 : (a_{01} : R_{01}\, a_0\, a_1)(a_{02} : R_{02}\, a_0\, a_2) \to T\, a_{01}\, a_{02}\, (comp_0^2\, a_{01}\, a_{02}))\\
&\times (Comp_1^2 : (a_{01} : R_{01}\, a_0\, a_1)(a_{12} : R_{12}\, a_1\, a_2) \to T\, a_{01}\, (comp_1^2\, a_{01}\, a_{12})\, a_{12})\\
&\times (Comp_2^2 : (a_{02} : R_{02}\, a_0\, a_2)(a_{12} : R_{12}\, a_1\, a_2) \to T\, (comp_2^2\, a_{02}\, a_{12})\, a_{02}\, a_{12})
\end{aligned}
$$

A truncated version of the Kan semisimplicial set model of weak MLTT has been formalised in Coq [2]. This model also validates functional extensionality and a univalent universe of small types.

A small type is modelled as a Kan semisimplicial type of level ≤ 1 (which is equivalent to a setoid), a context is modelled as a Kan semisimplicial type of level ≤ 2. A universe is defined as a context with isomorphisms as equalities.

References

[1] Thorsten Altenkirch, Conor Mcbride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–58. ACM, 2007.

[2] Bruno Barras, Thierry Coquand, and Simon Huber. A generalization of Takeuti-Gandy interpretation. `http://uf-ias-2012.wikispaces.com/file/view/semi.pdf`, 2013.

[3] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[4] James Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, January 2009.

[5] Thierry Coquand. About the setoid model. `http://www.cse.chalmers.se/~coquand/setoid.pdf`, 2013.

[6] Thierry Coquand and Simon Huber. Simplicial sets model of type theory. `http://www.cse.chalmers.se/~coquand/decuniv.pdf`, 2013.

[7] Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

[8] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, 1995.

[9] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[10] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations, 2012. arXiv:1211.2851.

[11] Per Martin-Löf. About models for intuitionistic type theories and the notion of de

nitional equality. In Stig Kanger, editor, *Proceedings of the Third Scandinavian Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 81–109. North-Holland, 1975.

[12] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[13] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[14] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013. Available online at homotopytypetheory.org/book.

Generalizations of Hedberg's Theorem Away Day Version*

Nicolai Kraus
joint work with
Martín Escardó, Thierry Coquand and Thorsten Altenkirch

June 2013

Abstract

As the groupoid model by Hofmann and Streicher shows, *uniqueness of identity proofs* (UIP) is not provable. Generalizing a theorem by Hedberg, we give new characterizations of types that satisfy UIP. It turns out to be natural in this context to consider constant endofunctions. For such a function, we can look at the type of its fixed points. We show that this type has at most one element, which is a nontrivial lemma in the absence of UIP. As an application, a new notion of anonymous existence can be defined. One further main result is that, if every type has a constant endofunction, then all equalities are decidable. All the proofs have been formalized in Agda.

1 Introduction

Although the identity types in Martin-Löf type theory (MLTT) are defined by one constructor refl and by one eliminator J that matches the constructor, the statement that every identity type has at most one inhabitant is not provable [9]. Thus, *uniqueness of identity proofs* (UIP), or, equivalently, *Streicher's axiom K* are principles that have to be assumed, and have often been assumed, as additional rules of MLTT. In recent years, there is a growing interest in type theory without these assumptions, in particular with the development of *Homotopy Type Theory* (HoTT) and *Univalent Foundations* (UF) - see [4] for a brief and [13] for a detailed introduction. While we do not use any axioms of HoTT or UF (other than those of standard MLTT), we make use of their notation and intuition. For a better understanding of our arguments, is useful to think of a type as a space, and a propositional equality proof as a path. Notation and some basic definitions are listed in Section 2.

As said above, we do not assume the principle of unique identity proofs. However, certain types do satisfy it naturally, and such types are often called *h-sets*. A sufficient condition for a type to be an h-set, given by Hedberg [8], is that it has decidable propositional equality. In Section 3, we analyze Hedberg's original argument, which consists of two steps:

*the original version was submitted to TLCA 2013

1. A type X is an h-set iff for all $x, y : X$ there is a constant map $x = y \to x = y$.

2. If X has decidable equality then such constant endomaps exist.

Here, we write $x = y$ for the identity type $\mathrm{Id}_X(x, y)$ of an implicitly given type X.

Decidable equality means that, for all x and y, we have $(x = y) + (x \neq y)$. Thus, a natural weakening is $\neg\neg$-*separated* equality,

$$\neg\neg(x = y) \to x = y,$$

which occurs often in constructive mathematics. In this case we say that the type X is *separated*. For example, going beyond MLTT, the reals and the Cantor space in Bishop mathematics and topos theory are separated. In MLTT, the Cantor type of functions from natural numbers to booleans is separated under the assumption of functional extensionality,

$$\forall\, f\, g\, : X \to Y,\, (\forall\, x\, : X,\, f\, x = g\, x) \to f = g.$$

We observe that under functional extensionality, a separated type X is an h-set, because there is always a constant map $x = y \to x = y$.

In order to obtain a further characterization of the notion of h-set, we consider *truncations* (also known as *bracket* or *squash* types), written $\|X\|$ in accordance with recent HoTT notation. The idea is to collapse all inhabitants of X so that $\|X\|$ has at most one inhabitant. We refer the reader to the technical development for a precise definition. We observe that

1′. A type X is an h-set iff $\|x = y\| \to x = y$ for all $x, y : X$,

and we mention a couple of other simple, but noteworthy, connections.

While Section 3 gives properties and arguments involving path spaces (i. e. equality types), we go beyond that in Section 4. Dealing with a path space opens up many possibilities that are not available for a general type. For that reason, we find it somewhat surprising that the equivalence of two of the above mentioned properties can be translated to general spaces, though that requires a nontrivial argument. This is done in Section 4:

> A type X satisfies $\|X\| \to X$ iff it has a constant endomap.

We find this interesting, as it says that from the anonymous existence of a point of X, that is, from the inhabitedness of $\|X\|$, one can get an inhabitant of X, provided a constant endomap is available. It is important here (and above) that our definition of constant function does not require X to be inhabited: we say that a function is constant if any two of its values are equal, and this may happen vacuously. The main technical lemma to prove this, which is noteworthy on its own right, is our Fixed Point Lemma:

> For any type X and any constant map $f : X \to X$, the type of fixed points of f is an h-proposition.

Here, an h-proposition is defined to be a type with at most one element. The proof of this lemma would be trivial if UIP was assumed, but in its absence, it is not.

Section 5 can, together with the just described results, be seen as the highlight of this paper. The assumption that every type has a constant endomap has an interesting status. It is not a constructive principle, but at the same time, it is seemingly weaker that typical classical statements. But this is only partially true: While we cannot make a strong conclusion for arbitrary types, such as excluded middle, we prove that the assumption implies that all equalities are decidable.

The just discussed section depends crucially on the Fixed Point Lemma, and so does Section 6: We describe how the lemma gives rise to another notion of anonymous existence, which we call *populatedness*. We say that X is populated, written $\langle\langle X \rangle\rangle$, if every constant endofunction on X has a fixed point. Unlike $\|X\|$, this new notion is thus defined internally, instead of using a postulate.

In our final Section 7, we discuss the relationship between the different notions of existence, starting with a chain of implications:

$$X \longrightarrow \|X\| \longrightarrow \langle\langle X \rangle\rangle \longrightarrow \neg\neg X.$$

We have formalized and proved all our statements in the dependently typed programming language Agda [3] and presented parts on the HoTT blog [1].

2 Preliminaries

We work in a standard version of Martin-Löf Type Theory with dependent sums, dependent function types and identity types. For the latter, we assume the eliminator J and, as it is standard, its computational β-rule, but not the definitional η-law. We further do not assume the eliminator K, or the principle of unique identity proofs. Summarized, our setting is very minimalistic. Sometimes, additional principles (*function extensionality* and *truncation*, as introduced later) are assumed, but this will be stated clearly.

We use standard notation whenever it is available. Regarding the identity types, we write, for two elements $a, b : A$, the expression $a = b$ for the type of *equality proofs*, or *paths* from a to b, keeping A implicit. Other common notations for the same thing are $a =_A b$, as well as $\mathrm{Id}(a, b)$ and $\mathrm{Id}_A(a, b)$. If $a = b$ is inhabited, it is standard to say that a and b are *propositionally equal*. In contrast, *definitional equality* is a meta-level concept, referring to two terms, rather than two (hypothetical) elements, with the same β (and, sometimes, η in a restricted sense) normal form. Recently, it has become standard to use the symbol $\equiv$ for definitional equality.

Propositional equality satisfies the *Groupoid Laws*: If we have $p : a = b$ and $q : b = c$, there is a canonical path $p \bullet q : a = c$ (the *composition* of p and q). Further, we have $p^{-1} : b = a$. There always is $\mathrm{refl}_a : a = a$, which behaves as a neutral element when composed with another path. Pairs of inverses cancel each other out when composed, and the obvious associativity law holds. In general, these statements are valid only up to propositional equality.

An important special case of the J eliminator is *substitution*, for which the name *transport* has been established in HoTT: If P is a family of types over A, and there are two elements (or *points*) $a, a' : A$, together with some $p : a = a'$, then a point $x : P(a)$ can be "transported along the path p" to get an element of $P(a')$:

$$\mathrm{transport}\, p\, x : P(a').$$

Another useful function, easily derived from the J eliminator, is the follwing: If we have a function $f : A \to B$ and a path $p : a = a'$ in A, we get a path of type $f(a) = f(a')$ in B:

$$\mathrm{ap}_f\ p : f(a) = f(a')$$

Our hope is that all of the notions in the following definition are as intuitive as possible, if not already known. The only notions that are not standard are *collapsible*, meaning that a type has a constant endomap, and *path-collapsible*, saying that every path space over the type is collapsible.

Definition 1. *We say that a type X is an* h-proposition *if all its inhabitants are equal:*

$$\mathrm{hprop}\, X \equiv \forall\, x\, y\ :X,\ x = y.$$

Further, X satisfies UIP *(uniqueness of identity proofs), or is an* h-set*, if its path spaces are all h-propositional:*

$$\text{h-set}\, X \equiv \forall\, x\, y\ :X,\ \mathrm{hprop}(x = y).$$

The property of being h-propositional or an h-set are all h-propositional themselves, which the following properties are not.

X is decidable *if it is either inhabited or empty:*

$$\mathrm{decidable}\, X \equiv X + \neg X.$$

We therefore say that X has decidable equality*, if the equality type of any two inhabitants of X is decidable:*

$$\mathrm{discrete}\, X \equiv \forall\, x\, y\ :X,\ \mathrm{decidable}(x = y).$$

Based on the terminology in [11], we also call a type with decidable equality discrete*.*

A function (synonymously, map) $f : X \to Z$ is constant *if it maps any two elements to the same inhabitant of Y:*

$$\mathrm{const}\, f \equiv \forall\, x\, y\ :X,\ f(x) = f(y).$$

We call a type X collapsible *if it has a constant endomap:*

$$\mathrm{coll}\, X \equiv \Sigma_{f:X\to X}\ \mathrm{const}\, f.$$

Finally, X is called path-collapsible *if any two points x, y of X have a collapsible path space:*

$$\text{path-coll}\, X \equiv \forall\, x\, y\ :X,\ \mathrm{coll}\,(x = y).$$

For some statements, but only if clearly indicated, we use *functional extensionality*. This principle says that two functions f, g of the same type are equal as soon as they are pointwise equal:

$$(\forall x\,,\ f\, x = g\, x) \to f = g.$$

An important equivalent formulation (see Voevodsky [14]) is that the set of h-propositions is closed under $\forall$. More precisely,

$$(\forall\, a\ :A,\ \mathrm{hprop}\, B)\ \to\ \mathrm{hprop}\,(\forall\, a\ :A,\ B)\,.$$

In the case of non-dependent function types, this can be read as follows: If B is h-propositional, then so is $A \to B$.

3 Hedberg's Theorem

Before discussing possible generalizations, we discuss Hedberg's Theorem.

Theorem 1 (Hedberg)**.** *Every discrete type has unique identity proofs,*

$$\text{discrete}\, X \to \text{h-set}\, X.$$

We shortly state Hedberg's original proof [8], consisting of two steps.

Lemma 1. *If a type has decidable equality, it is path-collapsible:*

$$\text{discrete}\, X \to \text{path-coll}\, X.$$

Proof. Given inhabitants x and y of X, the assumptions provide an inhabitant of $\text{decidable}(x = y) \equiv (x = y) + \neg(x = y)$. If it is an inhabitant of $x = y$, we construct the required constant map $(x = y) \to (x = y)$ by mapping everything to this path. If it is an inhabitant of $\neg(x = y)$, there is only a unique such map which is constant automatically. □

Lemma 2. *If a type is path-collapsible, it has unique identity proofs:*

$$\text{path-coll}\, X \to \text{h-set}\, X.$$

Proof. Assume f is a parametrized constant endofunction on the path spaces. Let p be a path from x to y. We claim that $p = (f\, p) \bullet (f\ \text{refl}_x)^{-1}$. Using the equality eliminator on (x, y, p), we only have to give a proof for the triple (x, x, refl_x), which is one of the groupoid laws that equality satisfies. Using the fact f is constant on every path space, the right-hand side expression is independent of p, and in particular, equal to any other path of the same type. □

Hedberg's proof [8] is just the concatenation of the two lemmas. A slightly more direct proof can be found in a post on the HoTT blog [10], and in the HoTT Coq repository [12]. The first of the two lemmas uses the rather strong assumption of decidable equality. In contrast, the assumption of the second lemma is equivalent its conclusion, which means that we cannot do much there. We include a proof of this simple claim in Theorem 2 below and concentrate on weakening the assumption of the first lemma. Let us first introduce the notions of *stability* and *separatedness*.

Definition 2. *For any type X, define*

$$\begin{aligned} \text{stable}\, X &\equiv \neg\neg X \to X, \\ \text{separated}\, X &\equiv \forall\, x\, y : X, \text{stable}(x = y). \end{aligned}$$

We can see $\text{stable}\, X$ as a *classical* condition, similar to $\text{decidable}\, X \equiv X + \neg X$, but strictly weaker. Indeed, we get a first strengthening of Hedberg's Theorem as follows:

Lemma 3. *If functional extensionality holds, any separated type has unique identity proofs,*

$$\text{separated}\, X \to \text{h-set}\, X.$$

Proof. There is, for any $x, y : X$, a canonical map $(x = y) \to \neg\neg(x = y)$. Composing this map with the proof that X is separated yields an endofunction on the path spaces. With functional extensionality, the first map has an h-propositional codomain, which implies that the endofunction is constant, fulfilling the requirements of lemma 2. □

We remark that full functional extensionality is actually not needed here. Instead, a weaker version that only works with the empty type is sufficient. Similar statements hold true for all further applications of extensionality in this paper. Details can be found in the Agda file [3].

In a constructive setting, the question how to express that "there exists something" in a type X is very subtle. One possibility is to ask for an inhabitant of X, but in many cases, this is stronger than one can hope. A second possibility, which corresponds to our above definition of *separated*, is to ask for a proof of $\neg\neg X$. Then again, this is very weak, and often too weak, as one can in general only prove negative statements from double-negated assumptions.

This fact has inspired the introduction of *squash types* (the Nuprl book [6]), and similar, *bracket types* (Awodey and Bauer [5]). These lie in between of the two extremes mentioned above. In our intensional setting, we talk of *h-propositional truncations*: For any type X, we postulate that there is a type $\|X\|$ that is an *h-proposition*, representing the statement that X is inhabited. The rules are that if we have a proof of X, we can, of course, get a proof of $\|X\|$, and from $\|X\|$, we can conclude the same statements as we can conclude from X, but only if the actual representative of X does not matter:

Definition 3. *For a given type $X : \boldsymbol{Type}$, we postulate the existence of a type $\|X\| : \boldsymbol{Type}$, satisfying the following properties:*

1. $\eta : X \to \|X\|$
2. $\mathrm{hprop}(\|X\|)$
3. $\forall\, P : \boldsymbol{Type}, \mathrm{hprop}\, P \to (X \to P) \to \|X\| \to P$.

We say that X is h-inhabited *if $\|X\|$ is inhabited.*

Note that this amounts to saying that the operator $\|\cdot\|$ is left adjoint to the inclusion of the subcategory of h-propositions into the category of all types. Therefore, it can be seen as the *h-propositional reflection*.

There is a type expression that is equivalent to h-inhabtedness:

Proposition 1. *For any given $X : \boldsymbol{Type}$, we have*

$$\|X\| \longleftrightarrow \forall\, P : \boldsymbol{Type}, \mathrm{hprop}\, P \to (X \to P) \to P.$$

The trouble with the expression on the right-hand side is that it is not living in universe **Type**. This size issue is really the only thing that is disturbing here, as the expression satisfies all the properties of the above definition, at least under the assumption of functional extensionality. Voevodsky [14] uses *resizing rules* to get rid of the problem.

Proof. The direction "$\to$" of the statement is not more than a rearrangement of the assumptions of property (3). For the other direction, we only need to instantiate P with $\|X\|$ and observe that the properties (1) and (2) in the definition of $\|X\|$ are exactly what is needed. □

With this definition at hand, we can provide an even stronger variant of Hedberg's Theorem. Completely analogous to the notions of stability and separatedness, we define *h-stable* and *h-separated*:

Definition 4. *For any type X, define*

$$\begin{aligned} \text{h-stable}\, X &\equiv \|X\| \to X, \\ \text{h-separated}\, X &\equiv \forall\, x\, y : X,\, \|x = y\| \to (x = y). \end{aligned}$$

In fact, h-separated X is a strictly weaker condition than separated X. Not only can we conclude h-set X from h-separated X, but even the converse. We also include the simple, but until here unmentioned fact that path-collapsibility is also equivalent to these statements:

Theorem 2. *For a type X in MLTT with h-propositional truncation, the following are equivalent:*

(i) X is an h-set.

(ii) X is path-collapsible.

(iii) X is h-separated.

Proof. *(ii)* $\Rightarrow$ *(i)* is just Lemma 2.

(i) $\Rightarrow$ *(iii)* uses simply the the definition of the h-propositional truncation: Given $x, y : X$, the fact that X is an h-set tells us exactly that $x = y$ is h-propositional, implying that we have a map $\|x = y\| \to (x = y)$.

Concerning *(iii)* $\Rightarrow$ *(ii)*, it is enough to observe that the composition of $\eta : (x = y) \to \|x = y\|$ and the map $\|x = y\| \to (x = y)$, provided by the fact that X is h-separated, is a parametrized constant endofunction. □

As a conclusion of this part of the paper, we observe that h-propositional truncation has some kind of extensionality built-in: In Lemma 3, we have given a proof for the simple statement that separated types are h-sets in the context of functional extensionality. This is not true in pure MLTT. Let us now drop functional extensionality and assume instead that h-propositional truncation is available. Every separated type is h-separated - more generally, we have

$$(\neg\neg A \to A) \to \|A\| \to A$$

for any type A -, and every h-separated space is an h-set. Notice that the mere availability of h-propositional truncation suffices to solve a gap that functional extensionality would usually fill.

4 Collapsibility implies H-Stability

If we unfold the definitions in the statements of Theorem 2, they all involve the path spaces over some type X:

(i) $\forall\, x\, y : X$, hprop$(x = y)$

(ii) $\forall\, x\, y : X$, coll $(x = y)$

(iii) $\forall\, x\, y : X$, h-stable$(x = y)$.

We have proved that these statements are logically equivalent. It is a natural question to ask whether the properties of path spaces are required. The possibilities that path spaces offer are very powerful and we have used them heavily. Indeed, if we formulate the above properties for an arbitrary type A instead of path types

(i') $\mathrm{hprop}(A)$

(ii') $\mathrm{coll}\,(A)$

(iii') h-stable A,

we notice immediately that (i') is significantly and strictly stronger than the other two properties. (i') says that A has at most one inhabitant, (ii') says that there is a constant endofunction on A, and (iii') gives us a possibility to get an explicit inhabitant of A from the proposition that A has an anonymous inhabitant. An h-propositional type has the other two properties trivially, while the converse is not true. In fact, as soon as we know an inhabitant $a : A$, we can very easily construct proofs of (ii') and (iii'), while it does not help at all with (i').

The implication $(iii') \Rightarrow (ii')$ is also simple: If we have $h : \|A\| \to A$, the composition $h \circ \eta : A \to A$ is constant, as for any $a, b : A$, we have $\eta(a) = \eta(b)$ and therefore $h(\eta(a)) = h(\eta(b))$.

In summary, we have $(i') \Rightarrow (iii') \Rightarrow (ii')$ and we know that the first implication cannot be reversed. What is less clear is the reversibility of the second implication: If we have a constant endofunction on A, can we get a map $\|A\| \to A$? Put differently, what does it take to get out of $\|A\|$? Of course, a proof that A is h-stable is fine for that, but does a constant endomap on A also suffice? Surprisingly, the answer is positive, and there are interesting applications (Section 6). The main ingredient of our proof, and of much of the rest of the paper, is the following crucial lemma about fixed points:

Lemma 4 (Fixed Point Lemma)**.** *Given a constant endomap f on a type X, the type of fixed points is h-propositional, where this type is defined by*

$$\mathrm{fix}\, f \equiv \Sigma_{x:X}\, x = f(x).$$

Before we can give the proof, we first need to formulate two observations. Both of them are simple on their own, but important insights for the Fixed Point Lemma. Let X and Y be two types.

Proposition 2. *Assume $h, k : X \to Y$ are two functions and $t : x = y$ as well as $p : h(x) = k(x)$ are paths. Then, substituting along t into p can be expressed as a composition of paths:*

$$(\mathrm{transport}\ t\, p) = \left((\mathrm{ap}_h\ t)^{-1} \bullet p \bullet (\mathrm{ap}_k\ t)\right).$$

Proof. This is immediate if t is the trivial reflexivity path, i.e. if (x, y, t) is just (x, x, refl_x), and for all other cases, it follows as a direct application of the equality eliminator J. □

Even if the latter proof is trivial, the statement is essential. In the proof of Lemma 4, we need a special case, were x and y are the same. However, this special version cannot be proved directly. We consider the second observation the key insight for the Fixed Point Lemma:

Proposition 3. *If $f : X \to Y$ is constant and $x : X$ some point, then* ap_f *maps every path between x and x to* $\mathrm{refl}_{f(x)}$*, up to propositional equality.*

Proof. It is not possible to prove this directly. Instead, we state a slight generalization: If c is the proof of $\mathrm{const}\, f$, then ap_f maps a path $p : x = y$ to $(c\, x\, x)^{-1} \bullet c\, x\, y$. This is easily seen to be correct for (x, x, refl_x), which is enough to apply the eliminator. As the expression is independent of p, but only depends on its endpoints, it is for $p : x = x$ equal to $\mathrm{refl}_{f(x)}$, as claimed. Note that the proposition can also be stated as: For all x and y, the function $\mathrm{ap}_f\ x\, y : (x = y) \to (f\, x = f\, y)$ is constant. □

With these lemmas at hand, the rest is fairly simple:

of the Fixed Point Lemma. Assume $f : X \to X$ is a function and $c : \mathrm{const}\, f$ is a proof that it is constant. For any two pairs (x, p) and $(x', p') : \mathrm{fix}\, f$, we need to construct a path connection them.

First, we simplify the situation by showing that we can assume that x and x' are the same: By composing $p : x = f\, x$ with $c\, x\, x' : f\, x = f\, x'$ and $(p')^{-1} : f\, x' = x'$, we get a path $p'' : x = x'$. A path between two pairs corresponds to two paths: One path between the first components, and one between the second, where a substitution along the first path is needed. We therefore now get that $(x, \mathrm{transport}\ (p'')^{-1}\, p')$ and (x', p') are propositionally equal: p'' is a path between the first components, which makes the second component trivial. Write q for the term $\mathrm{transport}\ (p'')^{-1}\, p'$.

We are now in the (nicer) situation that we have to construct a path between (x, p) and $(x, q) : \mathrm{fix}\, f$. Again, such a path has to consist of two paths, for the two components. Let us assume that we use some path $t : x = x$ for the first component. We then have to show that transport $t\, p$ equals q. In the situation with (x, p) and (x', p'), it might have been tempting to use p'' as a path between the first components, and that would correspond to choosing refl_x for t. However, one quickly convinces oneself that this cannot work in the general case.

By Proposition 2, with the identity for h and f for k, the first of the two terms, i. e. transport $t\, p$, corresponds to $t^{-1} \bullet p \bullet \mathrm{ap}_f\ t$. With Proposition 3, that term can be further simplified to $t^{-1} \bullet p$. What we have to prove is now just $\left(t^{-1} \bullet p\right) = q$, so let us just choose $q \bullet p^{-1}$ for t, thereby making it into a straight-forward application of the standard lemmas. □

We are now finally in the position to prove the statement that is announced in Section 4:

Theorem 3. *A type A is collapsible, i.e. has a constant endomap, iff it is h-stable in the sense that $\|A\| \to A$.*

Proof. As already mentioned in Section earlier, the "if-part" is simple: If there is a map $\|A\| \to A$, we just need to compose it with $\eta : A \to \|A\|$ to get a constant endomap on A.

For the other direction, let c be the proof that f is constant, just as before. Observe that we have $A \to \mathrm{fix}\, f$ by mapping a on $(f\, a, c\, a\, (f\, a))$. As $\mathrm{fix}\, f$ is an h-proposition by the previous lemma, we get a map $\|A\| \to \mathrm{fix}\, f$ by the elimination rule for h-propositional truncation. That map can be composed

with the first projection of type $\mathrm{fix}\, f \to A$, yielding a function $\|A\| \to A$ as required. □

Looking at the just proved theorem, it makes sense to ask the following question: Given a constant function $f : A \to B$, is it possible to construct a function $\overline{f} : \|A\| \to B$? We can do that if B is an h-set. For the general case, we have evidence that the answer is likely to be negative.

5 Global Collapsibility implies Decidable Equality

If X is some type, having a proof of $\|X\|$ is, intuitively, much weaker than a proof of X. While the latter consists of a concrete element of X, the first is given by an *anonymous* inhabitant of X. This is actually nothing more than the intention of the truncation: $\|X\|$ allows us to make the statement that "there exists something in X", without giving away a concrete element. It is therefore unreasonable to suppose that

$$\forall\, X\ :\mathbf{Type},\, \|X\| \to X,$$

can be proved, but it is interesting to consider what it would imply. Using Theorem 3, the above type is logically equivalent to the statement

Every type has a constant endomap.

From a constructive type of view, this is an interesting statement. It clearly follows from the *Principle of Excluded Middle*, $\forall\, X\ :\mathbf{Type},\, X + \neg X$: If we know an inhabitant of a type, we can immediately construct a constant endomap, and for the empty type, considering the identity function is sufficient. Thus, we understand *"Every type has a constant endomap"* as a weak form of the excluded middle: It seems to use that every type is either empty or inhabited, but there is no way of knowing in which case we are. We are unable to show that it implies excluded middle.

However, what we can conclude is excluded middle for all path spaces. We can prove the following statement in basic MLTT, without h-propositional truncation, without extensionality, and even without a universe:

Lemma 5. *Let A be a type and $a_0, a_1 : A$ two points. If for all $x : A$ the type $(a_0 = x) + (a_1 = x)$ is collapsible, then $a_0 = a_1$ is decidable.*

Before giving the proof, we state an immediate corollary:

Theorem 4. *If every type has a constant endomap (equivalently, is h-stable), then every type has decidable equality.*

of Lemma 5. Let us define $E_x \equiv (x = a_0) + (x = a_1)$. The assumption says that we have a family of endomaps $f_x : E_x \to E_x$, together with proofs of their constancy $c_x : \mathrm{const}\, f_x$. We show that the identity map on $\Sigma_{x:A}\, \mathrm{fix}\, f_x$ factorizes pointwise through **Bool**. Note that an element of $\Sigma_{x:A}\, \mathrm{fix}\, f_x$ is a pair of an $x : A$ and a point in $\mathrm{fix}\, f_x$; and such a point consists itself of a pair (c, p), where $c : E_x$ and $p : c = f_x(c)$. There is a canonical inhabitant of $\mathrm{fix}\, f_{a_0}$, given by

$f_{a_0}(\mathrm{inl}\,\mathrm{refl}_{a_0})$ for the first component, and $c_{a_0}\,(\mathrm{inl}(\mathrm{refl}_{a_0}))\,(f_{a_0}(\mathrm{inl}(\mathrm{refl}_{a_0}))$ for the second. We call it k_0, and analogously, we write k_1 for the canonical inhabitant of $\mathrm{fix}\,f_{a_1}$.

$$\begin{array}{rcl@{\qquad}rcl} r : \Sigma_{x:A}\,\mathrm{fix}\,f_x & \to & \mathbf{Bool} & s : \mathbf{Bool} & \to & \Sigma_{x:A}\,\mathrm{fix}\,f_x \\ (x, (\mathrm{inl}\,q, p)) & \mapsto & \mathrm{true}, & \mathrm{true} & \mapsto & (a_0, k_0), \\ (x, (\mathrm{inr}\,q, p)) & \mapsto & \mathrm{false}, & \mathrm{false} & \mapsto & (a_1, k_1). \end{array}$$

We claim that any pair (x, k) is equal to $s \circ r(x, k)$. An equality of pairs corresponds to a pair of equalities. As the second component is, by the Fixed Point Lemma, an equality over an h-propositional type, it is enough to show that x equals the first component of $s \circ r(x, k)$. Let k be (c, p). We can now perform case analysis on c: If c is of the form $\mathrm{inl}\,q$, we need to prove $x = a_0$; but this is shown by q. If c is $\mathrm{inr}\,q$, we proceed analogously. Therefore, equality of any two such pairs is decidable, as we just have to check whether r maps them to the same value in **Bool**.

Again because $\mathrm{fix}\,f_x$ is an h-proposition, the pairs (a_0, k_0) and (a_1, k_1) are equal iff $a_0 = a_1$, and, therefore, $a_0 = a_1$ is decidable. □

6 Populatedness

In this section we discuss a notion of *anonymous existence*, similar, but weaker (see Section 7.2) than h-propositional truncation. It crucially depends on the Fixed Point Lemma 4. Let us start by discussing another perspective of what we have explained in the previous section.

Trivially, for any type X, we can prove the statement

$$\|X\| \to (\|X\| \to X) \to X. \tag{1}$$

By Lemma 3, this is equivalent to

$$\|X\| \to \mathrm{coll}\,X \to X, \tag{2}$$

which can be read as: If we have a constant endomap on X and we wish to get an inhabitant of X (or, equivalently, a fixed point of the endomap), then $\|X\|$ is sufficient to do so. Now, we can ask whether it is also necessary: Can we replace the first assumption $\|X\|$ by something weaker? Looking at formula 1, it would be natural to conjecture that this is not the case, but it is. In this section, we discuss by what it can be replaced, and in Section 7.2, we give a proof that it is indeed weaker.

For answering the question what is needed to get from h-stable A to A, let us define the following notion:

Definition 5 (populatedness)**.** *For a given type X, we say that X is populated, written $\langle\langle X \rangle\rangle$, if every constant endomap on X has a fixed point:*

$$\langle\langle X \rangle\rangle \equiv \forall\, f\, : X \to X, \mathrm{const}\, f \to \mathrm{fix}\, f,$$

where $\mathrm{fix}\, f$ is the type of fixed points, defined as in Lemma 4.

This definition allows us to comment on the question risen above. If $\langle\langle X \rangle\rangle$ is inhabited and X is collapsible, then X has an inhabitant, as such an inhabitant

can be extracted from the type of fixed points by projection. Hence, $\langle\langle X \rangle\rangle$ instead of $\|X\|$ in 2 would be sufficient as well (we discuss in Section 7 whether it is weaker). Therefore,

$$\langle\langle X \rangle\rangle \to (\|X\| \to X) \to X.$$

Next we draw a parallel between populatedness and h-inhabitedness.

Theorem 5. *For any given X : **Type**, the following holds:*

$$\langle\langle X \rangle\rangle \longleftrightarrow \forall\, P : \textbf{\textit{Type}},\, \mathrm{hprop}\, P \to (P \to X) \to (X \to P) \to P.$$

This statement can be read as "X is populated iff every h-proposition logically equivalent to X is inhabited." Note that the only difference to the type expression in Proposition 1 is that we only quantify over *sub-propositions* of X, i. e. over those that satisfy $P \to X$, while we quantify over all propositions in the case of $\|X\|$. Therefore, $\|X\|$ is clearly at least as strong as $\langle\langle X \rangle\rangle$.

Proof. Let us first prove the direction "$\to$". Assume an h-propositional P is given, together with functions $X \to P$ and $P \to X$. Composition of these gives us a constant endomap on X, exactly as in the proof of Theorem 2. But then $\langle\langle X \rangle\rangle$ makes sure that this constant endomap has a fixed point, which is (or allows us to extract) an inhabitant of X. Using $X \to P$ again, we get P.

For the direction "$\leftarrow$", assume we have a constant endomap f. We need to construct an inhabitant of $\mathrm{fix}\, f$. In the expression on the right-hand side, choose P to be $\mathrm{fix}\, f$. By the Fixed Point Lemma, this is an h-proposition. Further, P and X are logically equivalent (i.e. there are maps in both directions), where the non-trivial direction makes use of Theorem 3. Then, the right-handed expression shows P, which is just the required $\mathrm{fix}\, f$. □

This proof uses the Fixed Point Lemma twice: Once, as we needed P to be an h-proposition, and once hidden, as we used Theorem 3.

The similarities between $\|X\|$ and $\langle\langle X \rangle\rangle$ do not stop here. The following statement, together with the direction "$\to$" of the theorem that we have just proved, is worth to be compared to the definition of $\|X\|$ (that is, Definition 3):

Proposition 4. *For any type X, the type $\langle\langle X \rangle\rangle$ has the following properties:*

(1) $X \to \langle\langle X \rangle\rangle$

(2) $\mathrm{hprop}(\langle\langle X \rangle\rangle)$ *(if functional extensionality holds).*

The proof is fairly simple, and, of course, again an application of the Fixed Point Lemma.

Proof. Regarding (1), given $x : X$ and a constant endomap f, we need to prove that f has a fixed point. We just take $f\, x$ and use the fact that $f\, x$ is propositionally equal to $f(f\, x)$, by constancy of f.

For (2), we need to use that $\mathrm{fix}\, f$ is an h-proposition, by Lemma 4. By functional extensionality, a (dependent) function type is h-propositional if the codomain is (see Section 2) and we are done. □

7 Taboos and Counter-Models

In this final section we look at the differences between the various notions of (anonymous) inhabitedness we have encountered. We have, for any type X, the following chain of implications:

$$X \longrightarrow \|X\| \longrightarrow \langle\langle X \rangle\rangle \longrightarrow \neg\neg X.$$

The first implication is trivial and the second has already been mentioned after Theorem 5. Maybe somewhat surprisingly, the last implication does not require functional extensionality, as we do not need to prove that $\neg\neg X$ is h-propositional: To show

$$\langle\langle X \rangle\rangle \to \neg\neg X \ ,$$

let us assume $f : \neg X$. But then, f can be composed with the unique function from the empty type into X, yielding a constant endomap on X, and obviously, this function does not have a fixed point. Therefore, the assumption of $\langle\langle X \rangle\rangle$ would lead to a contradiction, as required.

Intuitively, none of the implications should be reversible. To make that precise, we use two techniques: Taboos, showing that the provability of a statement would imply the provability of another, better understood statement, that is known to be not provable. As the second technique, we use HoTT models.

1. Theorem 4 shows that, if the first implication can be reversed, then all types have decidable equality. Using Hedberg's Theorem, this immediately implies that every type is an h-set, and thus, it is inconsistent with the Univalence Axiom of HoTT. But the conclusion that every type is an h-set can be derived much more directly: If we assume $\|X\| \to X$ for all types X, we have this in particular for all path spaces. Then, by Theorem 2, every type is an h-set.

   As an alternative argument, if every type is h-stable, a form of choice that does not belong to type theory is implied.

2. It would be wonderful if the second implication could be reversed, as this would imply that h-propositional truncation is definable in MLTT. However, this is equivalent to a certain h-propositional axiom of choice discussed below, which is not provable but holds under excluded middle.

3. If the last implication can be reversed, excluded middle for h-propositions holds (a constructive taboo, which is not valid in recursive models).

7.1 Inhabited and H-Inhabited

The question whether the first implication in the chain above can be reversed has already been analyzed in Section 5. This cannot be possible as long as equality is not globally decidable. Here, we want to state another noteworthy consequence of

$$\forall\, X \,:\, \mathbf{Type},\, \|X\| \to X.$$

In [2], we show that this assumption allows us to show that any relation has a functional subrelation with the same domain. This is a form of the axiom of

choice that does not pertain to intuitionistic type theory. Here, we only sketch the proof. Given a binary relation A on the type X. Define

$$A_x \equiv \Sigma_{y:X}\, A(x,y), \qquad F(x,y) \equiv \Sigma_{a:A(x,y)}\, (y,a) = k_x(y,a),$$

where $k_x : A_x \to A_x$ is the constant map induced by the hypothesis $\|A_x\| \to A_x$. By the Fixed Point Lemma, $F(x,y)$ is an h-proposition. If $(a,p) : F(x,y)$ and $(a',p') : F(x,y')$, then

$$(y,a) = k_x(y,a) = k_x(y',a') = (y',a')$$

because k_x is constant and hence $y = y'$, and so F is single-valued. But in fact, with a subtler argument, it is single-valued in the stronger sense that F_x is an h-proposition. Moreover, F has the same domain as A in the sense that F_x is inhabited iff A_x is inhabited.

7.2 H-Inhabited and Populated

Assume that the second implication can be reversed, meaning that we have

$$\forall\, X : \mathbf{Type},\, \langle\langle X \rangle\rangle \to \|X\|.$$

Repeated use of the Fixed Point Lemma leads to a couple of interesting equivalent statements. We discuss one that is particularly interesting: Every populated type is h-inhabited iff for every type, the statement that it is h-stable is h-inhabited.

In the previous subsection, we have discussed that we cannot prove the statement that every type is h-stable. However, we can always populate it:

Lemma 6. $\forall\, X : \textbf{\textit{Type}},\, \langle\langle \|X\| \to X \rangle\rangle$.

Proof. Assume we are given a constant endomap f on h-stable X. We need to construct a fixed point of that endomap, which amounts to construction an inhabitant of h-stable X. By the Fixed Point Lemma, a constant endomap $g : X \to X$ is enough for this. From f, we can construct g easily: Given $x : X$, we get a canonical inhabitant of h-stable X. We apply f on this inhabitant, and we apply the result on $\eta(x)$, yielding an inhabitant of X. We define $g\,x$ to be this inhabitant. It is easy to see that g is constant. □

An alternative proof is available in the Agda file.

Theorem 6. *The implication $\|X\| \to \langle\langle X \rangle\rangle$ can always be reversed iff the statement that that a type is h-stable can always be h-inhabited:*

$$(\forall\, X : \textbf{\textit{Type}},\, \langle\langle X \rangle\rangle \to \|X\|) \longleftrightarrow (\forall\, X : \textbf{\textit{Type}},\, \|\|X\| \to X\|)\,.$$

Proof. The direction "→" is an immediate application of Lemma 6 above. The other direction is slightly trickier: If we knew h-stable X, we would have a constant endomap on X, and with the assumption $\langle\langle X \rangle\rangle$, this constant endomap would have a fixed point. Hence, we would have an inhabitant of X, and therefore and inhabitant of $\|X\|$. We observe that $\|X\|$ is h-propositional, so, by definition, we do not necessarily need h-stable X, but $\|$ h-stable $X\|$ is enough, and that completes the proof. □

It is also easy to see (cf. our Agda file [3]) that

$$\langle\langle X \rangle\rangle \quad \longleftrightarrow \quad \|\|X\| \to X\| \to \|X\|,$$

which gives an alternative route to the above theorem. Moreover, the statement $\forall\, X\ :\ \mathbf{Type},\ \|\|X\| \to X\|$ is equivalent to the *h-propositional axiom of choice*: For every h-proposition P and any family $Y : P \to \mathbf{Type}$,

$$(\forall\, p\ :\ P,\ \|Yp\|) \to \|\forall\, p\ :\ P,\ Yp\|,$$

which clearly holds under h-propositional excluded middle. When Yp is a set with exactly two elements for every $p : P$, this amounts to *the world's simplest axiom of choice* [7], which fails in some toposes. Thus, by the above theorem, $\forall\, X\ :\ \mathbf{Type},\ \langle\langle X \rangle\rangle \to \|X\|$ is not provable.

7.3 Populated and Non-Empty

If we can reverse the last implication of the chain, we have

$$\forall\, X\ :\ \mathbf{Type},\ \neg\neg X \to \langle\langle X \rangle\rangle.$$

To show that this is not provable, we prove that it is a taboo from the point of view of constructive mathematics, in the sense that it implies Excluded Middle for h-propositions,

$$\text{hprop-EM} \equiv \forall P\,,\ \text{hprop}\, P \to P + \neg P.$$

Lemma 7. *With functional extensionality, the following implication holds:*

$$(\forall\, X\ :\ \textbf{\textit{Type}},\ \neg\neg X \to \langle\langle X \rangle\rangle)\ \to\ \text{hprop-EM}\,.$$

Proof. Assume P is an h-proposition. Then so is the type $P + \neg P$ (where we require functional extensionality to show that $\neg P$ is an h-proposition). Hence, the identity function on $P + \neg P$ is constant.

On the other hand, it is straightforward to construct a proof of $\neg\neg\,(P + \neg P)$. By the assumption, this means that $P + \neg P$ is populated, i.e. every constant endomap on it has a fixed point. Therefore, we can construct a fixed point of the identity function, which is equivalent to proving $P + \neg P$. □

Acknowledgments.

The first-named author would like to thank Paolo Capriotti, Ambrus Kaposi, Nuo Li and especially Christian Sattler for interesting discussions and technical assistance.

References

[1] T. Altenkirch, T. Coquand, M. Escardó, and N. Kraus. On h-propositional reflection and hedbergs theorem, November 2012. Blog post at homotopy-typetheory.org.

[2] T. Altenkirch, T. Coquand, M. Escardó, and N. Kraus. Constant choice (Agda file), 2012/2013. Available at the third-named author's institutional webpage.

[3] T. Altenkirch, T. Coquand, M. Escardó, and N. Kraus. Generalizations of Hedberg's theorem (Agda file), 2012/2013. Available at the third-named author's institutional webpage.

[4] S. Awodey. Type theory and homotopy. Technical report, 2010.

[5] S. Awodey and A. Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

[6] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nurpl Proof Development System*. Prentice-Hall, NJ, 1986.

[7] M. P. Fourman and A. Ščedrov. The "world's simplest axiom of choice" fails. *Manuscripta Math.*, 38(3):325–332, 1982.

[8] M. Hedberg. A coherence theorem for Martin-Löf's type theory. *J. Functional Programming*, pages 413–436, 1998.

[9] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[10] N. Kraus. A direct proof of Hedberg's theorem, March 2012. Blog post at homotopytypetheory.org.

[11] R. Mines, F. Richman, and W. Ruitenberg. *A Course in constructive algebra*. Universitext. Springer-verlag, New York, 1988.

[12] The HoTT and UF community. HoTT github repository. Available online.

[13] Univalent Foundations Program, IAS. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.

[14] V. Voevodsky. Coq library. Availabe at the author's institutional webpage.

An Implementation of Syntactic Weak ω-Groupoids in Agda

Li Nuo

June 12, 2013

Abstract

In Homotopy Type Theory, a variant of Martin-Löf Type Theory, we reject proof-irrelevance so that the common interpretation of types as setoids has to be generalised. With the univalence axiom, we treat equivalence as equality and interpret types as ω-groupoids. Inspired by Altenkirch's work [3] and Brunerie's notes [6], we study and implement a syntactic definition of Grothendieck weak ω-groupoids in Agda which is a popular variant of Martin-Löf Type Theory and a famous theorem prover. It is the first step to model type theory with weak ω-groupoids so that we could eliminate the univalence axiom.

1 Introduction

In Type Theory, a type could be interpreted as a setoid which is a set equipped with an equivalence relation [1]. The equivalence proof of the relation consists of reflexivity, symmetry and transitivity whose proof terms namely inhabitants are unique. However in Homotopy Type Theory, we reject the principle of uniqueness of identity proofs (UIP). Instead we accept the univalence axiom which says that equality of types is weakly equivalent to weak equivalence. Weak equivalence can been seen as a refinement of isomorphism without UIP [3]. To make it more precise, a weak equivalence between two objects A and B in a 2-category is a morphism $f : A \to B$ which has a corresponding inverse morphism $g : B \to A$, but instead of the proofs of isomorphism $f \circ g = 1_B$ and $g \circ f = 1_A$ we have two 2-cell isomorphisms $f \circ g \cong 1_B$ and $g \circ f \cong 1_A$.

It has been proved by Vladimir Voevodsky that the univalence axiom implies functional extensionality (a coq proof of this could be found in [5]) which results in the problem of non-canonical terms in Type Theory. Altenkirch has proposed a solution in [1] to solve the problem caused by functional extensionality based on setoid model and also refines it [2] to Observational Type Theory to justify functional extensionality. However as mentioned before, setoids require UIP which is incompatible with Homotopy Type Theory. To solve the problem we should generalise the notion of setoids, namely to enrich the structure of the identity proofs.

The generalised notion is called Grothendieck ω-groupoids. Grothendieck introduced the notion of ω-groupoids in 1983 in a famous Manuscript *Pursuing Stacks* [7]. Maltsiniotis continued his work and suggested a simplification of the original definition wihch can be found in [8]. Later Ara also present a slight variation of the simplication of weak ω-groupoids in [4]. Categorically speaking an ω-groupoid is an ω-category in which morphisms on all levels are equivalences. As we know that a set can be seen as a discrete category, a setoid is a category where every morphism is unique between two objects. A groupoid is more generalised, every morphism is isomorphism but the proof of isomorphism is unique, namely the composition of a morphism with its inverse is equal to an identity morphism. Similarly, an n-groupoid is an n-category in which morphisms on all levels are equivalence. ω-groupoids which are also called ∞-groupoids is an infinite version of n-groupoids. To model Type Theory without UIP we also require the equalities to be non-strict, in other words, they are not definitionally equalities. Finally we should use weak ω-groupoids to interpret types and eliminate the univalence axiom.

There are several approaches to formalise weak ω-groupoids in Type Theory. For instance, Altenkirch [3], and Bruneries' notes [6]. This paper mainly explains an implementation of weak ω-groupoids following Brunerie's approach in Agda which is a well-known theorem prover and also a variant of intensional Martin-Löf Type Theory. The approach is to specify when a globular set is a weak ω-groupoid by first defining a type theory called $\tau_{\infty-groupoid}$ to describe the internal language of Grothendieck weak ω-groupoids, then interpret it with a globular set and a dependent function. All coherence laws of the weak ω-groupoids should be derivable from the syntax, we will present some basic ones, for example reflexivity. One of the main contribution of this paper is to use the heterogeneous equality for terms to overcome some very difficult problems when we used the normal homogeneous one. In this paper, we omit some complicated and less important programs, namely the proofs of some lemmas or the definitions of some auxiliary functions. it is still possible for the reader who is interested in the details to check the code online, in which there are only some minor differences.

2 Syntax

Since the definitions of contexts, types and terms involve each others, we adopt a more liberal way to do mutual definition in Agda which is a feature available since version 2.2.10. Something declared is free to use even it has not been completely defined.

Basic Objects We first declare the syntax of our type theory which is called $\tau_{\infty-groupoid}$ namely the internal language of weak ω-groupoids. The following declarations in order are contexts as sets, types are sets dependent on contexts, terms and variables are sets dependent on types, Contexts morphisms and the contractible contexts.

```
data Con            : Set
data Ty (Γ : Con)   : Set
data Tm             : {Γ : Con}(A : Ty Γ) → Set
data Var            : {Γ : Con}(A : Ty Γ) → Set
data _⇒_            : Con → Con → Set

data isContr        : Con → Set
```

Altenkirch also suggests to use Higher Inductive-Inductive definitions for these sets which he coined as Quotient Inductive-Inductive Types (QIIT), in other words, to given an equivalence relation for each of them as one constructor. However we do not use it here.

It is possible to complete the definition of contexts and types first. Contexts are inductively defined as either an empty context or a context with a type of it. Types are defined as either $*$ which we call it 0-cell, or a morphism between two terms of some type A. If the type A is n-cell then we call the morphism $n+1$-cell.

```
data Con where
  ε   : Con
  _,_ : (Γ : Con)(A : Ty Γ) → Con


data Ty Γ where
  *   : Ty Γ
  _=h_ : {A : Ty Γ}(a b : Tm A) → Ty Γ
```

Heterogeneous Equality for Terms One of the big challenge we encountered at first is the difficulty to formalise and to reason about the equalities of terms. When we used the common identity types which is homogeneous, we had to use *subst* function in Agda to unify the types on both sides of the equation. It created a lot of technical issues that made the encoding too involved to proceed. However we found that the syntactic equality of types of given context which will be introduced later, is decidable which means that it is an h-set. In other words, the equalities of types is unique, so that it is safe to use the JM equality (heterogeneous equality) for terms of different types. The equality is inhabited only when they are definitionally equal.

```
data _≅_ {Γ : Con}{A : Ty Γ}
      : {B : Ty Γ} → Tm A → Tm B → Set where
  refl : (b : Tm A) → b ≅ b
```

Once we have the heterogeneous equality for terms, we could define a proof-irrelevant substitution which we call coercion here since it gives us a term of type A if we have a term of type B and the two types are equal. We can also prove that the coerced term is heterogeneously equal to the original term. Combined these definitions, it is much more convenient to formalise and to reason about term equations.

```
_⟦_⟫ : {Γ : Con}{A B : Ty Γ}(a : Tm B) → A ≡ B → Tm A
a ⟦ refl ⟫ = a

cohOp : {Γ : Con}{A B : Ty Γ}{a : Tm B}(p : A ≡ B)
  → a ⟦ p ⟫ ≅ a
cohOp refl = refl _
```

Substitutions With context morphism, we could define substitutions for types variables and terms. Indeed the composition of contexts can be understood as substitution for context morphisms as well.

```
_[_]T  : {Γ Δ : Con}(A : Ty Δ)   (δ : Γ ⇒ Δ) → Ty Γ
_[_]V  : {Γ Δ : Con}{A : Ty Δ}(a : Var A)(δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_[_]tm : {Γ Δ : Con}{A : Ty Δ}(a : Tm A) (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_◎_    : {Γ Δ Θ : Con}   →   Δ ⇒ Θ → Γ ⇒ Δ → Γ ⇒ Θ
```

Weakening Rules we could freely add types to the contexts of given any type judgments, term judgments or context morphisms. We call these rules weakening rules.

```
_+T_   : {Γ : Con}(A : Ty Γ)   → (B : Ty Γ) → Ty (Γ , B)
_+tm_ : {Γ : Con}{A : Ty Γ}(a : Tm A) → (B : Ty Γ) → Tm (A +T B)
_+S_   : {Γ : Con}{Δ : Con}(δ : Γ ⇒ Δ) → (B : Ty Γ) → (Γ , B) ⇒ Δ
```

To define the variables and terms we have to use the weakening rules. A Term can be either a variable or a J-term. We use the unnamed way to define variables as either the immediate variable at the right most of the context, or some variable in the context which can be found by cancelling the right most variable along with each vS. The J-terms are one of the major part of this syntax, which are primitive terms of the primitive types in contractible contexts which will be introduced later. Since contexts, types, variables and terms are all mutually defined, most of the properties of them have to be proved simultaneously as well.

```
data Var where
  v0 : {Γ : Con}{A : Ty Γ}   → Var (A +T A)
```

```
    vS : {Γ : Con}{A B : Ty Γ}(x : Var A) → Var (A +T B)

data Tm where
  var : {Γ : Con}{A : Ty Γ} → Var A → Tm A
  JJ  : {Γ Δ : Con} → isContr Δ → (δ : Γ ⇒ Δ) → (A : Ty Δ)
      → Tm (A [ δ ]T)
```

Another core part of the syntactic framework is contractible contexts. Intuitively speaking, a context is contractible if its geometric realization is contractible to a point. It either contains one variable of the 0-cell $*$ which is the base case, or we can extend a contractible context with a variable of an existing type and an n-cell, namely a morphism, between the new variable and some existing variable.

```
data isContr where
  c*  : isContr (ε , *)
  ext : {Γ : Con}
      → isContr Γ → {A : Ty Γ}(x : Var A)
        → isContr ((Γ , A) , (var (vS x) =h var v0))
```

Context morphisms are defined inductively similar to contexts. A context morphism is a list of terms corresponding to the list of types in the context on the right hand side of this morphism.

```
data _⇒_ where
  •   : {Γ : Con} → Γ ⇒ ε
  _,_ : {Γ Δ : Con}(δ : Γ ⇒ Δ){A : Ty Δ}(a : Tm (A [ δ ]T))
      → Γ ⇒ (Δ , A)
```

Lemmas The following four lemmas state that to substitute a type, a variable, a term, or a context morphism with two context morphisms consecutively, is equivalent to substitute with the composition of substitution.

```
[◎]T : {Γ Δ Θ : Con}
  {ϑ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}{A : Ty Θ}
  → A [ ϑ ◎ δ ]T ≡ (A [ ϑ ]T)[ δ ]T

[◎]v : {Γ Δ Θ : Con}
     (ϑ : Δ ⇒ Θ)(δ : Γ ⇒ Δ)(A : Ty Θ)(x   : Var A)
     → x [ ϑ ◎ δ ]V ≅ (x [ ϑ ]V) [ δ ]tm

[◎]tm : {Γ Δ Θ : Con}
  (ϑ : Δ ⇒ Θ)(δ : Γ ⇒ Δ)(A : Ty Θ)(a : Tm A)
  →    a [ ϑ ◎ δ ]tm ≅ (a [ ϑ ]tm) [ δ ]tm
```

```
⊚assoc : {Γ Δ Θ Δ₁ : Con}
  (γ : Θ ⇒ Δ₁)(ϑ : Δ ⇒ Θ)(δ : Γ ⇒ Δ)
  → (γ ⊚ ϑ) ⊚ δ ≡ γ ⊚ (ϑ ⊚ δ)
```

Weakening inside substitution is equivalent to weakening outside.

```
[+S]T : {Γ Δ : Con}
  {A : Ty Δ}{δ : Γ ⇒ Δ}
  {B : Ty Γ}
  → A [ δ +S B ]T ≡ (A [ δ ]T) +T B

[+S]tm : {Γ Δ : Con}{A : Ty Δ}
    (a : Tm A){δ : Γ ⇒ Δ}
    {B : Ty Γ}
    → a [ δ +S B ]tm ≅ (a [ δ ]tm) +tm B
```

They are useful to derive some auxiliary functions. The following is one of them which is used a lot in proofs.

```
wk-tm+ : {Γ Δ : Con}
  {A : Ty Δ}{δ : Γ ⇒ Δ}
  (B : Ty Γ)
  → Tm (A [ δ ]T +T B) → Tm (A [ δ +S B ]T)
wk-tm+ B t = t ⟦ [+S]T ⟩
```

We could cancel the last term in the substitution for weakened objects since weakening doesn't introduce new variables in types and terms.

```
+T[,]T : {Γ Δ : Con}
  {A : Ty Δ}{δ : Γ ⇒ Δ}
  {B : Ty Δ}{b : Tm (B [ δ ]T)}
  → (A +T B) [ δ , b ]T ≡ A [ δ ]T

+tm[,]tm : {Γ Δ : Con}{A : Ty Δ}
  (a : Tm A)(δ : Γ ⇒ Δ)(B : Ty Δ)
  (c : Tm (B [ δ ]T))
  → (a +tm B) [ δ , c ]tm ≅ a [ δ ]tm
```

Most of the substitutions are defined as usual, except the one for J-terms. We do substitution in the context morphism part of the J-terms.

```
var x    [ δ ]tm = x [ δ ]V
```

```
JJ cΔ γ A [ δ ]tm = JJ cΔ (γ ⊚ δ) A ⟦ sym [⊚]T ⟩
```

3 Some Important Derivable Constructions

There are some important notions which are missing but are derivable from the syntax. The groupoid laws on all levels should also be derivable using the J-terms. We will show some of them in this section.

Identity context morphism is not a primitive notion in this framework. To define it, we have to declare all the properties it should hold as an identity morphism. In other words, substitution with identity morphism should keep everything unchanged.

```
IdCm : ∀ Γ → Γ ⇒ Γ

IC-T   : ∀ {Γ : Con}(A : Ty Γ) → A [ IdCm Γ ]T ≡ A
IC-v : ∀{Γ : Con}{A : Ty Γ}(x : Var A) →   x [ IdCm Γ ]V ≅ var x
IC-⊚ : ∀{Γ Δ : Con}(δ : Γ ⇒ Δ) → δ ⊚ IdCm Γ ≡ δ
IC-tm : ∀{Γ : Con}{A : Ty Γ}(a : Tm A) → a [ IdCm Γ ]tm ≅ a
```

One of the most important feature of the contractible contexts is that any type in a contractible context is inhabited. It can be simply proved by using J-term and identity morphism.

```
anyTypeInh : ∀{Γ} → {A : Ty Γ} → isContr Γ → Tm {Γ} A
anyTypeInh {A = A} ctr = JJ ctr (IdCm _)   A ⟦ sym (IC-T _) ⟩
```

To show the syntax framework is a valid internal language of weak ω-groupoids, as a first step, we should produce a reflexivity term for the equality of any type.

We use a context with a type to denote the non-empty context.

```
Con* = Σ Con Ty

preCon : Con* → Con
preCon = proj₁

‖_‖ : Con* → Con
‖_‖ = uncurry _,_

lastTy : (Γ : Con*) → Ty (preCon Γ)
lastTy   = proj₂

lastTy' : (Γ : Con*) → Ty ‖ Γ ‖
```

```
lastTy' (_ ,, A) = A +T A
```

We tried several ways to get the reflexivity terms. One of them is to define the suspension of contexts first and then suspend a term of the base type n times to get the n-cell reflexivity. The encoding of the suspension is finished and will be showed later, however we found that there is a simpler way to do it.

Loop Context Here we are going to use some special contexts which are called *loop context* in this paper. For any type in any context, we could always filter out the unrelated part to get a minimum context for this type which is a loop context.

We will use a function to show what is a loop context.

```
ΩCon : ℕ → Con*
ΩCon 0 = ε ,, *
ΩCon (suc n) = let (Γ ,, A) = ΩCon n in
   (Γ , A , A +T A) ,, (var (vS v0) =h var v0)
```

A variable of the base type is a 0-cell, and the morphism between any two n-cells is an $n+1$-cell as mentioned before. A level-n loop context is the minimum non-empty context where the last variable is n-cell. We could easily prove such a context is contractible. Intuitively it is a special kind of contractible context where the branching approach is unique as we always create an $n+1$ cell for level-n loop context to get a level-(n+1) loop context. Since a loop context is contractible, all types are inhabited. The approach to get the reflexivity is to use J-term with a corresponding loop context. The idea is easy but there are three difficult steps.

First we need to define a function called *loop*Ω to get the required loop context. *loop*Ω' is the complete version which returns five different things, the previous context, the last type, a context morphism between the input previous context and output previous context, a proof term that substitute the output last type with the context morphism is equal to the input last type and another proof term that it is a contractible context. It is necessary to combine them together, because it will become much more involved if they are defined separately.

```
loopΩ' : (Γ : Con)(A : Ty Γ)
       → Σ[ Ω : Con ] Σ[ ω : Ty Ω ] Σ[ γ : Γ ⇒ Ω ]
   Σ[ prf : ω [ γ ]T ≡ A ] isContr (Ω , ω)
loopΩ' Γ * = ε    ,, * ,, • ,, refl ,, c*
loopΩ' Γ (_=h_ {A} a b) with loopΩ' Γ A
... | (Γ' ,, A' ,, γ' ,, prf' ,, isc) =
      Γ' , A' , A' +T A' ,,
      (var (vS v0) =h var v0) ,,
      γ' , (a [[ prf' ]]) , wk-tm (b [[ prf' ]]) ,,
```

```
      (trans wk-hom (trans wk-hom (cohOp-hom prf'))) ,,
      ext isc v0

loopΩ : Con* → Con*
loopΩ (Γ ,, A) with (loopΩ' Γ A)
... |   (Γ' ,, A' ,, γ' ,, prf' ,, isc) = Γ' ,, A'
```

The second problem is to define the context morphism, namely the substitution between the orignal context and the corresponding loop context. And the third problem is to prove type unification for the J-terms. The auxiliary functions make the proofs look much simpler that it was earlier.

```
Tm-refl :   (ne : Con*) → Tm {‖ ne ‖} (var v0 =h var v0)
Tm-refl (Γ ,, A) with loopΩ' Γ A
... | ΩΓ ,, ΩA ,, γ ,, prf ,, isc =
  JJ isc (γ +S A , wk-tm+ A (var v0 ⟦ wk-T prf ⟫))) (var v0 =h var v0)
  ⟦ sym (trans wk-hom (trans wk-hom+ (hom≡ (cohOp (wk-T prf))
  (cohOp (wk-T prf))))) ⟫
```

The one above is special for the reflexivity of the last variable in a non-empty context. We also define a more general version which is the reflexivity for any term of any type in given context. In addition we also obtain the symmetry for the morphism between the last two variables.

```
Tm-refl' : (Γ : Con)(A : Ty Γ)(x : Tm A) → Tm (x =h x)
Tm-refl' Γ A x =
        (Tm-refl (Γ ,, A) [ (IdCm _) , (x ⟦ IC-T A ⟫) ]tm)
        ⟦ sym (trans wk-hom (hom≡ (cohOp (IC-T A)) (cohOp (IC-T A)))) ⟫
```

```
Tm-sym : (Γ : Con)(A : Ty Γ)
        → Tm {Γ , A , A +T A} (var (vS v0) =h var v0)
        → Tm {Γ , A , A +T A} (var v0 =h var (vS v0))
Tm-sym Γ A t = (t [ (((IdCm _ +S A) +S (A +T A)) ,
  (var v0 ⟦ trans [+S]T (wk-T (trans [+S]T (wk-T (IC-T _)))) ⟫))) ,
  (var (vS v0) ⟦ trans +T[,]T
    (trans [+S]T (wk-T (trans [+S]T (wk-T (IC-T _))))) ⟫) ]tm)
  ⟦ sym (trans wk-hom (hom≡ (htrans (cohOp +T[,]T)
  (cohOp (trans [+S]T (wk-T (trans [+S]T (wk-T (IC-T A)))))))
  (cohOp (trans +T[,]T (trans [+S]T (wk-T
    (trans [+S]T (wk-T (IC-T A)))))))))) ⟫
```

There are still a lot of coherence laws to prove but it is going to be very sophisticated. We also tried to construct the J-eliminator for equality in this syntactic approach but have not found a solution. To construct more of them,

Altenkirch suggests to use a polymorphism theorem which says that given any types, terms and context morphisms, we could replace the base type $*$ in the context of the objects by some type in another context. With this theorem, any lemmas or term inhabited in contractible context should also be inhabited in higher dimensions.

Even though we didn't choose suspension to generate the reflexivity, it should be still useful in the future work.

Like all the other definitions, we have to define a set of operations together. In addition we could also prove that the suspension of a contractible context is still contractible.

```
ΣC : Con → Con
ΣT : {Γ : Con} → Ty Γ → Ty (ΣC Γ)
Σv : {Γ : Con}(A : Ty Γ) → Var A → Var (ΣT A)
Σtm : {Γ : Con}(A : Ty Γ) → Tm A → Tm (ΣT A)
Σs : {Γ Δ : Con} → Γ ⇒ Δ → ΣC Γ ⇒ ΣC Δ
ΣC-Contr : (Δ : Con) → isContr Δ → isContr (ΣC Δ)
```

The suspension of a context is to substitute the base type with the equality of two variables of base type for all occurences. So the base case for a suspension is a context contains two variables of base type. That means we can declare new variables whose type is the equality of these two variables.

```
ΣC ε = ε , * , *
ΣC (Γ , A) = ΣC Γ , ΣT A

*' : {Γ : Con} → Ty (ΣC Γ)
*' {ε} = var (vS v0) =h var v0
*' {Γ , A} = *' {Γ} +T ΣT A

_=h'_ : {Γ : Con}{A : Ty Γ}(a b : Tm A) → Ty (ΣC Γ)
a =h' b = Σtm _ a =h Σtm _ b

ΣT {Γ} * = *' {Γ}
ΣT (a =h b) = a =h' b
```

There are some lemmas which are neceesary for the definitions. The suspension of terms and context morphisms are too cumbersome to present here.

```
ΣT[+T]   : {Γ : Con}(A : Ty Γ)(B : Ty Γ)
```

```
                → ΣT (A +T B) ≡ ΣT A +T ΣT B
Σtm[+tm] : {Γ : Con}{A : Ty Γ}(a : Tm A)(B : Ty Γ)
                → Σtm _ (a +tm B) ≅ Σtm _ a +tm ΣT B
```

4 Semantics

Globular Sets To interpret the syntax, we need globular sets. Globular sets are defined coinductively as follows.

```
record Glob : Set₁ where
  constructor _||_
  field
    |_|  : Set
    homo : |_| →  |_| → ∞ Glob
open Glob public
```

Indeed we should assume the 0-level object to be an h-set, namely the equality of any two terms of it should be unique.

As an example, we could contruct the identity globular set called Idw.

```
Idω : (A : Set) → Glob
Idω A = A || (λ a b → ♯ Idω (a ≡ b))
```

Then given a globular set G, we could interpret the objects in syntactic frameworks.

```
⟦_⟧C : Con → Set
⟦_⟧cm : ∀{Γ Δ : Con} → (Γ ⇒ Δ) → ⟦ Γ ⟧C → ⟦ Δ ⟧C
⟦_⟧T : ∀{Γ}(A : Ty Γ)(γ : ⟦ Γ ⟧C) → Glob
⟦_⟧tm : ∀{Γ A}(v : Tm A)(γ :  ⟦ Γ ⟧C) → | ⟦ A ⟧T γ |
```

Another necessary thing is a dependent function Coh [1] should also comes with the globular set. It returns an object for every type in any contractible context, namely what is called a valid coherence in Brunerie's paper. This actually enables us to interpret J-terms in syntax.

```
Coh : (Θ : Con)(ic : isContr Θ)(A : Ty Θ) → (ϑ : ⟦ Θ ⟧C) → | ⟦ A ⟧T ϑ |
```

We temporarily postulate Coh function so that we could define the interpretations. However we would adopt the correct way later by defining a record type including the globular set, the interpretations and this function.

[1] it was called J but to make it less ambiguous we renamed it

There are also some lemmas for weakening to prove as before. The semantic weakening rules tell us how to deal with the weakening inside interpretation.

```
semWK-ty : ∀ {Γ : Con}(A B : Ty Γ)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
  → ⟦ A ⟧T γ ≡ ⟦ A +T B ⟧T (γ , v)

semWK-tm : ∀ {Γ : Con}(A B : Ty Γ)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
  (a : Tm A) → subst |_| (semWK-ty A B γ v) (⟦ a ⟧tm γ)
    ≡ ⟦ a +tm B ⟧tm (γ , v)

semWK-cm : ∀ {Γ Δ : Con}(B : Ty Γ)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
  (δ : Γ ⇒ Δ) → ⟦ δ ⟧cm γ ≡ ⟦ δ +S B ⟧cm (γ , v)
```

5 Conclusion

In this paper, we present an implementation of weak ω-groupoids following the Brunerie's work. Briefly speaking, we define the syntax of the type theory $\tau_{\infty-groupoid}$, then a weak ω-groupoid is a globular set with the interpretation of the syntax. To overcome some technical problems, we use heterogeneous equality for terms, some auxiliary functions and loop context in all implementation. We construct the identity morphisms and verify some groupoid laws in the syntactic framework. The suspensions for all sorts of objects are also defined for other later constructions.

There are still a lot of work to do within the syntactic framework. For instance, we would like to investigate the relation between the $\tau_{\infty-groupoid}$ and a Type Theory with equality types and J eliminator which is called τ_{eq}. One direction is to simulate the J eliminator syntactically in $\tau_{\infty-groupoid}$ as we mentioned before, the other direction is to derive J using Coh if we can prove that the τ_{eq} is a weak ω-groupoid. The syntax could be simplified by adopting categories with families. Altenkirch also suggests to use explicit substitution and QIIP which is an alternative way to define the syntax.

We would like to formalise a proof of that Idω is an weak ω-groupoids, but the base set in a globular set is an h-set which is incompatible with Idω. Perhaps we could solve the problem by making a syntactic proof. Finally, to model the Type Theory with weak ω-groupoids and to eliminate the univalence axiom would be the most challenging task in the future.

References

[1] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM.

[3] Thorsten Altenkirch and Ondrej Rypacek. A syntactical Approach to Weak ω-groupoids. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012*, 2012.

[4] D. Ara. On the homotopy theory of Grothendieck ∞-groupoids. *ArXiv e-prints*, June 2012.

[5] Andrej Bauer and Peter LeFanu Lumsdaine. A Coq proof that Univalence Axioms implies Functional Extensionality. 2013.

[6] Guillaume Brunerie. Syntactic Grothendieck weak ∞-groupoids. http://uf-ias-2012.wikispaces.com/file/view/SyntacticInfinityGroupoidsRawDefinition.pdf, 2013.

[7] Alexander Grothendieck. Pursuing Stacks. 1983. Manuscript.

[8] G. Maltsiniotis. Grothendieck ∞-groupoids, and still another definition of ∞-categories. *ArXiv e-prints*, September 2010.

Equations Over Groups: An Interdisciplinary Approach

Christian Sattler

June 11, 2013

Introduction

A central topic in combinatorial group theory are the so-called equations over groups. Given a group G and one or more equations over G in one or more "unknowns", one may ask whether G may be "extended" to, i.e. faithfully embedded into, a larger group H such that the equational system can be solved over H. In that case, we call the system solvable.

An early result by Neumann [7] in 1949, the equation $x^n = g$ can be solved for any $g \in G$ and $n \geq 1$. Perhaps surprisingly, two elements $a, b \in G$ of equal order can always be made conjugate, corresponding to solvability of the equation $xa = bx$ [4]. In general, one would like to know conditions on G and the structure of equations that imply solvability. For example, if G is abelian, solvability of a non-trivial equation in a single variable x simply corresponds to the sum of exponents of x being non-zero. In spite of many such individual results, the central conjecture in the field still remains unsolved:

Conjecture (Kervaire-Laudenbach Conjecture)**.** *Let a group G and an equation $a_1 x^{t_1} \dots a_i x^{t_i} = e$ over G be given ($a_1, \dots, a_i \in G$). If $t_1 + \dots + t_i \neq 0$, then the equation is solvable.*

The special case $t_1, \dots, t_i \geq 0$ was proven in 1962 by Levin [6] with a direct constructing using the wreath product. The value $|t_1| + \dots + |t_i|$ is known as the *length* of the equation.

The canonical generalization to systems of equations goes as follows:

Conjecture (Kervaire-Laudenbach-Howie Conjecture)**.** *Let G be a group and $x_1, \dots, x_n$ free variables ("unknowns"). Consider a system of equations*

$$\begin{aligned} A_1[x_1, \dots, x_n] &= e, \\ &\dots \\ A_m[x_1, \dots, x_n] &= e \end{aligned}$$

where $A_1, \dots, A_m$ denote expressions in G and the unknowns. Let $w_{ij} \in \mathbb{Z}$ denote the sum of exponents of x_j in A_i. If the rows of the matrix w are linearly independent, then the system is solvable.

If $m = n$, then the condition states that w must be invertible, i.e. have non-zero determinant. This case is known for G finite or a compact Lie group, or more generally locally residually finite. [8]

In what follows, however, we will restrict ourselves to the original Kervaire-Laudenbach conjecture, i.e. the case of one equation in one unknown. Furthermore, we will focus on conditions for the shape of the equation instead of conditions for the underlying group G. Note that the conjecture is non-obvious only for equations of length at least 3.

We now give a short summary of past progress: Howie [5] settled the conjecture for equations of length 3 in 1983. Edjvet [1] proved the case of only two occurrences of the unknown, i.e. equations of the form $at^m bt^n = e$ with $m + n \neq 0$ in 1991. Shortly afterwards, working with Howie [2] they dealt with the remaining cases for equations of length 4. Continuing the methods of Edjvet, in their dissertation [3] Evangelidou filled in the holes remaining in the case of length 5. However, already in the last mentioned incremental improvement, the bulk of over 200 pages was spent on chasing many different cases in a giant tree of case distinctions.

Basics

Given groups G and H, the group generated by elements of both G and H with the obvious relations is known as the *free product* of G and H. In the category of groups, it can be characterized as the coproduct of G and H. The canonical morphisms from G and H into $G * H$ are always injective and their images share only the neutral element.

A common way of representing a group G called (finite) *presentation* consists of a (finite) set S of generators S together with a (finite) set $R \subseteq \mathrm{Free}(S)$ of relations, where $\mathrm{Free}(S)$ is the free group on S (the image of the right adjoint of the forgetful functor from groups to sets). Formally, G is defined as the quotient of $\mathrm{Free}(S)$ and the normal closure $\langle R^{\mathrm{Free}(S)} \rangle$ of R, which is the subgroup of $\mathrm{Free}(S)$ generated by all conjugates of elements of R. However, deciding whether a given word (an element of $\mathrm{Free}(S)$) is trivial modulo a set of relations is in general undecidable, even for finitely presented groups. This hints at such a representation not being all that explicit, often leaving in the dark many properties of the group in question.

The Kervaire-Laudenbach can now formally be stated as follows:

Conjecture (Kervaire-Laudenbach Conjecture, Categorically)**.** *For a given group G, consider the morphism $s := [\mathrm{const}\, 0, \mathrm{id}] : G * \mathbb{Z} \to \mathbb{Z}$. Fix r such that $s(r) \neq 0$. Then the composition of the embedding $G \hookrightarrow G * \mathbb{Z}$ with the projection $G * \mathbb{Z} \to (G * \mathbb{Z})/\langle r^{G*\mathbb{Z}} \rangle$ is an embedding, i.e. monic.*

The unknown x is implicitely modelled as $\langle x \rangle \cong \mathbb{Z}$. Note that for a word $r \in G * \mathbb{Z}$, the value $s(r)$ of the sum-of-exponents morphism s will tell us the previously awkwardly defined sum of exponents of the unknown in r.

The previous version of the conjecture in fact constructs the *universal* solution to the given equation $r = e$: All other group extensions solving it will have a unique morphism from $(G * \mathbb{Z})/\langle r^G \rangle$. This is also a first hint at a perhaps more symbolic nature of the conjecture in which the concrete nature of the group G will retreat to the background, with the central arguments taking place over words in a finitely presented framework. Hence, instead of just replacing the solving extension H of G with a synthetic universal object, we might as well try the same with G.

The last version of the conjecture can also be stated as saying that the intersection of the image of G in $G * \mathbb{Z}$ with $r^{G*\mathbb{Z}}$ is trivial. Note that $r^{G*\mathbb{Z}}$ consists of all words of the form $(a_1 r^{d_1} a_1^{-1}) \dots (a_i r^{d_i} a_i^{-1})$ where $a_i \in G * \mathbb{Z}$ and $d_i \in \mathbb{Z}$. Whenever this expression reduces to a "normal form" in $G * \mathbb{Z}$ with no occurrences of the unknown $x := \mathrm{inr}(1)$ contained in r, it must be equals to e.

Dealing with normal subgroups and normal closures entails cyclic closure: Whenever $u_1 \dots u_i = e$, we can conjugate both sides with u_1 and obtain $u_2 \dots u_i u_1 = e$. We will often regard a word as an oriented cyclic structure with no canonical starting point. In topology, this situation is common when dealing with the oriented boundary of a 2-cell in a cell complex. This suggests modelling the normal closure of r in topological terms. A 2-cell will correspond to either r or r^{-1}. Each 1-cell corresponds to an element of either G or $\mathbb{Z}$.

Conjecture (Kervaire-Laudenbach Conjecture, Topologically)**.** *Let G be a group and $r \in G * \mathbb{Z}$ such that $s(r) \neq 0$. Let C be an oriented 2-complex topologically equivalent to the sphere S^2. Consider each edge of C to be marked with $x := \mathrm{inr}(1)$, and every corner of each face marked with an element of G. Assume the oriented boundaries of each face read as a word in $G * \mathbb{Z}$ yield (cyclical permutations of) r or r^{-1}. For each vertex v, let $w(v)$ be the G-word obtained (up to cyclic permutation) by reading off the corner labels in order (formally, the oriented boundary of v in the dual complex. If $w(v) = e$ in G for all v but a single vertex v_0, then also $w(v_0) = e$.*

Conjecture (Kervaire-Laudenbach Conjecture, Combinatorically)**.** *Fix a set $\{a_1, \dots, a_n\}$ and an n-gon T with corners labelled $a_1, \dots, a_n$ and each edge having a specific orientation such that the respective numbers of positively and negatively oriented boundary edges are distinct. Consider a directed tiling of the sphere with copies of T and its mirror image. For each vertex v, let $w(v)$ denote the element of* Free(A) *given (up to cyclic permutation) by reading off the corner labels in order. Then for any vertex v_0, the word $w(v_0)$ is contained in the normal closure of $\{w(v) \mid v \neq v_0\}$.*

Equivalently, there is a directed tiling of the sphere by tiles T_v and T_v^{-1} with boundary given by $w(v)$ such that T_{v_0} or $T_{v_0}^{-1}$ is only used once. This is just a combinatorial reformulation of the proposition that the relations $w(v) = e$ for $v \neq v_0$ induce $w(v_0) = e$.

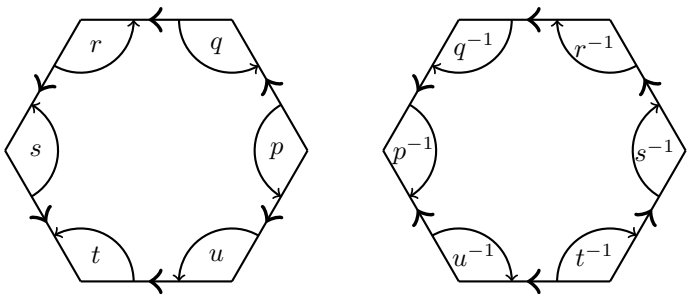


Figure 1: Allowed tiles for $p \,\mathbf{x}\, q \,\mathbf{x}\, r \,\mathbf{x}\, s \,\mathbf{x}\, t \,\mathbf{x}^{-1}\, u \,\mathbf{x}^{-1} = e$

Figure 1 shows an example of how the tiles T and T^{-1} look like. Note the orientations of the edges and angles. Instead of explicitely inverting the angle labels in the right tile, we might as well have reverse the orientation of

the respective angles. This would have produced a complete geometric mirror image of the left tile.

The following simplification is probably wrong, but the author did not yet check any examples.

Conjecture (Strengthened)**.** *In the above setting, before quantifying over v_0, there exists a directed tiling of disjoint copies of S^2 using just one tile T_v per original vertex v.*

Methods

In this section, we will briefly present the technical tools introduced by Edjvet and others to tacke the cases of low equation length. Edjvet believes the main obstacle is the exponentially increasing number of cases to analyze, with the same general techniques still applicable. This is were our research comes into play, lending a helping hand in computerizing large strategizable portions of the proof.

Despite the high-profile nature of the general conjecture, the problem itself as well as the following techniques do not require a large library of group theoretical facts for implementation, consisting mainly of combinatorics. The problem may hence be viewed as an ideal case study for the use of algorithmized proving using dependently typed frameworks in non-calculational proofs. The general plan is to implement a number of abstract standard argument templates by which a given case may be solved or reduced, if necessary together with a heuristic to limit tree depth exploration. The actual program therefore will model the process a human mathematician would follow going through hundreds or even thousands of cases, with the benefit of not causing sleep deprivation.

It remains to be seen what level of reflection is required for this undertaking. However, since the combinatorial assertions concern a somewhat closed area of objects and should be of limited shapes, we probably do not require an internalization of dependently typed syntax.

Star Graphs

Fix an equation $a_1 t^{d_1} \dots a_n t^{d_n} = 1$ with $d_1, \dots, d_n \in \{-1, 1\}$ and symbolic coefficients $a_1, \dots, a_n$. We assume we have a tiling of the sphere as in the combinatorial version of the conjecture and are interested in the possible shapes of the words $w(v)$ for vertices v. The *star graph* Γ of the equation consists of two vertices $+$ and $-$. For $i = 1, \dots, n$, we draw a directed edge labelled by a_i. Source and target of the edge are determined respectively by the sign of the exponent of t directly after and the *inverse* sign of the exponent of t directly before a_i in the cyclic word $a_1 t^{d_1} \dots a_n t^{d_n}$. Symmetrically, for $i = 1, \dots, n$, we draw a directed edge labelled by a_i^{-1} according to the "connection" information contained in the inverse cyclic word $t^{-d_n} a_n^{-1} \dots t^{-d_1} a_1^{-1}$.

The possible shapes of $w(v)$ for a vertex v can now be read off as the directed circles in Γ. The simple reason for this is that each incident face the vertex v is a corner of has boundary $a_1 t^{d_1} \dots a_n t^{d_n}$ and adjacent faces must have complementary exponents of t at their shared edge. For example, this allows a_i to directly *precede* a_j in orientation around the vertex v only if the exponent of t directly *before* a_i in the equation matches the inverse of the exponent of t

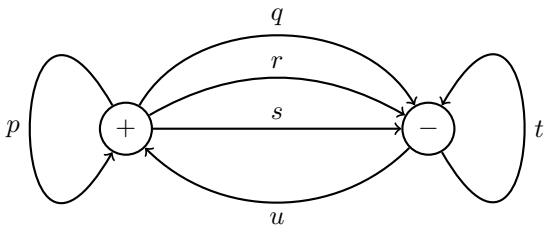


Figure 2: Star graph for $p \mathbf{x} q \mathbf{x} r \mathbf{x} s \mathbf{x} t \mathbf{x}^{-1} u \mathbf{x}^{-1} = e$

directly *after* a_j in the equation (note the inversion of orientation when relating vertices and faces). This is exactly what is expressed in the source and target vertex information of the star graph. Figure 2 shows an example of a star graph for an equation of length 6.

Vertices of Degree 2

Information on low degree vertices in the above tiling of the sphere is valuable because of the easier to apply structure of the relations induced by their corner words. In particular, the central resting pillars of the techniques of Edjvet is the use of vertices of degree 2. By a straightforward extremal argument, we may assume that a corner a_i is never directly followed by a corner a_i^{-1} or vice-versa in orientation around a vertex for otherwise we are able to collapse an adjacent face pair. Hence, a vertex of degree 2 must induce a relation of the form $a_i = a_j$ or $a_i = a_j^{-1}$ for i different from j in the latter case. This effectively takes out one of the six coefficients, reducing the "degree of freedom" by one, except in the special case $i = j$, where $a_i = a_i^{-1}$ restricts the order of a_i to 2.

The central question therefore is how to somehow force the existence of low degree vertices. By Eulers formula, the average vertex degree in a spherical graph with hexagonal faces is $3-\frac{6}{m}$ where m is the number of vertices. This alone is enough to deduce the existence of a single vertex of degree 2. To get better estimates, in particular taking into account the topological-combinatorical tiling information, we must look for more fine-grained methods.

Curvature

A basic fact in geometry is that the internal angles of an n-gon always sum to $(n-2)\pi$. Equivalently (and in fact generalizing to piecewise differentiably closed intersection-free curves), the sum of the amounts *missing* from each angle to π, the amount required to straighten it, will always equal 2π, a full rotation.

It should not come as a surprise that this can be generalized to higher dimensions. For three-dimensional polytopes, it can in fact be read as a geometric

interpretation of Eulers formula (a prime example for the usefulness of topology in abstracting from geometry): Fix a polytope. For each individual vertex, compute the sum of the angles of the incident corners. Subtracting this value from 2π yields the amount missing from the vertex to completely flatten it. Now the sum of these missing amounts over all vertices will always equal 4π. [1]

We will apply the concept for curvature, a more formal notion for the missing amounts alluded to above, to the dual graph of our tiling. Each corner of a vertex v will receive a synthetic angle $\frac{2\pi}{\deg(v)}$, making it completely flat (an even more fine-grained control is achieved by assigning angles to corners such that the both faces and vertices have non-trivial curvature). The curvature $c(\Delta)$ of each face Δ is given by

$$\pi\left(2-\deg(\Delta)+2\sum_{v}\frac{1}{\deg(v)}\right)$$

where the sum runs over the corner vertices of Δ of. By the geometrical interpretation of Euler's formula, we must have $\sum_{\Delta} c(\Delta) = 4\pi$, where the sum runs over all faces of the tiling.

However, a total positive curvature in particular means that we should expect to find many faces of individual positive curvature. Rewriting the face curvature as

$$\begin{aligned} c(\Delta) &= 2\pi\left(1+\sum_{v}\left(\frac{1}{\deg(v)}-\frac{1}{\deg(2)}\right)\right) \\ &= 2\pi\left(1-\sum_{v}\frac{\deg(v)-2}{2\deg(v)}\right). \end{aligned}$$

For the easiest unsolved case of hexagonal faces, all vertices incident to a face having degree at least 3 is already enough for a non-positive curvature. In general, the more vertices our faces have, the more vertices of low degree we need to compensate for this curvatory negativity. However, the different types of vertices of low degree are restricted by combinatorial arguments, and the tiling itself comes with restrictions as to where these vertices can occur. Furthermore, detailed case analyses are undertaken to examine how a face with positive curvature is compensated for by its neighbouring faces. The final goal will be to show that the only way to "close the sphere" at any vertex v_0 and achieve a total curvature of 4π is to have sufficiently many distinct corner words $w(v)$ such that the group presented by $\langle a_1, \ldots, a_n \mid w(v) = 1 \text{ for } v \neq v_0\rangle$ is locally residually finite or otherwise restricted to one of several known cases, or if that fails, that the word $w(v_0)$ is contained in the normal closure of $\{w(v) \mid v \neq v_0\}$.

Diamond Moves

Diamond moves form a basic set of "rewriting" strategies with which any given tiling may be locally modified. They can be applied in any scenario where a direct application of the above techniques is not possible.

[1] Generalizations to even higher dimensions and continuous shapes form beautiful and well-studied areas of homological topology and differential geometry.

Conclusion

We gave an outline of the current state of the art of the techniques for solving the equations over groups problems for equations of fixed (low) length. Since the number of case distinctions necessary even for equations of length 6 already appears intractable for a human mathematician to do manually, it is our hope that these techniques are elementary enough to be implemented in a reasonable amount of time in a dependent type system like Agda, facilitating not only a systematic way of resolving many cases automatically (albeit not completely algorithmically), but a way of integrating these programmatic components of the proof with manually supplied components based on traditional mathematical reasoning. Furthermore, correspondence has shown Edjvet to be optimistically open for this kind of approach.

References

[1] Martin Edjvet. Equations over groups and a theorem of Highman, Neumann, and Neumann. In *Proc. London. Math. Soc.*, volume 62, pages 563–589, 1991.

[2] Martin Edjvet and James Howie. The solution of length four equations over groups. *Trans. Amer. Math. Soc.*, 326:345–369, 1991.

[3] Anastasia Evangelidou. *Equations of length five over groups.* PhD thesis, University of Nottingham, 2003.

[4] Graham Higman, Bernhard Hermann Neumann, and Hanna Neumann. Embedding theorems for groups. *Journal London Math. Soc.*, 24:247–254, 1949.

[5] James Howie. The solution of length three equations over groups. In *Proc. Edinburgh Math. Soc.*, volume 26, pages 89–96, 1983.

[6] Frank Levin. Solution of equations over groups. *Bull. Ameri. Math. Soc.*, 68:603–604, 1962.

[7] Bernhard Hermann Neumann. Adjuction of elements to groups. *Journal London Math. Soc.*, 18:4–11, 1943.

[8] Oscar S. Rothaus. On the non-triviality of some group extensions given by generators and relations. *The Annals of Mathematics*, 106:599–612, 1977.