LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-458

TYPECHECKING IS UNDECIDABLE WHEN 'TYPE' IS A TYPE

Mark B. Reinhold

December 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Typechecking Is Undecidable When 'Type' Is a Type

Mark B. Reinhold

Technical Report 458
December 1989

Abstract. A function has a *dependent type* when the type of its result depends upon the value of its argument. The *type of all types* is the type of every type, including itself. In a typed λ-calculus, these two features synergize in a conceptually clean and uniform way to yield enormous expressive power at very little apparent cost. By reconstructing and analyzing a paradox due to Girard, we argue that there is no effective typechecking algorithm for such a language.

Categories and subject descriptors: F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*type structure*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*lambda calculus and related systems.*

Additional key words and phrases: Polymorphism, propositions-as-types analogy, intuitionistic logic, Girard's paradox, type theory.

General terms: Languages, theory.

Copyright © 1989, Massachusetts Institute of Technology. All rights reserved.

This report is a revised version of a Master's thesis of the same title, which was supervised by Professor Albert R. Meyer and submitted to the Department of Electrical Engineering and Computer Science in August 1987. This research was supported by the National Science Foundation under grant numbers DCR-8010707 and DCR-8511190.

1. Introduction

1.1. Dependent function 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 with a dependent type is the unary function *zero-vector* that, when applied to an integer n, returns an n-vector of zeroes. If all vectors are given the same type **vector**, then *zero-vector* can be given the simple functional type

$$\mathbf{int} \rightarrow \mathbf{vector}\ .$$

On the other hand, we may choose to make vector types more informative by having the type of a vector reflect its length. In this case, **vector** is not a type constant but a *parameterized type*: $(\mathbf{vector}\, n)$ denotes the type of vectors of length n. We can thus describe the type of the value $(\mathit{zero\text{-}vector}\ n)$, for a given n, as $(\mathbf{vector}\, n)$; in symbols,

$$(\mathit{zero\text{-}vector}\ n) : (\mathbf{vector}\, n)\ ,$$

where ':' is read as 'has type.' But now *zero-vector* no longer has the simple type given above, for the type of $(\mathit{zero\text{-}vector}\ n)$ depends on n; *i.e.*, the type of $(\mathit{zero\text{-}vector}\ n)$ is a function of n. This suggests that we use the type-returning function

$$\lambda n{:}\mathbf{int}.(\mathbf{vector}\, n)\ ,$$

often called a *type generator* or a *type constructor*, as the type of *zero-vector*, for it accurately describes the type of the value of *zero-vector* at the argument n. However, to maintain the distinction between functions and types, we use Π in place of λ as an abstraction symbol for type expressions. We therefore have

$$\mathit{zero\text{-}vector} : \Pi n{:}\mathbf{int}.(\mathbf{vector}\, n)\ .$$

The dependent function type $\Pi x{:}A.B$, also called a *general product type*, is the type of functions that map an element a of type A to an element of type $B[a/x]$, where $B[a/x]$ stands for B with a substituted for all free occurrences of x. Ordinary function types are a special case of dependent types, for the type $A \rightarrow B$ is simply $\Pi x{:}A.B$, where x is chosen to be an identifier not free in B.

Terms (*i.e.*, programs), which denote functions and values, and type expressions, which classify terms, are syntactically and semantically distinct in most explicitly typed programming languages. This distinction permits *static typechecking*: A term can be typechecked (*e.g.*, by a compiler) in advance of performing the computation that it describes. In type systems that support dependent function types this distinction is blurred, for terms can appear in type expressions and it may be necessary to perform some computation in order to typecheck a term. For example, consider the function

$$\mathit{neg\text{-}vector} : \Pi n{:}\mathbf{int}.(\mathbf{vector}\, n) \rightarrow (\mathbf{vector}\, n)$$

which, given an integer n, returns a function that negates each element of a given n-vector. When we apply *neg-vector* to a particular integer, say 23, we obtain a function

$$(\mathit{neg\text{-}vector}\ 23) : (\mathbf{vector}\, 23) \rightarrow (\mathbf{vector}\, 23)\ .$$

Now suppose that M is a term of type **int** and that x is a variable of type $(\mathbf{vector}\, M)$. In order to typecheck the application

$$((\mathit{neg\text{-}vector}\ 23)\ x)\ ,$$

we must verify that x is of type $(\mathbf{vector}\, 23)$, and this requires checking that M computes to 23. While typechecking may require computation, it is still possible to distinguish between the process of typechecking a term, which may require performing the computations described by some other terms, and the process of performing the computation described by the term itself.

1.2. The type of all types

The *type of all types* is the type of every type, including itself. In symbols, writing '$\star$' for the type of all types, we have $A : \star$ for every type A and, since $\star$ is its own type, $\star : \star$.

By admitting the type of all types, we admit types as true "first-class" citizens, allowing types to be treated just like ordinary values. In particular, functions on types can be defined by ordinary λ-abstraction; *e.g.*, writing '$\times$' for pair (*i.e.*, product) types, we can define

$$\mathit{three\text{-}tuple} \equiv \lambda A{:}\star . A \times A \times A$$

and then write (*three-tuple* **int**) for $\mathbf{int} \times \mathbf{int} \times \mathbf{int}$, the type of three-tuples of integers. Just as functions involving integers have types containing **int**, so functions involving types have types containing $\star$. Thus the *three-tuple* type generator has type $\star \rightarrow \star$; similarly, the parameterized type **vector** used above has type $\mathbf{int} \rightarrow \star$.

In the presence of dependent function types, the type of all types allows the expression of *parametric polymorphism* [Strachey 1967, Reynolds 1983], which can be found in the programming languages CLU [Liskov *et al.* 1981], Ada [Ada 1980], Russell [Boehm *et al.* 1980, Donahue & Demers 1985], and ML [Milner 1983]. Intuitively, we can think of a polymorphic function as a function mapping types to values; a polymorphic function has a dependent type because the type of its result depends upon the value of its argument, which in this case is a type. Polymorphic functions are easily expressed in a λ-calculus with dependent function types and the type of all types because types can be treated as values: Type variables are just variables of type $\star$, abstraction with respect to a type variable is done with ordinary λ-abstraction, and the application of a polymorphic function to a type is ordinary function application. For example, let A be a type variable in the function

$$\lambda x{:}A.x\ .$$

By λ-abstracting over this function with respect to A, we obtain the polymorphic identity function

$$\begin{aligned} id &\equiv \lambda A{:}\star.\lambda x{:}A.x \\ &: \Pi A{:}\star.A \to A\ . \end{aligned}$$

When *id* is applied to a type, it yields the identity function on that type; *e.g.*, for the type **int** $\to$ **int**,

$$id\,(\mathbf{int} \to \mathbf{int}) \;=\; \lambda x{:}\mathbf{int} \to \mathbf{int}.x\ ,$$

which in this case is a function of type $(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})$.

Thus a λ-calculus with dependent function types and the type of all types subsumes the polymorphic λ-calculus of [Girard 1972] and [Reynolds 1974], which is itself a very rich and powerful language. Any recursive numeric function that is provably total in second-order Peano arithmetic can be represented in this calculus [Girard 1972, Statman 1981, Fortune *et al.* 1983]; there are no known uncontrived examples of total recursive functions that are not in this class. Many interesting generic data types can be defined, including pairs, unions, and homogeneous lists and trees [Reynolds 1985, Böhm & Berarducci 1985].

Going beyond the polymorphic λ-calculus, in a λ-calculus with dependent function types and the type of all types we can apply the programming techniques of the more powerful Calculus of Constructions [Coquand 1985b, Mohring 1986, Coquand & Huet 1988]. For example, we can define a form of *dependent pair* type, also called a *general sum* or an *existential* type, that is useful for modelling abstract data types [Mitchell & Plotkin 1985, MacQueen 1986].

The combination of dependent function types and the type of all types yields enormous expressive power at very little apparent cost, and it does so in a conceptually clean and uniform way by treating types as values. Unfortunately, we shall see that the typechecking problem for a language with these features is undecidable. That is, there is no effective algorithm for computing the type of a program, even though the type of every identifier is explicitly declared.

1.3. The $\lambda^\star$-calculus

This thesis is a study of dependent function types and the type of all types in the setting of the $\lambda^\star$-calculus, which is intended to be a minimal typed λ-calculus incorporating these two features. The $\lambda^\star$-calculus, defined in §2, can be seen as a strongly- and explicitly-typed functional programming language in which functions are first-class values. Functions (λ-terms) are classified by dependent function types (Π-terms), which in turn have type $\star$. As we shall see, many of the familiar properties of typed λ-calculi hold for the $\lambda^\star$-calculus:

(1) Reduction is Church-Rosser, *i.e.*, confluent.

(2) Reduction is *type preserving*: If $M : A$ and $M \to N$, then $N : A$, where '$\to$' denotes the one-step reduction relation on $\lambda^\star$-terms. In other words, evaluation does not change the type of a term.

(3) The *static typing* property holds: The processes of typechecking a term and of performing the computation that it describes are distinct, *i.e.*, types need not be checked during evaluation.

However, a number of expected properties *fail* to hold for the $\lambda^\star$-calculus. In §3 and §4 we shall establish that:

(4) Strong normalization fails: There exist terms that have no normal form (*i.e.*, their evaluation does not terminate).

(5) The normal-form relation is undecidable: For two terms M and N such that N is a normal form, it is undecidable whether $M \twoheadrightarrow N$, where '$\twoheadrightarrow$' denotes the multi-step reduction relation on $\lambda^\star$-terms.

(6) The equational theory is undecidable: For two terms M and N, it is undecidable whether $M =_\beta N$, where '$=_\beta$' denotes provable equality (*i.e.*, program equivalence) under β-conversion.

(7) The typing theory is undecidable: For a term M and a type A, it is undecidable whether $M : A$.

We normally associate these failed properties with languages in which divergent recursive computations can be expressed, so it is surprising that (4)–(7) apply to the $\lambda^\star$-calculus, which has no primitives for recursion and in which recursion is not obviously definable. The positive forms of (4)–(7) are enjoyed by a variety of typed λ-calculi, including the simply-typed λ-calculus [Barendregt 1984, Appendix A], the polymorphic λ-calculus [Girard 1972, Reynolds 1974], and the Calculus of Constructions [Coquand 1985b, Coquand & Huet 1988].

That strong normalization fails (4) for languages like the $\lambda^\star$-calculus has been known for some time; the central result of this thesis is that the normal-form relation is undecidable (5). The undecidability of the equational theory (6) follows from (5) because, by the Church-Rosser property (1), the equational theory is characterized by the reduction rules. The undecidability of the typing theory (7) follows from (5) because computation is required, in general, in order to typecheck a term.

1.4. Overview

Girard's paradox. The $\lambda^\star$-calculus is not a recent invention: Martin-Löf's first Intuitionistic Theory of Types [Martin-Löf 1971] is a higher-order constructive logic that is essentially equivalent to the $\lambda^\star$-calculus. Martin-Löf wanted a single formalism that would be adequate for the formalization of a substantial portion of constructive mathematics, and he believed that the $\lambda^\star$-calculus, with its type of all types, had sufficient expressive power to meet this requirement. Martin-Löf's type theory is based upon a connection between intuitionistic logics and typed λ-calculi known as the *propositions-as-types* analogy. Under this analogy, logical propositions are written as types, and a proof of a proposition is a λ-term of the appropriate type.

[Girard 1972] showed Martin-Löf's type theory to be inconsistent by reformulating the Burali-Forti paradox [Burali-Forti 1897] of classical set theory; this reformulation has come to be known as *Girard's paradox.* The upshot of the paradox is a proof of the absurd proposition, which asserts that every proposition is provable. In the $\lambda^\star$-calculus, the absurd proposition is written as the type $\Pi A{:}\star.A$, and Girard's argument describes how to construct a proof of this proposition, *i.e.*, a closed $\lambda^\star$-term of this type. It is straightforward to show that no term of this type has a normal form, and this implies the failure of strong normalization.

Looping combinators. The failure of strong normalization does not, however, imply the undecidability of normal forms, equations, or typings in the $\lambda^\star$-calculus. In order to examine the actual non-normalizing term described by Girard's proof, in §3 we follow the paradox to construct a term Z of type $\Pi A{:}\star.A$, using an improved formulation adapted from [Coquand 1986] and [Howe 1987]. Being too complex to be carried out by hand, the construction is done with the assistance of a computer program called LP, which is a typechecking and reduction engine for general typed λ-calculi. Given the typechecking rules for a particular calculus, LP provides a Lisp-like interface that accepts an identifier definition, typechecks the definition, and enters it into the global environment for use in later definitions. LP and its input language, which is essentially a sugared version of the $\lambda^\star$-calculus, are described in Appendix A.

Observation of the reduction behavior of Z shows that certain of its subterms, which correspond to formal proofs of lemmas used in the paradox proof, continuously reappear at the front of Z as it is reduced. We show how to build a polymorphic *looping combinator* $\mathbf{L}_0$ by making minor modifications to the paradox construction; this term, written in the LP input language, is contained in Appendix B. A polymorphic looping combinator $\mathbf{L}_i$ is a term of type $\Pi A{:}\star.(A \to A) \to A$ such that, for any type A and function $f : A \to A$, we have

$$(\mathbf{L}_i\, A\, f) \twoheadrightarrow f\, (\mathbf{L}_{i+1}\, A\, f)\ ,$$

where $\mathbf{L}_{i+1}$ is another looping combinator. Intuitively, a looping combinator is "just as good as" a fixed-point combinator for the purpose of expressing recursive computations. Recall that the polymorphic fixed-point combinator $\mathbf{Y}$ has the reduction behavior

$$(\mathbf{Y}\, A\, f) \to f\, (\mathbf{Y}\, A\, f)\ .$$

The only difference between a fixed-point combinator $\mathbf{Y}$ and a looping combinator $\mathbf{L}$ is that an application of $\mathbf{Y}$ reduces to a term involving $\mathbf{Y}$ itself, while an application of $\mathbf{L}$ reduces to a term involving another, possibly different, looping combinator.

Undecidability. One way of showing that the normal-form relation of the $\lambda^\star$-calculus is undecidable is to show that all partial recursive functions can be defined within it. As is well known, any partial recursive function can be expressed in terms of the initial functions and the function-forming operations of composition, primitive recursion, and minimalization. It would therefore suffice to show that we can compute, in the $\lambda^\star$-calculus, all functions constructed in this manner. The primitive recursion and minimalization operators are typically implemented using a fixed-point combinator, but true fixed points are not required; all that is necessary is the ability to iterate a function an arbitrary number of times. Looping combinators have exactly this ability, so the undecidability result would follow immediately by reduction from the halting problem.

Unfortunately, as we discuss in §4, we don't know quite enough about the behavior of the constructed looping combinator $\mathbf{L}_\Theta$ to be able to apply this method. However, we can show that all *total* recursive functions can be computed in the λ^*-calculus. A simple complexity-theoretic argument then proves that the normal-form relation of the λ^*-calculus is undecidable, from which it follows that the equational and typing theories are undecidable as well.

2. The $\lambda^\star$-calculus

Terms can appear in type expressions in the $\lambda^\star$-calculus, so we cannot first define type expressions and then define the set of terms, as is possible for simpler typed λ-calculi. The syntactic machinery of the $\lambda^\star$-calculus is therefore much like that of AUTOMATH [Barendregt & Rezus 1983] and the Calculus of Constructions [Coquand 1985b, Coquand & Huet 1988]. We first define the set $\Lambda^\star$ of *raw terms* and a notion of reduction upon them. We then restrict the raw terms to the *well-typed terms* through a proof system for *typing statements*, which are assertions about the type of a term relative to the types of its free variables. Thus the raw terms include both the well-typed terms and many other terms that are not well-typed.

2.1. Raw terms

Fix a countably infinite set of variables. In what follows, x, y, and z are metavariables for variables and M and N are metavariables for terms. Other capital Roman letters are occasionally used for terms; in particular, A, B, and C stand for terms intended to be types. The set $\Lambda^\star$ of raw terms is the smallest set defined by the following inductive clauses:

$x \in \Lambda^\star$	every variable is a term
$\star \in \Lambda^\star$	'$\star$' is a term
$(M\ N) \in \Lambda^\star$	application
$(\lambda x{:}A.M) \in \Lambda^\star$	λ-abstraction
$(\Pi x{:}A.B) \in \Lambda^\star$	Π-abstraction

In a raw abstraction it is possible to have occurrences of the bound variable x free in the binding type A; for definiteness we say that such occurrences are *not* bound by the binding symbol of the abstraction, but in fact the typing rules will forbid such occurrences.

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 [Barendregt 1984, Appendix C]. 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. When used between terms, '$=$' denotes syntactic equality modulo α-conversion.

The set of free variables of M is denoted $\mathrm{fv}(M)$, and the substitution of N for all free occurrences of x in M is denoted by $M[N/x]$; both are defined inductively on the structure of terms in the usual way.

The function type expression $A \to B$ abbreviates $(\Pi x{:}A.B)$, where x does not occur free in B. We follow the familiar convention that $\to$ associates to the right, so that $A \to B \to C$ abbreviates $A \to (B \to C)$. Application associates to the left so that $(F\ M\ N)$ abbreviates $((F\ M)\ N)$. The vector notation $(M\ \vec{N})$ abbreviates $(M\ N_1\ N_2 \cdots N_k)$; similarly, $\lambda\vec{x}{:}\vec{A}.M$ abbreviates $\lambda x_1{:}A_1.\lambda x_2{:}A_2.\cdots\lambda x_k{:}A_k.M$.

2.2. Reduction

The one-step reduction relation $\to$ is inductively defined as the least relation satisfying the axiom of β-contraction,

$$(\lambda x{:}A.M)\,N \to M[N/x]\ ,$$

and the inference rules:

$$\begin{array}{rcl} M \to M' & \Longrightarrow & M\,N \to M'\,N \\ N \to N' & \Longrightarrow & M\,N \to M\,N' \\ M \to M' & \Longrightarrow & (\lambda x{:}A.M) \to (\lambda x{:}A.M'),\ (\Pi x{:}A.M) \to (\Pi x{:}A.M') \\ A \to A' & \Longrightarrow & (\lambda x{:}A.M) \to (\lambda x{:}A'.M),\ (\Pi x{:}A.M) \to (\Pi x{:}A'.M)\ . \end{array}$$

Note that there is no notion of β-contraction for Π-abstractions; this is intentional, as Π-abstractions are dissected only by the typing rules. A term M is a *normal form* (nf) iff there is no term N such that $M \to N$. The multi-step reduction relation $\twoheadrightarrow$ is the transitive, reflexive closure of $\to$, and the conversion relation $\twoheadleftarrow\!\twoheadrightarrow$ is the equivalence relation generated by $\twoheadrightarrow$. If N is a nf and $M \twoheadleftarrow\!\twoheadrightarrow N$, we say that N is a normal form *of* M. As expected, the Church-Rosser property holds for reduction:

Lemma 2.1. If $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$, then there exists an N such that $M_1 \twoheadrightarrow N$ and $M_2 \twoheadrightarrow N$.

Proof. See [Martin-Löf 1971]. ■

We have the usual corollaries, proved by the same methods as for the untyped λ-calculus [Barendregt 1984, §3.1].

Corollary 2.2. $M_1 \twoheadleftarrow\!\twoheadrightarrow M_2$ iff there exists an N such that $M_1 \twoheadrightarrow N$ and $M_2 \twoheadrightarrow N$.

Corollary 2.3. Normal forms of given terms are unique, if they exist.

The notion of *head normal form* (hnf) for raw terms, useful in the analysis of non-normalizing terms, is adapted from the untyped λ-calculus [Barendregt 1984, Def. 2.2.11]. A term of $\Lambda^\star$ is a hnf iff it is of one of the following forms:

$$\lambda\vec{x}{:}\vec{A}.(\star\ \vec{M})$$
$$\lambda\vec{x}{:}\vec{A}.(y\ \vec{M})$$
$$\lambda\vec{x}{:}\vec{A}.((\Pi z{:}A.B)\ \vec{M})$$

Note that $\vec{x}$, $\vec{A}$, and $\vec{M}$ may be empty, and y may be free or bound. A term M is said to *have a hnf* iff there exists a hnf N such that $M \twoheadleftarrow\!\!\twoheadrightarrow N$.

Proposition 2.4. If $M = (M_1\ M_2)$ is a nf, then M is of the form $(Z\ \vec{N})$, where Z is either $\star$, a variable, or a Π-abstraction.

Proof by induction on the structure of terms, analyzing the structure of M_1. If M_1 is $\star$, a variable, or a Π-abstraction, then $Z = M_1$ and $N = M_2$. M_1 cannot be a λ-abstraction since M is a nf. If M_1 is an application, then by induction it is of the form $(Z\ \vec{N})$ for some appropriate Z, so $M = ((Z\ \vec{N})\ M_2)$. ∎

Proposition 2.5. If M is a nf, then M is a hnf.

Proof by induction on the structure of terms. The cases for $\star$, variables, and Π-abstractions are obvious. If M is a λ-abstraction, then by induction its body must be a hnf, making M itself a hnf. If $M = (M_1\ M_2)$ then, by Proposition 2.4, M is a hnf. ∎

2.3. Well-typed terms

A typing statement consists of a *typing* and a *context*. The typing associates a term with a type, and is a pair of terms written $M : A$. The context* records the types of the free variables in the typing, and is a sequence of typings of variables (*e.g.*, $\langle x{:}A, y{:}B, z{:}C\rangle$). The empty context is written $\langle\rangle$, and the context Δ with the typing $x{:}A$ appended is written $\Delta, x{:}A$. A complete typing statement is written $\Delta \vdash M : A$ and can be read, 'under the variable declarations in Δ, the term M has type A.'

The proof system for typing statements contains one axiom and seven inference rules. Each inference rule consists of a set of *antecedent* statements and a *consequent* statement, graphically separated by a horizontal line. A typing statement is *provable* iff it is the axiom, or iff it is the consequent of an inference rule

*Sometimes called a *type assignment* or *type environment*.

whose antecedents are all provable. A term M is said to *have type* A iff there is a context Δ such that $\Delta \vdash M : A$ is provable, and M is said to be *well-typed* iff there is a type A such that M has type A. The proof system is carefully designed so that every provable typing statement $\Delta \vdash M : A$ is *well-formed*, meaning that every free variable in Δ, M, and A is declared in Δ, no variable is declared more than once, and the type of every declared variable is provably of type $\star$.

The typing rules can be divided into those that manipulate contexts and those that construct types. The context manipulation rules make use of *empty statements*, written $\Delta \vdash$, to indicate that Δ alone is well-formed. The context manipulation rules are:

(ci) $$\langle\rangle \vdash$$ context initialization

($\star$:$\star$) $$\frac{\Delta \vdash}{\Delta \vdash \star : \star}$$ type of all types

(cx) $$\frac{\Delta \vdash A : \star}{\Delta, x{:}A \vdash} \quad x \notin \Delta$$ context extension

(cp) $$\frac{\Delta \vdash}{\Delta \vdash x : A} \quad x{:}A \in \Delta$$ context projection

Note that $x{:}A \in \Delta$ means that the typing $x{:}A$ occurs in Δ, and $x \notin \Delta$ means that there is no typing $y{:}A \in \Delta$ such that $x = y$. The (ci) rule simply says that the empty context is well-formed. The ($\star$:$\star$) rule allows the introduction of the type of all types. The (cx) rule is used to declare a new variable x of type A after A is proved to be a type, and the (cp) rule allows a previously declared variable to be projected from the context into a typing.

The rules for type construction are:

(Πf) $$\frac{\Delta, x{:}A \vdash B : \star}{\Delta \vdash (\Pi x{:}A.B) : \star}$$ Π-formation

(Πi) $$\frac{\Delta, x{:}A \vdash M : B}{\Delta \vdash (\lambda x{:}A.M) : (\Pi x{:}A.B)}$$ Π-introduction

(Πe) $$\frac{\Delta \vdash M : (\Pi x{:}A.B), \quad \Delta \vdash N : A}{\Delta \vdash (M\ N) : B[N/x]}$$ Π-elimination

(tc) $$\frac{\Delta \vdash M : A, \quad \Delta \vdash B : \star}{\Delta \vdash M : B} \quad A \twoheadleftarrow\!\!\twoheadrightarrow B$$ type conversion

The first three rules construct the type of a term from the types of its subterms, and correspond exactly to the term formation operations of Π-abstraction, λ-abstraction, and application. The first two rules also modify the context, discharging the newly bound variable x. The final rule (tc) allows the type of a term to be replaced with a type to which it converts, establishing the connection between computation and the process of constructing the type of a term.

Notice that each group of rules depends upon the other. The antecedent of the (cx) rule is a typing statement that can be proved by any of the type construction rules, as well as by the (cp) rule. The antecedents of the type construction rules are typing statements that can be proved by the (cp) rule, as well as by the other type construction rules. In general, the proof of a typing statement requires the mixed use of both kinds of rules.

2.4. Properties of provable typing statements

The notation $\Delta \vdash \varphi$ stands for an arbitrary statement; φ is either a typing, in the case of a typing statement, or φ is empty, in the case of an empty statement. If $\varphi = M : A$, then $\mathrm{fv}(\varphi) \equiv \mathrm{fv}(M) \cup \mathrm{fv}(A)$; otherwise, $\mathrm{fv}(\varphi) \equiv \varnothing$. Unless otherwise noted, the proofs in this subsection proceed by induction on the definition of statement provability.

Well-formedness. Although contexts are sequences of typings rather than sets of typings or functions from variables to terms, the proof system is designed so that a context in a provable statement will never contain more than one typing for any variable.

Lemma 2.6. If $\Delta \vdash \varphi$ is provable then, for every $x{:}A \in \Delta$, there is no other $x{:}B \in \Delta$ for any B.

Thus we can conveniently think of contexts as partial functions from variables to their declared types; *i.e.*, $\Delta(x) = A$ iff $x{:}A \in \Delta$.

An important property of the proof system is that in a provable statement $\Delta \vdash \varphi$, the type of any free variable in φ or of any declared variable in Δ is guaranteed to contain only previously declared variables that are themselves declared in Δ. Certain nonsensical situations can arise if this property does not hold; *e.g.*, a variable can have a type containing itself or an undeclared variable. Because contexts are sequences, they record the history of the declarations that they contain. Relative to a variable x, the previously declared variables in a context Δ—*i.e.*, those comprising the context in which x was declared—are just those variables that precede x in Δ. Let Δ^x denote the proper prefix of Δ preceding $x{:}A$ if $x{:}A \in \Delta$. Then we have

Lemma 2.7. If $\Delta \vdash \varphi$ is provable, then $\mathrm{fv}(\varphi) \subseteq \mathrm{dom}\,\Delta$ and, for each $x \in \mathrm{dom}\,\Delta$, we have that $\mathrm{fv}(\Delta(x)) \subseteq \mathrm{dom}\,\Delta^x$.

These two lemmas are equivalent to the "variable restrictions" of [Martin-Löf 1971].

Finally, we verify that each term associated with a declared variable x in a provable statement is actually a type, *i.e.*, provably of type $\star$ in the context in which x was declared.

Lemma 2.8. If $\Delta \vdash \varphi$ is provable, then $\Delta^x \vdash \Delta(x) : \star$ is provable for each $x \in \mathrm{dom}\,\Delta$.

The type of all types. The type of all types, $\star$, is the type of every type, including itself. In the $\lambda^\star$-calculus, a type is any term appearing on the right-hand side of a colon in a provable typing statement. Therefore any such term should be provably of type $\star$ in the same context.

Lemma 2.9. If $\Delta \vdash M : A$ is provable, then $\Delta \vdash A : \star$ is provable.

Reduction and typing. The following lemmas, or stronger versions of them, are proved in [Martin-Löf 1971]; essentially the same proofs work for the $\lambda^\star$-calculus.

Lemma 2.10 (*Replacement*). If $\Delta, x{:}A \vdash M : B$ and $\Delta \vdash N : A$ are provable, then $\Delta \vdash M[N/x] : B[N/x]$ is provable.

Lemma 2.11 (*Type Preservation*). If $\Delta \vdash M : A$ is provable and $M \twoheadrightarrow N$, then $\Delta \vdash N : A$ is provable.

Type preservation is sometimes called *subject reduction*, but we prefer to reserve that name for its original meaning as a similar property of *untyped* λ-terms [Curry & Feys 1958, Hindley & Seldin 1986].

Lemma 2.12 (*Unique Typing*). If $\Delta \vdash M : A$ and $\Delta \vdash M : B$ are provable, then $A \twoheadleftarrow\!\!\twoheadrightarrow B$.

Static typing. Let any type convertible to the form $\Pi\vec{x}{:}\vec{A}.\star$ be called a $\star$-*type*. A term with a $\star$-type A is either a type, if $A = \star$, or a function returning a type, *i.e.*, a type generator. If $\Delta \vdash M : A$ and A is not a $\star$-type, then the *erasure* of M, written $|M|$, is defined inductively as follows:

$$|x| = x$$

$$|(M\ N)| = \begin{cases} |M| & \text{if } \Delta \vdash N : A \text{ and } A \text{ is a } \star\text{-type,} \\ (|M|\ |N|) & \text{otherwise.} \end{cases}$$

$$|(\lambda x{:}A.M)| = \begin{cases} |M| & \text{if } A \text{ is a } \star\text{-type,} \\ (\lambda x.|M|) & \text{otherwise.} \end{cases}$$

The erasure map removes all type information from a given term, including both types and type generators, yielding a term of the untyped λ-calculus. (This definition is adapted from the definition of *stripping* given in [Coquand & Huet 1988].)

Lemma 2.13 (*Static Typing*). If $\Delta \vdash M : A$ is provable and A is not a $\star$-type, then $M \twoheadrightarrow N$ iff $|M| \twoheadrightarrow |N|$.

In other words, the reduction path of a well-typed term coincides with the reduction path of its erasure. Therefore it is not necessary to consider any of the type information in a term when performing the computation that it describes.

Non-normalizing terms. It seems to be well known that in the $\lambda^\star$-calculus, and in other λ-calculi with type abstraction, no term of type $\Pi A{:}\star.A$ has a nf; for completeness we prove this fact here. It suffices to show that no term of type $\Pi A{:}\star.A$ *is* a nf, from which it follows by Lemma 2.11 that no such term *has* a nf.

Lemma 2.14. If $\langle\rangle \vdash M : (\Pi A{:}\star.A)$ is provable, then M is not a nf.

Proof. We show that M is not a hnf; by contraposition from Proposition 2.5, this implies that M is not a nf.

Suppose that M is a hnf; we shall analyze its structure and see that it cannot possibly have the required type. M cannot be $\star$ because $\star$ is not of type $\Pi A{:}\star.A$ (the type of $\star$ is $\star$, and $\star \not\twoheadleftarrow\!\!\twoheadrightarrow \Pi A{:}\star.A$, since both are nfs). By Lemma 2.7, M must be closed, therefore it cannot be a variable. M cannot be a Π-abstraction because any well-typed Π-abstraction has type $\star$.

If M is an application then, being a hnf, it must be of the form $(Z\ \vec{N})$, where Z is either $\star$, a Π-abstraction, or a variable. The first two cases are impossible because such terms can never have Π-types, and therefore could not be used as operators in the (Πe) rule. Since M is closed, Z cannot be a variable.

If M is a λ-abstraction, say $\lambda A{:}\star.N$, then by (Πi) it must be that $\langle A{:}\star\rangle \vdash N : A$ is provable. Again, N cannot be $\star$ since $\star$ is not of type A. If N is a variable then, by Lemma 2.7, it must be A, but A is not of type A. N cannot be a λ-

or Π-abstraction because neither could be of type A. If N is an application then the analysis of the previous paragraph applies, except that Z cannot be a variable because the only declared variable is A, which does not have a Π-type. ■

2.5. Some constructions

We now define some of the constructs mentioned in §1, all of which are given in [Martin-Löf 1971]. Named $\lambda^\star$-combinators will be defined in the following format:

$$(f\ a\ b) \equiv M\ .$$

This is shorthand for $f \equiv \lambda a{:}A.\lambda b{:}B.M$ for appropriate types A and B that can be inferred from the context.

Pairs. An element of a pair, or product, type $A \times B$ contains an element of type A and an element of type B. A pair type has an injection operation to create pairs of that type from elements of the component types, and two projection operations, one for each component.

$$A \times B \equiv \Pi X{:}\star.(A \to B \to X) \to X$$

$$\begin{aligned}(\mathit{pair}\ A\ B) &\equiv \lambda a{:}A.\lambda b{:}B.\lambda X{:}\star.\lambda f{:}(A \to B \to X).f\ a\ b\\ &:\ A \to B \to (A \times B)\end{aligned}$$

$$\begin{aligned}(\mathit{left}\ A\ B) &\equiv \lambda z{:}(A \times B).z\ A\ (\lambda a{:}A.\lambda b{:}B.a)\\ &:\ (A \times B) \to A\end{aligned}$$

$$\begin{aligned}(\mathit{right}\ A\ B) &\equiv \lambda z{:}(A \times B).z\ B\ (\lambda a{:}A.\lambda b{:}B.b)\\ &:\ (A \times B) \to B\end{aligned}$$

This is essentially a typed version of the pairing combinator of the untyped λ-calculus. For $a : A$ and $b : B$, the following pairing axioms are easily verified:

$$(\mathit{left}\ A\ B\ (\mathit{pair}\ A\ B\ a\ b)) = a$$
$$(\mathit{right}\ A\ B\ (\mathit{pair}\ A\ B\ a\ b)) = b$$

This definition easily generalizes to tuples of arbitrary length [Reynolds 1985].

Unions. An element of a union, or sum, type $A + B$ contains either an element of A or an element of B, together with an indication of which kind of element it contains. A union type has one injection operation for each of the component types, and a projection operation that allows the element within a union to be accessed in a type-safe way.

$$A + B \equiv \Pi X{:}\star.(A \to X) \to (B \to X) \to X$$

$$\begin{aligned}(\mathit{inleft}\ A\ B) &\equiv \lambda a{:}A.\lambda X{:}\star.\lambda f{:}A \to X.\lambda g{:}B \to X.f\ a\\ &:\ A \to (A + B)\end{aligned}$$

$$\begin{aligned}(\mathit{inright}\ A\ B) &\equiv \lambda b{:}B.\lambda X{:}\star.\lambda f{:}A \to X.\lambda g{:}B \to X.g\ a\\ &:\ B \to (A + B)\end{aligned}$$

$$\begin{aligned}(\mathit{case}\ A\ B) &\equiv \lambda X{:}\star.\lambda f{:}A \to X.\lambda g{:}B \to X.\lambda z{:}(A + B).z\ X\ f\ g\\ &:\ \Pi X{:}\star.(A \to X) \to (B \to X) \to (A + B) \to X\end{aligned}$$

For $a : A$, $b : B$, a type X, and functions $f : A \to X$ and $g : B \to X$, these definitions satisfy the following union axioms:

$$(\mathit{case}\ A\ B\ X\ f\ g\ (\mathit{inleft}\ A\ B\ a)) = f\ a$$
$$(\mathit{case}\ A\ B\ X\ f\ g\ (\mathit{inright}\ A\ B\ b)) = g\ b$$

As with pairs, it is straightforward to generalize this definition to the union of an arbitrary number of types [Reynolds 1985].

Dependent pairs. A dependent pair type, also called a general sum or an existential type, is to an ordinary pair type as a dependent function type is to a simple function type. A dependent pair type is written $\Sigma x{:}A.B$, with the Σ binding x in B. An element of $\Sigma x{:}A.B$ can be thought of as a pair consisting of an element $a : A$ and an element of type $B[a/x]$; the type of the second element depends upon the value of the first. When treating dependent pairs as a defined notion, Σ cannot be used as a binding operator; instead, it is a function that maps a type A and a function B of type $A \to \star$ into a type:

$$(\Sigma\ A\ B) \equiv \Pi X{:}\star.(\Pi a{:}A.(B\ a) \to X) \to X\ .$$

The injection operation for dependent pairs is defined by the combinator

$$\begin{aligned}(\mathit{dpair}\ A\ B) &\equiv \lambda a{:}A.\lambda b{:}(B\ a).\lambda X{:}\star.\lambda f{:}(\Pi a{:}A.(B\ a) \to X).f\ a\ b\\ &:\ \Pi a{:}A.(B\ a) \to (\Sigma\ A\ B)\ ,\end{aligned}$$

which is much like the injection for pairing except that the type of b has been made dependent upon a.

The left element of a dependent pair, called the *witness*, can be extracted via

$$\begin{aligned}(\mathit{wit}\ A\ B) &\equiv \lambda s{:}(\Sigma\ A\ B).s\ A\ (\lambda a{:}A.\lambda b{:}(B\ a).a)\\ &: (\Sigma\ A\ B) \to A\ ,\end{aligned}$$

and it is straightforward to verify that

$$(\mathit{wit}\ A\ B\ (\mathit{dpair}\ A\ B\ a\ b)) = a$$

for $a : A$ and $b : (B\ a)$.

The dependent pairs definable in the $\lambda^\star$-calculus have no right-hand projection operation. One might try defining

$$\begin{aligned}(\mathit{dright}\ A\ B) &\equiv \lambda s{:}(\Sigma\ A\ B).s\ (B\ (\mathit{wit}\ A\ B\ s))\ (\lambda a{:}A.\lambda b{:}(B\ a).b)\\ &: \Pi s{:}(\Sigma\ A\ B).B\ (\mathit{wit}\ A\ B\ s)\ ,\end{aligned}$$

but *dright* does not have the indicated type; in fact, *dright* has no type in the $\lambda^\star$-calculus. Attempting to typecheck the body of *dright*, under the declarations $A : \star$, $B : A \to \star$, and $s : (\Sigma\ A\ B)$, we have

$$\begin{aligned}s\ (\mathit{wit}\ A\ B\ s) &: \big(\Pi a{:}A.(B\ a) \to (\mathit{wit}\ A\ B\ s)\big) \to (\mathit{wit}\ A\ B\ s)\ ,\\ (\lambda a{:}A.\lambda b{:}(B\ a).b) &: \big(\Pi a{:}A.(B\ a) \to (B\ a)\big)\ ;\end{aligned}$$

the body of *dright* is the application of the first term above to the second, and this application is not well-typed.

Although there is no right-hand projection for dependent pairs, the right-hand element can be indirectly accessed via the *out* operation,

$$\begin{aligned}(\mathit{out}\ A\ B) &\equiv \lambda s{:}(\Sigma\ A\ B).\lambda X{:}\star.\lambda f{:}(\Pi a{:}A.(B\ a) \to X).s\ X\ f\\ &: (\Sigma\ A\ B) \to \Pi X{:}\star.(\Pi a{:}A.(B\ a) \to X) \to X\ ,\end{aligned}$$

which obeys the axiom

$$(\mathit{out}\ A\ B\ (\mathit{dpair}\ A\ B\ a\ b)\ X\ f) = (f\ a\ b)$$

for $a : A$, $b : (B\ a)$, a type X, and a function $f : \Pi a{:}A.(B\ a) \to X$.

We shall make extensive use of dependent pairs in the construction of Girard's paradox. A generalization of dependent pairs to dependent tuples, *i.e.*, elements of type $\Sigma\vec{x}{:}\vec{A}.B$, is described in §A.

The above definitions of Σ, *dpair*, *wit*, and *out* can be used without change in the Calculus of Constructions [Coquand & Huet 1988], but the typing constraints of that system do not allow the witness of a dependent pair to be a type. A weaker form of dependent pair, which is definable in both calculi, allows the witness to be a type but lacks a witness operator.

2.6. Related systems

The $\lambda^\star$-calculus is a rather special-purpose system, having been adapted from [Martin-Löf 1971] specifically for the investigation of dependent function types and the type of all types. In [Meyer & Reinhold 1986] we discussed a more flexible system, the λ^Π-calculus, that was designed for the general study of dependent function types, both with and without the type of all types. For example, we considered an extension of the λ^Π-calculus that would subsume the polymorphic λ-calculus while preserving the properties lost when the type of all types is added. The design of the λ^Π-calculus has been an ongoing project, evolving according to the needs of our research, and we do not wish to address here the complex issues involved in designing a truly general λ-calculus with dependent types. Another advantage of the simplicity of the $\lambda^\star$-calculus is that our results can be carried over to other systems by showing that they are conservative extensions of the $\lambda^\star$-calculus. This is a straightforward task in many cases; it would not be as easy if the $\lambda^\star$-calculus were more general.

The $\lambda^\star$-calculus and the λ^Π-calculus differ in two ways. First, the context manipulation rules of the λ^Π-calculus are much like the thinning rules of Gentzen's sequent calculus [Kleene 1950]; there is no rule for context projection. We originally thought that this method was superior to the style of the $\lambda^\star$-calculus, as it does not require special empty statements to assert the well-formedness of a context. However, further experience with the proof theory of the λ^Π-calculus and similar systems [Barendregt & Rezus 1983, Coquand 1985b] convinced us otherwise. The second difference is that the λ^Π-calculus uses typed equations between λ-terms rather than the untyped reduction used in the $\lambda^\star$-calculus. This choice complicates the definition of the proof system for typing statements, since it must also contain inference rules for typed equations; the inference rules for contexts, typings, and equations are all interdependent and the system as a whole is more difficult to reason about. There are other situations in which typed equations are more appropriate (*e.g.*, in the study of conservative extensions of algebraic theories by typed λ-calculi [Breazu-Tannen & Meyer 1987]), but for our purposes the simpler calculus suffices.

3. Looping with Girard's paradox

3.1. The $\lambda^\star$-calculus as a logic

[Martin-Löf 1971] defines an intuitionistic type theory, *i.e.*, a higher-order constructive logic, that is essentially equivalent to the $\lambda^\star$-calculus. Central to this theory is the *propositions-as-types analogy*, which establishes a connection between intuitionistic logics and typed λ-calculi. Also known as the 'formulae-as-types analogy' and the 'Curry-Howard isomorphism,' the analogy was first discovered for the case of positive implicational propositional logic by [Curry & Feys 1958] and extended to first- and second-order intuitionistic logic by [Howard 1969] and [Girard 1972]. Martin-Löf's type theory is based upon a particularly close reading of the analogy, which we briefly review here. More general and complete presentations of the analogy can be found in [Stenlund 1972] and in [Huet 1986].

An intuitionistic type theory is a formal system for doing constructive reasoning about constructible objects. By 'constructible objects' we mean *effectively* constructible objects, including effectively computable functions. As we have seen, a rich class of computable functions can be defined in the $\lambda^\star$-calculus, and other constructible objects, such as pairs and unions, can be built up from functions. These functions and objects, classified by their types, make up the universe of discourse of Martin-Löf's type theory.

In constructive reasoning, each logical connective is interpreted as a specification of a class of constructible objects. A proposition is proved by exhibiting an object in the class that it describes, so proofs themselves are constructible objects. It turns out that the relationship between a proof and the proposition that it proves is very much like the relationship between an object and its type, as shown in the table below.

Proposition	*Proof*	*Type*
$P \wedge Q$	A proof of P, and a proof of Q.	$P \times Q$
$P \vee Q$	A proof of P, or a proof of Q, and an indication of which is proved.	$P + Q$
$P \Rightarrow Q$	An effective method for transforming a proof of P into a proof of Q.	$P \rightarrow Q$
$\forall x{:}A.P$	An effective method that, given any a of type A, produces a proof of $P[a/x]$.	$\Pi x{:}A.P$
$\exists x{:}A.P$	A particular a of type A, and a proof of $P[a/x]$.	$\Sigma x{:}A.P$

This correspondence justifies the key idea of Martin-Löf's type theory: Propositions and types are identified, as are proofs and objects. Paraphrasing Martin-Löf, a proposition is the type of its proofs; conversely, a type is the proposition that is proved by exhibiting an object of that type. When using the $\lambda^{\star}$-calculus as a type theory, we construct objects, which have types, and we prove propositions about them by constructing other objects, whose types are the propositions (according to the above correspondence) being proved.

The type of all types plays a central role in Martin-Löf's type theory. Since types and propositions are identified, $\star$ is not only the type of all types, but also the type of all propositions.* A predicate is a parameterized proposition, so a predicate on a type A is just a function on A returning a proposition, *i.e.*, a function of type $A \rightarrow \star$. Relations are obtained by Currying; *e.g.*, $A \rightarrow A \rightarrow \star$ is the type of a binary relation on the type A. Higher-order predicates, *i.e.*, predicates on predicates, can also be defined; *e.g.*, a predicate on predicates on a type A is a function of type $(A \rightarrow \star) \rightarrow \star$. Higher-order quantification is expressible because the type of a quantifier-bound variable can be any term of type $\star$, including $\star$ as well as the type of any predicate. Thus the absurd proposition $\forall P.P$ translates directly into $\bot \equiv \Pi P{:}\star.P$; the negation of a proposition P is written $\neg P \equiv P \rightarrow \bot$.

3.2. The paradox

The informal construction of Girard's paradox given here is intended to serve as a guide to the full formal construction, given in §B. We give the essential formal definitions in this section and prove the required lemmas informally; a lemma numbered $n.m$ in this section corresponds to lemma m in the full construction. To aid in the reading of the formal construction, we consistently use the outer parenthesization convention of λ-calculus, *e.g.*, $(f\ x)$, rather than the traditional mathematical notation, *e.g.*, $f(x)$.

The essence of the paradox is this: In a type theory with quantification over all types, we can construct the set of well-founded sets. We can then define an ordering on well-founded sets and show that, under this ordering, the set of well-founded sets is a well-founded set that precedes itself—a contradiction.

An ordered set $\mathcal{A}$ will be represented by a type A, a *domain predicate* $d_{\mathcal{A}} : A \rightarrow \star$, and a (strict) *ordering relation* $r_{\mathcal{A}} : A \rightarrow A \rightarrow \star$. An element $a : A$ is *in* the ordered set $\mathcal{A}$ iff $(d_{\mathcal{A}}\ a)$ holds. $\mathcal{A}$ will often be written in place of the three elements A, $d_{\mathcal{A}}$, and $r_{\mathcal{A}}$ that represent $\mathcal{A}$. We stress that this is a strictly visual device; in a term, $\mathcal{A}$ stands not for one term constructed from these three elements

*Moreover, since $\star$ is itself a type, $\star$ is also a proposition. As a proposition, $\star$ is provable because there is a term, namely $\star$, of type $\star$, so $\star$ is a propositional constant that proves itself (!).

but exactly for '$A\ d_{\mathcal{A}}\ r_{\mathcal{A}}$'. $\mathcal{B}$ is another ordered set represented by B, $d_{\mathcal{B}}$, and $r_{\mathcal{B}}$, and similarly for other calligraphic capital letters.

An ordered set $\mathcal{A}$ is *transitive* iff its ordering relation is transitive:

$$(\mathit{Trans}\ \mathcal{A}) \equiv \Pi x, y, z{:}A.(r_{\mathcal{A}}\ x\ y) \to (r_{\mathcal{A}}\ y\ z) \to (r_{\mathcal{A}}\ x\ z)\ .$$

A predicate $C : A \to \star$ on the elements of an ordered set $\mathcal{A}$ is a *chain* iff there is a *base element* $z : A$ such that z is in $\mathcal{A}$ and z satisfies C, and, for every $x : A$ satisfying C, there is a $y : A$ satisfying C that is smaller than x:

$$(\mathit{Chain}\ \mathcal{A}\ C) \equiv \big[\Sigma z{:}A.(d_{\mathcal{A}}\ z) \times (C\ z)\big] \times \big[\Pi x{:}A.(C\ x) \to \Sigma y{:}A.(C\ y) \times (r_{\mathcal{A}}\ y\ x)\big]\ .$$

An ordered set is *well-founded* iff no chains exist within it, *i.e.*, iff the existence of a chain implies absurdity:

$$(\mathit{WF}\ \mathcal{A}) \equiv \Pi C{:}A \to \star.(\mathit{Chain}\ \mathcal{A}\ C) \to \bot\ .$$

An ordered set $\mathcal{A}$ is *embedded into* an ordered set $\mathcal{B}$ by a function $f : A \to B$ and a *bound element* $b : B$ iff b is in the domain of $\mathcal{B}$, f is domain-preserving and monotonic, and the range of f is dominated by b:

$$\begin{aligned}(\mathit{Embed}\ \mathcal{A}\ \mathcal{B}\ f\ b) \equiv\ & (d_{\mathcal{B}}\ b)\\ & \times \big[\Pi x{:}A.(d_{\mathcal{A}}\ x) \to (d_{\mathcal{B}}\ (f\ x))\big]\\ & \times \big[\Pi x, y{:}A.(d_{\mathcal{A}}\ x) \to (d_{\mathcal{A}}\ y) \to (r_{\mathcal{A}}\ x\ y) \to (r_{\mathcal{B}}\ (f\ x)\ (f\ y))\big]\\ & \times \big[\Pi x{:}A.(d_{\mathcal{A}}\ x) \to (r_{\mathcal{B}}\ (f\ x)\ b)\big]\ .\end{aligned}$$

The ordering on ordered sets is defined in terms of embedding: $\mathcal{A}$ is less than $\mathcal{B}$ iff there exists a function f and an element b such that f and b embed $\mathcal{A}$ into $\mathcal{B}$.

$$(\mathit{Embed\text{-}ord}\ \mathcal{A}\ \mathcal{B}) \equiv \Sigma f{:}A \to B.\Sigma b{:}B.\mathit{Embed}\ \mathcal{A}\ \mathcal{B}\ f\ b\ .$$

We abbreviate $(\mathit{Embed}\ \mathcal{A}\ \mathcal{B}\ f\ b)$ by $\mathcal{A} <_{f,b} \mathcal{B}$, and $(\mathit{Embed\text{-}ord}\ \mathcal{A}\ \mathcal{B})$ by $\mathcal{A} < \mathcal{B}$.

Two elements x and y of a type A are *intensionally equal* iff every predicate true of x is true of y:

$$(\mathit{Eq}\ A\ x\ y) \equiv \Pi P{:}(A \to \star).(P\ x) \to (P\ y)\ .$$

It is easy to show that intensional equality is reflexive, symmetric, and transitive. We shall abbreviate $(\mathit{Eq}\ A\ x\ y)$ by $x = y$ when A is obvious.

To represent the ordered set of well-founded sets, we need a type U of ordered sets, a domain predicate $d_{\mathcal{U}} : U \to \star$ that is true only of the well-founded sets, and an ordering relation $r_{\mathcal{U}} : U \to U \to \star$ defined in terms of embeddings. Defining

U presents some difficulties, as we must somehow merge the three components A, $d_{\mathcal{A}}$, and $r_{\mathcal{A}}$ representing an ordered set $\mathcal{A}$ into a single element of type U. It is tempting to use dependent pair types, but these cannot be made to work due to the lack of a right-hand projection. The solution is to arrange that the injection of a particular ordered set $\mathcal{A}$ into U is a predicate true of precisely those predicates true of $\mathcal{A}$. Thus, defining *OS-pred* as the type of predicates on ordered sets,

$$\mathit{OS\text{-}pred} \equiv \Pi A{:}\star.(A \to \star) \to (A \to A \to \star) \to \star\ ,$$

we have $U \equiv \mathit{OS\text{-}pred} \to \star$, and the injection *inj* is simply

$$(\mathit{inj}\,\mathcal{A}) \equiv \lambda S{:}\mathit{OS\text{-}pred}.S\ \mathcal{A}\ .$$

If the injections of two ordered sets are intensionally equal, then any property true of the first is true of the second.

Lemma 3.1. If $(\mathit{inj}\ \mathcal{A}) = (\mathit{inj}\ \mathcal{B})$ and $(P\ \mathcal{A})$, then $(P\ \mathcal{B})$.

Proof. Apply the proof of $(\mathit{inj}\ \mathcal{A}) = (\mathit{inj}\ \mathcal{B})$ to $(\lambda u{:}U.u\ P)$ and the proof of $(P\ \mathcal{A})$. ■

The domain predicate on U is defined

$$\begin{aligned}(d_{\mathcal{U}}\ u) \equiv\ & \Sigma A{:}\star.\Sigma d_{\mathcal{A}}{:}A \to \star.\Sigma r_{\mathcal{A}}{:}A \to A \to \star.\\ & (\mathit{Eq}\ U\ u\ (\mathit{inj}\ \mathcal{A})) \times (\mathit{Trans}\ \mathcal{A}) \times (\mathit{WF}\ \mathcal{A})\ ,\end{aligned}$$

and the ordering relation on U is defined

$$\begin{aligned}(r_{\mathcal{U}}\ u\ v) \equiv\ & \Sigma A{:}\star.\Sigma d_{\mathcal{A}}{:}A \to \star.\Sigma r_{\mathcal{A}}{:}A \to A \to \star.\\ & \Sigma B{:}\star.\Sigma d_{\mathcal{B}}{:}B \to \star.\Sigma r_{\mathcal{B}}{:}B \to B \to \star.\\ & \quad (\mathit{Eq}\ U\ u\ (\mathit{inj}\ \mathcal{A})) \times (\mathit{Eq}\ U\ v\ (\mathit{inj}\ \mathcal{B}))\\ & \quad \times (\mathit{Embed\text{-}ord}\ \mathcal{A}\,\mathcal{B})\ .\end{aligned}$$

We shall write $\mathcal{U}$ for the ordered set represented by U, $d_{\mathcal{U}}$, and $r_{\mathcal{U}}$.

We now prove the lemmas necessary to show that $\mathcal{U}$ is a well-founded set. Most lemmas are stated informally, in which case a more formal version of the same statement follows in square brackets.

Lemma 3.2. Any ordered set in $\mathcal{U}$ is transitive and well-founded. [If $(d_{\mathcal{U}}(\mathit{inj}\ \mathcal{A}))$ then $(\mathit{Trans}\ \mathcal{A})$ and $(\mathit{WF}\ \mathcal{A})$.]

Proof. Follows from Lemma 3.1 and the symmetry of intensional equality. ■

Lemma 3.3. If $\mathcal{A} < \mathcal{B}$ and $\mathcal{B} <_{f_{\mathcal{BC}},b_{\mathcal{BC}}} \mathcal{C}$, then there exist $f_{\mathcal{AC}}$ and $b_{\mathcal{AC}}$ such that $\mathcal{A} <_{f_{\mathcal{AC}},b_{\mathcal{AC}}} \mathcal{C}$ and $(r_{\mathcal{C}}\ b_{\mathcal{AC}}\ b_{\mathcal{BC}})$.

Proof. Suppose that $f_{\mathcal{AB}}$ and $b_{\mathcal{AB}}$ embed $\mathcal{A}$ into $\mathcal{B}$. Let $f_{\mathcal{AC}} \equiv f_{\mathcal{BC}} \circ f_{\mathcal{AB}}$, and let $b_{\mathcal{AC}} \equiv (f_{\mathcal{BC}}\ b_{\mathcal{AB}})$. It is straightforward to check that the required properties hold. ■

Corollary 3.4. The relation $<$ is transitive.
[If $\mathcal{A} < \mathcal{B}$ and $\mathcal{B} < \mathcal{C}$, then $\mathcal{A} < \mathcal{C}$.]

The formal proofs of the remaining lemmas require uses of Lemma 3.1 and the properties of intensional equality; these will not be mentioned in the informal proofs given here.

Lemma 3.5. $\mathcal{U}$ is transitive. [$(\mathit{Trans}\,\mathcal{U})$.]

Proof. Follows immediately from Corollary 3.4. ■

Lemma 3.6. $\mathcal{U}$ is well-founded. [$(\mathit{WF}\,\mathcal{U})$.]

Proof by contradiction. Suppose that C is a chain in $\mathcal{U}$. The base of C is an ordered set, say $\mathcal{Z}$, that satisfies $d_{\mathcal{U}}$. By Lemma 3.2, $\mathcal{Z}$ is well-founded; we shall contradict this fact by using C to construct a chain D in $\mathcal{Z}$. Define D to be the predicate

$$\begin{aligned} D \equiv \lambda a{:}Z.\Sigma Y{:}\star.\Sigma dy{:}Y \to \star.\Sigma ry{:}Y \to Y \to \star.\Sigma f{:}Y \to Z. \\ (C\ (\mathit{inj}\ \mathcal{Y})) \times (\mathit{Embed}\ \mathcal{Y}\ \mathcal{Z}\ f\ a)\ . \end{aligned}$$

To prove that D is a chain in $\mathcal{Z}$, we must show that D has a base element $a_0 : Z$ such that $(d_{\mathcal{Z}}\ a_0)$, and that every element of $\mathcal{Z}$ in D has a predecessor in D.

Let $\mathcal{W}$ be a predecessor of $\mathcal{Z}$ in C. Taking a_0 to be the bound on the embedding from $\mathcal{W}$ to $\mathcal{Z}$, we immediately see that $(d_{\mathcal{Z}}\ a_0)$ and that a_0 is in D.

If $a : Z$ is in D, then there exists an ordered set $\mathcal{Y}$ and a function $f : Y \to Z$ such that $\mathcal{Y}$ is in C and $\mathcal{Y} <_{f,a} \mathcal{Z}$. Furthermore, there must exist an $\mathcal{X}$ in C such that $\mathcal{X} < \mathcal{Y}$ since $\mathcal{Y}$ is in C. By Lemma 3.3, we have an $f' : X \to Z$ and $a' : Z$ such that $\mathcal{X} <_{f',a'} \mathcal{Z}$. The predecessor of a in D is a'. ■

By Lemma 3.5 and Lemma 3.6, $\mathcal{U}$ is a transitive, well-founded set, hence $\mathcal{U}$ is in $\mathcal{U}$, i.e., $(d_{\mathcal{U}}\ (\mathit{inj}\,\mathcal{U}))$.

Next we show that, under the embedding ordering, any ordered set in $\mathcal{U}$ is less than $\mathcal{U}$. Given an ordered set $\mathcal{A}$ and an element $a : A$, the *initial segment* of $\mathcal{A}$ relative to a is the ordered set $\mathcal{A}_a$ represented by the type A, the ordering relation $r_{\mathcal{A}}$, and the new domain predicate

$$(d_{\mathcal{A}_a}\ x) \equiv (d_{\mathcal{A}}\ x) \times (r_{\mathcal{A}}\ x\ a)\ .$$

Lemma 3.7. Any initial segment of an ordered set in $\mathcal{U}$ is itself in $\mathcal{U}$.
[If $(d_{\mathcal{U}}(\mathit{inj}\ \mathcal{A}))$ and $(d_{\mathcal{A}}\ a)$ for some $a : A$, then $(d_{\mathcal{U}}\ (\mathit{inj}\ \mathcal{A}_a))$.]

Proof. Let $\mathcal{A}$ be in $\mathcal{U}$, and let $a : A$ be such that $(d_{\mathcal{A}}\ a)$. The initial segment $\mathcal{A}_a$ is transitive because $\mathcal{A}$ is transitive; similarly, any chain in $\mathcal{A}_a$ is clearly a chain in $\mathcal{A}$. Hence $\mathcal{A}_a$ is transitive and well-founded. ■

Lemma 3.8. $\mathcal{A} < \mathcal{U}$ for any ordered set $\mathcal{A}$ in $\mathcal{U}$.
[If $(d_{\mathcal{U}}\ (\mathit{inj}\ \mathcal{A}))$, then $\mathcal{A} < \mathcal{U}$.]

Proof. Let $\mathcal{A}$ be in $\mathcal{U}$, and let $f : A \rightarrow U$ be the function mapping an element $a : A$ to the ordered set $\mathcal{A}_a$. Then f embeds $\mathcal{A}$ into $\mathcal{U}$ with the embedding bound being $\mathcal{A}$ itself. $\mathcal{A}$ satisfies $d_{\mathcal{U}}$ by hypothesis, and f is domain-preserving by Lemma 3.7. Let x and y be elements of $\mathcal{A}$ such that $(r_{\mathcal{A}}\ x\ y)$; the identity function embeds $\mathcal{A}_x$ into $\mathcal{A}_y$ with bound x, so f is monotonic. Similarly, for any element x of $\mathcal{A}$, the identity function embeds $\mathcal{A}_x$ into $\mathcal{A}$ with bound x, hence the range of f is dominated by $\mathcal{A}$. ■

By Lemma 3.5, Lemma 3.6, and Lemma 3.8, we have that $\mathcal{U} < \mathcal{U}$, and thus $(r_{\mathcal{U}}\ (\mathit{inj}\ \mathcal{U})\ (\mathit{inj}\ \mathcal{U}))$. The predicate

$$\lambda u{:}U.(u = (\mathit{inj}\ \mathcal{U}))$$

describes a chain in $\mathcal{U}$: The base of the chain is $(\mathit{inj}\ \mathcal{U})$, and the predecessor of $(\mathit{inj}\ \mathcal{U})$ is just $(\mathit{inj}\ \mathcal{U})$. The existence of this chain contradicts Lemma 3.6, and leads to a proof of $\bot$ in the formal construction.

3.3. Building a looping combinator

Let Z denote the proof of $\bot$ obtained by constructing the formal proof of Girard's paradox; *i.e.*, $Z : (\Pi A{:}\star.A)$. By Lemma 2.14, Z has no normal form, and hence the $\lambda^\star$-calculus is not strongly normalizing. Reducing Z, we find that the following pattern emerges:

$$\begin{aligned} Z \equiv (\mathit{lemma6}\ C\ c) &\twoheadrightarrow (\mathit{lemma2b}\ \cdots) \\ &\twoheadrightarrow (\mathit{lemma6}\ \cdots) \\ &\twoheadrightarrow (\mathit{lemma2b}\ \cdots) \twoheadrightarrow \cdots\ , \end{aligned}$$

where *lemma6* is the formal proof of Lemma 3.6 and *lemma2b* is the formal proof of the second consequence of Lemma 3.2. To convert Z into a looping combinator, and thus harness this repetitive pattern, we make the following minor changes to the paradox construction. First, the absurd type $\bot$ is replaced by the type of polymorphic looping combinators, namely

$$\mathit{Loop} \equiv \Pi A{:}\star.(A \rightarrow A) \rightarrow A\ ,$$

in the definition of *WF* and in the formal proof of Lemma 3.7. With the redefinition of *WF*, Lemma 3.6 becomes

$$\Pi C{:}U \rightarrow \star.(\mathit{Chain}\,\mathcal{U}\;C) \rightarrow \mathit{Loop}\ ;$$

the formal proof *lemma6* of Lemma 3.6 must be changed so that, given a chain C in $\mathcal{U}$ and a proof that C is a chain, it returns a looping combinator. The redefinition of *WF* further implies that *lemma2b*, which is used in the body of Lemma 3.6, will also return a looping combinator. We thus modify *lemma6* to take two more arguments, a type A and a function $f : A \rightarrow A$, and we change the body of *lemma6* to supply A and f as additional arguments to the proof of Lemma 3.2 and to apply f to the result of that application. In symbols,

$$\begin{aligned} \mathit{lemma6} \equiv\ & \lambda C{:}U \rightarrow \star.\lambda c{:}(\mathit{Chain}\,\mathcal{U}). \\ & \cdots (\mathit{lemma2b}\ \cdots) \end{aligned}$$

is transformed into

$$\begin{aligned} \mathit{lemma6} \equiv\ & \lambda C{:}U \rightarrow \star.\lambda c{:}(\mathit{Chain}\,\mathcal{U}).\lambda A{:}\star.\lambda f{:}A \rightarrow A. \\ & \cdots f\,((\mathit{lemma2b}\ \cdots)\,A\,f)\ . \end{aligned}$$

Making these changes to Z yields an initial looping combinator $\mathbf{L}_0$, which is called *loop* in §B. Reduction of $\mathbf{L}_0$ reveals the pattern

$$\begin{aligned} (\mathbf{L}_0\,A\,f) & \twoheadrightarrow (\mathit{lemma6}\ C\,c\,A\,f) \\ & \twoheadrightarrow f\,((\mathit{lemma2b}\ \cdots)\,A\,f) \\ & \twoheadrightarrow f\,(\mathit{lemma6}\ \cdots\ A\,f) \\ & \twoheadrightarrow f\,(f\,((\mathit{lemma2b}\ \cdots)\,A\,f)) \twoheadrightarrow \cdots\ , \end{aligned}$$

which is exactly the behavior required of a looping combinator.

We have very convincing evidence that this cyclic pattern continues forever, *i.e.*, that $\mathbf{L}_0$ is a true looping combinator. After 970,000 reductions of $\mathbf{L}_0$ by the LP program, we obtain the combinator

$$\mathbf{L}_{69} = \lambda A{:}\star.\lambda f{:}A \rightarrow A.f^{69}(\mathbf{L}_{70}\,A\,f)\ ,$$

which iterates the given function f at least 69 times. Up to this point, the number of reductions involved is exponential in the number of iterations obtained. (We were forced to stop after 970,000 reductions due to virtual memory limitations.)

This empirical analysis of $\mathbf{L}_0$ is quite persuasive, but it does not constitute a proof that $\mathbf{L}_0$ is in fact a looping combinator. [Howe 1987] carried out a similar construction of Girard's paradox with the assistance of the NUPRL proof development system [Constable 1986]. Howe applied the looping combinator construction method described above [Reinhold 1986], and proved that the resulting term is a true looping combinator by reasoning about its reduction behavior. More recently, [Coquand 1987] has proposed an argument that *any* term of type $\perp$ must behave in a regenerative manner similar to that of Z, and that it will do so forever. Assuming that this argument can be made rigorous, the correctness of our construction method follows.

Theorem 3.9. $\mathbf{L}_0$ is a looping combinator.

4. Undecidability

We now prove that the existence of looping combinators in the $\lambda^{\star}$-calculus implies that the normal-form relation of the $\lambda^{\star}$-calculus is undecidable, from which it follows that its equational and typing theories are undecidable as well.

As our undecidability proof is somewhat novel, we wish to make it generally applicable. Therefore, we first prove the result for abstract *reduction systems*, of which all typed λ-calculi are instances. We show that if a reduction system is rich enough to compute all total recursive functions, then the normal-form relation of the system is undecidable. We then show that a particular reduction system, a typed λ-calculus with integers and looping combinators (the $\lambda^{\mathbf{L}}$-calculus), satisfies these requirements. Finally, we introduce a notion of *embedding* between reduction systems and prove that embeddings preserve the computability of classes of functions; by showing how to embed $\lambda^{\mathbf{L}}$ into $\lambda^{\star}$, we obtain the undecidability result for the $\lambda^{\star}$-calculus.

Henceforth, the term 'recursive function' will mean *total* recursive function.

4.1. Reduction systems

A *reduction system* is a three-tuple $(R, \rightarrow_R, \cdot_R)$ consisting of a set R of terms, a binary *reduction relation* $\rightarrow_R \subseteq R \times R$, and a binary *application operation* $\cdot_R \in R \times R \rightarrow R$. The reflexive, transitive closure of $\rightarrow_R$ is written $\twoheadrightarrow_R$. We shall write $\cdot_R$ as though it were left-associative, and *sans* subscript when it is possible to do so unambiguously. A reduction system will often be specified by mentioning only its set of terms.

A term M of a reduction system R is a *normal form* iff there is no $N \in R$ such that $M \rightarrow_R N$.

The *normal-form relation* $\text{NF}(R)$ of a reduction system R is defined as

$$\text{NF}(R) \equiv \{\langle M, N\rangle \in R \times R \mid M \twoheadrightarrow N \text{ and } N \text{ a nf}\} \ ;$$

i.e., the set of pairs of terms such that the first reduces to the second, and the second is a normal form.

A reduction system *with numerals* is a four-tuple $(R, \rightarrow_R, \cdot_R, num_R)$ such that $(R, \rightarrow_R, \cdot_R)$ is a reduction system and num_R is a recursive bijection mapping integers to normal forms of R; the range of num_R is the set of numerals of R. The fact that num_R is a bijection implies that the numerals are distinct, that every numeral is associated with a unique integer, and vice-versa. For an integer n, $num_R(n)$ will be written as $\llcorner n \lrcorner_R$, dropping the subscript when the context determines R.

4.2. Undecidability in reduction systems

We must first precisely define what it means for a recursive function to be computable in a reduction system. We say that a total numeric function $\varphi \in \mathbb{N}^k \to \mathbb{N}$ for some $k \geq 0$ is *numeralwise representable* [Kleene 1950] in a reduction system with numerals R if there is a term $F \in R$ such that

$$\varphi(n_1, n_2, \ldots, n_k) = m \iff F \cdot \llcorner n_1 \lrcorner \cdot \llcorner n_2 \lrcorner \cdot \ldots \cdot \llcorner n_k \lrcorner \twoheadrightarrow \llcorner m \lrcorner \, . \tag{1}$$

This definition is quite general in that it does not specify whether the R-terms have types nor whether they are open or closed. If we define $\vec{n} \equiv n_1, n_2, \ldots, n_k$ for integers n_i $(1 \leq i \leq k)$, define $F \cdot \vec{M} \equiv F \cdot M_1 \cdot M_2 \cdot \ldots \cdot M_k$ for terms F and M_i $(1 \leq i \leq k)$, and extend the numeral notation $\llcorner \; \lrcorner$ to work on vectors in the obvious way, then (1) becomes

$$\varphi(\vec{n}) = m \iff F \cdot \llcorner \vec{n} \lrcorner \twoheadrightarrow \llcorner m \lrcorner \, . \tag{2}$$

The forward direction of (2) is often called *definability* [Barendregt 1984, p. 135]. Given a reduction system R and a recursive function φ, finding an $F \in R$ that defines φ is usually straightforward. But the fact that F defines φ does not imply that F completely characterizes φ; if F defines φ but does not numeralwise represent it (*i.e.*, the reverse direction of (2) does not hold), then it is possible that $F \cdot \llcorner \vec{n} \lrcorner \twoheadrightarrow \llcorner x \lrcorner$ for some $x \neq m$.

One way to prove that such an F numeralwise represents φ is to show that the Church-Rosser property holds for R. However, a weaker condition suffices. Define the *unique answer condition* for a term F as

$$F \cdot \llcorner \vec{n} \lrcorner \twoheadrightarrow \llcorner m_1 \lrcorner, \quad F \cdot \llcorner \vec{n} \lrcorner \twoheadrightarrow \llcorner m_2 \lrcorner \implies m_1 = m_2 \, .$$

Then we have:

Lemma 4.1. Let $\varphi \in \mathbb{N}^k \to \mathbb{N}$ be defined by a term F in a reduction system with numerals. The unique answer condition holds for F iff F numeralwise represents φ.

Proof. Straightforward. ■

While the Church-Rosser property applies to an entire reduction system, the unique answer condition applies only to the particular terms used to define particular functions. Of course, if the Church-Rosser property holds for a reduction system R, then the unique answer condition holds for all terms in R.

At the end of this section, we shall prove that the recursive functions are numeralwise representable in the $\lambda^{\star}$-calculus. Once this is established, we have the undecidability result via the following:

Theorem 4.2. If the recursive functions are numeralwise representable in a reduction system R, then $\mathrm{NF}(R)$ is undecidable.

Proof. Every decidable set has a recursive characteristic function and, by numeralwise representability, the graph of any such function is recursive in $\mathrm{NF}(R)$. Therefore every decidable set is recursive in $\mathrm{NF}(R)$. Now suppose that $\mathrm{NF}(R)$ is decidable. Then there is a recursive function $T \in \mathbb{N} \to \mathbb{N}$ such that every decidable set is in $\mathrm{TIME}(T)$, the class of sets of deterministic time complexity T. But this contradicts the time hierarchy theorem of complexity theory [Hopcroft & Ullman 1979], so $\mathrm{NF}(R)$ must be undecidable. ■

Representing partial functions. Why not prove the undecidability result by showing that the *partial* recursive functions are numeralwise representable in the $\lambda^{\star}$-calculus? To begin with, the previous definition of numeralwise representability is not adequate for partial functions. If a partial recursive function φ is undefined at some argument vector $\vec{n}$, to what shall the representing term F reduce when applied to $\llcorner\vec{n}\lrcorner$? We think of a λ-term as having computed some value when it reduces to a normal form, so it is natural to propose that undefined values be represented by terms with no normal form.*

If we accept this proposal, then we must ensure that $F \cdot \llcorner\vec{n}\lrcorner$ has no normal form when $\varphi(\vec{n})$ is undefined. Recall that in the $\lambda^{\star}$-calculus, as in many reduction systems, there is more than one way to go about reducing a term; *i.e.*, there are multiple *reduction strategies.* In particular, there is more than one way to proceed from $F \cdot \llcorner\vec{n}\lrcorner$. Hence, we must ensure that no reduction strategy allows $F \cdot \llcorner\vec{n}\lrcorner$ to reach a normal form when $\varphi(\vec{n})$ is undefined.

The minimalization operation is the only possible source of divergence in a partial recursive function expressed in terms of the initial functions and the standard function-forming operations, and the only known way of implementing minimalization in the $\lambda^{\star}$-calculus is with a looping combinator. Therefore, to ensure that minimalization diverges when appropriate, we must ensure that the looping combinator itself, in any context, cannot reach a normal form under any reduction strategy. The known results about looping combinators in the $\lambda^{\star}$-calculus

*Church made this identification early in the development of the untyped λ-calculus; Barendregt and Wadsworth later suggested that undefined values be represented by terms with no *head* normal form (the so-called *unsolvable* terms) [Barendregt 1984, §2.2].

only assert the existence of a single non-normalizing reduction path. If some untried strategy allows the looping combinator to reach a normal form, then a term representing a partial recursive function φ might have a normal form where φ is undefined.

4.3. The $\lambda^{\mathbf{L}}$-calculus

The typed λ-calculus with integers and looping combinators, the $\lambda^{\mathbf{L}}$-calculus, is an instance of the finitely-typed λ-calculus [Barendregt 1984, Appendix A]. Rather than use the traditional syntax, we prefer to use the syntax of λ-terms given in §2. The only significant difference is that the type of a variable is not part of its name in our syntax; the type of a variable is given where it is bound, or by the context in the case of free variables.

The $\lambda^{\mathbf{L}}$-calculus has a single base type $\mathbf{N}$, representing the natural numbers, and the following typed constants for integer arithmetic:

$\mathbf{0} : \mathbf{N}$	zero constant
$\mathbf{s} : \mathbf{N} \to \mathbf{N}$	successor
$\mathbf{c} : \mathbf{N} \to \mathbf{N} \to \mathbf{N} \to \mathbf{N}$	conditional
$\mathbf{p} : \mathbf{N} \to \mathbf{N}$	predecessor

The numerals of $\lambda^{\mathbf{L}}$ are formed from $\mathbf{0}$ and $\mathbf{s}$. For any terms M and N, define $(M^0\ N) \equiv N$ and $(M^{i+1}\ N) \equiv (M\ (M^i\ N))$; then $\llcorner i \lrcorner_{\mathbf{L}} \equiv (\mathbf{s}^i\ \mathbf{0})$. Each numeral is a normal form simply because none of the δ-reduction rules for the arithmetic constants, defined below, are applicable to numerals.

$$\begin{aligned}
(\mathbf{c}\ \mathbf{0}\ x\ y) &\to_{\mathbf{L}} x \\
(\mathbf{c}\ (\mathbf{s}\ z)\ x\ y) &\to_{\mathbf{L}} y \\
(\mathbf{p}\ \mathbf{0}) &\to_{\mathbf{L}} \mathbf{0} \\
(\mathbf{p}\ (\mathbf{s}^i\ 0)) &\to_{\mathbf{L}} (\mathbf{s}^{i-1}\ 0) \qquad (i \geq 1)
\end{aligned}$$

The final line above is actually a reduction rule *scheme*, used here because the simpler rule

$$(\mathbf{p}\ (\mathbf{s}\ x)) \to_{\mathbf{L}} x$$

expresses a stronger condition than is necessary. In particular, this rule is not satisfied by the predecessor combinator to be defined in the $\lambda^{\star}$-calculus, which must mimic $\mathbf{p}$ in order to embed $\lambda^{\mathbf{L}}$ into $\lambda^{\star}$. The $\lambda^{\star}$-combinator, adapted from the polymorphic λ-calculus, can compute the predecessor of a numeral, but it cannot compute the predecessor of $(\mathbf{s}\ x)$ when x is a variable.

The $\lambda^{\mathbf{L}}$-calculus also has a countably infinite set of looping combinators. There is an initial looping combinator

$$\mathbf{L}_0 : (\mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{N}$$

and, for every looping combinator $\mathbf{L}_i$, there is a successor looping combinator $\mathbf{L}_{i+1}$ of the same type such that $(\mathbf{L}_i\ F) \rightarrow_{\mathbf{L}} F\ (\mathbf{L}_{i+1}\ F)$.

From the remarks made above, we have:

Lemma 4.3. The $\lambda^{\mathbf{L}}$-calculus is a reduction system with numerals.

We shall also need:

Lemma 4.4. Reduction in the $\lambda^{\mathbf{L}}$-calculus is Church-Rosser.

Proof. Follows immediately from [Barendregt 1984, Theorem 15.3.3]. ■

4.4. Representing recursive functions in the $\lambda^{\mathbf{L}}$-calculus

Closely following [Barendregt 1984, §6.4], we show that the functions that are numeralwise representable in $\lambda^{\mathbf{L}}$ include the recursive functions. For $k \geq 1$, define

$$\mathbf{N}^k \rightarrow \mathbf{N} \equiv \underbrace{\mathbf{N} \rightarrow \mathbf{N} \rightarrow \cdots \rightarrow \mathbf{N}}_{k} \rightarrow \mathbf{N}\ .$$

The type $\mathbf{N}^k \rightarrow \mathbf{N}$ will be the type of a term representing a k-ary numeric function; rather than introducing cartesian product (tuple) types, the k arguments are "Curried."

Say that a total function $\varphi \in \mathbb{N}^k \rightarrow \mathbb{N}$ is *$\lambda^{\mathbf{L}}$-representable* iff it is numeralwise representable by a closed term of type $\mathbf{N}^k \rightarrow \mathbf{N}$ when the $\lambda^{\mathbf{L}}$-calculus is viewed as a reduction system with numerals. The condition that F be closed and of the appropriate type is necessary because $\lambda^{\mathbf{L}}$ is a typed calculus and we shall construct new representing terms by applying and abstracting over previously constructed representing terms.

In each of the propositions leading up to the main lemma, all of the real work involves proving that some recursive function is definable in $\lambda^{\mathbf{L}}$. That the function is also representable in $\lambda^{\mathbf{L}}$ follows immediately by Lemma 4.4 and Lemma 4.1.

Proposition 4.5. The initial functions are $\lambda^{\mathbf{L}}$-representable.

Proof. For the successor function and the zero constant, simply use the successor and zero constants of $\lambda^{\mathbf{L}}$. For $0 \leq i \leq k$, the i^{th} projection function on k-tuples is represented by the term

$$\pi_i^k \equiv \lambda x_0{:}\mathbf{N}.\lambda x_1{:}\mathbf{N}.\cdots.\lambda x_k{:}\mathbf{N}.x_i \ . \ \blacksquare \tag{1}$$

If we define

$$\lambda \vec{x}{:}\mathbf{N}^k.M \equiv \lambda x_1{:}\mathbf{N}.\lambda x_2{:}\mathbf{N}.\cdots.\lambda x_k{:}\mathbf{N}.M \ ,$$

then (1) becomes

$$\pi_i^k \equiv \lambda \vec{x}{:}\mathbf{N}^k.x_i \ .$$

Proposition 4.6. The $\lambda^{\mathbf{L}}$-representable functions are closed under composition.

Proof. Let $\varphi \in \mathbb{N}^l \to \mathbb{N}$ be defined by

$$\varphi(\vec{n}) \equiv \chi(\psi_1(\vec{n}), \psi_2(\vec{n}), \ldots, \psi_k(\vec{n})) \ .$$

Assume that the k-argument function χ is represented by a term $G : \mathbf{N}^k \to \mathbf{N}$, and that the l-argument functions $\psi_1, \psi_2, \ldots \psi_k$ are represented by $H_1, H_2, \ldots, H_k$ of type $\mathbf{N}^l \to \mathbf{N}$. Then φ is represented by the term

$$\lambda \vec{x}{:}\mathbf{N}^k.G\,(H_1\,\vec{x})\,(H_2\,\vec{x})\cdots(H_k\,\vec{x}) \ . \ \blacksquare$$

Proposition 4.7. The $\lambda^{\mathbf{L}}$-representable functions are closed under primitive recursion.

Proof. Suppose that $\varphi \in \mathbb{N}^{l+1} \to \mathbb{N}$ is defined by the primitive recursive scheme

$$\begin{aligned} \varphi(\vec{n}, 0) &\equiv \chi(\vec{n}) \\ \varphi(\vec{n}, k+1) &\equiv \psi(\vec{n}, k, \varphi(\vec{n}, k)) \ , \end{aligned}$$

that χ is represented by a term $G : \mathbf{N}^l \to \mathbf{N}$, and that ψ is represented by a term $H : \mathbf{N}^{l+2} \to \mathbf{N}$. We shall compute $\varphi(\vec{n}, k)$ by using a looping combinator to iteratively compute $\varphi(\vec{n}, k-1)$. Letting $\vec{y}$ stand for the arguments $\llcorner\vec{n}\lrcorner$, define

$$F \equiv \lambda f{:}\mathbf{N} \to \mathbf{N}.\lambda x{:}\mathbf{N}.\mathbf{c}\,x\,(G\,\vec{y})\,\big(H\,\vec{y}\,(\mathbf{p}\,x)\,(f\,(\mathbf{p}\,x))\big) \ .$$

If we had a true $\mathbf{Y}$ combinator, then we could represent φ by $(\lambda \vec{y}{:}\mathbf{N}^l.\mathbf{Y}F)$. However, we only need the looping behavior here in order for F to work as required, so a looping combinator $\mathbf{L}_i$ suffices. We shall show by induction on k that $\varphi(\vec{y}, k)$ is represented by $(\mathbf{L}_i\ F\ \llcorner k \lrcorner)$ for any i. By λ-abstraction, it follows immediately that φ is represented by $(\lambda \vec{y}{:}\mathbf{N}^l.\mathbf{L}_i\ F\ \llcorner k \lrcorner)$.

For the basis, $k = 0$ and we have

$$\begin{aligned}
\mathbf{L}_i\ F\ \llcorner 0 \lrcorner &= \mathbf{L}_i\ F\ \mathbf{0} \\
&\to F\,(\mathbf{L}_{i+1}\ F)\,\mathbf{0} \\
&\twoheadrightarrow \mathbf{c}\ \mathbf{0}\,(G\,\vec{y})\,\big(H\,\vec{y}\,(\mathbf{p}\ \mathbf{0})\,(\mathbf{L}_{i+1}\ F\,(\mathbf{p}\ \mathbf{0}))\big) \\
&\to G\,\vec{y}\ .
\end{aligned}$$

Since G represents χ by hypothesis, we have that $(\mathbf{L}_i\ F\ \llcorner 0 \lrcorner)$ represents $\varphi(\vec{y}, 0)$.

For $k + 1 > 0$, we have

$$\begin{aligned}
\mathbf{L}_i\ F\ \llcorner k+1 \lrcorner &= \mathbf{L}_i\ F\,(\mathbf{s}\ \llcorner k \lrcorner) \\
&\to F\,(\mathbf{L}_{i+1}\ F)\,(\mathbf{s}\ \llcorner k \lrcorner) \\
&\twoheadrightarrow \mathbf{c}\,(\mathbf{s}\ \llcorner k \lrcorner)\,(G\,\vec{y})\,\big(H\,\vec{y}\,(\mathbf{p}\,(\mathbf{s}\ \llcorner k \lrcorner))\,(\mathbf{L}_{i+1}\ F\,(\mathbf{p}\,(\mathbf{s}\ \llcorner k \lrcorner)))\big) \\
&\to \big(H\,\vec{y}\,(\mathbf{p}\,(\mathbf{s}\ \llcorner k \lrcorner))\,(\mathbf{L}_{i+1}\ F\,(\mathbf{p}\,(\mathbf{s}\ \llcorner k \lrcorner)))\big) \\
&\twoheadrightarrow (H\,\vec{y}\ \llcorner k \lrcorner\,(\mathbf{L}_{i+1}\ F\ \llcorner k \lrcorner))\ .
\end{aligned}$$

By induction, we know that $(\mathbf{L}_{i+1}\ F\ \llcorner k \lrcorner)$ represents $\varphi(\vec{y}, k)$; since H represents ψ by hypothesis, it follows that $(\mathbf{L}_i\ F\ \llcorner k+1 \lrcorner)$ represents $\varphi(\vec{y}, k+1)$. ∎

Proposition 4.8. The $\lambda^{\mathbf{L}}$-representable functions are closed under minimalization.

Proof. Suppose that $\varphi \in \mathbb{N}^k \to \mathbb{N}$ is defined by

$$\varphi(\vec{n}) \equiv \mu m.\chi(\vec{n}, m) = 0$$

for a numeric function $\chi \in \mathbb{N}^{k+1} \to \mathbb{N}$ such that $\forall \vec{n}.\exists m.\chi(\vec{n}, m) = 0$. To compute $\varphi(\vec{n})$, we shall use a looping combinator to recursively check $\chi(\vec{n}, 0)$, then $\chi(\vec{n}, 1)$, and so on until an m is found such that $\chi(\vec{n}, m) = 0$. The search will always terminate because we have assumed that such an m exists. It is this assumption that restricts the representability result to the total, as opposed to partial, recursive functions.

First, let us see how to search in the simple case when $k = 0$.

Claim. Suppose that $\psi \in \mathbb{N} \to \mathbb{N}$ is represented by $P : \mathbf{N} \to \mathbf{N}$. Then there is a term *search* such that $(\mathbf{L}_0\ (\textit{search}\ P)\ \mathbf{0}) \twoheadrightarrow \llcorner m \lrcorner$, where $m = (\mu m.\psi(m) = 0)$.

Proof. Define

$$\textit{search} \equiv \lambda g{:}\mathbf{N} \to \mathbf{N}.\lambda f{:}\mathbf{N} \to \mathbf{N}.\lambda z{:}\mathbf{N}.\mathbf{c}\,(g\ z)\ z\ (f\ (\mathrm{s}\ z))\ .$$

By induction on $m - i$, we prove that $(\mathbf{L}_i\,(\textit{search}\ P)\,\llcorner i \lrcorner) \twoheadrightarrow \llcorner m \lrcorner$ if $i \leq m$. If $m = i$ then $\psi(i) = 0$; hence $(P\,\llcorner i \lrcorner) \twoheadrightarrow \mathbf{0}$ and we have the following reductions:

$$\begin{aligned}
\mathbf{L}_i\,(\textit{search}\ P)\,\llcorner i \lrcorner &\to (\textit{search}\ P)\,(\mathbf{L}_{i+1}\,(\textit{search}\ P))\,\llcorner i \lrcorner \\
&\twoheadrightarrow \mathbf{c}\,(P\,\llcorner i \lrcorner)\,\llcorner i \lrcorner\,(\mathbf{L}_{i+1}\,(\textit{search}\ P)\,(\mathrm{s}\,\llcorner i \lrcorner)) \\
&\twoheadrightarrow \mathbf{c}\,\mathbf{0}\,\llcorner i \lrcorner\,(\mathbf{L}_{i+1}\,(\textit{search}\ P)\,(\mathrm{s}\,\llcorner i \lrcorner)) \\
&\to \llcorner i \lrcorner\ .
\end{aligned}$$

On the other hand, if $i < m$ then $\psi(i) = n \neq 0$ (n exists because ψ is total); therefore $(P\,\llcorner i \lrcorner) \twoheadrightarrow \llcorner n \lrcorner$ and we have

$$\begin{aligned}
\mathbf{L}_i\,(\textit{search}\ P)\,\llcorner i \lrcorner &\twoheadrightarrow \mathbf{c}\,(P\,\llcorner i \lrcorner)\,\llcorner i \lrcorner\,(\mathbf{L}_{i+1}\,(\textit{search}\ P)\,(\mathrm{s}\,\llcorner i \lrcorner)) \\
&\twoheadrightarrow \mathbf{c}\,\llcorner n \lrcorner\,\llcorner i \lrcorner\,(\mathbf{L}_{i+1}\,(\textit{search}\ P)\,(\mathrm{s}\,\llcorner i \lrcorner)) \\
&\to \mathbf{L}_{i+1}\,(\textit{search}\ P)\,(\mathrm{s}\,\llcorner i \lrcorner) \\
&= \mathbf{L}_{i+1}\,(\textit{search}\ P)\,\llcorner i + 1 \lrcorner\ .
\end{aligned}$$

By induction, $(\mathbf{L}_{i+1}\,(\textit{search}\ P)\,\llcorner i+1 \lrcorner) \twoheadrightarrow \llcorner m \lrcorner$. Thus, if we begin searching at 0, $(\mathbf{L}_0\,(\textit{search}\ P)\,\mathbf{0})$ will reduce to $\llcorner m \lrcorner$ since $m \geq 0$ for any ψ. ∎

To complete the proof of the proposition, assume that χ is represented by a term $G : \mathbf{N}^{k+1} \to \mathbf{N}$. For a particular vector of arguments $\vec{n}$, we can compute $\mu m.\chi(\vec{n}, m) = 0$ by defining $\psi(m) = \chi(\vec{n}, m)$ and computing $\mu m.\psi(m) = 0$. Now $(G\,\llcorner \vec{n} \lrcorner)$ represents ψ, therefore φ is represented by the term

$$F \equiv \lambda \vec{x}{:}\mathbf{N}^k.\mathbf{L}_0\,(\textit{search}\,(G\,\vec{x}))\,\mathbf{0}\ .$$

By the claim, $(F\,\llcorner \vec{n} \lrcorner) \twoheadrightarrow \llcorner \mu m.\chi(\vec{n}, m) = 0 \lrcorner$ for any $\vec{n}$. ∎

Lemma 4.9. The recursive functions are $\lambda^{\mathbf{L}}$-representable.

Proof. By the preceding Propositions. ∎

Corollary 4.10. The normal-form relation of the $\lambda^{\mathbf{L}}$-calculus is undecidable.

Proof. By Lemma 4.9 and Theorem 4.2. ∎

4.5. Embedding reduction systems

Let R and S be two reduction systems with numerals, and let $trans_{R,S} \in R \to S$ be a translation function mapping terms of R to terms of S. If $M \in R$, then $\overline{M}$ abbreviates $trans_{R,S}(M)$. We say that $trans_{R,S}$ *embeds R into S* iff

(1) For every $m \geq 0$, $\overline{\llcorner m \lrcorner_R} \twoheadrightarrow_S \llcorner m \lrcorner_S$,

(2) For any $M \in R$ and $n \geq 0$, $M \twoheadrightarrow_R \llcorner n \lrcorner \implies \overline{M} \twoheadrightarrow_S \overline{\llcorner n \lrcorner}$, and

(3) For $M, N \in R$, $\overline{M \cdot_R N} = \overline{M} \cdot_S \overline{N}$.

In other words, the translation function preserves numerals and reduction paths ending at numerals, and commutes with the application operators.

Lemma 4.11. Let $\mathcal{F}$ be a class of numeric functions numeralwise representable in R. If R can be embedded into S and, for every $\varphi \in \mathcal{F}$ represented by some $F \in R$, the unique answer condition holds for $\overline{F} \in S$, then the functions in $\mathcal{F}$ are numeralwise representable in S.

Proof. Suppose that $\varphi \in \mathbb{N}^k \to \mathbb{N}$ is numeralwise represented by a term $F \in R$. Then, for any $\vec{n} \in \mathbb{N}^k$ and $m \in \mathbb{N}$, we have

$$\begin{aligned} \varphi(\vec{n}) = m \quad &\iff \quad F \cdot_R \llcorner \vec{n} \lrcorner_R \twoheadrightarrow_R \llcorner m \lrcorner_R \\ &\implies \quad \overline{F \cdot_R \llcorner \vec{n} \lrcorner_R} \twoheadrightarrow_S \overline{\llcorner m \lrcorner_R} \ , \end{aligned}$$

since R can be embedded into S. The embedding translation preserves numerals and commutes with term application in S, therefore $\overline{F} \cdot_S \llcorner \vec{n} \lrcorner_S \twoheadrightarrow_S \llcorner m \lrcorner_S$, so φ is defined by $\overline{F} \in S$. By Lemma 4.1, φ is in fact represented by $\overline{F}$. ∎

Corollary 4.12. If R can be embedded into S and the recursive functions are numeralwise representable in R, then the normal-form relation of S is undecidable.

Proof. By Theorem 4.2 and Lemma 4.11. ∎

4.6. Undecidability in the $\lambda^\star$-calculus

We are now ready to prove that the normal-form relation of the $\lambda^\star$-calculus is undecidable. We can compute on integers in $\lambda^\star$ by adapting the polymorphic Church numerals [Fortune *et al.* 1983]. Define the numeral type

$$\dot{\mathbb{N}} \equiv \Pi t{:}\star.(t \to t) \to (t \to t) \ ;$$

then the numerals of $\lambda^\star$ are defined as

$$\llcorner i \lrcorner_\star \equiv \lambda t{:}\star.\lambda f{:}t \to t.\lambda x{:}t.f^i\,x \ ,$$

with the zero constant $\dot{\mathbf{0}} \equiv \llcorner 0 \lrcorner_\star$. By inspection, it is evident that the numerals are distinct and that each is a closed normal form. The $\lambda^\star$-combinators for the successor and conditional operators are

$$\dot{\mathbf{s}} \equiv \lambda n{:}\dot{\mathbf{N}}.\lambda t{:}\star.\lambda f{:}t \to t.\lambda x{:}t.f\,(n\;t\;f\;x) \ ,$$
$$\dot{\mathbf{c}} \equiv \lambda n{:}\dot{\mathbf{N}}.\lambda a{:}\dot{\mathbf{N}}.\lambda b{:}\dot{\mathbf{N}}.n\;\dot{\mathbf{N}}\;(\lambda z{:}\dot{\mathbf{N}}.b)\;a \ .$$

To define predecessor, we first need a special pairing operation on integers that is definable in terms of $\dot{\mathbf{c}}$. For terms $M, N : \dot{\mathbf{N}}$, define

$$\langle M, N\rangle \equiv \lambda n{:}\dot{\mathbf{N}}.\dot{\mathbf{c}}\;n\;M\;N \ .$$

The pair $\langle M, N\rangle$ is such that $(\langle M, N\rangle\;\dot{\mathbf{0}}) \twoheadrightarrow M$ and $(\langle M, N\rangle\;(\dot{\mathbf{s}}\,x)) \twoheadrightarrow N$. The basic trick behind the predecessor operation is due to [Kleene 1979] and is explained in [Fortune *et al.* 1983, p. 161].

$$\dot{\mathbf{p}} \equiv \lambda n{:}\dot{\mathbf{N}}.n\;(\dot{\mathbf{N}} \to \dot{\mathbf{N}})\;\big(\lambda z{:}\dot{\mathbf{N}} \to \dot{\mathbf{N}}.\langle(\dot{\mathbf{s}}\,(z\,\dot{\mathbf{0}})), (z\,\dot{\mathbf{0}})\rangle\big)\;\langle\dot{\mathbf{0}},\dot{\mathbf{0}}\rangle\;(\dot{\mathbf{s}}\,\dot{\mathbf{0}}) \ .$$

From these observations, we have

Lemma 4.13. The $\lambda^\star$-calculus is a reduction system with numerals.

Proposition 4.14. The reduction rules defined for the $\lambda^{\mathbf{L}}$-constants $\mathbf{c}$ and $\mathbf{p}$ are observed by the $\lambda^\star$-combinators $\dot{\mathbf{c}}$ and $\dot{\mathbf{p}}$, and $(\dot{\mathbf{s}}^i\,\dot{\mathbf{0}}) \twoheadrightarrow \llcorner i \lrcorner_\star$.

Proof by verifying the appropriate reductions. To show $(\dot{\mathbf{s}}^i\,\dot{\mathbf{0}}) \twoheadrightarrow \llcorner i \lrcorner_\star$, proceed by induction on i. ■

To embed the $\lambda^{\mathbf{L}}$-calculus into the $\lambda^\star$-calculus, we must define a translation not only on terms but on types as well, since $\lambda^{\mathbf{L}}$-terms may contain types. The translation from $\lambda^{\mathbf{L}}$-types to $\lambda^\star$-terms is defined:

$$\overline{\mathbf{N}} \equiv \dot{\mathbf{N}} \ ,$$
$$\overline{A \to B} \equiv \Pi x{:}A.B \ .$$

Because all $\lambda^{\mathbf{L}}$-types are built from the base type $\mathbf{N}$, there is no possibility that x will be free in B, so we need not state this as a restriction. The $\lambda^{\mathbf{L}}$-terms are translated as follows, where $\dot{\mathbf{L}}$ denotes the polymorphic looping combinator constructed in §3:

$$\begin{aligned}
\overline{\mathbf{0}} &\equiv \dot{\mathbf{0}} \\
\overline{\mathbf{s}} &\equiv \dot{\mathbf{s}} \\
\overline{\mathbf{c}} &\equiv \dot{\mathbf{c}} \\
\overline{\mathbf{p}} &\equiv \dot{\mathbf{p}} \\
\overline{\mathbf{L}_i} &\equiv \dot{\mathbf{L}}_i\,\overline{(\mathbf{N} \to \mathbf{N}) \to \mathbf{N}} \\
\overline{x} &\equiv x \\
\overline{(M\,N)} &\equiv (\overline{M}\,\overline{N}) \\
\overline{(\lambda x{:}A.M)} &\equiv (\lambda x{:}\overline{A}.\overline{M})
\end{aligned}$$

Lemma 4.15. The above translation embeds $\lambda^{\mathbf{L}}$ into $\lambda^{\star}$, when these calculi are viewed as reduction systems.

Proof. The translation obviously commutes with the application operators. To see that numerals are preserved, note that, by Proposition 4.14,

$$\overline{\llcorner i \lrcorner_{\mathbf{L}}} = \overline{(\mathbf{s}^i\,\mathbf{0})} = (\dot{\mathbf{s}}^i\,\dot{\mathbf{0}}) \twoheadrightarrow_{\star} \llcorner i \lrcorner_{\star}\ .$$

To show that $M \twoheadrightarrow_{\mathbf{L}} \llcorner n \lrcorner \implies \overline{M} \twoheadrightarrow_{\star} \overline{\llcorner n \lrcorner}$, we proceed by induction on the length of the $\lambda^{\mathbf{L}}$-reduction path from M to $\llcorner n \lrcorner$. If $M \to_{\mathbf{L}} N$ by a β-reduction, then the same β-reduction will take place in $\lambda^{\star}$. If $M \to_{\mathbf{L}} N$ by one of the arithmetic constant reduction rules, then, by Proposition 4.14, a chain of one or more β-reductions will take $\overline{M}$ to $\overline{N}$ in $\lambda^{\star}$. If $M \to_{\mathbf{L}} N$ by a looping combinator reduction, then, by Theorem 3.9, a chain of one or more β-reductions will take $\overline{M}$ to $\overline{N}$ in $\lambda^{\star}$. ■

Theorem 4.16. The normal-form relation of the $\lambda^{\star}$-calculus is undecidable.

Proof. By Lemma 2.1, Lemma 4.9, Corollary 4.12, and Lemma 4.15. ■

5. Conclusion

The main results of this thesis can be summarized as follows.

- A family of polymorphic looping combinators can be constructed in the $\lambda^{\star}$-calculus.
- The existence of these looping combinators implies that the reduction, equational, and typing theories of the calculus are all undecidable.

We now discuss some practical and theoretical consequences of these results.

Dependent function types in programming languages. The failure of strong normalization and the undecidability of the reduction and equational theories of terms is not surprising for real programming languages, because in such languages we expect to be able to express possibly divergent recursive computations using primitives such as **letrec**. Furthermore, since typechecking may require arbitrary computation, typechecking will be undecidable for a language with recursion and dependent function types whether it has a type of all types or not.

This might seem to disqualify dependent function types as a useful feature of practical programming languages, but there are a number of ways to make them more palatable. One extreme is taken by CLU, in which *parameterized procedures* are similar to functions with dependent types. In CLU, a parameterized procedure can be applied only to a type or to a constant expression of some built-in type, the value of which can be computed at compile time. At the other extreme, a clever compiler could, in principle, detect any recursive computation that arises during typechecking by watching for invocations of **letrec** or other recursion constructs. Upon detecting a recursive computation, the compiler could either warn that the compilation may not terminate or simply refuse to compile the program.

The primary consequence of our result for programming language practice is that this latter strategy will not work for a language with recursion that subsumes the $\lambda^{\star}$-calculus. Recursive computations can be expressed in such a language using looping combinators, which cannot be detected by a compiler. One might object that it is unlikely that a programmer would, intentionally or otherwise, formalize Girard's paradox and thereby subvert the typechecker, but we do not know that simpler looping combinators do not exist.

Dependent pair types. As we saw in §3, dependent pair types correspond to existential propositions under the propositions-as-types analogy. Dependent pair types seem to have first appeared in [Howard 1969], who distinguished between *weak* dependent pairs, which only have an *out*-like accessing operation, and *strong* dependent pairs, which have both left- and right-hand projections. (Having a left-hand projection and *out*, but lacking a right-hand projection, the dependent pairs definable in the $\lambda^\star$-calculus lie between the two.)

[Mitchell & Plotkin 1985] observe that implementations of abstract data types in languages such as Ada, CLU, and ML can be modelled by weak dependent pairs. [MacQueen 1986] argues that weak dependent pairs are not flexible enough for full-fledged "programming in the large," and proposes instead the use of strong dependent pairs. Unfortunately, the familiar polymorphic λ-calculus of [Girard 1972] and [Reynolds 1974] augmented with strong dependent pairs admits the formalization of Girard's paradox, which implies the failure of strong normalization, the ability to construct looping combinators, and thereby the undecidability of normal forms, equality, and typing. This was discovered simultaneously by [Harper & Mitchell 1986] and [Hook & Howe 1986], who demonstrated that the $\lambda^\star$-calculus can be simulated in the polymorphic λ-calculus with strong dependent pairs, and by [Coquand 1986], who argued that Girard's paradox can be carried out in such a language directly. The difficulties stemming from Girard's paradox can be avoided either by retreating to weak dependent pairs, or by settling for a less flexible form of polymorphism [Mitchell & Harper 1988].

Kernel languages. Our interest in programming languages with dependent function types and the type of all types was originally sparked by [Burstall & Lampson 1984], which describes Pebble, a "kernel language" for abstract data types and modules. Pebble is a richer language than the $\lambda^\star$-calculus; it supports dependent function types and the type of all types, but it also includes recursion, integer and boolean base types, tuple types, and strong dependent pairs. Yet the type system of Pebble does not seem to have the full power of that of the $\lambda^\star$-calculus. One major difference is that Pebble does not allow α-conversion in dependent function types, so that $\Pi x{:}A.B$ and $\Pi y{:}A.B[y/x]$ are considered to be two distinct types. The $\lambda^\star$-calculus, which does not make this distinction, cannot be simulated under this restriction, and so our results for the $\lambda^\star$-calculus are not directly applicable to Pebble.

Related to Pebble is the typed λ-calculus of [Cardelli 1986], which is essentially the $\lambda^\star$-calculus enriched with strong existential and recursive types. This language subsumes the $\lambda^\star$-calculus, so our results extend to it trivially.

5.1. Context and history of this work

In [Meyer & Reinhold 1986] we claimed the existence of a polymorphic fixed-point combinator in a typed λ-calculus essentially equivalent to the $\lambda^\star$-calculus. Shortly before that paper was presented, I found two major flaws in the construction of the combinator. First, the construction relied upon features not present in the pure calculus. My construction was based on the presentation in [Martin-Löf 1972], which implicitly uses a richer language that contains strong dependent pairs and a flexible primitive recursion operator for functions on integers. It appears that neither of these constructs can be simulated in the pure $\lambda^\star$-calculus, although this has not been proved. Second, even under the assumption that these features existed, I could not find a convincing, rigorous proof that the constructed term really was a fixed-point operator. On the positive side, I observed that the constructed term was a polymorphic looping combinator [Reinhold 1986], and I conjectured that the construction would lead to a true fixed-point combinator when carried out in the pure system.

In the fall of 1985 we learned that Thierry Coquand had constructed, and checked by machine, a formalization of Girard's paradox in an extension of his Calculus of Constructions [Coquand 1985a]. Encouraged by this result, in the spring of 1986 I began working from a preprint of [Coquand 1986] and from [Girard 1972] to obtain a formalization of the paradox in the pure $\lambda^\star$-calculus. It had been apparent beforehand that machine assistance would be required in order to completely check the formal construction. Even the original construction, carried out in a richer language, requires about five pages to write down, and checking it by hand is a tedious and error-prone process. The obvious complexity of the construction in the pure $\lambda^\star$-calculus lead to the initial design and development of LP in the summer of 1986. With this system, I checked my original construction and empirically verified that it loops under reduction.

In the meantime, Doug Howe had been working on formalizing and analyzing Girard's paradox with the aid of the NUPRL proof development system [Constable 1986]. In the fall of 1986 a preprint of [Howe 1987] arrived, in which a more complete formulation of the paradox than given in [Coquand 1986] is presented. Howe applied my looping combinator construction method and proved that he obtained a looping combinator that was *not* a fixed-point operator.

Using Howe's formulation of the paradox, I was quickly able to complete my own formalization with the aid of LP and verify that our method was applicable to it, obtaining the looping combinator $\mathbf{L}_0$. The paradox and the construction method were presented in §3, and a complete listing of $\mathbf{L}_0$ is given in §B. In light of Howe's proof and Coquand's more recent argument, mentioned in §3, it does not seem worth duplicating their work in order to rigorously prove that $\mathbf{L}_0$ is in fact a looping combinator.

An important detail that has been missing from the published work on looping combinators in the $\lambda^\star$-calculus is the fact that the existence of a looping combinator does not immediately imply the undecidability of normal forms, equations, and typings in the calculus. Indeed, as discussed in §4, the obvious method for proving undecidability, namely by showing that all partial recursive functions can be computed, relies upon knowing more than we do about the reduction behavior of looping combinators in the $\lambda^\star$-calculus. In §4 we proved by an alternative method that normal forms, equations, and typings are undecidable in the $\lambda^\star$-calculus.

5.2. Open questions

The claim of [Meyer & Reinhold 1986] that fixed-point operators exist in the $\lambda^\star$-calculus was premature, leaving open the following question:

(1) Do fixed-point operators exist in the $\lambda^\star$-calculus?

It seems likely that a nonexistence proof will require semantic rather than syntactic methods.

A major theme of [Meyer & Reinhold 1986] that we have not addressed here is the *conservative extension* of algebraic theories by typed λ-calculi [Breazu-Tannen & Meyer 1987, Breazu-Tannen 1987]. An algebraic theory is conservatively extended by a typed λ-calculus if, when the function symbols and axioms of the theory are added to the calculus, the equations provable between algebraic terms in the calculus are just those that were provable in the original theory. The value of conservative extension theorems for reasoning about programs is that they allow the familiar methods of equational reasoning to be used instead of the more complex methods that have been developed for reasoning about divergent computations [Gordon *et al.* 1979, Paulson 1984]. In [Meyer & Reinhold 1986] we concluded that the $\lambda^\star$-calculus does not conservatively extend every algebraic theory, but this claim depends upon the existence of a fixed-point operator. The unknown status of fixed-point operators and the presence of looping combinators in the $\lambda^\star$-calculus raises the question:

(2) Do looping combinators conservatively extend algebraic theories?

To start, one might consider whether the $\lambda^{\mathbf{L}}$-calculus of §4 is a conservative extension of every algebraic theory.

We mention here that a related open question of [Meyer & Reinhold 1986] has been answered: The pure $\lambda^\star$-calculus *is* a conservative extension of the pure λ^{Π}-calculus, the typed λ-calculus with dependent function types [Breazu-Tannen 1986].

Finally, a more pragmatic question is:

(3) Are dependent function types useful of themselves, in the absence of the type of all types or similar constructs?

While dependent types are not explicitly available in many current programming languages, they occur implicitly in a variety of ways. For example, a well-known limitation of procedures in Pascal is the restriction that an array parameter must be of a fixed size that can be determined at compile time. Many implementations of Pascal overcome this problem by extending the procedure call mechanism so that the dimensions of an array are passed along with the array itself; the dimensions are made available to the called procedure via special "dimension" variables.

There are many examples of interesting functions in real programming languages that are typically untyped or weakly typed, but which can be accurately described by dependent types. Consider the *format* procedure available in most dialects of Lisp [Steele 1984] (equivalently, the *printf* procedure of C [Kernighan & Ritchie 1978]). *Format* is applied to a format string and some number of arguments, and patterns in the format string specify how the arguments are to be printed. While the first argument to *format* must be a string, the types of the remaining arguments depend upon the particular patterns given in the string. If we think of *format* as a function mapping a string to another function that will actually perform the output, then *format* can be given a dependent type, since the type of the returned function depends upon the value of the string argument.

A. The LP program

A.1. Data structures and algorithms

The core of the LP program is a reduction engine for typed λ-terms. Terms are represented as graphs, much as in [Wadsworth 1971], and reduction is performed by a closure-based algorithm similar to that of [Aiello & Prini 1981].

Once reduction is implemented, it is a simple matter to implement a type derivation algorithm that computes the type, if any, of an arbitrary term. Due to the type conversion (tc) rule, the proof system for typing statements presented in §2 is non-deterministic. However, it is not hard to demonstrate that any provable typing statement has a proof in which the (tc) rule is used only at the very end of the proof or immediately before the (Πe) rule, to convert the type of the operand of an application. This observation leads to the algorithm shown below, expressed as a recursively defined function $\mathcal{T}$ of type *context* $\rightarrow$ *term* $\rightarrow$ *term*. The algorithm traverses an input term by recursive descent, extending the context Δ whenever an abstraction term is entered.

```
$\mathcal{T}\Delta(x) \equiv$ **if** $x \in \mathrm{dom}\,\Delta$
          **then** $\Delta(x)$
          **else error**: undeclared variable $x$

$\mathcal{T}\Delta(\star) \equiv \star$

$\mathcal{T}\Delta(\Pi x{:}A.B) \equiv$ **if** $\mathcal{T}\Delta(A) \neq \star$
    **then error**: type of declared variable is not of type $\star$
    **else if** $\mathcal{T}(\Delta, x{:}A)(B) \neq \star$
        **then error**: $\Pi$-body does not have type $\star$
        **else** $\star$

$\mathcal{T}\Delta(\lambda x{:}A.B) \equiv$ **if** $\mathcal{T}\Delta(A) \neq \star$
    **then error**: type of declared variable is not of type $\star$
    **else** $(\Pi x{:}A.\mathcal{T}(\Delta, x{:}A)(B))$

$\mathcal{T}\Delta(M\ N) \equiv$ **let** $A = \mathcal{T}\Delta(M)$ **and** $B = \mathcal{T}\Delta(N)$
    **in if** $\neg$*pi-term?*$(A)$
        **then error**: operator does not have a $\Pi$-type
        **else if** *bound-var-type*$(A) \not\twoheadleftarrow\!\!\twoheadrightarrow B$
            **then error**: ill-typed application
            **else** *pi-body*$(A)[N/x]$
```

A.2. Syntax and sugar

The notation for terms of the $\lambda^\star$-calculus presented in §2 is pleasant enough for human consumption, but it takes some time to teach a computer how to read it. As LP was written in Lisp, and we already have text editors with powerful facilities for editing Lisp-like programs, it was decided to use the list-reading facilities of the Lisp implementation to parse terms input to LP. Thus the LP input language is different in appearance from the usual notation. Variables of the $\lambda^\star$-calculus are represented by symbols in LP (alphabetic case *is* significant); beyond that, the correspondence is:

$\star$	*
$(\lambda x{:}A.M)$	`(\x A M)`
$(\Pi x{:}A.B)$	`(!x A B)`
$(M\,N)$	`(M N)`

Some syntactic sugar is defined. Application, as usual, is left-associative, so `(f a b)` is equivalent to `((f a) b)`. An abbreviation for `(!x A B)` is `(A -> B)` if x does not occur free in B; '`->`' is right-associative. A chain of λ-abstractions such as `(\x A (\y B (\z C ...)))` may be abbreviated:

```
(\ ((x A)
    (y B)
    (z C))
   ...)
```

Chains of Π-abstractions may be abbreviated in a similar fashion. The `let` construct

```
(let ((x A M)
      (y B N)
      (z C O))
   ...)
```

is sugar for the application

```
((\ ((x A)
     (y B)
     (z C))
    ...)
 M N O) .
```

Like Lisp's "read-eval-print" loop, LP has a top-level "read-type-eval-print" loop that reads a term, attempts to compute its type, and, if the term is well-typed, prints its type and its normal form. Certain symbols are predefined as LP commands; the most important of these is the `def` command. A term of the form (`def` x M) defines the symbol x in the global environment to be the term M, if M is well-typed. After this definition, any free occurrence of x, in any term, will refer to M. Terms in the environment are kept in normal form for efficiency, but some terms (*e.g.*, any looping combinator) do not have normal forms. The `defx` command is used in such cases; it works just like `def` except that it does not attempt to reduce the defined term to normal form.

LP has a macro facility that is used to define local abbreviations within the paradox construction (they are also used in the implementation of dependent tuples, described below). A term of the form

$$(\texttt{mlet}\ ((x_1\ M_1)\ (x_2\ M_2)\ \cdots\ (x_k\ M_k))\ N)$$

expands into $N[\vec{M}/\vec{x}]$. Macros are expanded from the outside inwards, and expansion takes place before typechecking is done.

A.3. Dependent tuple types

The formulation and proof of Girard's paradox makes frequent use of dependent pair types. The paradox could, in principle, be formalized using the definitions of §2, but the resulting term would be rather complex. Certain propositions of the paradox are expressed as nested dependent pair types, entailing corresponding nested uses of *out* to get at the innermost dependent component of an element of such a type (*e.g.*, consider $r_{\mathcal{U}}$, in which dependent pair types are nested six deep). Each use of *out* must mention the result type X of the function f that accesses the components of the dependent pair;* moreover, the dependent parts of most of the dependent pair types used in the paradox are themselves composed of conjunctions of two or three types.

The complexity of terms involving nested dependent pair types motivated the introduction of *dependent tuple types*, a generalization of dependent pair types that is supported in part by the macro expansion mechanism of LP. An element of a dependent tuple type can be thought of as a record containing named fields $a_1{:}A_1, a_2{:}A_2, \ldots, a_n{:}A_n$ in which the type t_i of field a_i can depend upon the values of the preceding fields $a_1, a_2, \ldots, a_{i-1}$. Dependent pair types can also be parameterized, and the type of each parameter may depend upon the values of previous parameters. (This parameterization could have been done with ordinary λ-abstraction, but it was more convenient for the paradox construction to include it as part of the dependent tuple type construct.)

*Uses of *out*, which require mentioning the type A and the function B, can be eliminated by directly applying a dependent pair to the desired X and f, but X must still be given.

A dependent tuple type is declared to LP by a command of the form

```
(def-dtuple (G ((p1 s1) (p2 s2) ··· (pk sk)))
    ((a1 t1) (a2 t2) ··· (an tn))) ,
```

where G is the name of the tuple type, each parameter p_i has type s_i, and each field a_i has type t_i. The declaration of a tuple type G causes three things to be defined: G is bound to a term describing the actual type of the tuple, $\hat{}G$ is bound to a *constructor function* that produces an element of the tuple when applied to actual parameters and field values of the appropriate type, and @G is bound to an *accessor macro* that can be used to "open up" an entire tuple with a single operation, allowing a field a of a tuple variable x to be accessed as $x.a$.

Specifically, the above declaration causes G to be defined as

$$G \equiv \lambda\vec{p}{:}\vec{s}.\Pi X{:}\star.(\Pi\vec{a}{:}\vec{t}.X) \rightarrow X \ ,$$

and the constructor function $\hat{}G$ to be defined as

$$\begin{aligned}\hat{}G &\equiv \lambda\vec{p}{:}\vec{s}.\lambda\vec{a}{:}\vec{t}.\lambda X{:}\star.\lambda f{:}(\Pi\vec{a}{:}\vec{t}.X).f\,\vec{a}\\ &:\ \Pi\vec{p}{:}\vec{s}.\Pi\vec{a}{:}\vec{t}.G\,\vec{p}\ .\end{aligned}$$

Finally, the accessor macro @G is defined so that (@$G\,x\,\vec{q}\,A\,M$), for some variable x of type $(G\,\vec{q})$, expands into

$$\begin{aligned}&(x\,A\,(\lambda x.a_1{:}t_1[x\vec{.}a/\vec{a}\,][\vec{q}/\vec{p}\,]\\ &\qquad(\lambda x.a_2{:}t_2[x\vec{.}a/\vec{a}\,][\vec{q}/\vec{p}\,]\cdots\\ &\qquad\quad(\lambda x.a_n{:}t_n[x\vec{.}a/\vec{a}\,][\vec{q}/\vec{p}\,].M)\cdots))\ ,\end{aligned}$$

where $t[x\vec{.}a/\vec{a}\,]$ denotes the simultaneous substitution of $x.a_i$ for a_i in t $(1 \le i \le n)$. In words, the application of @G evaluates the body M of type A in an environment with $x.a_i$ bound to the appropriate element of x, for each declared field a_i.

B. The looping combinator

```
;;; polymorphic looping combinator type

(def Loop (!Z * ((Z -> Z) -> Z)))


;;; type of relations on a type A

(def Rel (\A * (A -> A -> *)))


;;; type of predicates on a type A

(def Pred (\A * (A -> *)))


;;; transitivity

(def Trans (\ ((A *)
               (dA (Pred A))                  ; unused argument, to get Trans: OS-pred
               (rA (Rel A)))
              (!x A (!y A (!z A ((rA x y) -> (rA y z) -> (rA x z)))))))


;;; chains

(def-dtuple (Predec ((A *)                    ; predecessor existence predicate
                     (dA (Pred A))
                     (rA (Rel A))
                     (C (Pred A))             ; in chain C,
                     (x A)                    ;   for every x
                     (cx (C x))))             ;   in the chain,
  ((y A)                                      ;   there is a y
   (cy (C y))                                 ;   that is in the chain
   (ryx (rA y x))))                           ;   and smaller than x


(def-dtuple (Chain ((A *)
                    (dA (Pred A))
                    (rA (Rel A))
                    (C (Pred A))))            ; chain C
  ((z A)                                      ; base of chain
   (cz (C z))                                 ; base is in chain
   (dz (dA z))                                ; base is in A
   (pr (! ((x A)                              ; predecessors exist
           (cx (C x)))
          (Predec A dA rA C x cx)))))
```

```
;;; well-foundedness

(def WF (\ ((A *)
            (dA (Pred A))
            (rA (Rel A)))
           (!C (Pred A) ((Chain A dA rA C) -> Loop))))


;;; embedding ordered sets

(def-dtuple (Embed ((A *)
                    (dA (Pred A))
                    (rA (Rel A))
                    (B *)
                    (dB (Pred B))
                    (rB (Rel B))
                    (f (A -> B))                 ; embedding function
                    (b B)))                      ; embedding bound
  ((db (dB b))                                   ; b is in domain of B
   (pres-dom (!x A ((dA x) -> (dB (f x)))))      ; f is domain-preserving
   (mono (!x A (!y A ((dA x)                     ; f is monotonic
                      -> (dA y)
                      -> (rA x y)
                      -> (rB (f x) (f y))))))
   (dominate (!x A ((dA x) -> (rB (f x) b)))))) ; b dominates ran f


;;; ordering on ordered sets based on embedding

(def-dtuple (Embed-ord ((A *)
                        (dA (Pred A))
                        (rA (Rel A))
                        (B *)
                        (dB (Pred B))
                        (rB (Rel B))))
  ((f (A -> B))                                  ; embedding function exists
   (b B)                                         ; embedding bound exists
   (m (Embed A dA rA B dB rB f b))))             ; together they define an embedding
```

```
;;; intensional equality (directed from left to right)

(def Eq (\ ((A *)
           (x A)
           (y A))
          (!P (Pred A) ((P x) -> (P y)))))


;;; intensional equality is reflexive, symmetric, and transitive

(def eq-ref (\ ((A *)
                (x A))
               (: (Eq A x x)
                  (\ ((P (Pred A))
                      (p (P x)))
                     p))))


(def eq-sym (\ ((A *)
                (x A)
                (y A)
                (e (Eq A x y)))
               (: (Eq A y x)
                  (e (\z A (Eq A z x))
                     (eq-ref A x)))))


(def eq-trans (\ ((A *)
                  (x A)
                  (y A)
                  (z A)
                  (p (Eq A x y))
                  (q (Eq A y z)))
                 (: (Eq A x z)
                    (q (\w A (Eq A x w)) p))))
```

```
;;; universal type of ordered sets, and the injection into it

(def OS-pred (!A * ((Pred A) -> (Rel A) -> *))) ; type of predicates on ordered sets

(def U (OS-pred -> *))                          ; type of predicates on OS-predicates

(def inj (\ ((A *)                              ; injection into U
             (dA (Pred A))
             (rA (Rel A)))
            (: U
              (\x OS-pred
                  (x A dA rA)))))


;;; Lemma 1.  If (inj OA) = (inj OB) and (P OA), then (P OB).

(def lemma1 (\ ((A *)
                (dA (Pred A))
                (rA (Rel A))
                (B *)
                (dB (Pred B))
                (rB (Rel B))
                (e (Eq U (inj A dA rA) (inj B dB rB)))
                (P OS-pred)
                (pA (P A dA rA)))
               (: (P B dB rB)
                  (e (\u U (u P)) pA))))


;;; domain predicate on U: transitive, well-founded ordered sets only

(def-dtuple (dU ((u U)))
  ((A *)                                        ; ordered set OA exists
   (dA (Pred A))
   (rA (Rel A))
   (e (Eq U u (inj A dA rA)))                   ; u = (inj OA)
   (t (Trans A dA rA))                          ; OA is transitive
   (w (WF A dA rA))))                           ; OA is well-founded


;;; ordering on elements of U: by embedding

(def-dtuple (rU ((u U)
                 (v U)))
  ((A *)                                        ; ordered set OA exists
   (dA (Pred A))
   (rA (Rel A))
   (B *)                                        ; ordered set OB exists
   (dB (Pred B))
   (rB (Rel B))
   (i (Eq U u (inj A dA rA)))                   ; u = OA
   (j (Eq U v (inj B dB rB)))                   ; v = OB
   (o (Embed-ord A dA rA B dB rB))))            ; OA < OB
```

```
;;; Lemma 2a.  If (dU (inj 0A)), then (Trans 0A).

(def lemma2a
  (\ ((A *)
      (dA (Pred A))
      (rA (Rel A))
      (d (dU (inj A dA rA))))
     (@dU d
          (inj A dA rA)
          (Trans A dA rA)
          ;; apply lemma1
          (lemma1 d.A d.dA d.rA
                  A dA rA
                  (eq-sym U (inj A dA rA) (inj d.A d.dA d.rA) d.e)
                  Trans
                  d.t))))


;;; Lemma 2b.  If (dU (inj 0A)), then (WF 0A).
;;; Identical in structure to lemma2a, except for result.

(def lemma2b
  (\ ((A *)
      (dA (Pred A))
      (rA (Rel A))
      (d (dU (inj A dA rA))))
     (@dU d
          (inj A dA rA)
          (WF A dA rA)
          ;; apply lemma1
          (lemma1 d.A d.dA d.rA
                  A dA rA
                  (eq-sym U (inj A dA rA) (inj d.A d.dA d.rA) d.e)
                  WF
                  d.w))))
```

```
;;; Lemma 3.  If OA < OB, and fBC, bBC embed OB into OC, then there exist fAC, bAC
;;; embedding OA into OC such that (rC bAC bBC).

(def-dtuple (L3-conseq ((A *)                   ; type of consequence of lemma3
                        (dA (Pred A))
                        (rA (Rel A))
                        (C *)
                        (dC (Pred C))
                        (rC (Rel C))
                        (bBC C)))
  ((f (A -> C))
   (b C)
   (m (Embed A dA rA C dC rC f b))
   (rb (rC b bBC))))


(def lemma3
  (\ ((A *)
      (dA (Pred A))
      (rA (Rel A))
      (B *)
      (dB (Pred B))
      (rB (Rel B))
      (C *)
      (dC (Pred C))
      (rC (Rel C))
      (oAB (Embed-ord A dA rA B dB rB))
      (fBC (B -> C))
      (bBC C)
      (mBC (Embed B dB rB C dC rC fBC bBC)))

     (mlet ((RT (L3-conseq A dA rA C dC rC bBC))) ; result type
       (@Embed-ord                              ; open oAB
        oAB
        A dA rA B dB rB RT
        (@Embed                                 ; open oAB.m
         oAB.m
         A dA rA B dB rB oAB.f oAB.b RT
         (@Embed                                ; open mBC
          mBC
          B dB rB C dC rC fBC bBC RT
          (mlet ((f (\x A (fBC (oAB.f x))))     ; function mapping A -> C
                 (b (fBC oAB.b)))               ; upper bound on its range
            (^L3-conseq A dA rA C dC rC bBC     ; construct consequence
                        f
                        b
                        ;; embedding of A into C
                        (^Embed A dA rA C dC rC
                                f b
                                ;; b in domain of C
                                (mBC.pres-dom oAB.b oAB.m.db)
                                ;; f preserves domain
                                (\ ((x A)
                                    (dx (dA x)))
                                   (mBC.pres-dom (oAB.f x)
                                                 (oAB.m.pres-dom x dx)))
                                ;; f is monotonic
                                 (\ ((x A)
                                     (y A)
                                     (dx (dA x))
                                     (dy (dA y))
                                     (rxy (rA x y)))
                                    (mBC.mono (oAB.f x)
                                              (oAB.f y)
                                              (oAB.m.pres-dom x dx)
                                              (oAB.m.pres-dom y dy)
                                              (oAB.m.mono x y dx dy rxy)))
                                 ;; ran f dominated by b
                                 (\ ((x A)
                                     (dx (dA x)))
                                    (mBC.mono (oAB.f x)
                                              oAB.b
                                              (oAB.m.pres-dom x dx)
                                              oAB.m.db
                                              (oAB.m.dominate x dx))))
                         ;; proof that (rC b bBC)
                         (mBC.dominate oAB.b oAB.m.db))))))))


;;; Corollary 4.  Embed-ord is transitive.

(def cor4
  (\ ((A *)
      (dA (Pred A))
      (rA (Rel A))
      (B *)
      (dB (Pred B))
      (rB (Rel B))
      (C *)
      (dC (Pred C))
      (rC (Rel C))
      (oAB (Embed-ord A dA rA B dB rB))
      (oBC (Embed-ord B dB rB C dC rC)))

     (mlet ((RT (Embed-ord A dA rA C dC rC)))   ; result type
       (@Embed-ord oBC
                   B dB rB C dC rC RT
                   (let ((l (L3-conseq A dA rA C dC rC oBC.b)
                            (lemma3 A dA rA B dB rB C dC rC
                                    oAB oBC.f oBC.b oBC.m)))
                     (@L3-conseq l
                                 A dA rA C dC rC oBC.b RT
                                 (^Embed-ord A dA rA C dC rC
                                             l.f
                                             l.b
                                             l.m)))))))
```

```
;;; Lemma 5.  (Trans OU).

(def lemma5
  (\ ((u1 U)
      (u2 U)
      (u3 U)
      (r (rU u1 u2))
      (s (rU u2 u3)))

     (mlet ((RT (rU u1 u3)))                    ; result type
       (@rU                                     ; open r
        r
        u1 u2 RT
        (@rU                                    ; open s
         s
         u2 u3 RT
         (^rU u1 u3                             ; construct consequence
              r.A r.dA r.rA
              s.B s.dB s.rB
              ;; proof that u1 = (inj r.OA)
              r.i
              ;; proof that u3 = (inj s.OB)
              s.j
              ;; (Embed-ord r.OA s.OB)
              (cor4 r.A r.dA r.rA               ; by transitivity of Embed-ord
                    s.A s.dA s.rA
                    s.B s.dB s.rB
                    ;; r.OA < s.OA follows from r.OB = s.OA and r.o : r.OA < r.OB
                    (lemma1 r.B r.dB r.rB       ; connects r.OB to s.OA
                            s.A s.dA s.rA
                            ;; proof that (inj r.OB) = (inj s.OA)
                            (eq-trans U (inj r.B r.dB r.rB) u2 (inj s.A s.dA s.rA)
                                      ;; proof that (inj r.OB) = u2
                                      (eq-sym U u2 (inj r.B r.dB r.rB) r.j)
                                      s.i)      ; proof that u2 = (inj s.OA)
                            ;; OS-pred true of r.OB; lemma1 will prove it for s.OA
                            (\ ((X *)
                                (dX (Pred X))
                                (rX (Rel X)))
                               (Embed-ord r.A r.dA r.rA
                                          X dX rX))
                            ;; proof that above OS-pred is true of r.OB
                            r.o)
                    ;; proof of s.OA < s.OB
                    s.o)))))))
```

```
;;; Lemma 6.  (WF U dU rU).

(def-dtuple (D-chain ((C (Pred U))              ; chain constructed in an ordered set OA
                      (A *)
                      (dA (Pred A))
                      (rA (Rel A))
                      (a A)))                   ; element of D chain to consider
  ((B *)                                        ; preceding ordered set OB
   (dB (Pred B))
   (rB (Rel B))
   (f (B -> A))                                 ; embedding function from B to A
   (ci (C (inj B dB rB)))                       ; proof that OB is in C chain
   (m (Embed B dB rB A dA rA f a))))            ; proof that f embeds OB into OA,
                                                ; bounded by a


(def lemma6
  (\ ((C (Pred U))                              ; chain in U
      (c (Chain U dU rU C))                     ; proof that C is a chain
      (T *)                                     ; ** looping type **
      (fT (T -> T)))                            ; ** looping function **

     (@Chain                                    ; open c
      c
      U dU rU C
      T                                         ; result type

      (mlet ((uZ c.z)                           ; uZ is base of chain C
             (duZ c.dz)                         ; uZ in dU
             (cuZ c.cz))                        ; uZ in chain
        (let ((cp (Predec U dU rU C c.z c.cz)   ; predecessor of uZ exists and is in C
                  (c.pr c.z c.cz)))
          (@Predec                              ; open predecessor proof
           cp
           U dU rU C c.z c.cz T
           (@rU
            cp.ryx
            cp.y c.z T
            (mlet ((Z cp.ryx.B) (dZ cp.ryx.dB) (rZ cp.ryx.rB) ; the actual OZ
                   (W cp.ryx.A) (dW cp.ryx.dA) (rW cp.ryx.rA)) ; OW is used in basis

              ;; prove that OZ is WF, then apply this to a chain constructed within OZ
              (fT                               ; ** one loop iteration **
               (lemma2b                         ; this will prove (WF OZ)
                Z dZ rZ
                (cp.ryx.j dU duZ)               ; proof that (inj OZ) is in dU
                (D-chain C Z dZ rZ)             ; chain in OZ
```

```
;; proof that (D-chain C Z dZ rZ) is a chain
 (@Embed-ord
  cp.ryx.o
  W dW rW Z dZ rZ
  (Chain Z dZ rZ (D-chain C Z dZ rZ))
  (@Embed
   cp.ryx.o.m
   W dW rW Z dZ rZ cp.ryx.o.f cp.ryx.o.b
   (Chain Z dZ rZ (D-chain C Z dZ rZ))
   (^Chain
    Z dZ rZ (D-chain C Z dZ rZ)

    ;; base of chain in OZ is embedding bound from proof of OW < OZ
    cp.ryx.o.b

    ;; proof that base is in D-chain
    (^D-chain C Z dZ rZ cp.ryx.o.b
              W dW rW             ; preceding ordered set W
              cp.ryx.o.f          ; embedding from W to Z
              (cp.ryx.i C cp.cy) ; W is in C
              cp.ryx.o.m)         ; f embeds W into Z

    ;; proof that base satisfies dZ (comes from proof of OW < OZ)
    cp.ryx.o.m.db
   ;; proof that predecessors exist
    (\ ((a Z)                   ; for every a in OZ
        (da (D-chain C Z dZ rZ a))) ; that is in chain D...
       (mlet ((RT (Predec Z dZ rZ (D-chain C Z dZ rZ) a da)))
         (@D-chain                ; open da
          da
          C Z dZ rZ a RT
          (mlet ((Y da.B) (dY da.dB) (rY da.rB))
            ;; OY < OZ and (C (inj OY)) by da; since OY is in the
            ;; chain C, we can find its predecessor by applying c.pr
            (let ((dp (Predec U dU rU C (inj Y dY rY) da.ci)
                      (c.pr (inj Y dY rY) da.ci)))
              (@Predec            ; open dp
               dp
               U dU rU C (inj Y dY rY) da.ci RT
               (@rU               ; open dp.ryx
                dp.ryx
                dp.y (inj Y dY rY) RT
                (mlet ((X dp.ryx.A) (dX dp.ryx.dA) (rX dp.ryx.rA)
                       (Y~ dp.ryx.B) (dY~ dp.ryx.dB) (rY~ dp.ryx.rB))
                  ;; OX < OY~ and (C (inj OX)),
                  ;; so apply lemma3 to get OX < OZ
                  (let ((l (L3-conseq X dX rX Z dZ rZ a)
                           (lemma3 X dX rX Y dY rY Z dZ rZ
                                   ;; proof that OX < OY from OX < OY~
                                   (lemma1 Y~ dY~ rY~ Y dY rY
                                           (eq-sym U (inj Y dY rY)
                                                   (inj Y~ dY~ rY~)
                                                   dp.ryx.j)
                                           (\ ((A *)
                                               (dA (Pred A))
                                               (rA (Rel A)))
                                             (Embed-ord X dX rX A dA rA))
                                           dp.ryx.o)
                                   ;; embedding from OY to OZ
                                   da.f a da.m)))
                    (@L3-conseq  ; open lemma3 result
                     l
                     X dX rX Z dZ rZ a RT
                     ;; proof that l.b is predecessor of a in D chain
                     (^Predec
                      Z dZ rZ (D-chain C Z dZ rZ) a da
                      l.b
                      ;; proof that l.b is in D
                      (^D-chain C Z dZ rZ l.b
                                X dX rX       ; preceding set is OX
                                l.f           ; embedding function is f
                                (dp.ryx.i C dp.cy) ; OX is in C
                                l.m)          ; f, l.b embed OX into OZ
                      ;; proof that l.b precedes a in OZ
                      l.rb)
                     ))))))))))))
 T fT)                          ; ** looping arguments to lemma2b **
)))))))))
```

```
;;; Initial segments of ordered sets.

(def-dtuple (Seg ((A *)                          ; segment of OA
                  (dA (Pred A))
                  (rA (Rel A))
                  (a A)                          ; determined by a
                  (x A)))
  ((d (dA x))                                    ; x is in OA
   (r (rA x a))))                                ; x less than a


;;; Lemma 7.  If (dU OA) and (dA a) for some a : A, then (dU (inj OAa)).

(def lemma7
  (\ ((A *)
      (dA (Pred A))
      (rA (Rel A))
      (a A)
      (di (dU (inj A dA rA)))
      (da (dA a)))
     (mlet ((iA (inj A dA rA))                   ; injection of OA into U
            (sA (Seg A dA rA a)))                ; segment of OA we're interested in
      (@dU di iA                                 ; open di
            (dU (inj A sA rA))                   ; result type
            (let ((t (Trans A dA rA)             ; given proof that OA is transitive
                     (lemma2a A dA rA di))
                  (w (WF A dA rA)                ; given proof that OA is well-founded
                     (lemma2b A dA rA di)))
              ;; prove that (inj OAa) satisfies dU
              (^dU (inj A sA rA)
                   A sA rA
                   ;; proof that (inj OAa) = (inj OAa)
                   (eq-ref U (inj A sA rA))
                   ;; OAa is transitive
                   t
                   ;; OAa is WF since a chain in the segment is a chain in the set
                   (\ ((C (Pred A))
                       (c (Chain A sA rA C)))
                      (@Chain c A sA rA C        ; open c
                              Loop               ; result type
                              (@Seg c.dz A dA rA a c.z ; open c.dz
                                    Loop         ; result type
                                    (w C
                                       (^Chain A dA rA C
                                               c.z
                                               c.cz
                                               c.dz.d
                                               c.pr))))))))))
```

```
;;; Lemma 8.  If (dU (inj OA)), then OA < OU.

(def lemma8
  (\ ((A *)
      (dA (Pred A))
      (rA (Rel A))
      (di (dU (inj A dA rA))))
     (mlet ((f (\x A (inj A (Seg A dA rA x) rA))) ; mapping A -> U
            (b (inj A dA rA)))                    ; bound on mapping
       ;; prove that f embeds OA into U, bounded by b
       (^Embed-ord A dA rA U dU rU f b
        (^Embed A dA rA U dU rU f b

                ;; embedding bound satisfies domain
                di

                ;; f preserves domain
                (\ ((x A)
                    (dx (dA x)))
                   (lemma7 A dA rA x di dx))

                ;; f is monotonic
                (\ ((x A)
                    (y A)
                    (dx (dA x))
                    (dy (dA y))
                    (rxy (rA x y)))
                   (mlet ((sx (Seg A dA rA x))  ; segment determined by x
                          (sy (Seg A dA rA y))) ; segment determined by y
                     (^rU (f x) (f y)
                          A sx rA
                          A sy rA
                          (eq-ref U (inj A sx rA))
                          (eq-ref U (inj A sy rA))
                          (mlet ((fA (\x A x))) ; mapping A -> A
                            ;; prove that fA embeds (f x) into (f y), bounded by x
                            (^Embed-ord A sx rA
                                        A sy rA
                                        fA x
                                        (^Embed A sx rA
                                                A sy rA
                                                fA x
                                                ;; bound x is in domain of (f y)
                                                (^Seg A dA rA y x dx rxy)
                                                ;; fA preserves domain
                                                (\ ((a A)
                                                    (da (sx a)))
                                                   (@Seg da
                                                         A dA rA x a
                                                         (Seg A dA rA y a)
                                                         (^Seg A dA rA y a
                                                               da.d
                                                               ((lemma2a A dA rA di)
                                                                a x y
                                                                da.r rxy))))
```

```
                                ;; fA is monotonic
                                (\ ((a A)
                                    (b A)
                                    (da (sx a))
                                    (db (sx b))
                                    (rab (rA a b)))
                                   rab)
                                ;; x dominates ran fA
                                (\ ((a A)
                                    (da (sx a)))
                                   (@Seg da
                                         A dA rA x a
                                         (rA a x)
                                         da.r)))))))

;; b dominates ran f
(\ ((x A)
    (dx (dA x)))
   (mlet ((sx (Seg A dA rA x))) ; segment determined by x
     (^rU (f x) b
          A sx rA
          A dA rA
          (eq-ref U (inj A sx rA))
          (eq-ref U (inj A dA rA))
          (mlet ((fA (\x A x))) ; mapping A -> A
            ;; prove that fA embeds (f x) into OA, bounded by x
            (^Embed-ord A sx rA
                        A dA rA
                        fA x
                        (^Embed A sx rA
                                A dA rA
                                fA x
                                ;; bound is in domain
                                dx
                                ;; fA preserves domain
                                (\ ((a A)
                                    (da (sx a)))
                                   (@Seg da
                                         A dA rA x a
                                         (dA a)
                                         da.d))
                                ;; fA preserves order
                                (\ ((a A)
                                    (b A)
                                    (da (sx a))
                                    (db (sx b))
                                    (rab (rA a b)))
                                   rab)
                                ;; x dominates ran fA
                                (\ ((a A)
                                    (da (sx a)))
                                   (@Seg da
                                         A dA rA x a
                                         (rA a x)
                                         da.r)))))))) )))))
```

```
;;; Contradiction.

(def u (inj U dU rU))                           ; injection of PU into itself


(def du                                         ; (dU u)
  (^dU u U dU rU
       (eq-ref U u)
       lemma5
       lemma6))


(def ru                                         ; (rU u u)
  (^rU u u
       U dU rU
       U dU rU
       (eq-ref U u)
       (eq-ref U u)
       (lemma8 U dU rU du)))


(def C (\v U (Eq U v u)))                       ; chain in U


(def cC (^Chain U dU rU C                       ; proof that C is a chain
                 u                              ; base of chain
                 (eq-ref U u)                   ; base is in C,
                 du                             ;      and in dU
                 (\ ((v U)                      ; predecessor of any v
                     (cv (C v)))                ;   that is in C
                    (^Predec U dU rU C v cv
                             u                  ;   is u,
                             (eq-ref U u)       ;   which is in C also,
                             ((eq-sym U v u cv) ;   and is smaller than u
                              (\w U (rU u w)) ru)
                             ))))

(defx loop (lemma6 C cC))                       ; Q.E.D.
```

References

[Ada 1980] *Reference Manual for the Ada Programming Language.* 1980. G.P.O. 008-000-00354-8.

[Aiello & Prini 1981] Luigia Aiello and Gianfranco Prini. An efficient interpreter for the lambda-calculus. *Journal of Computer and System Sciences*, 23(3):383–424, December 1981.

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

[Barendregt 1984] 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.

[Boehm *et al.* 1980] Hans J. Boehm, Alan Demers, and James Donahue. *An Informal Description of Russell.* Technical Report 80-430, Department of Computer Science, Cornell University, Ithaca, New York, October 1980.

[Böhm & Berarducci 1985] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39(2/3):135–154, August 1985.

[Breazu-Tannen & Meyer 1987] Valeriu Breazu-Tannen and Albert R. Meyer. Computable values can be classical (preliminary report). In *Principles of Programming Languages*, ACM, January 1987.

[Breazu-Tannen 1986] Valeriu Breazu-Tannen. Personal communication, February 1986. Given two λ-calculi λ_1 and λ_2 such that λ_2 is Church-Rosser, λ_2-reductions preserve λ_1-terms, and the new reductions of λ_2 are not applicable to λ_1-terms, it is not hard to show that λ_2 is a conservative extension of λ_1.

[Breazu-Tannen 1987] Valeriu Breazu-Tannen. *Conservative Extensions of Type Theories.* Ph.D. thesis, Massachusetts Institute of Technology, February 1987.

[Burali-Forti 1897] Cesare Burali-Forti. Une questione sui numeri transfiniti. *Rendiconti del Circolo Matematico di Palermo*, 11:154–164, 1897.

[Burstall & Lampson 1984] R. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, pages 1–50, Volume 173 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, June 1984.

[Cardelli 1986] Luca Cardelli. *A Polymorphic λ-calculus with Type:Type.* Technical Report 10, Digital Equipment Corporation Systems Research Center, Palo Alto, California, May 1986.

[Constable 1986] Robert L. Constable. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Coquand & Huet 1988] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Control*, 76(2/3):95–120, February/March 1988.

[Coquand 1985a] Thierry Coquand. Personal communication, December 1985.

[Coquand 1985b] Thierry Coquand. *Une Théorie des Constructions.* Ph.D. thesis, Université de Paris, January 1985. Thèse de troisième cycle.

[Coquand 1986] Thierry Coquand. An analysis of Girard's paradox. In *Logic in Computer Science*, pages 227–236, IEEE, June 1986.

[Coquand 1987] Thierry Coquand. Personal communication, July 1987.

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

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

[Fortune *et al.* 1983] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, January 1983.

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

[Gordon *et al.* 1979] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1979.

[Harper & Mitchell 1986] Robert Harper and John C. Mitchell. Message in the TYPES electronic forum (types@theory.lcs.mit.edu), April 1986.

[Hindley & Seldin 1986] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Volume 1 of *London Mathematical Society Student Texts*, Cambridge University Press, 1986.

[Hook & Howe 1986] James G. Hook and Douglas J. Howe. *Impredicative Strong Existential Equivalent to Type:Type.* Technical Report 86-760, Department of Computer Science, Cornell University, Ithaca, New York, June 1986.

[Hopcroft & Ullman 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Howard 1969] W. A. Howard. The formulae-as-types notion of construction. 1969. Recently published as [Howard 1980].

[Howard 1980] 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.

[Howe 1987] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Logic in Computer Science*, pages 205–214, IEEE, June 1987.

[Huet 1986] Gérard Huet. Formal structures for computation and deduction. May 1986. First edition of an unpublished manuscript.

[Kernighan & Ritchie 1978] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kleene 1950] Stephen C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, New Jersey, 1950.

[Kleene 1979] Stephen C. Kleene. Origins of recursive function theory. In *Foundations of Computer Science*, pages 371–382, IEEE, 1979.

[Liskov *et al.* 1981] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981.

[MacQueen 1986] David B. MacQueen. Using dependent types to express modular structure. In *Principles of Programming Languages*, pages 277–286, ACM, January 1986.

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

[Martin-Löf 1972] Per Martin-Löf. *An Intuitionistic Theory of Types.* Report, Department of Mathematics, University of Stockholm, 1972.

[Meyer & Reinhold 1986] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type: preliminary report. In *Principles of Programming Languages*, pages 287–295, ACM, January 1986.

[Milner 1983] Robin Milner. A proposal for standard ML. *Polymorphism*, 1(3), December 1983.

[Mitchell & Harper 1988] John C. Mitchell and Robert Harper. The essence of ML. In *Principles of Programming Languages*, pages 28–46, ACM, January 1988.

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

[Mohring 1986] Christine Mohring. Algorithm development in the calculus of constructions. In *Logic in Computer Science*, pages 84–91, IEEE, June 1986.

[Paulson 1984] Lawrence Paulson. Deriving structural induction in LCF. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, pages 197–214, Volume 173 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, June 1984.

[Reinhold 1986] Mark B. Reinhold. Looping with Girard's paradox. June 1986. Unpublished manuscript.

[Reynolds 1974] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Volume 19 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1974.

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

[Reynolds 1985] John C. Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development*, Volume 185 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

[Statman 1981] Rick Statman. Number theoretic functions computable by polymorphic programs. In *Foundations of Computer Science*, pages 279–282, IEEE, 1981.

[Steele 1984] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, 1984.

[Stenlund 1972] Sören Stenlund. *Combinators, λ-Terms, and Proof Theory.* D. Reidel, Dordrecht, Holland, 1972.

[Strachey 1967] Christopher C. Strachey. *Fundamental Concepts in Programming Languages.* Lecture notes, International Summer School in Computer Programming, Copenhagen, August 1967.

[Wadsworth 1971] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* Ph.D. thesis, Oxford University, 1971.