'Type' Is Not a Type: Preliminary Report

Albert R. Meyer and Mark B. Reinhold
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract. A function has a *dependent type* when the type of its result depends upon the value of its argument. Dependent types originated in the type theory of intuitionistic mathematics and have reappeared independently in programming languages such as CLU, Pebble, and Russell. Some of these languages make the assumption that there exists a *type-of-all-types* which is its own type as well as the type of all other types. Girard proved that this approach is inconsistent from the perspective of intuitionistic logic. We apply Girard's techniques to establish that the type-of-all-types assumption creates serious pathologies from a programming perspective: a system using this assumption is inherently not normalizing, term equality is undecidable, and the resulting theory fails to be a conservative extension of the theory of the underlying base types. The failure of conservative extension means that classical reasoning about programs in such a system is not sound.

1. Introduction

Dependent types. A function has a dependent type when the type of its result depends upon the value of its argument.

A simple example of a function which has a dependent type is the unary function *zero_vector* which when applied to an integer n returns an n-vector of zeroes. No particularly appropriate type presents itself for the range of *zero_vector*, and hence *zero_vector* itself is typically not assigned a simple functional type. For this reason, a parameterized type constructor like $\mathrm{vector}(n)$, which denotes the type of integer vectors of length n, is a built-in feature of many programming languages. In programming languages with richer type systems, *e.g.*, CLU [18], a type constructor (or *parameterized cluster* in CLU terminology) such as $\lambda n{:}\mathrm{int}.\mathrm{vector}(n)$ can even be user-defined. In any case, the type of the value $\mathit{zero\_vector}(n)$ can be described as $\mathrm{vector}(n)$.

The function $\lambda n{:}\mathrm{int}.\mathrm{vector}(n)$ defines the type of the value of *zero_vector* at its argument n and thus is a good candidate for specifying the dependent type of *zero_vector*. However, to maintain a useful distinction between $\lambda n{:}\mathrm{int}.\mathrm{vector}(n)$ in its role as a function and in its role as a type, we use Π in place of λ as a syntactic marker for type expressions. That is, we write

$$\mathit{zero\_vector}{:}(\Pi n{:}\mathrm{int}.\mathrm{vector}(n)),$$

where ':' is read as 'has type'.

Ordinary function types can be regarded as a special case of dependent types. That is, the type $(s \to t)$ of functions from arguments of type s to values of type t is simply $(\Pi x{:}s.t)$ where x is chosen to be some fresh identifier.

A more provocative example involves finding a type for the function

$$f = (\lambda x{:}\mathrm{int}.\mathrm{if}\ (x = 0)\ \mathrm{then}\ 4\ \mathrm{else}\ \mathrm{true}).$$

That is, $f(0) = 4$, and $f(n) = \mathrm{true}$ for any integer $n \neq 0$. Typically, f would be considered untypable (or perhaps would be assigned some loophole type such as $\mathrm{int} \to \mathrm{any}$), but we can easily assign an informative dependent type to f, namely,

$$f{:}(\Pi x{:}\mathrm{int}.\mathrm{if}\ (x = 0)\ \mathrm{then}\ \mathrm{int}\ \mathrm{else}\ \mathrm{bool}).$$

Of course, allowing such a type for f may undermine the utility of the type system. The difficulty is that the type

This research was supported by NSF grant no. DCR-8511190.

involves the test $(x = 0)$ which may require the computation of an arbitrary integer expression. If that is the case, then constructive typechecking becomes unlikely, since the distinction between type computation and arbitrary computation disappears. Nevertheless, under suitable restrictions it is possible to define rich systems of dependent types, in which functions such as f above are typable, while retaining effective typechecking.

Polymorphic types. A particularly rich effectively-typed calculus is λ^Δ, the *polymorphic* (or *second-order*) λ-calculus [15, 29]. The polymorphic λ-calculus allows variables ranging over the class of all types (*second-order* variables), reflecting facilities available in Ada [1], CLU, Russell [4, 11], and other programming languages. Variables ranging over objects of some particular type are called *first-order* variables. (Note that a variable of type $\mathbf{int} \to \mathbf{int}$ ranging over functions would be called first-order in this context.) In λ^Δ, the first-order binding operators λ and Π are paralleled by the corresponding operators Λ and Δ, for second-order variables. For example, the polymorphic identity function I can be written $(\Lambda x.\lambda y{:}x.y)$. This means that I can be applied to any argument t which is a type, and returns as its value the identity function on t, *e.g.*, $I(\mathbf{int})$ is the identity function on integers which when in turn applied to any integer n yields n. Since the type of $I(t)$ is $t \to t$, we assign I itself the polymorphic type $(\Delta x.x \to x)$.

A striking feature of the polymorphic calculus is that a polymorphic function may be applied to any type, including its own: $I(\Delta x.x \to x)$ is the identity function on identity functions. This turns out to give the system enormous expressive power while retaining easy typechecking of a very informative kind [12]. (This kind of indirect circularity, reminiscent of the directly circular self-application possible in the untyped λ-calculus, rules out any naive set-theoretic explanation of the semantics of terms or types [5, 28].)

In particular, λ^Δ retains five surprising and valuable properties possessed by the more familiar finitely-typed λ-calculus [2, Appendix A]:

(1) Provable equality is characterized by a set of directed rewrite rules (reduction) satisfying the confluence (Church-Rosser) property.

(2) Reduction is strongly normalizing (terminating), *i.e.*, every way of applying reductions to a term eventually leads to a term in normal form which cannot be further reduced.

(3) The equational theory of terms is decidable.

(4) The theory of type assertions $a{:}t$ is decidable.

(5) The polymorphic theory of any algebraic structure is a conservative extension of the ordinary (*i.e.*, first-order in the usual sense) equational theory of the structure; in other words, the equations between terms of the algebra provable using λ^Δ are just those in the original theory.

Several other rich calculi with first- and second-order dependent types have been developed which retain these properties while allowing type constructors like those in the examples above [19, 21, 10, 3, 23, 24, 7].

The type-of-all-types assumption. The application of a polymorphic function to a type argument is evaluated by substitution (β-reduction) exactly as for an ordinary λ-expression; *e.g.*, $(\Lambda x.a)s$ can be reduced by substituting s for all free occurrences of the variable x in a. Because of the distinction between the first- and second-order binding operators λ and Λ (likewise Π and Δ), a duplicate set of computation rules is generally needed for first- and second-order terms. The system would be simpler if we could use λ- and Π-binding for second-order variables.

One comprehensive way to do this is to introduce a type-of-all-types, τ, such that $\tau{:}\tau$. Then, for example, the polymorphic identity function can be rewritten as $(\lambda x{:}\tau.\lambda y{:}x.y)$ and its type as $(\Pi x{:}\tau.x \to x)$. Further, $(\Pi x{:}\tau.x \to x)$ has in turn type τ since it is a type expression. Thus the combination of first-order dependent types and the type-of-all-types assumption subsumes λ^Δ, *cf.* [3]. Moreover, the vector type constructor used above can now easily be typed as $\mathbf{int} \to \tau$, and the if-then-else construct within the type expression

$$(\Pi x{:}\mathbf{int}.\mathbf{if}\ (x = 0)\ \mathbf{then}\ \mathbf{int}\ \mathbf{else}\ \mathbf{bool})$$

has type $(\mathbf{bool} \times \tau \times \tau) \to \tau$.

In short, the $\tau{:}\tau$ assumption extends and apparently simplifies systems of first- and second-order dependent types.

We will be concerned below with four distinct λ-calculi, namely:

λ^τ	the finitely typed (first-order) λ-calculus,
λ^Δ	the polymorphic (second-order) λ-calculus,
λ^Π	the typed λ-calculus with Π-types, and
$\lambda^{\tau\tau}$	λ^Π extended by the $\tau{:}\tau$ assumption.

The new system, λ^Π, defined in Section 2, is designed to be one of the simplest systems with first-order dependent types. Indeed, without the addition of further

axioms, λ^Π will be shown to be a notational variant of the finitely-typed λ-calculus, λ^τ. In other words, dependent types appear only in degenerate form in λ^Π. Choosing such a system strengthens our (negative) results, since the pathologies we exhibit when λ^Π is extended with the type-of-all-types assumption will surely apply to any less degenerate system.

Penalties of the type-of-all-types assumption. Programming languages such as Pebble [6] and λ-calculi such as Martin-Löf's theory of types [20] with type systems incorporating a type-of-all-types seem capable of satisfying two principal goals of a type system:

(6) Freedom from runtime type-errors.

(7) Representation independence of abstract data types [30, 11, 22].

Our objective in the rest of this paper is to demonstrate that other valuable properties, namely (2)–(5) above, fail even for the minimal type-of-all-types system $\lambda^{\tau\tau}$.

The failure of property (2) (strong normalization) is not by itself surprising, since in a general programming language one expects terms which define divergent computations not to have normal forms. Similarly, the failure of property (3) (undecidability of term equivalence) is to be expected. But the failure of property (4) means that it is not decidable whether a term has a given type, and more generally whether two types are equal. This undermines the possibility of effective "static" typechecking before runtime. Finally, the failure of property (5), conservative extension, is also serious as we indicate in Section 4 below.

The finitely typed λ-calculus, λ^τ, is well-known to satisfy the confluence and decidability properties (1)–(4), and can be proved trivially to satisfy property (5) (conservative extension) by an easy model-theoretic argument given in Section 4. It follows that "pure" λ^Π, which is a notational variant of λ^τ, also satisfies (1)–(5), whereas the extension of λ^Π to $\lambda^{\tau\tau}$ by the single axiom $\tau{:}\tau$ destroys everything but confluence.

We remark that in order to identify $\tau{:}\tau$ as the main culprit, we plan in our full paper also to argue that λ^Π is not so near the brink of undecidability that less forceful jolts than $\tau{:}\tau$ will throw it over as well. In particular, we will show that λ^Π can be safely extended in other ways, for example to subsume λ^Δ, while preserving properties (1)–(5).

Girard's "paradox". There is a purely formal correspondence, known as the *formulae-as-types* analogy, between some λ-calculi and intuitionistic logics [8, 16]. Types are seen as logical propositions, and (closed) λ-expressions are proofs of the propositions which are their types. The intuitionistically provable propositions are exactly those types t for which closed terms of type t exist. Intuitionistic absurdity is the proposition that all propositions are provable; in a λ-calculus this corresponds to the type $(\Pi x{:}\tau.x)$. So in λ-calculi where the formulae-as-types analogy holds, there are no closed λ-expressions of type $(\Pi x{:}\tau.x)$.

An early formulation of higher-order intuitionistic logic based upon the type-of-all-types assumption was laid out by Martin-Löf [20]; the λ-calculus corresponding to this logic subsumes $\lambda^{\tau\tau}$. Shortly after Martin-Löf's proposal, Girard [15] showed that the system was intuitionistically inconsistent. A closed term of type $(\Pi x{:}\tau.x)$ can actually be extracted from Girard's proof.

Intuitionistic inconsistency by no means implies that the associated λ-calculus is trivial; on the contrary, the theory is so rich that it is undecidable. From a programming perspective, the ability to define a term of type $(\Pi x{:}\tau.x)$ in a λ-calculus is not intrinsically objectionable and does not appear to have the negative consequences noted above.

However, one technical consequence of intuitionistic inconsistency turns out to be fairly immediate: the calculus is not strongly normalizing. Indeed, it is easy to show that any pure closed term of type $(\Pi x{:}\tau.x)$ has no normal form under the usual reduction rules.

One might hope to restore normalization by introducing some new notion of reduction. However, the construction of the term of type $(\Pi x{:}\tau.x)$ from Girard's proof indicates how to construct a fixed-point combinator which can then be used to show that arbitrary partial recursive functions are numeralwise representable in $\lambda^{\tau\tau}$. This implies that term equality in $\lambda^{\tau\tau}$ is undecidable, and therefore further implies that no complete set of effective reduction rules exists which are both confluent and normalizing [2]. The presence of a fixed-point operator also easily implies that the calculus is not a conservative extension of algebraic equational theories.

2. A λ-calculus with dependent types

We now formalize λ^Π, the λ-calculus with dependent types sketched in Section 1.

The syntax of terms. Let $V = \{x_1, x_2, \ldots\}$ be a countably infinite set of variables. In what follows a, b, e, f, and g are metavariables for terms, u, v, and w are

metavariables for variables, and s and t are metavariables for terms appearing as types. The set of *raw* untyped terms Λ^R is the smallest set defined by the following inductive clauses:

$$V \subseteq \Lambda^R \qquad \text{(every variable is a term)},$$
$$\tau \in \Lambda^R \qquad \text{('}\tau\text{' is a term)},$$
$$(a\,b) \in \Lambda^R \qquad \text{(application)},$$
$$(\lambda v{:}s.e) \in \Lambda^R \qquad (\lambda\text{-abstraction}),$$
$$(\Pi v{:}s.t) \in \Lambda^R \qquad (\Pi\text{-abstraction}).$$

Let $\mathrm{fv}[\![a]\!]$ denote the set of free variables of a, defined inductively in the usual way on the structure of terms. In raw abstractions it is possible to have occurrences of v free in s; for definiteness we adopt the convention that such occurrences are in the scope of 'λv' or 'Πv', but in fact our typing rules will forbid such occurrences. The function-space expression $s \to t$ stands for $(\Pi v{:}s.t)$ where v does not occur free in s or t. We follow the usual convention that when $\to$ appears as a binary connective, it associates to the right, so that $s \to s' \to s''$ abbreviates $s \to (s' \to s'')$. Application associates to the left so that $f\,a\,b$ abbreviates $(f\,a)\,b$.

We adopt the following *variable convention*: if a set of terms occurs together, for example, in a definition, then all bound variables in these terms are distinct from each other and from the free variables [2]. We also identify terms modulo the uniform renaming of bound variables (α-conversion); in combination with the variable convention this allows us to work with representatives of the α-equivalence classes of terms rather than terms themselves.* Henceforth '$\equiv$' denotes syntactic term equality modulo α-conversion. The substitution operator $[a/v]$ denoting the replacement of all free occurrences of v with a is defined recursively in the usual way.

Typed λ-calculi as proof systems. We formulate our various typed λ-calculi, λ^α for α one of τ, Δ, Π, or $\tau\tau$, as proof systems for *statements* about terms [3]. A statement is a pair consisting of a *context*, Γ, and a *sentence*, φ, and is written $\Gamma \vdash \varphi$. There are two kinds of sentences, namely *equations* of the form $a = b$ and *typings* of the form $a{:}t$.

Since sentences may have free variables, contexts Γ are needed to specify the types of the free variables. Raw contexts are defined to be partial functions from V to raw terms. The empty context is written Γ_0, and $\Gamma[v{:}t]$ denotes the context Γ modified or extended so that $\Gamma(v) \equiv t$. (Contexts do *not* record their history; $\Gamma[v{:}s][v{:}t]$ equals $\Gamma[v{:}t]$.)

In the most general situation, *well-formed* contexts, equations, and typings must be mutually recursively defined. This is because well-formed equations are constrained to be between typable terms (of the same type), so the inference rules for equations typically have antecedents which are typings. The inference rules for typings must in turn allow for equations between types, so these rules may have equational antecedents. Finally, the range of a context is intended to contain only τ or terms t which are type expressions, *i.e.*, terms such that the typing $t{:}\tau$ is provable (in an appropriate context), so the definition of well-formed context depends on provability of typings.

We write $\Gamma \vdash^\alpha \varphi$ when the statement $\Gamma \vdash \varphi$ is λ^α-provable. If (A) is a set of statements, then $\lambda^\alpha(A)$ denotes the extended system obtained by adding (A) to the axioms of λ^α. In particular, $\lambda^{\tau\tau}$ will be defined to be $\lambda^\Pi(\Gamma_0 \vdash \tau{:}\tau)$.

The λ^Π-calculus. The axioms and inference rules of the λ^Π-calculus are presented in a form similar to Gentzen's calculus of sequents [14, 27, 21]. Each rule consists of a set of statements (which have already been defined using the usual notation, $\vdash$, for sequents) called the *antecedents* and a statement called the *consequent*, graphically separated by a horizontal line. In the case of an axiom or axiom scheme, there are no antecedents and no horizontal line is drawn. The consequent of an inference rule is *provable* if each antecedent is itself provable.

All λ^Π-proofs begin with an instance of the type variable introduction (τ-vi) axiom scheme since it is the only axiom in the system. The pair of statements $\Gamma \vdash e{:}t$ and $\Gamma \vdash t{:}\tau$ and is abbreviated $\Gamma \vdash e{:}t{:}\tau$.

Rules for typings.

(τ-vi) τ-variable introduction

$$\Gamma_0[v{:}\tau] \vdash v{:}\tau$$

(vi) variable introduction

$$\frac{\Gamma \vdash t{:}\tau,\ v \notin \mathrm{dom}(\Gamma)}{\Gamma[v{:}t] \vdash v{:}t}$$

(Πi) Π-introduction

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma[v{:}s] \vdash t{:}\tau}{\Gamma \vdash (\Pi v{:}s.t){:}\tau}$$

(λi) λ-introduction

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma[v{:}s] \vdash e{:}t{:}\tau}{\Gamma \vdash (\lambda v{:}s.e){:}(\Pi v{:}s.t)}$$

*In the full paper we simplify the handling of bound variables by using a variant of de Bruijn's *nameless terms* [9].

(Πe) Π-elimination

$$\frac{\Gamma \vdash a{:}s,\ \Gamma \vdash f{:}(\Pi v{:}s.t){:}\tau}{\Gamma \vdash (f\ a){:}t[a/v]}$$

(τc) type conversion

$$\frac{\Gamma \vdash a{:}s,\ \Gamma \vdash s = t}{\Gamma \vdash a{:}t}$$

Rules for equations.

(r) reflexivity

$$\frac{\Gamma \vdash a{:}t}{\Gamma \vdash a = a}$$

(s) symmetry

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash b = a}$$

(t) transitivity

$$\frac{\Gamma \vdash a = b,\ \Gamma \vdash b = c}{\Gamma \vdash a = c}$$

(cl) left congruence

$$\frac{\Gamma \vdash a{:}s,\ \Gamma \vdash f{:}(\Pi v{:}s.t){:}\tau,\ \Gamma \vdash f = f'}{\Gamma \vdash (f\ a) = (f'\ a)}$$

(cr) right congruence

$$\frac{\Gamma \vdash a{:}s,\ \Gamma \vdash f{:}(\Pi v{:}s.t){:}\tau,\ \Gamma \vdash a = a'}{\Gamma \vdash (f\ a) = (f\ a')}$$

($\lambda\beta$) β-conversion of λ-terms

$$\frac{\Gamma \vdash a{:}s,\ \Gamma \vdash (\lambda v{:}s.e){:}(\Pi v{:}s.t){:}\tau}{\Gamma \vdash ((\lambda v{:}s.e)a) = e[a/v]}$$

($\lambda\xi$) weak extensionality for λ-terms

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma[v{:}s] \vdash e{:}t{:}\tau,\ \Gamma[v{:}s] \vdash e = e'}{\Gamma \vdash (\lambda v{:}s.e) = (\lambda v{:}s.e')}$$

($\Pi\xi$) weak extensionality for Π-terms

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma[v{:}s] \vdash t{:}\tau,\ \Gamma[v{:}s] \vdash t = t'}{\Gamma \vdash (\Pi v{:}s.t) = (\Pi v{:}s.t')}$$

($\lambda\eta$) η-conversion of λ-terms

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma \vdash f{:}(\Pi w{:}s.t){:}\tau,\ v \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda v{:}s.(f\ v)) = f}$$

(λ-τc) binding type conversion in λ-terms

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma[v{:}s] \vdash e{:}t{:}\tau,\ \Gamma \vdash s = s'}{\Gamma \vdash (\lambda v{:}s.e) = (\lambda v{:}s'.e)}$$

(Π-τc) binding type conversion in Π-terms

$$\frac{\Gamma \vdash s{:}\tau,\ \Gamma[v{:}s] \vdash t{:}\tau,\ \Gamma \vdash s = s'}{\Gamma \vdash (\Pi v{:}s.t) = (\Pi v{:}s'.t)}$$

Rules for context extension.

(cx) context extension

$$\frac{\Gamma \vdash \varphi,\ \Gamma \vdash t{:}\tau,\ v \notin \mathrm{dom}(\Gamma)}{\Gamma[v{:}t] \vdash \varphi}$$

(τ-cx) τ-context extension

$$\frac{\Gamma \vdash \varphi,\ v \notin \mathrm{dom}(\Gamma)}{\Gamma[v{:}\tau] \vdash \varphi}$$

The rules may seem cumbersome at first, but most of them are formulations, using dependent types, of familiar rules of the λ^τ-calculus.

We do not have time here to explain in detail the several design choices embodied in the rules. However, the context extension rules, (cx) and (τ-cx), are worth noting. These rules assert that what is provable in a context may also be concluded when that context is extended by assigning a type to a fresh variable. Such rules are generally not necessary in systems which provide a *context projection* rule

$$\frac{\Gamma(v) = t}{\Gamma \vdash v{:}t}\ .$$

However, including context projection seems to force complications in other rules in order to preserve the intended behavior of the system. Thus, in systems with context projection (*e.g.*, various formulations of AUTOMATH [3, 31]), the notion of context well-formedness is explicitly introduced into the proof system as a third kind of sentence. Inference rules which extend contexts must assert that the extended context is well-formed, and rules which project contexts must check that their antecedent contexts are well-formed.

A key aspect of well-formed contexts is that circular type assignments in which the type of a variable depends on the value of the variable do not occur. For example, a circular assignment such as

$$\Gamma(x) \equiv (\mathbf{if}\ (x = 0)\ \mathbf{then\ int\ else\ bool})$$

can lead to undesirable λ-abstractions such as

$$\lambda x{:}(\mathbf{if}\ (x = 0)\ \mathbf{then\ int\ else\ bool}).x$$

which are not typable in the system. It is helpful to realize in reading the axioms and rules of λ^Π that they are formulated so that if $\Gamma \vdash v{:}s$, $\Gamma \vdash (\lambda v{:}s.a){:}t$, or $\Gamma \vdash (\Pi v{:}s.t){:}\tau$, then Γ will not be circular and $v \notin \mathrm{fv}[\![s]\!]$.

Equivalence of λ^τ and pure λ^Π. The pure λ^Π system is designed to provide the basic facilities for a rich dependent type system through its introduction and elimination rules for manipulating these types. However, these

facilities have no real opportunity to come into play in the pure system because the only axioms are those for type variable introduction. So until we add more axioms as in the next section, the system is very limited.

More precisely, we can prove by (a surprisingly intricate but ultimately routine) induction on the length of λ^{Π}-proofs that if $\Gamma \vdash (\Pi v{:}s.t){:}\tau$, then in fact v *does not occur free* in s *or* t; consequently this typing statement may be abbreviated as $\Gamma \vdash (s \to t){:}\tau$. It follows that the only variables v which can appear free in a type expression s, *i.e.*, an s such that $\Gamma \vdash s{:}\tau$, are free type variables, *viz.*, $\Gamma \vdash v{:}\tau$. Such free type variables behave essentially as the ground types of the finitely typed λ-calculus, λ^{τ}.

From this observation we can establish a translation $\mathcal{T}$ from terms and contexts of λ^{Π} to terms and type expressions of λ^{τ}, assuming that λ^{τ} allows an infinite number of ground types.

Lemma 1. $\Gamma \vdash^{\Pi} a{:}t{:}\tau$ iff $\vdash^{\tau} \mathcal{T}[\![a]\!]\Gamma{:}\mathcal{T}[\![t]\!]\Gamma$, and $\Gamma \vdash^{\Pi} a = b$ iff $\vdash^{\tau} \mathcal{T}[\![a]\!]\Gamma = \mathcal{T}[\![b]\!]\Gamma$.

Hence λ^{Π} may be regarded as a notational variant of λ^{τ}. From this it follows that λ^{Π} has a confluent system of reduction rules, is strongly normalizing, and consequently decidable.

Theorem 1. It is decidable whether a raw statement is λ^{Π}-provable.

3. Non-normalizability and undecidability

By assuming that 'type' is a type, we obtain from λ^{Π} a system essentially equivalent to Martin-Löf's 1971 theory of types.

The proof system for the $\lambda^{\tau\tau}$-calculus is just the λ^{Π} proof system augmented with the axiom

$$(\tau\tau) \quad \Gamma_0 \vdash^{\tau\tau} \tau{:}\tau.$$

(We remark that in the presence of $(\tau\tau)$, the (τ-vi) axiom and the (τ-cx) rule are become redundant.)

Analysis of the properties of $\lambda^{\tau\tau}$ hinges on replacing the equational rules of $\lambda^{\tau\tau}$ by directed rewrite rules (reductions). Namely, let $a \triangleright b$ be a new kind of sentence called a *reduction*. We modify the $\lambda^{\tau\tau}$ proof system to prove reduction statements. Rules for reductions are obtained from the rules of λ^{Π} by replacing all equations $a = b$ by reductions $a \triangleright b$ and deleting the reflexive and symmetric rules (r), (s). The key technical fact about reduction is that it is confluent: two terms are provably equal in some context iff they have a common reduct in that context.

As indicated in the introduction, we can establish a translation $\mathcal{U}$ from terms and contexts of λ^{Δ} to those of $\lambda^{\tau\tau}$ by replacing 'Λv' and 'Γv' by '$\lambda v{:}\tau$' and '$\Pi v{:}\tau$', respectively. Under this translation it is easy to see that λ^{τ} reductions can simulate λ^{Δ} reductions. Given that $\lambda^{\tau\tau}$-reduction is confluent, it is not hard to verify that the simulation is faithful:

Theorem 2. $\Gamma \vdash^{\Delta} a{:}t$ iff $\mathcal{U}\Gamma \vdash^{\tau\tau} \mathcal{U}[\![a]\!]{:}\mathcal{U}[\![t]\!]{:}\tau$, and $\Gamma \vdash^{\Delta} a = b$ iff $\mathcal{U}\Gamma \vdash^{\tau\tau} \mathcal{U}[\![a]\!] = \mathcal{U}[\![b]\!]$.

It follows that the data types and operations definable in the polymorphic λ-calculus are available in $\lambda^{\tau\tau}$. In particular, using

$$N \equiv (\Pi x{:}\tau.((x \to x) \to (x \to x)))$$

as the type of the polymorphic Church numerals, we conclude that the primitive recursive (and many more) functions on the integers are numeralwise representable by terms of $\lambda^{\tau\tau}$ [12].

So for any primitive recursive function f with one integer argument, choose a term a_f such that $\Gamma_0 \vdash^{\tau\tau} a_f{:}(N \to N)$ and a_f numeralwise represents f. It is well-known to be undecidable whether $f(n) = 0$ for some integer n. Let g to be the partial recursive function of one integer argument such that $g(m) = 0$ if $f(n) = 0$ for some $n \geq m$ and is undefined otherwise; hence it is undecidable whether $g(0) = 0$.

With a fixed-point operator $\mathbf{Y}_N{:}((N \to N) \to N)$, it is easy to construct from a_f a term a_g in $\lambda^{\tau\tau}$ which numeralwise represents g. (The confluence of reduction and the fact that numerals are normal forms make it easy to show that if $g(0) = 0$ then $\Gamma_0 \vdash^{\tau\tau} (a_g\,\mathbf{0}) = \mathbf{0}$ where $\mathbf{0}{:}N$ is the numeral for 0. Showing the converse requires more detailed information about the behavior of $\mathbf{Y}_N$ under reduction.) In fact, when $g(0)$ is undefined, the term $(a_g\,\mathbf{0})$ does not have a normal (or even head-normal) form. Thus, to prove the undecidability of $\lambda^{\tau\tau}$ it is sufficient to construct a fixed-point combinator $\mathbf{Y}_N$.

Analysis of Girard's proof enables us to construct a combinator $\mathbf{Y}$ satisfying the *polymorphic fixed-point* rule:

$$\frac{\Gamma \vdash f{:}(s \to s)}{\Gamma \vdash \mathbf{Y}\,s\,f = f(\mathbf{Y}\,s\,f)}.$$

Moreover, this is essentially the only rule needed to derive all the $\lambda^{\tau\tau}$-provable statements about $\mathbf{Y}$. Now $\mathbf{Y}_N$

is simply $(\mathbf{Y}\,N)$. Time limitations prevent us from including the construction of $\mathbf{Y}$ and a precise analysis of its behavior under reduction.

This completes the outline of the proof of

Theorem 3. The equational theory of $\lambda^{\tau\tau}$ is undecidable. In fact, it is undecidable whether

$$\Gamma_0 \vdash^{\tau\tau} a = 0$$

where $0 \equiv \lambda x{:}\tau.\lambda y{:}x.y$ and a is a raw term.

Corollary 1. There is no decidable, confluent, and normalizing set of reduction rules for the equational theory of $\lambda^{\tau\tau}$.

4. Conservative extension

We illustrate conservative extension metatheorems with a simple example suggested by Gordon Plotkin [26].

Consider an algebraic structure ι containing distinct elements 0 and 1, a quaternary operator *cond* (conditional equality), and a unary operator *suc* (successor) satisfying the pair of equational axioms (A):

$$cond\,x\,x\,y\,z = y$$
$$cond\,x\,(suc\,x)\,y\,z = z.$$

It is easy to find such structures, and since $1 = 0$ is not true in them, it cannot be derived from (A) by sound inference rules.

Any algebraic structure can be chosen as the base type of a λ^{τ} model using the classical interpretation of function types $s \to t$ as all functions from s to t. Since the rules of λ^{τ} are sound in any such model, it follows that all equations between algebraic terms over the signature of ι (*i.e.*, ordinary first-order terms with free variables of type ι) provable in $\lambda^{\tau}(A)$ are valid in all models of (A). Substituting equals for equals is a derived rule of λ^{τ}, and the rule of substituting equals for equals is well-known to be sufficient for proving all the logical consequences of any set of equational axioms. Hence, the equations between algebraic terms over the signature of ι provable in $\lambda^{\tau}(A)$ coincide with those provable by substituting equals for equals—which are, in turn, exactly the equations valid in all models of (A). In sum, the provable equations between algebraic terms do not change when we switch to λ^{τ} rules, *e.g.*, $1 = 0$ is not provable in $\lambda^{\tau}(A)$. This is what is meant by the assertion that $\lambda^{\tau}(A)$ yields a *conservative extension* of the (first-order) equational theory of (A). More generally,

Theorem 4. $\lambda^{\tau}(E)$ yields a conservative extension of the equational theory of (E) for any set (E) of algebraic equations.

Now let (A^{Π}) be the set of statements

$$\Gamma_0 \vdash \iota{:}\tau,$$
$$\Gamma_0 \vdash cond{:}(\iota \to \iota \to \iota \to \iota \to \iota),$$
$$\Gamma_0 \vdash suc{:}\iota \to \iota,$$
$$\Gamma_0 \vdash cond\,u\,u\,v\,w = v,$$
$$\Gamma_0 \vdash cond\,u\,(suc\,u)\,v\,w = w.$$

The correspondence between λ^{τ} and λ^{Π} in Lemma 1 is easily seen to continue to hold in the presence of algebraic axioms, so $\lambda^{\Pi}(A^{\Pi})$, like $\lambda^{\tau}(A)$, also yields a conservative extension of the equational theory of (A).

(We remark that we expect to prove in the full paper that a similar conservative extension theorem holds for $\lambda^{\Delta}(E^{\Delta})$. Since it is not known whether an arbitrary algebraic structure can be fully and faithfully embedded in some model of λ^{Δ}, the proof of conservative extension cannot proceed along the model-theoretic lines of the argument used above for λ^{τ}. Instead we use a proof-theoretic argument based on the confluence and strong normalizability of λ^{Δ}.)

On the other hand, in $\lambda^{\tau\tau}(A^{\Pi})$, the polymorphic combinator $\mathbf{Y}$ supplies a closed term a of type ι such that $a = (suc\,a)$ is provable, namely $a \equiv \mathbf{Y}\,\iota\,suc$. But now, from (A) we have

$$cond\,a\,a\,v\,w = v$$

and

$$cond\,a\,(suc\,a)\,v\,w = w,$$

and since $a = (suc\,a)$, we conclude $v = w$. Hence *all* equations between terms of the same type are provable in $\lambda^{\tau\tau}(A^{\Pi})$. In particular, $1 = 0$ is not true in ι but is provable in $\lambda^{\tau\tau}(A^{\Pi})$ even though ι satisfies the set of axioms (A^{Π})!

Theorem 5. $\lambda^{\tau\tau}(A^{\Pi})$ does not yield a conservative extension of the equational theory of the set (A^{Π}) of algebraic equational statements above.

This failure of conservative extension in $\lambda^{\tau\tau}$ is actually a familiar one. In a theory of computation on the integers, say, one expects to have divergent computations of integer type. In the usual approach of denotational semantics, such divergent computations are said to compute a special divergent integer value, $\perp$. Now $\perp$ serves

as a fixed point of the successor function. Although the theory of integers with $\perp$ may not seem any harder than the standard theory, the technical complications in proofs about integer computations caused by $\perp$ has prompted recent efforts to find better theories [25, 26].

We expect that the introduction of fixed points is not the only source of failure of conservative extension. Hence we raise the following open problem:

(1) Let (E) be a set of algebraic (classical first-order) typing and equational statements, and let (fix) be

$$\Gamma_0 \vdash \mathbf{Y} = \lambda v{:}(\iota \to \iota).(\mathbf{Y}\, v).$$

Is $\lambda^{\Pi}((\text{fix}), E)$ conservatively extended by $\lambda^{\tau\tau}((\text{fix}), E)$?

We also mention that we do not know whether the $\tau{:}\tau$ axiom, *in the absence of any other axioms*, violates conservative extension, namely,

(2) Is $\lambda^{\tau\tau}(\varnothing)$ a conservative extension of $\lambda^{\Pi}(\varnothing)$?

In general, conservative extension theorems offer the opportunity to carry classical mathematics into a computational setting without change. The need to reason about divergent values and more generally about the ordering and topological properties typically superimposed on discrete structures in computational settings would be avoided. It remains to be seen how much computation theory we can fully develop while preserving classical reasoning.

References

[1] Reference manual for the Ada programming language. G.P.O. 008-000-00354-8, 1980.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, Amsterdam, second edition, 1984.

[3] H. P. Barendregt and A. Rezus. Semantics for classical AUTOMATH and related systems. *Information and Control*, 59:127–147, 1983.

[4] H. Boehm, A. Demers, and J. Donahue. *An Informal Description of Russell.* Technical Report TR80-430, Cornell University, Computer Science Department, 1980.

[5] K. B. Bruce and A. R. Meyer. The semantics of second order polymorphic lambda calculus. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 131–144, Springer-Verlag, Berlin, June 1984.

[6] R. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 1–50, Springer-Verlag, Berlin, June 1984.

[7] T. Coquand and G. Huet. *Constructions: A Higher Order Proof System for Mechanizing Mathematics.* Rapport de Recherche 401, INRIA, Domaine de Voluceau, 78150 Rocquencourt, France, May 1985. Presented at EUROCAL 85, Linz, Austria.

[8] H. B. Curry and R. Feys. *Combinatory Logic.* Volume 1, North-Holland, Amsterdam, 1958.

[9] N. G. de Bruijn. Lambda-calculus notation with nameless dummies. *Indag. Math.*, 34(5):381–392, 1972.

[10] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606, Academic Press, New York, 1980.

[11] J. Donahue and A. Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[12] S. Fortune, D. Leivant, and M. O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, January 1983.

[13] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, 1969.

[14] G. Gentzen. Investigations into logical deduction. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1934. See [13] for an English translation.

[15] J. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* Ph.D. thesis, Université Paris VII, 1972.

[16] W. A. Howard. The formulae-as-types notion of construction. 1969. Recently published as [17].

[17] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490, Academic Press, New York, 1980.

[18] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981.

[19] P. Martin-Löf. An intuitionistic theory of types: predicative part. In F. Rose and J. Sheperdson, editors, *Logic Colloquium III*, pages 73–118, North-Holland, Amsterdam, July 1973.

[20] P. Martin-Löf. *A Theory of Types.* Report 71-3, Department of Mathematics, University of Stockholm, February 1971.

[21] N. J. McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* Ph.D. thesis, Syracuse University, Syracuse, New York, June 1979.

[22] A. R. Meyer and J. C. Mitchell. Second-order logical relations (extended abstract). In R. Parikh, editor, *Logics of Programs*, pages 225–236, Springer-Verlag, Berlin, June 1985.

[23] J. C. Mitchell. *Lambda Calculus Models of Typed Programming Languages.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1984.

[24] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Principles of Programming Languages*, pages 37–51, ACM, New York, January 1985.

[25] L. Paulson. Deriving structural induction in LCF. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 197–214, Springer-Verlag, Berlin, June 1984.

[26] G. D. Plotkin. Personal communication, March 1985.

[27] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Volume 3 of *Stockholm Studies in Philosophy*, Almqvist and Wiksell, Stockholm, 1965.

[28] J. C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, pages 145–156, Springer-Verlag, Berlin, June 1984.

[29] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Springer-Verlag, Berlin, 1974.

[30] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, North-Holland, 1983.

[31] A. Rezus. *Abstract AUTOMATH.* Mathematical Centre Tract 160, Mathematisch Centrum, Amsterdam, 1983.