Reasoning about functional programs and complexity classes associated with type disciplines

Daniel Leivant

Computer Science Department

Carnegie-Mellon University

Abstract.

We present a method of reasoning directly about functional programs in Second-Order Logic, based on the use of explicit second-order definitions for inductively defined data-types. Termination becomes a special case of correct typing.

The formula-as-type analogy known from Proof Theory, when applied to this formalism, yields λ-expressions representing objects of inductively defined types, as well as λ-expressions representing functions between such types. A proof that a functional closed expression e is of type T maps into a λ-expression representing (the value of) e; and a proof that a function f is correctly typed maps into a λ-expression representing f (modulo the representations of objects of those types). When applied to integers and to numeric functions the mapping yields Church's numerals and the traditional function representations over them.

The λ-expressions obtained under the isomorphism are typed (in the Second-Order Lambda Calculus). This implies that, for functions defined over inductively defined types, the property of being proved everywhere-defined in Second-Order Logic is equivalent to the property of being representable in the Second-Order Lambda Calculus.

Extensions and refinements of this result lead to other characterizations of complexity classes by type disciplines. For example, log-space functions over finite structures are precisely the functions over finite-structures definable by λ-*pairing*-expressions in a predicative version of the Second-Order Lambda Calculus.

Introduction.

- *Second-order reasoning about functional programs.*

Computable functions given by functional mechanisms, such as recursion equations or lambda expressions, have come into prominence through the advent of Denotational Semantics, as well as by an independent interest in Functional Programming as a new methodology. Some formalisms for reasoning about functional programs, such as the PPLAMBDA portion of Edinburgh LCF [GMW], follow the traditional framework of first order algebraic equational theories, augmented with a type structure. Types are intended to denote domains, with, say, a type **natural** denoting the flat domain of the natural numbers. The very nature of the denotational semantics intended here implies, then, that non-converging expressions are distinguished from converging ones not in their type, but in value only.

A different approach is possible within Second-Order Logic, where quantification is permitted over predicates as well as individuals. Inductively defined types can then be defined explicitly, rather than via algebraic closure conditions. For instance, the well-known definition of the natural numbers is

$$N(x) \equiv \forall R\ [(\forall z(R(z) \rightarrow R(Sz))) \rightarrow (R(0) \rightarrow R(x))],$$

where R ranges over all unary relations, and S denotes the successor function. In every structure in which Peano's axioms for the successor function hold,

$$S0 \neq 0, \quad Sx = Sy \rightarrow x = y,$$

$N(x)$ defines an isomorphic copy of the natural numbers; there are no non-standard models. The same holds for the explicit definition of any other inductively defined type, such as the types of lists, trees, complete binary trees, and so on.

The distinction between converging and non-converging expressions is here a distinction of type, not only of value. For example, if certain computation rules for the unary function f imply that the statement $N(f(0)))$ holds in all models, then the function f evaluates on input 0 to a *standard* natural number, and the expression $f(0)$ converges. Similarly, if the statement $\forall x.\ N(x) \rightarrow N(f(x))$ holds in all models, then the function f is total. More generally, if $T_0, \ldots, T_k$ are formulas defining inductively-defined types explicitly, and f is a function from the types (defined by the formulas) $T_1, \ldots, T_k$ to T_0, then the validity of

$$T_1(x_1) \wedge \cdots \wedge T_k(x_k) \rightarrow T_0(f(x_0,\ldots,x_k))$$

implies that the function denoted by f is total over the function-domain $T_1 \times \cdots \times T_k$.

The particular formalism we propose for reasoning about functionally defined programs in Second-Order Logic is very simple. We assume that computable functions are given by *functional programs*, i.e., by systems of recursion equations in the style of Herbrand-Gödel [Kle,Men]. To reason about a functional program P, we introduce as non-logical axioms the formulas of the form

$$\forall R(R(s) \leftrightarrow R(t))$$

for all equations $s = t$ in P. (Actually we use inference rules equivalent to these axioms.)

The use of Second-Order, rather than First-Order Logic, for reasoning about programs, although not completely new (compare e.g. [Par]), is as yet controversial. We cannot possibly do justice to the subject in the present framework, but we can try to summarize the most relevant aspects of our position. The pleasant properties of First-Order Logic, such as compactness and effective enumerability of the valid formulas, are indeed lost for Second-Order Logic. However, these properties are lost also for True Arithmetic, i.e. the set of first order formulas in Peano's Language that are true in the (standard) structure $\langle \omega, 0, S, +, \times \rangle$. Since counting, by *standard* integers, is intimately intertwined with the semantics of programming languages, nothing much (if anything) is left of the niceties of First-Order Logic in formal calculi for reasoning about programs.[1].

There is also a positive aspect of Second-Order Logic that often passes un-noticed by computer scientists. Although the set of valid second-order formulas is not effectively enumerable (nor even definable in Second-Order Arithmetic), the Second-Order Predicate Calculus (as formalized, for instance, in [Pra]) is a perfectly pedestrian formalism, which generates precisely all the second-order formulas that are valid not merely in all standard second-order structures, but in all Henkin structures [Hen] as well. (In a Henkin structure the collection of sets may be a proper subset of the power set of the universe.)

[1] The relevant theory here is in fact the set of first order formulas *with predicate letters* over the structure $\langle \omega, 0, S \rangle$. This is because logics of programs blend first order relational statements with counting (as the only component of Arithmetic). The set of valid formulas is here complete Π^1_1 (as opposed to True Arithmetic, which is only Δ^1_1), a level of complexity often encountered in relation to logics of programs (compare for example [MP, MD]). Two results where the blend mentioned above is particularly striking are Harel's theorem on arithmetical completeness [Har], and Clarke's incompleteness result [Cla].

0272-5428/83/0000/0460$01.00 © 1983 IEEE

• *Proofs of type-correctness as typed λ-algorithms*

The simplicity and economy of the kind of theories of computation we propose have technical consequences, arising from the relation between Second-Order Logic and the Second-Order Lambda Calculus, via the formula-as-type analogy known from Proof Theory. An analogy between Gentzen-style natural deduction proofs (in Minimal Propositional Logic) and simply typed λ-expressions was discovered independently by Curry, Howard, de Bruin and Lauchli [CF, How, Lau]. The analogy is based on the observation that a derivation of a formula $\varphi \Rightarrow \psi$ may be thought of (constructively) as a method for converting any proof of φ into a proof of ψ. (We use here momentarily $\Rightarrow$ to denote implication.) Thus, if we think of the formula derived by a derivation as being that derivation's *type*, then a derivation of the formula $\varphi \Rightarrow \psi$ has the type $\varphi \rightarrow \psi$, where the single arrow is read as *functional* type formation. Now, if Δ_1 is a derivation of the formula $\varphi \rightarrow \psi$, and Δ_1 is a derivation of the formula φ, then the application $\Delta_1\Delta_2$ should correspond to a derivation of "type" ψ. Indeed, this application is meaningful in natural deduction: it corresponds to combining Δ_1 and Δ_2 into a derivation of ψ by the rule of implication elimination. Similarly, λ-abstraction corresponds to the rule of implication *introduction*.

Does this isomorphism hide anything interesting about the computational character of the λ-expressions corresponding to derivations? The question is of course meaningful, because it has been known almost since the inception of the λ-calculus that λ-expressions can be used to represent numeric functions, and that the functions representable are precisely the general recursive ones [Bar]. As it turns out, the Simply-Typed Lambda Calculus, to which the analogy as stated restricts us, is computationally quite weak. W.W. Tait has shown that, if e is a simply-types λ-expression of size n, then the size of the normal form of e is bounded by the function $e(n) \equiv$ an exponential stack of n 2's [Tai, FLO]. It follows that no function majorizing e can be represented in the Simply-Typed Lambda calculus, as long as the coding of numerals is bounded by an elementary function.

The formula-as-type analogy can be extended from Propositional to First Order Logic, at the cost of introducing additional constructs to the Lambda Calculus, with corresponding conversion rules. This approach has been pursued by Jean-Yves Girard and Per Martin-Löf [Gir,Mar]. By further extending the type structure with the type of the natural numbers, and with conversion rules corresponding to the rules and axioms of Peano's Arithmetic, Martin-Löf has obtained the representation of all recursive functions that can be proved total in First-Order Arithmetic, and exactly those [Mar,Sch]. However, when the entire machinery of First-Order Arithmetic is explicitly injected into the Lambda Calculus and to the type structure imposed on it, there is little left of the purely functional character of the Lambda Calculus and much of the analogy's interest is lost.

We wish to demonstrate that a much simpler and more flexible extension of the formula-as-type analogy is obtained by passing to Second-Order Logic. We map natural-deduction derivations of Second-Order Logic [Pra] into λ-expressions in the Second-Order Lambda Calculus, a discipline invented independently by Jean-Yves Girard and John Reynolds ([Gir, Rey]; or see [FLO]). Our injection corresponds Λ-abstraction (abstraction over types) to second order universal quantifier introduction, and Λ-application to second order universal quantifier elimination. First-order quantification rules are ignored.

From this mapping we immediately obtain a simple method for representing inductively-defined data-types in the Second-Order Lambda Calculus. Suppose given a data-type T, and a functional expression e representing canonically an object v of type T (for example, $SS0$ represents canonically 2). The proof that e is of type T maps into an expression of the Second-Order Lambda Calculus which we may take to represent v. By stripping e of its type structure we obtain a representation of v in the untyped Lambda Calculus. For example, for the natural numbers we obtain the Fortune-O'Donnell numerals in the typed calculus [FLO], and Church's numerals in the untyped calculus [Bar].

The true interest of the method lies, however, in its extension to the λ-representation of functions. Suppose that f is a computable function, say from natural numbers to natural numbers, and that P is a functional program for f, with f as the (principal) function-letter of P, denoting f. Then any proof of $N(x) \rightarrow N(f(x))$ in Second-Order Logic, using the program P in the manner outlined above, maps under our correspondence into a second order λ-expression that represents f in the Second-Order Lamdba Calculus, modulo the representation of the natural numbers. Again, by stripping the expression of its type structure we obtain a representation of f in the untyped calculus.

By considering straightforward proofs for the type correctness of basic arithmetic functions, such as the successor, addition, multiplication and predecessor functions, we recover the familiar representations of these functions in the Lambda Calculus, invented by Church and Kleene half a century ago [Bar]. Similarly, we recover the familiar λ-representation of non-arithmetic functions, such as pairing, conditional and the Boolean operations. The gain is twofold. On the one hand we obtain insight into and justification for Church's and Kleene's representations of functions in the Lambda Calculus. On the other we have a method for automatically generating typed λ-representation for arbitrary functions from proofs of their type-correctness. In particular, we have a direct proof that all the recursive functions, over inductively defined types, that are provably total in Second-Order Logic (or Second-Order Arithmetic), are representable in the Second-Order Lambda Calculus This result, for number-theoretic functions and Second-Order Arithmetic, was first proved by Richard Statman [Sta]. The converse, that every function representable in the Second-Order Lambda Calculus is provably total in Second-Order Arithmetic, as been known by the methods of Girard [Gir,FLO].

The translation from derivations to λ-terms can be rephrased in terms more relevant to programming, as follows. Given a functional program defining a function in an applicative language, using recursion to any desired extent, one can convert a proof in Second-Order Logic (equivalently — in Second Order Arithmetic) that execution terminates for all input, into a strongly typed and recursion-free program that represents that function. The input program converges everywhere by virtue of a correctness proof; the convergence for all input of the output λ-algorithm is earmarked in its syntax.

• *Characterization of complexity classes by type disciplines*

(Discussion and proofs of the results below will be given elsewhere.)

The main result described above may be read as a characterization of a complexity class by a type discipline imposed on λ-expressions. The complexity class is the class of recursive functions provably total in Second-Order Logic (or Second-Order Arithmetic), and the type discipline is the Second-Order type discipline of Girard and Reynolds. The method yields similar characterizations for other complexity classes of interest to logicians. For example, the provably recursive functions of Type Theory [Chu] are precisely the functions defined by λ-expressions that can be typed in the full ω-order type discipline of Girard [Gir].

Other characterizations of complexity classes are obtained by restrictions on the higher-order type disciplines to their appropriately defined *predicative* fragment. The term "predicative" is used here in complete analogy to its use for higher-order logical calculi. Roughly, type abstractions come in "levels", and a type application $e\tau$ is permitted only if all abstractions within the type argument τ are of levels inferior to that of the main type abstraction of e. Because the conjunction connective can not be expressed in terms of implication and universal quantification, as in [Pra], in predicative versions of higher-order logical calculi, we need to consider an extension of the λ-calculus with pairing and projections as primitive operations.

We find that the recursive functions provably total in Predicative Second-Order Logic are precisely the functions definable by λ-*pairing*-expressions typed in a correspondingly predicative version of the Second-Order type discipline of Girard and Reynolds. The latter

class is easily verified to be identical with the class of primitive-recursive functions, for which we therefor obtain an attractive characterization in terms of provability in Predicative Second-Order Logic.

An interesting corollary of this is obtained via the recent result of Yuri Gurevitch [Gur], who has characterized the log-space functions over finite structures as precisely the functions defined by primitive recursive functions, with the size of the structure and its internal functions and relations as input parameters. Thus, the log-space functions over finite structures can also be characterized as the functions over finite structures that are representable in the predicative fragment of the Second-Order Lambda Calculus.

When the full ω-order Lambda Calculus of Girard is restricted to its predicative fragment, the class of λ-representable functions turns out to be exactly the class of recursive functions provably total in First Order (Peano) Arithmetic. The relation of this result to the characterization above of the primitive-recursive functions is analogous to the characterization by Gödel of the provably recursive functions of First-Order Arithmetic as the functions defined using primitive-recursion for functionals of all finite type [Göd].

The characterizations of the classes of functions representable in predicative fragments of Second- or Higher-Order Logics, *without* the pairing function, are open problems.

1. Second order theories of computation.

We use Gentzen's natural deduction style for Second-Order Logic [Pra]. The logical inference rules are the well known natural deduction rules for First-Order Logic (Classical, Constructive or Minimal), augmented with introduction and elimination rules for quantifiers over relations. Thus, the introduction rule for second order universal quantifier is

$$\frac{\varphi}{\forall R\,\varphi}\ ,$$

where the relational variable R is not free in any open assumption here, and the corresponding elimination rule is

$$\frac{\forall R^n \varphi}{\varphi[\psi/R]}\ ,$$

where ψ is a formula free for R in φ, whose free variables are $x_1,\ldots,x_n$, and where the replacement $[\psi/R]$ is performed by replacing each atomic subformula $Rt_1\ldots t_n$ of φ by $\psi[t_1/x_1 \cdots t_n/x_n]$[2].

A *system of data-types*, or *data-system*, is a finite collections of sets, the *data-types* of the system, generated by (possibly simultaneous) inductive definitions; that is, a finite collection of rules of the form

$$\mathbf{T}_1(z_1) \wedge \cdots \wedge \mathbf{T}_k(z_k) \rightarrow \mathbf{T}(\mathbf{f}(\vec{z})),$$

for each data-type $\mathbf{T}$ in the system, where $\mathbf{f}$ is a k-ary operator, with $k \geq 0$, and where each $\mathbf{T}_i$ is some data type of the system, possibly $\mathbf{T}$ itself. We assume that no operator $\mathbf{f}$ is used in two distinct rules. The elements of the data-types are the *data-objects*.

The model supporting the operations generating the data-types is the *implementation* of the data-system. An implementation of a data-system is *coherent* if $\mathbf{f}_1\vec{\mathbf{v}}_1 = \mathbf{f}_2\vec{\mathbf{v}}_2$ implies $\vec{\mathbf{v}}_1 = \vec{\mathbf{v}}_2$, for all operators $\mathbf{f}_1$, $\mathbf{f}_2$, and all tuples of data-objects $\vec{\mathbf{v}}_1$, $\vec{\mathbf{v}}_2$.[3]. A consequence of our assumption above is that in a coherent implementation every data-object determines uniquely the finite tree of initial objects and closure conditions that has generated it as a data-object.

Each particular *theory of computation* is determined by a data-system $\mathbf{D}$, and by a collection of functional programs. The data-system is represented in the theory using a collection of function symbols, one (k-ary) function letter f for each (k-ary) operator $\mathbf{f}$ of the system. These are the *primitive* symbols of the theory. The terms generated from the primitives are the *canonical terms* of the theory.

The data-types of the system are defined explicitly by second order predicates. Suppose there are m data-types $\mathbf{D}_1 \cdots \mathbf{D}_m$ in the system, and q rules of the kind above. Suppose the rule above is the j'th rule, with $\mathbf{T}_i \equiv \mathbf{D}_{ji}$, and $\mathbf{T} \equiv \mathbf{D}_j$. Then the corresponding *closure condition* $\Phi_j \equiv \Phi_j^{\mathbf{D}}[\vec{R}]$ is the first-order closure of the formula

$$R_{j1}(x_1) \rightarrow \cdots \rightarrow R_{jk}(x_k) \rightarrow R_j(f(\vec{x})),$$

and the *predicate defining* $\mathbf{D}_i$ is

$$T_i(z) \equiv \forall \vec{R}\,(\Phi_1 \rightarrow \cdots \rightarrow \Phi_m \rightarrow R_i(z)).$$

(Here $\rightarrow$ associates to the right, and $\vec{R}$ is $R_1 \cdots R_m$.)

PROPOSITION 1.1. Suppose given a theory of computation over a data-system $\mathbf{D}$. The canonical terms of the theory, together with the explicitly defined data-types, constitute a coherent syntactic implementation (the *canonical model*) of $\mathbf{D}$. All coherent implementations are isomorphic to the canonical model.

PROOF. Immediate from the definitions. □

A slightly more general definition of a theory of computation allows in the clauses above that some of the T_i's be free predicate variables. If this is the case we say that the explicit definition generated by the clauses is a *generic* type definition. Another generalization is to allow data types to be sets of k-ary tuples for $k \geq 2$. That is, an operator $\mathbf{f}$ in a closure condition may be in fact a k-ary tuple of operators $\vec{\mathbf{f}}$.

As mentioned in the introduction, the *functional programs* of a theory of computation are Herbrand-Gödel systems of recursion equations. Each recursion equation is of the form

$$f(\vec{t}) = s$$

where f is a non-primitive function letter, $\vec{t}$ is a tuple of terms, and s is a term. We do not restrict ourselves to the type of the natural numbers, to functions totally defined for any particular type, or to functions with arguments restricted to any particular type. However, if we wish to have coherent implementations to the data-types, as is most often the case, we must assume that the functional programs are consistent with the coherence conditions. That is, if $t = s$ is a closed instance of a recursion equation, then t and s reduce (modulo the programs of the theory) into exactly the same set of canonical terms. We call a system of functional programs *coherent* if it satisfies this condition[4].

The non-logical constants of a theory are the primitive symbols, plus the function letters and constants used in the systems of recursion equations. The non-logical part of the theory consists of a single inference rule schema that expresses extensionality for definitional equality:

Replacement Rule
$$\frac{\varphi[t/x]}{\varphi[t'/x]}\ ,$$

where $t' = t$ or $t = t'$ is a substitution instance of a recursion equation in one of the systems considered.[5]

Examples of theories of computation.

Example 1: Boolean Arithmetic.

The primitives are the individual constants $\mathbf{T}$ and $\mathbf{F}$. There is no primitive function. The only type is that of Booleans, defined by

$$\Phi_1^{\mathbf{B}} \equiv R\mathbf{F},\ \Phi_2^{\mathbf{B}} \equiv R\mathbf{T}$$

i.e.

$$B(x) \equiv \forall R^1(R\mathbf{F} \rightarrow R\mathbf{T} \rightarrow Rx).$$

The equations are:

$$neg(\mathbf{F}) = \mathbf{T};\ neg(\mathbf{T}) = \mathbf{F};$$
$$con(x,\mathbf{T}) = con(\mathbf{T},x) = x;$$
$$dis(x,\mathbf{F}) = dis(\mathbf{F},x) = x.$$

---

2) An equivalent, and more elegant, formulation uses predicates defined by abstraction, such as $\lambda x_1 \cdots \lambda x_n \psi$; it is less attractive for our purposes.

3) Thus, an implementation of the data-type **natural-number** is coherent iff it satisfies Peano's Third and Fourth Axioms, $Sx \neq 0$, and $Sx = Sy \rightarrow x = y$.

Related to boolean arithmetic are the generic *conditional* constructs:

$$ifthen(\mathrm{T},x) = x;$$
$$ifthenelse(\mathrm{T},x,y) = x;\ ifthenelse(\mathrm{F},x,y) = y.$$

Example 2: Arithmetic of general recursive functions.

The primitives are the constant 0 and the unary function letter S, denoting the successor function. The only type is that of natural numbers, determined by the closure conditions

$$\Phi_1^N \equiv R(x) \rightarrow R(Sx), \text{ and } \Phi_2^N \equiv R0,$$

i.e.,

$$N(x) \equiv \forall R^1[\forall z(Rz \rightarrow R(Sz)) \rightarrow (R0 \rightarrow Rx)].$$

Coherent systems can be generated effectively for all computable functions over the natural numbers, as the proof in [Kle,§57], that all μ-recursive function are Herbrand-Gödel computable, shows. Here is a somewhat simplified version of Kleene's construction. Every general recursive function $f(\vec{x})$ is definable as $f(\vec{x}) = \mu y.[g(\vec{x},y)=0]$, where g is primitive recursive and therefore defined by a (coherent) system of recursion equations. f is then defined by supplementing the system for g with the following equations (with fresh function letters).

$$a(u,Sv) = u;$$
$$b(\vec{x},0) = g(\vec{x},0);\ b(\vec{x},Su) = a(g(\vec{x},Su),b(\vec{x},u));$$
$$c(u,0)=u;$$
$$f(\vec{x}) = c(u,b(\vec{x},u)).$$

Note that $b(\vec{x},y)$ is the same as $g(\vec{x},y)$ for values of y up to and including the first zero of the function, and is undefined for larger values of y.

Example 3: S-expressions over a finite set of atoms.

The primitives are atoms $\mathrm{a}_1, \ldots, \mathrm{a}_k$, and a binary function letter *cons*. The type of *atoms* is defined by

$$A(x) \equiv \forall R\ [R\mathrm{a}_1 \rightarrow \cdots \rightarrow R\mathrm{a}_k \rightarrow Rx].$$

The type of *expressions* is defined by the conditions

$$\Phi_1^{SE} \equiv A(z) \rightarrow R(z)$$
$$\Phi_2^{SE} \equiv Rx \wedge Ry \rightarrow R(cons(x,y)).$$

Among the recursion equations we must include $head(cons(x,y))=x$, and $tail(cons(x,y))=y$.

Example 4: Pairing and tupling.

These are purely generic types that can be combined with any theory. The type of pairs (of objects of type Q, where Q is a free relation variable), is defined by

$$\Phi^{P(Q)} \equiv \forall x,y\ (Q(x) \wedge Q(y) \rightarrow R(x,y)).$$

2. Type-correctness and termination of programs.

Let P be a functional program, with principal function-letter f of arity k. A structure is a *model of* P if all equations in P are universally true therein. P is *correct for* data-types $\mathrm{D}_1, \cdots, \mathrm{D}_k; \mathrm{D}_0$ of a data-system $\mathbf{D}$ if the formula

$$T_1(x_1) \rightarrow \cdots \rightarrow T_k(x_k) \rightarrow T_0(f(x_1,\ldots,x_k)) \qquad (*)$$

is true in every model of P, where T_i is the explicit (second-order) definition of D_i.

PROPOSITION 2.1. Let P, $\mathbf{D}$ and D_i be as above. The following statements are equivalent.
(i) The least fixed point of P, over the canonical implementation of $\mathbf{D}$, converges on all input from $\mathrm{D}_1 \times \cdots \times \mathrm{D}_k$, with output in D_0.
(ii) P is correct for $\mathrm{D}_1, \cdots, \mathrm{D}_k; \mathrm{D}_0$. □

4) A coherent system always computes a (single-valued, partial) function, because if $t=s$ is *any* derived equation between closed terms, then t and s reduce to the same irreducible terms, in particular when s is itself an irreducible term.
5) In the system based on classical logic it suffices to use the inference rule only when $t'=t$ is an instance of a recursion equation (or only when $t=t'$ is).

A particular case of interest is the data-type of natural numbers. The proposition implies that a program converges in all models if it converges in all models satisfying $Sx \neq 0$ and $Sx=Sy \rightarrow x=y$. Since the proof of the proposition can be carried in Second-Order Logic, it follows that Peano's axioms can be eliminated from any proof that a given program over natural numbers is total.

More generally, we have

COROLLARY 2.2. Suppose that an algorithm is proved correct for a certain data-system, using as axioms statements about the data-objects that are valid in the canonical interpretation. Then there is a proof of the same statement which makes no use of those axioms. □

Traditionally, properties of number-theoretic general-recursive functions are proved in Second-Order Arithmetic via codes for algorithms and computations. The main tool is Kleene's (primitive-recursive) predicate $T(e,x,z)$, expressing that "e is the code of a μ-recursion algorithm which, on the x'th numeral as an input, generates a complete computation-list with numeric-code z". The companion of T is the "result-extracting" function U which, on z as above as input, produces the terminal value of the computation coded by z. This rather heavy-handed machinery has been extended in [Kle69], to permit semi-formal reasoning about general-recursive functions to proceed more smoothly. Using this notation, a μ-recursion algorithm with code e is everywhere converging exactly when $\forall x\ \exists\ y.T(e,x,y)$.

The connection between the two definitions is given by the following proposition, whose proof is straightforward but rather lengthy.

PROPOSITION 2.3. Let f be a unary general-recursive function, computed by the functional algorithm P, as well as by the μ-recursion algorithm with code e. (The restriction to unary function is inessential.) Then $\forall x\ \exists\ y.T(e,x,y)$ is true in the standard model of Arithmetic iff P is correct for N; N. □

The equivalence above is provable in (Minimal) Second-Order Logic, and therefore a general-recursive function is provably-total in (Minimal /Constructive /Classical) Second-Order Arithmetic in the sense of Kleene iff it has a functional algorithm provably correct for N; N in (Minimal /Constructive /Classical) Second-Order Logic.

Moreover, we know that if a general-recursive function is provably total in Classical Second-Order Arithmetic then it is provable total even in in Minimal Second-Order Arithmetic (see [Fri]). It follows that:

COROLLARY 2.4. For f, P and e as above, $\forall x\ \exists\ y.T(e,x,y)$ is provable in Classical Second-Order Arithmetic iff P can be proved correct for N; N in Minimal Second-Order Logic. □

3. Lambda-representation of type-correct programs.

3.1. Mapping from a theory of computation into the Second-Order Lambda Calculus.

We refer to the Second-Order Lambda Calculus as described in [FLO]. We use the relational variables of second order logic as type variables.

By corollary 2.4 we may restrict ourselves to Minimal Second-Order Logic, where there are no symbols nor rules for negation or false.

Let φ be a simple formula. To φ there corresponds a second-order type τ^{φ}, obtained by deleting from φ the first-order components, that is: replacing each atomic formula $R(\vec{t})$ by R, and deleting first order quantifiers. For example, to the formula $N(x)$ corresponds the type $\forall R.((R \rightarrow R) \rightarrow (R \rightarrow R))$, which is precisely the Fortune-O'Donnell type of the natural numbers [FLO].

Let Δ be a simple derivation deriving a formula φ. To Δ there corresponds an expression E_Δ of the Second-Order Lambda Calculus, of type τ_φ, as follows.

- If Δ consists of an open assumption φ, then E_Δ is a variable x_φ of type τ_φ, to be distinct from all variables assigned to other (labeled) open assumptions.
- If Δ is obtained from a derivation Δ_0 by implication introduction,

$$\dfrac{\begin{array}{c}[\psi]\\ \Delta_0\\ \chi\end{array}}{\psi\to\chi}$$

then E_Δ is $\lambda x_\psi.E_{\Delta_0}$.

- If Δ is obtained from derivations Δ_0 and Δ_1 by implication elimination,

$$\dfrac{\begin{array}{cc}\Delta_0 & \Delta_1\\ \psi\to\varphi & \psi\end{array}}{\varphi}$$

then E_Δ is $E_{\Delta_0}E_{\Delta_1}$.

- If Δ is obtained from a derivation Δ_0 by second order universal quantifier introduction, with quantification over the variable R, then E_Δ is $\Lambda R.E_{\Delta_0}$.
- If Δ is obtained from a derivation Δ_0 by second order universal quantifier elimination,

$$\dfrac{\begin{array}{c}\Delta_0\\ \forall R\,\psi\end{array}}{\psi[\chi/R]}$$

then E_Δ is $E_{\Delta_0}\tau_\chi$.

- If Δ is obtained from Δ_0 by a first order universal quantifier rule or by Replacement, then E_Δ is E_{Δ_0}.

This terminates the description of the mapping of derivations into λ-expressions.

The mapping can be extended to apply to the remaining logical constants, using the definition of these constant in terms of implication and universal quantification. One useful example is conjunction. The conjunction of formulas φ and ψ can be defined by

$$\varphi\&\psi \equiv \forall R^0((\varphi\to\psi\to R)\to R),$$

where we use $\&$ in place of $\wedge$ to emphasize that we think of a defined notion. The straightforward normal proof of $\varphi\&\psi$ from φ and ψ is mapped into the λ-expression

$$\Lambda R.\lambda x^{\tau_\varphi\to\tau_\psi\to R}xu^{\tau_\varphi}v^{\tau_\psi},$$

where u and v are the variables corresponding to the open assumptions φ and ψ.

Therefore, we can introduce to the definition of our mapping clauses for conjunction as follows.

- If Δ is obtained from derivations Δ_0 and Δ_1 by conjunction introduction,

$$\dfrac{\begin{array}{cc}\Delta_0 & \Delta_1\\ \varphi & \psi\end{array}}{\varphi\&\psi}$$

then E_Δ is $\Lambda R.\lambda x^{\tau_\varphi\to\tau_\psi\to R}xE_{\Delta_0}E_{\Delta_1}$.

- If Δ is obtained from a derivation Δ_0 by conjunction elimination, say

$$\dfrac{\begin{array}{c}\Delta_0\\ \varphi\,\&\,\psi\end{array}}{\varphi},$$

then E_Δ is $E_{\Delta_0}\tau_\varphi P$, where $P \equiv \lambda x^{\tau_\varphi}\lambda y^{\tau_\psi}.x$.

3.2. Lambda-representation of typed objects.

Suppose $\mathbf{D}$ is a data-type, p a canonical term which is a data-object in the canonical model of $\mathbf{D}$. Then p is uniquely determined by the tree of clauses used to put it in $\mathbf{D}$. Corresponding to this tree there is a natural-deduction proof Π that the term p satisfies the explicit definition T of type $\mathbf{D}$. By proposition 1.1 Π is uniquely determined. (Π is in Minimal Second-Order Logic, and in fact is essentially first-order, because the only seond-order quantifiers in $T(p)$ are positively-occuring universal quantifiers.) To that proof corresponds, in turn, an expression E_p of the Second-Order Lambda Calculus as above. Moreover, since the closure conditions Φ_i^T are given in a specific order, E_p determines the tree of clauses in the construction of p, and therefore determines p.

If T is a generic type, then there is no closed term of type T. However, if the free relation variables in the definition of T are $Q_1,..,Q_k$, all unary say, and t is a term where the free variables are $x_1,...,x_k$, then the λ-expression E corresponding to a derivation of

$$\forall Q_1...Q_k(Q_1(x_1)\to\cdots\to Q_k(x_k)\to T(t[x_1\cdots x_k]),$$

for example, uniquely determines the structure of t. Moreover, if T^* is obtained by instantiating each Q_i to a closed type T_i, and E_i is the λ-expression corresponding to a proof of $T_i(t_i)$, $t^* \equiv t[t_1/x_1,\ldots,t_k/x_k]$, then $ET_1...T_kE_1...E_k$ is an expression corresponding to a proof of $T^*(t^*)$. The λ-expression E codes, therefore, the meaning of t as a generic term of a generic type.

Example 1: Booleans.

A straightforward proof of $B(\mathbf{T})$ is

$$\begin{array}{c}R(\mathbf{T})\\ R(\mathbf{F})\to R(\mathbf{T})\\ R(\mathbf{T})\to R(\mathbf{F})\to R(\mathbf{T})\\ \forall R\,(R(\mathbf{T})\to R(\mathbf{F})\to R(\mathbf{T}))\end{array}$$

to which corresponds the λ-expression $\Lambda R.\lambda x^R.\lambda y^R.x$. Similarly, to $\mathbf{F}$ corresponds the λ-expression $\Lambda R.\lambda x^R.\lambda y^R.y$. Stripped of the type structure the expressions are $\lambda x\lambda y.x$ for $\mathbf{T}$, and $\lambda x\lambda y.y$ for $\mathbf{F}$, precisely the usual representations of the Booleans constants in the untyped Lambda Calculus.

Example 2: Natural numbers.

Consider the numeral for 2, $SS0$. A straightforward proof of $N(SS0)$ is given in figure 1. The λ-expression corresponding to this proof is

$$\Lambda R.\lambda x^{R\to R}.\lambda y^R.x(x(y)).$$

Clearly, the same process yields, for the n'th numeral $S^{(n)}(0)$, the λ-expression

$$\Lambda R.\lambda x^{R\to R}.\lambda y^R.x^{(n)}(y),$$

which is precisely the Fortune-O'Donnell numeral for n in the Second Order Lambda Calculus. Stripped of its type structure the expression is $\lambda x.\lambda y.x^{(n)}(y)$, that is — Church's numeral for n in the untyped Lambda Calculus.

Example 3: Even natural numbers.

This example will illustrate the possibility of representation of the same data-object by different λ-expressions for different data-types. This possibility arises when we slightly liberalize the definition of data-types, and allow, in place of a generated expression $f(\vec{x})$ in closure rules, an expression of the form $t[\vec{x}]$, where t is a term, with the stipulation of unique parsing *within each type* still applying. With this, we may define overlapping types too.

The closure rules for the even numbers are

$$\Phi_1^{\mathbf{E}} \equiv R(x)\to R(SSx), \text{ and } \Phi_2^{\mathbf{E}} \equiv R(0).$$

The λ-expression representing the natural number $2n$ in $\mathbf{E}$ is now the same as the λ-expression representing n in the type $\mathbf{N}$ of the natural numbers.

Example 4: S-expressions.

Suppose the atoms are $\mathbf{a}_1,\ldots,\mathbf{a}_k$. As for the Boolean constants one obtains the λ-expression $\Lambda R.\lambda x_1^R\cdots\lambda x_k^R.x_i$ as the representation $\mathbf{a}_i^A$ of the primitive constant $\mathbf{a}_i$ in as elements of the type A corresponding to the definition of the data type "atom", namely $\setminus\!/R.R\to R\cdots\to R$, with k $\to$'s. From the proof of $E(\mathbf{a}_i)$ we get the representation $\mathbf{a}_i^E$ of $\mathbf{a}_i$ *in* E, $\Lambda R.\lambda x^{R\to R\to R}.\lambda y^{A\to R}.y(\mathbf{a}_i^A)$.

If b and c are S-expressions with corresponding representations b^E and c^E in the Second Order Lambda Calculus, then from the proof of $E(cons(b,c))$ we obtain the representation $cons(b,c)^E$ of $cons(b,c)$,

$$\Lambda R.\lambda x^{R\to R\to R}.\lambda y^{A\to R}.x(b^E Rxy)(c_E Rxy),$$

which, depending on the forms of b^E and c^E, can be reduced by β-conversions.

For example, for the S-expressions on the left in figure 2 we obtain the corresponding representations on the right.

The pattern is obvious, and so is the representation obtained in the untyped λ-Calculus:

Example 5: Pairs and tuples.

The generic pair term is (x,y), where x and y are variables. Corresponding to the straightforward derivation of

$$\begin{aligned}&\forall Q_1^1Q_2^1.\forall x,y\ (Q_1(x)\to Q_2(y)\to P[Q_1,Q_2](x,y))\\ &\equiv \forall Q_1^1Q_2^1.\forall x,y\ (Q_1(x)\to Q_2(y)\to\\ &\quad \forall R^2((Q_1(x)\to Q_2(y)\to R(x,y))\to R(x,y))\end{aligned}$$

we obtain the λ-expression

$$\mathbf{pair}\equiv \Lambda Q_1Q_2.\lambda x^{Q_1}\lambda y^{Q_2}.\Lambda R.\lambda z^{\alpha}.\ zxy,$$

where $\alpha\equiv Q_1\to Q_2\to R$. Note that, given formulas φ, ψ, $\mathbf{pair}_{\tau_\varphi\tau_\psi}$ is the λ-expression corresponding to the inference rule of conjunction introduction, as described in §3.1. Stripped of its type structure **pair** becomes

$$\lambda x\lambda y\lambda z.zxy.$$

Similarly, to a k-tuple of variables corresponds the λ-expression obtained from a proof of

$$\begin{aligned}&\forall Q_1^1\ldots Q_k^1.\forall x_1,\ldots,x_k\,[Q_1(x_1)\to\cdots\to Q_k(x_k)\to P^k[\vec{Q}](x_1,\ldots,x_k)]\\ &\quad\equiv \forall\vec{Q}.\forall\vec{x}\ [\vec{Q}(\vec{x})\to\forall R^2((\vec{Q}(\vec{x})\to R(\vec{x}))\to R(\vec{x}))],\end{aligned}$$

that is,

$$\Lambda\vec{Q}.\lambda\vec{u}^{\vec{Q}}.\Lambda R\lambda v^{\vec{Q}\to R}.v\vec{u}.$$

3.3. Lambda-representation of functions.

Consider a k-ary recursive function $\mathbf{f}$ that, for arguments of types $\mathbf{T}_1,\ldots,\mathbf{T}_k$, always yields a result of type $\mathbf{T}_0$. Let $\mathbf{C}$ be a theory of computation whose data types include $\mathbf{T}_0,\ldots,\mathbf{T}_k$, with function letter f representing $\mathbf{f}$ by virtue of recusion equations. Then the formula

$$\begin{aligned}\varphi&\equiv \forall x_1^{T_1}\ldots x_k^{T_k}.\ T_0(f(x_1,\ldots,x_k))\\ &\equiv \forall x_1\ldots x_k\ [T_1(x_1)\to\cdots\to T_k(x_k)\to T_0(f(x_1,\ldots,x_k))]\end{aligned}$$

is true.

Suppose that φ is not only true, but provable, and let Δ_0 be a derivation (in $\mathbf{C}$) of φ. For $i=1\ldots k$ let t_i be a canonical term of type T_i. t_i is *provably* of type T_i, because it is canonical. Let Δ_i be a derivation of $T_i(t_i)$. We obtain a derivation Δ of $T_0(f(t_1,\ldots t_k))$ by composing the different proofs, as in figure 3.

Consider now the λ-expressions corresponding to all these proofs. If E_i is the expression corresponding to Δ_i $(i=0,\ldots,k)$, then the expression corresponding to Δ is $E_0E_1\ldots E_k$. We have thus proved:

THEOREM 3.1. Let Δ be a derivation of

$$\forall x_1^{T_1}\ldots x_k^{T_k}.\ T_0(f(x_1,\ldots,x_k)),$$

where f is a function letter denoting a k-ary function $\mathbf{f}$ from the types (defined by) $T_1,\ldots,T_k$ to type T_0. Let E be the second order lambda expression corresponding to Δ. Then E represents $\mathbf{f}$ in the Second Order Lambda Calculus.□

It is known that all functions representable in the Second-Order Lambda Calculus can be proved total in Second-Order Logic (see [Gir],[FLO,§6.4,6.5). Combining this with the theorem we obtain

COROLLARY 3.2. A recursive function between inductively defined types is provably type-correct in Second-Order Logic iff it is representable in the Second-Order Lambda Calculus. In particular, a number theoretic function is representable in the Second-Order Lambda Calculus iff it is provably recursive in (Classical) Second-Order Logic. □

The numeric case of the corollary for was proved by Richard Statman [Sta].

A slightly more general situation arises with generic functions, such as *if then else*. Suppose T_0 is a generic type whose definition uses the free relation variables $Q_1,\ldots,Q_m$, and suppose $T_1,\ldots,T_k$ are types. Assume that f is a function whose functional programs implies the following statement

$$\begin{aligned}&\forall x_1,\ldots,x_k\ [T_1(x_1)\to\cdots\to T_k(x_k)\to\\ &\quad\forall Q_1\ldots,Q_m\ \forall y_1\ldots,y_m\ [\vec{Q}(\vec{y})\to T_0[\vec{Q}](f(\vec{x},\vec{y}))]],\end{aligned}$$

Then to a proof of the latter formula corresponds a λ-expression E, with the property that, given terms $s_1,\ldots,s_k$ of types $S_1,\ldots,S_k$ respectively, if F_i is the λ-expression corresponding to a proof of $S_i(s_i)$, and σ_i is the type expression corresponding to the explicit definition of S_i, then $E\vec{\sigma}\vec{F}$ is an expression corresponding to a proof of $T^*(\iota[s_1,\ldots,s_k])$, where T^* is T with Q_i instantiated to S_i.

3.4. Examples of function representations.

We give here examples of functions over Booleans and over natural numbers only.

Example 1: The successor function.

A straightforward normal proof of $N(x)\to N(Sx)$ is given in figure 4. The λ-expression corresponding to this derivation is

$$\lambda n^\iota.\Lambda R.\lambda x^{R\to R}.\lambda y^R.x(nRxy),$$

where $\iota\equiv\setminus\!/R.(R\to R)\to(R\to R)$. The untyped form is $\lambda n\lambda x\lambda y.x(nxy)$, which is the usual representation of the successor function in the untyped Lambda Calculus.

Already for the successor function we can see how differnet (normal) derivations of the same correctness statement can lead to different λ representations. An alternative derivation of $N(x)\to N(Sx)$ is given in figuer 5.

The λ-expression corresponding to this derivation is

$$\lambda n^\iota.\Lambda R.\lambda x^{R\to R}.\lambda y^R.nRx(xy),$$

and the untyped form is

$$\lambda n\lambda x\lambda y.nx(xy).$$

Example 2: Conjunction.

A straightforward normal proof of

$$\forall xy\ (B(x)\to B(y)\to B(conj(x,y)))$$

is given in figure 6.

The corresponding λ-expression is

$$\lambda x^B.\lambda y^B.\Lambda R.\lambda u^R.\lambda v^R.((xR)(yRuv))v,$$

where $B\equiv\Delta R.R\to R\to R$, with the untyped form simplifying to

$$\mathbf{conj}\equiv\lambda x.\lambda y.\lambda u.\lambda v.x(yuv)v.$$

Example 3: if-then-else.

To the straightforward proof of

$$\forall z\ [B(z)\to\forall Q\forall x,y\ (Q(x)\to Q(y)\to Q(if then else(z,x,y)))]$$

corresponds the typed λ-expression

$$\lambda z^B.\Lambda Q.\lambda x^Q\lambda y^Q.\ zQxy,$$

which, stripped of its type structure, becomes $\lambda z\lambda x\lambda y.\ zxy$. This is the same as the untyped λ-expression for the generic pair, obtained in §2.3., which indeed is the usual representation of pairing in the untyped Lambda Calculus. Note, however, that the typed versions *differ*.

Example 4: Addition.

Given the usual recursion equations for addition,

$$a(x,0) = x;\ a(x,Sy) = Sa(x,y),$$

a natural normal derivation of $N(x) \to N(y) \to N(a(x,y)$ is given in figure 7.

The corresponding λ-expression is

$$\lambda x^N.\lambda y^N.\Lambda R.\lambda f^{R\to R}.\lambda z^R.\ yR(\lambda u^R.fu)(xRfz),$$

which reduces by an η-reduction to

$$\lambda x^N.\lambda y^N.\Lambda R.\lambda f^{R\to R}.\lambda z^R.\ yRf(xRfz),$$

the Fortune-O'Donnell representation of addition in the second order λ-calculus [FLO]. The untyped version is

$$\lambda x.\lambda y.\lambda f.\lambda z.\ yf(xfz),$$

precisely Church's representation of addition in the (untyped) λ-calculus [Bar].

From a somewhat different proof of type-correctness of addition we obtain a different λ-representation. Consider the derivation given in figure 8.

The typed λ-expression corresponding to this proof is

$$\lambda x^N.\lambda y^N.yN(\lambda u^N.\Lambda R.\lambda f^{R\to R}.\lambda z^R.f(uRfz))x,$$

with the corresponding untyped version

$$\lambda x.\lambda y.y(\lambda u.\lambda f.\lambda z.f(ufz))x.$$

Example 5: Predecessor.

The equations defining the predecessor function are

$$pred(0) = 0;\quad pred(Sx) = x.$$

To prove that $N(x) \to N(pred(x))$ is equivalent to proving $N(Sx) \to N(x)$. The latter is not completely trivial, because implicit in such a proof is the idea of "remembering one step back". A straightforward proof of $N(x) \to N(pred(x))$ that uses that idea is given in figure 9, where we use $\varphi[z]$ as an abbreviation for $R(pred(z)) \wedge R(z)$.

To define the corresponding λ expression we use the expression **pair** defined in §3.2, we let $Q_1 \& Q_2$ abbreviate the type

$$\forall R\ ((Q_1 \to Q_2 \to R) \to R,$$

and we define the expression

$$\mathbf{first} \equiv \Lambda Q_1 Q_2.\lambda u^{Q_1 \& Q_2} u Q_1 (\lambda x_1^{Q_1} \lambda x_2^{Q_2}.x_1).$$

Then **first** corresponds to the first rule of conjunction elimination, as described in §3.1. Given these abbreviations, the λ-expression corresponding to the proof above is

$$\lambda n^{\iota}.\Lambda R\,\lambda x^{R\to R}\lambda y^R$$
$$\mathbf{first}(n(R\&R)(\lambda u^{R\&R}\mathbf{pair}RR(\mathbf{first}RRu)(x(\mathbf{first}RRu)))(\mathbf{pair}RRyy)).$$

Stripped of its type structure, this expression becomes

$$\lambda n\lambda x\lambda y.\ \mathbf{first}(n(\lambda u\,\mathbf{pair}(\mathbf{first}u)(x(\mathbf{first}u))(\mathbf{pair}yy)),$$

where the expressions **pair** and **first** are understood to blso be stripped of their type structure. The reader will recognize here Kleene's definition of the successor function in the untyped Lambda Calculus [Bar].

References.

[Bar] Henk Barendregt, *The Lambda Calculus*, North Holland, Amsterdam-New York-Oxford, 1981, xiv+615pp.

[CF] H.B. Curry and R. Feys, *Combinatory Logic*, North-Holland, Amsterdam-New York-Oxford, 1958.

[Chu] Alonzo Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic 5* (1940), pp 56-68.

[Cla] Edmund M. Clarke, "Programming language constructs for which it is impossible to obtain good Hoare-like axioms, *Journal of the ACM 26* (1979), pp 129-147.

[FLO] S. Fortune, D. Leivant, and M. O'Donnell, "The expressiveness of simple and second order type structures," *Journal of the ACM 30* (1983), pp 151-185.

[Fri] Harvey Friedman, "Classically and intuitionistically provably recursive functions," in: *G. Muller and D. Scott (eds.), Higher Set Theory (LNM 669)*, Springer-Verlag, Berlin, 1977.

[Gir] J.-Y. Girard, *Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur*, These de Doctorat d'Etat, 1972, Paris.

[GMW] Michael J. Gordon, Arthur J. Nilner and Christopher P. Wadsworth, *Edinburgh LCF*, Springer, Berlin-Heidelberg-New York (LNCS #78), 1979.

[Göd] Kurt Gödel, "Ueber eine bisher noch nicht benutzte Erweiterung des finiten standpunktes," *Dialectica 12* (1958), pp 280-287.

[Gur] Yuri Gurevitch, "Algebras of feasible functions," these *Proceedings*.

[Har] David Harel, *First-Order Dynamic Logic*, Springer (LNCS #68), Berlin-Heidelberg-New York, 1979, 133pp.

[Hen] Leon Henkin, "Completeness in the Theory of Types," **Journal of Symbolic Logic 15** (1950), pp. 81-91.

[How] William A. Howard, "The formulae-as-types notion of construction," pp. 479-490 in [SH].

[Kle] S.C. Kleene, *Introduction to Metamathematics*, Noordhoff, Groningen, 1952.

[Kle69] S.C. Kleene, *Formalized Recursive Functions and Formalized Realizability*," Memoirs of the AMS 89 (1969).

[Lau] H. Lauchli, "An abstract notion of realizability for which intuitionistic predicate calculus is complete"; in *Intuitionism and Proof Theorey*, A. Kino, J. Myhill, R.E. Vesley, Editors (North-Holland, Amsterdam, 1970).

[Mar] P. Martin-Lof, "Constructive mathematics and computer programming," *Proceedings of the Sixth (1979) International Congress for Logic, Methodology and Philosophy of Science*, North-Holland, Amsterdam-New York-Oxford, 1979.

[MD] Kenneth L. Manders and Robert F. Daley, "The complexity of the validity problem for Dynamic Logic," manuscript, 1982.

[Men] Elliott Mendelson, *Introduction to Mathematical Logic*, Van Nostrand, Princeton, 1964.

[MP] Albert Meyer and Rohit Parikh, "Definability in Dynamic Logic," *Journal of Computer and System Science 23* (1981), pp. 271-298.

[Par] Rohit Parikh, "A decidability result for Second Order Process Logic", *Proceedings of the Seventeenth IEEE Symposium on Foundations of Computer Science*, Ann Arbor, 1978, 177-183.

[Pra] Dag Prawitz, *Natural Deduction*, Almqvist and Wiksell, Uppsala, 1965.

[Rey] J.C. Reynolds, "Towards a theory of type structures," in *Programming Symposium (Colloque sur la Programmation, Paris)*, Springer (Lecture Notes in Computer Science 19), Berlin, 1974, 408-425.

[Sch] Helmut Schwichtenberg, "The expressiveness of Martin-Löf programming language" [approximate title], manuscript, 1983.

[SH] J.P. Seldin and J.R. Hindley (editors), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, London, 1980, 606pp.*

[Sta] R. Statman, "Number theoretic functions computable by polymorphic programs," in *Twenty Second Annual Symposium on Foundations of Computer Science*, 1981, 279-282.

[Tai] W.W. Tait, "Infinitely long terms of transfinite type". In *Formal Systems and Recursive Functions*, Crossley and Dummett, Eds., North-Holland, Amsterdam, pp 176-185.

Figure 1

$$\dfrac{\dfrac{\dfrac{\forall z\,(Rz \to RSz)}{R(S0) \to R(SS0)} \qquad \dfrac{\forall z\,(Rz \to RSz)}{R(0) \to R(S0)} \qquad R(0)}{R(S0)}}{R(SS0)}$$

$$R(0) \to R(SS0)$$
$$\forall z\,(Rz \to RSz) \to R(0) \to R(SS0)$$
$$\forall R(\forall z\,(Rz \to RSz) \to R(0) \to R(SS0))$$

Figure 2

$cons(a_1,a_2)$

$$\Lambda R.\lambda x^{R\to R\to R}.\lambda y^{A\to R}.x(a_1^F Rxy)(a_2^F Rxy)$$
$$= \Lambda R.\lambda x^{R\to R\to R}.\lambda y^{A\to R}.x(ya_1^A)(ya_2^A)$$

$cons(cons(a_1,a_2),cons(a_3,a_4))$

$$\Lambda R.\lambda x^{R\to R\to R}.\lambda y^{A\to R}.x\quad (x(ya_1^A))(ya_2^A))$$
$$(x(ya_3^A))(ya_4^A))$$

$cons(a_1,cons(a_2,cons(a_3,a_4)))$

$$\Lambda R.\lambda x^{R\to R\to R}.\lambda y^{A\to R}.x\quad (ya_1^A))$$
$$(x\ (ya_2^A))$$
$$(x(ya_3^A))(ya_4^A)))$$

Figure 3

$$\dfrac{\dfrac{\dfrac{\forall x_1 \ldots x_k\,[T_1(x_1) \to \overset{\Delta_0}{\cdots} \to T_k(x_k) \to T_0(f(x_1,\ldots,x_k))]. \quad \overset{\Delta_1}{T_1(t_1)}}{T_1(t_1) \to \overset{\cdots}{\cdots} \to T_k(t_k) \to T_0(f(t_1,\ldots,t_k))]}}{T_2(t_2) \to \cdots \to T_k(t_k) \to T_0(f(t_1,\ldots,t_k))]} \qquad \dfrac{\overset{\cdots}{T_k(t_k) \to T_0(f(t_1,\ldots,t_k))} \quad \overset{\Delta_k}{T_k(t_k)}}{T(f(t_1,\ldots,t_k))}}{T(f(t_1,\ldots,t_k))}$$

Figure 4

$$\dfrac{\dfrac{\dfrac{\forall z(Rz \to RSz)}{Rx \to RSx} \qquad \dfrac{\dfrac{N(x)}{\forall z(Rz \to RSz) \to R0 \to Rx} \qquad \forall z(Rz \to RSz)}{R0 \to Rx} \qquad R0}{Rx}}{RSx}$$

$$R0 \to RSx$$
$$\forall z(Rz \to RSz) \to R0 \to Rx$$
$$\forall R[\forall z(Rz \to RSz) \to R0 \to Rx]$$

Figure 5

$$\dfrac{\dfrac{\dfrac{Nx}{\forall z(RSz \to RSSz) \to RS0 \to RSx} \qquad \dfrac{\forall z(Rz \to RSz)}{RSz \to RSSz} \qquad \forall z(RSz \to RSSz)}{RS0 \to RSx} \qquad \dfrac{\dfrac{\forall z(Rz \to RSz)}{R0 \to RS0} \qquad R0}{RS0}}{RSx}$$

$$R0 \to RSx$$
$$\forall z(Rz \to RSz) \to R0 \to Rx$$
$$\forall R[\forall z(Rz \to RSz) \to R0 \to Rx]$$

Figure 6

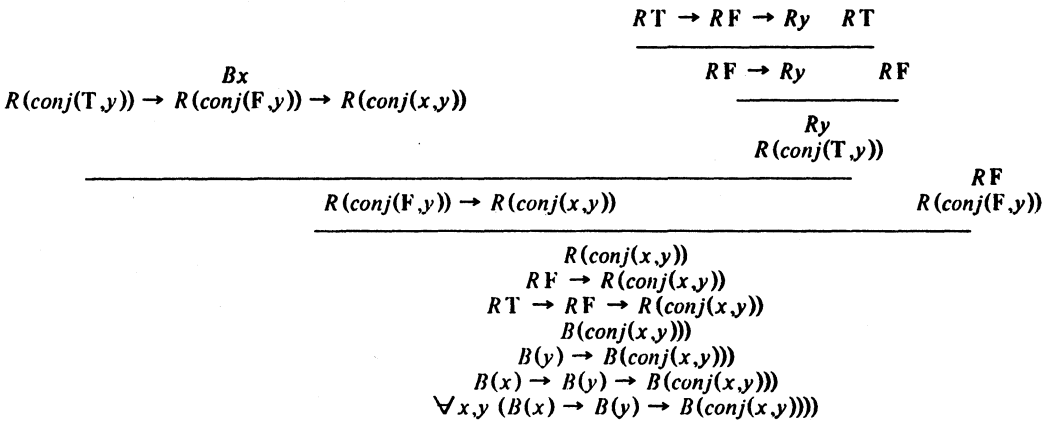


Figure 7

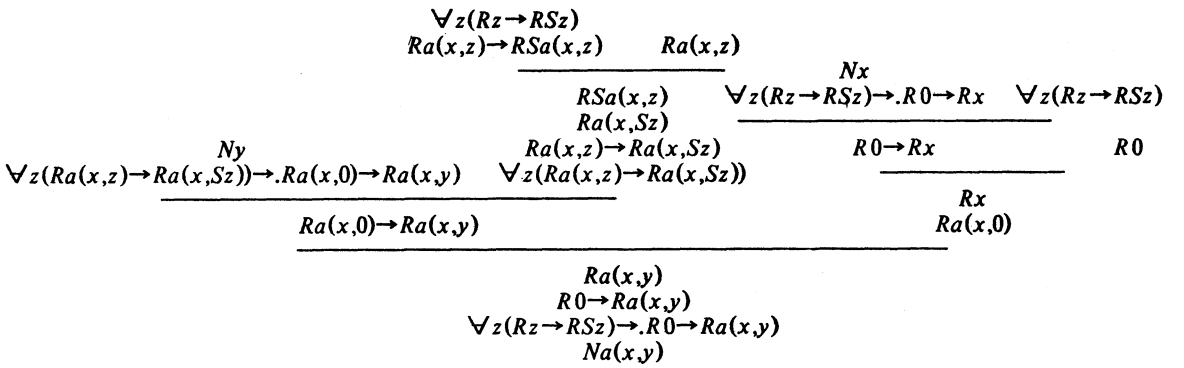


Figure 8

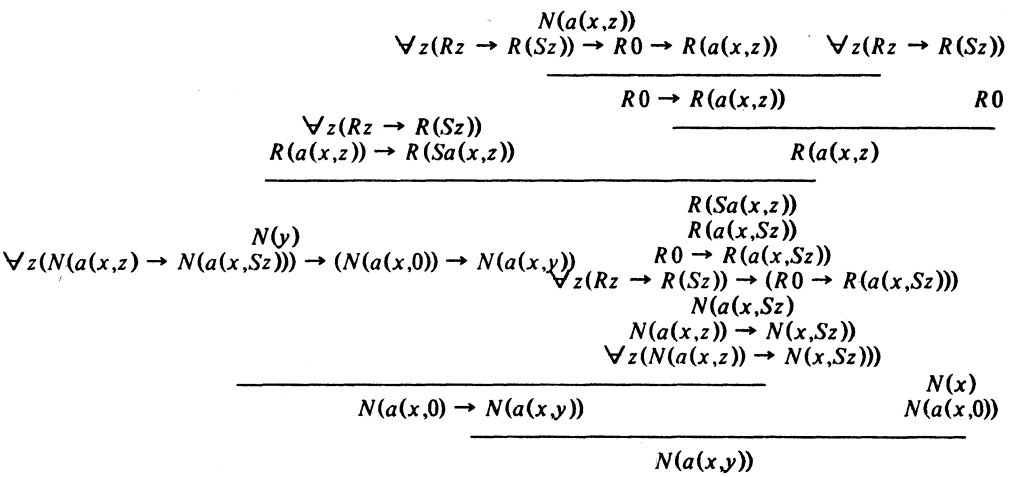
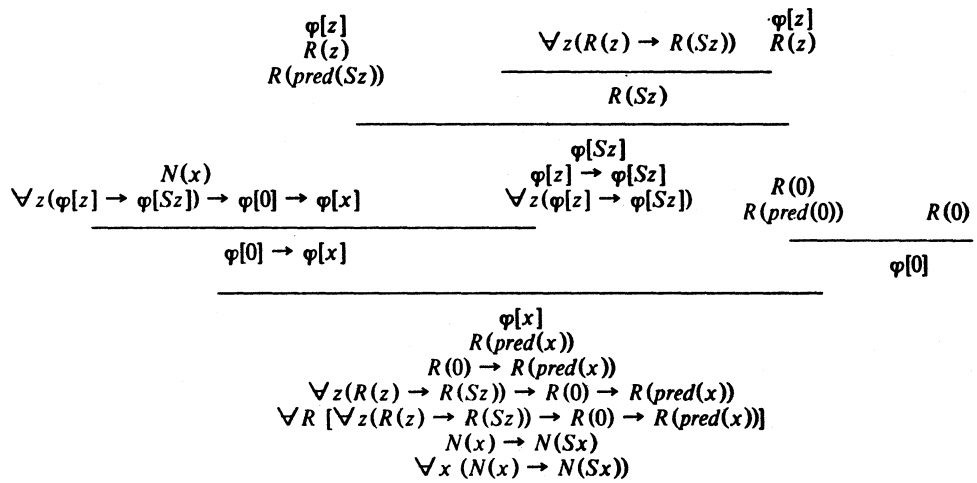


Figure 9


φ[z]
R(z)
R(pred(Sz))
∀z(R(z) → R(Sz))
φ[z]
R(z)
R(Sz)
φ[Sz]
φ[z] → φ[Sz]
∀z(φ[z] → φ[Sz])
N(x)
∀z(φ[z] → φ[Sz]) → φ[0] → φ[x]
φ[0] → φ[x]
R(0)
R(pred(0))
R(0)
φ[0]
φ[x]
R(pred(x))
R(0) → R(pred(x))
∀z(R(z) → R(Sz)) → R(0) → R(pred(x))
∀R [∀z(R(z) → R(Sz)) → R(0) → R(pred(x))]
N(x) → N(Sx)
∀x (N(x) → N(Sx))