Typability and Type Checking in System F Are Equivalent and Undecidable*

J. B. Wells†
jbw@cee.hw.ac.uk
http://www.cee.hw.ac.uk/~jbw
Heriot-Watt University, Dept. of Computing & Electrical Engineering
EDINBURGH, EH14 4AS, Scotland

October 26, 1998

Abstract

Girard and Reynolds independently invented System F (a.k.a. the second-order polymorphically typed lambda calculus) to handle problems in logic and computer programming language design, respectively. Viewing F in the *Curry style*, which associates types with untyped lambda terms, raises the questions of *typability* and *type checking*. Typability asks for a term whether there exists some type it can be given. Type checking asks, for a particular term and type, whether the term can be given that type. The decidability of these problems has been settled for restrictions and extensions of F and related systems and complexity lower-bounds have been determined for typability in F, but this report is the first to resolve whether these problems are decidable for System F.

This report proves that type checking in F is undecidable, by a reduction from *semi-unification*, and that typability in F is undecidable, by a reduction from type checking. Because there is an easy reduction from typability to type checking, the two problems are equivalent. The reduction from type checking to typability uses a novel method of constructing lambda terms that simulate arbitrarily chosen type environments. All of the results also hold for the λI-calculus.

1 Background and Summary

This report is divided into the non-technical and the technical. This section is the non-technical portion. All other sections are technical.

1.1 What are the problems?

Over the years, numerous type systems have been devised for Church's λ-calculus, generally as extensions of the simply typed λ-calculus. One particularly important type system, the "second-order polymorphically typed λ-calculus", was independently invented by Girard and Reynolds over twenty years ago [Gir72, Rey74]. Girard developed his system (named by chance "System F") to prove properties of second-order logic (hence F is sometimes called "the second-order λ-calculus") while Reynolds wanted to express polymorphic typing in programming explicitly. Both Girard and Reynolds formulated F in the *Church style* where types are embedded in terms which are no longer the terms of the pure λ-calculus. In the *Curry style*, where type information is kept distinct from the terms, it is meaningful to ask for an arbitrarily chosen λ-term M:

*Earlier versions of this material are [Wel94, Wel96b].

†This work is partly supported by NSF grants CCR–9113196 and CCR–9417382 and EPSRC grant GR/L 36963.

1. Is there any typing for M, i.e., do there exist assumptions A and type τ such that $A \vdash M : \tau$ is derivable?

2. Can M be given some particular type τ using some particular type assumptions A, i.e., for arbitrarily chosen A and τ, is $A \vdash M : \tau$ derivable?

We call the first problem TYP for *typability*[1] and the second problem TC for *type checking*[2]. The problem of *type inference* is the problem of actually finding all of the types and type assumptions that allow typability or type checking. Unless another type system is indicated, the names TYP and TC refer to these problems in System F.

1.2 What is the prior research?

Much research has been devoted to determining whether TYP and TC are decidable for F and related systems. Some of this research has focused directly on System F. Leivant first presented F in the Curry style in 1983 and made the first attempt to answer the question of decidability for TYP [Lei83]. The first interesting lower-bound for the computational complexity of TYP was given by Henglein [HM91, HM94] who showed that TYP is DEXPTIME-hard (where DEXPTIME means DTIME($2^{n^{O(1)}}$)). This was done by embedding Turing machine computations in the types of a λ-term.

Other research has considered TYP and TC for various restrictions and extensions of System F and related problems for F itself. Multiple stratifications of F have been proposed which restrict some parameter of derivations in F to finite values, e.g., depth of bound type variable from binding quantifier [GRDR91], the number of generations of instantiation of quantifiers themselves introduced by instantiation [Lei91], and the "rank" of polymorphic types (introduced in [Lei83], further studied in [McC84, KT92, KW94]). Urzyczyn recently showed TYP to be undecidable for F's powerful extension, the system F_ω [Urz93b, Urz97], which allows types to contain functions from types to types. The proof, which reduces the halting problem for Turing machines to TYP for F_ω, actually establishes the recursive inseparability of the F_ω-typable λ-terms and the λ-terms with β-normal forms. For F, proofs of undecidability have been given for both the problem of *partial polymorphic type inference* in the Church style [Boe85, Pfe93], which is related to TYP, and the problem of *conditional type inference* [GHRDR93], which is related to TC.

Only a fraction of the research over the years is mentioned here. Despite this intensive research program, until now the decidability of TYP and TC have remained "embarrassing open problems"[3] for System F itself. Both of these problems are proven undecidable in this article.

1.3 What are the implications?

Theoretical considerations have been part of the motivation for determining the decidability of TYP and TC. Of all of the type systems for the λ-calculus that result from extending the simply typed λ-calculus with some combination of universal, recursive, and intersection types and the β-equivalence and approximation rules, it is only for F that the decidability of TYP and TC has remained unknown until now, as depicted by the table in figure 1, which first appeared in [Bar92].[4]

[1]Typability has been called *type reconstruction* or *type inference*.

[2]Type checking has been called *derivation reconstruction*. In the Church style, "type checking" usually means checking that a derivation is valid, not finding a derivation.

[3]Robin Milner, winner of the 1991 Turing Award, quoted by Henk Barendregt [Bar92, p. 183].

[4]In figure 1, INH stands for the *inhabitation* problem, which asks for a type σ where there exist A and M such that $A \vdash M : \sigma$ is derivable. The β-equivalence rule extends a type system so that if $M =_\beta N$, any typing for M is also a typing for N. See [Bar92] for an explanation of the approximation rule ($\mathcal{A}$).

Constructors/Rules	TC	Typ	Inh
$\to$	yes	yes	yes
$(=_\beta)$	no	no	yes
$(\mathcal{A})$	no	no	yes*
$\to$, $\forall$	NO	NO	no
$(=_\beta)$	no	no	no
$(\mathcal{A})$	no	no	yes*
$\to$, μ	yes	yes*	yes*
$(=_\beta)$	no	yes*	yes*
$(\mathcal{A})$	no	yes*	yes*
$\to$, $\cap$, ω	no	yes*	no†
$(=_\beta)$	—	—	—
$(\mathcal{A})$	no	yes*	yes*

TC:	type checking problem
Typ:	typability problem
Inh:	type inhabitation problem
$(=_\beta)$:	plus β-equivalence rule
$(\mathcal{A})$:	plus approximation rule
*:	trivially decidable
—:	same as without β-rule
†:	proved since 1991
NO:	proved in this article

Figure 1: Decidability of Various Type System Problems.

Also, among the eight type systems in Barendregt's λ-cube [Bar92], when they are considered in Curry style, there are only three distinct sets of typable, pure, closed λ-terms corresponding to the simply typed λ-calculus, F, and F_ω [GHRDR93]. This is depicted in figure 2. Because it has been known that Typ is decidable for the simply typed λ-calculus [Hin69] and (recently) undecidable for F_ω [Urz93b, Urz97], it is once again only for F that the answer has been unknown. Thus, determining the decidability of Typ and TC for F completes our knowledge of these problems for a wide variety of λ-calculus type systems.

While the question of decidability of Typ has theoretical interest, a prime motivation for solving this problem has been its practical implications. The design of functional computer programming languages (which are based on the λ-calculus) makes the ability to perform type inference in System F (or even better, some extension of F) highly desirable. The primary reason for types in a functional programming languages is the usual desire to rule out nonsensical computations. Types can prevent undesirable run-time operations from occurring, e.g., types can rule out an application like "**+ true true**". Such operations are either unsafe or require run-time coercions. Many functional programming languages use type systems which are related to System F. The designers of many strongly typed functional programming languages (e.g., ML [MTH90], Miranda [Tur85], Haskell [HW88]) have chosen to use the Hindley/Milner type system whose core coincides with a fragment of F. There are also various programming languages using type systems based on variations of F_ω (e.g., Quest [Car91] and LEAP [PL89]).

The choice to use a type system similar to F or some portion of F in real-world programming languages has many reasons. The minimal requirement is that the type system rules out nonsense computations. Once that is achieved, there are other considerations. From a software engineering viewpoint, programmers wish to reuse program fragments rather than duplicating them. The choice of a type system determines how easily programmers can reuse code. In the worst case, the simply typed λ-calculus requires that even a simple function such as the identity (represented by the λ-term $(\lambda x.x)$) must be duplicated for each type on which it is used. A compiler would enforce this redundancy in a programming language based on this type system. To avoid this, the chosen type system generally supports some kind of *polymorphism*, which allows giving a function a polymorphic ("generic") type which represents a finite or infinite collection of types. System F exhibits a kind of polymorphism called *parametric polymorphism*, meaning

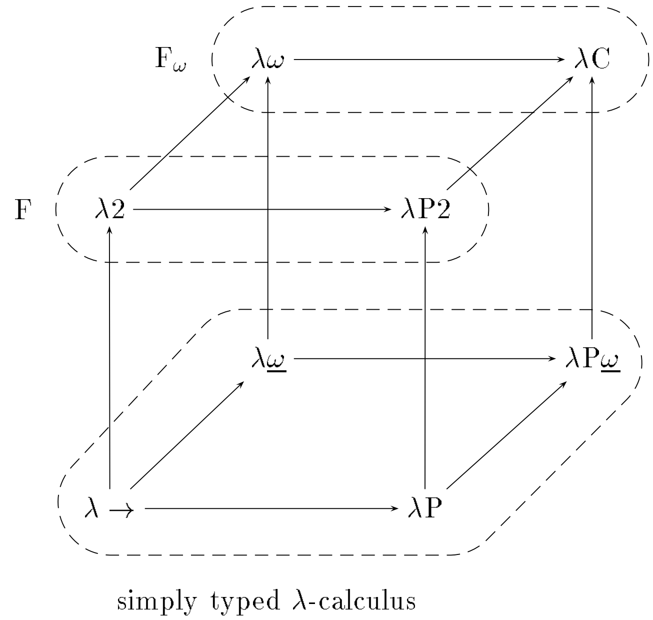


Figure 2: Curry/Church-Style Type System Correspondence.

that a λ-term operates in the same way over a range of types, the particular type used in any instance being determined by a type parameter. The Hindley/Milner type system supports a weak restriction of parametric polymorphism which allows a polymorphic function to be passed as a parameter, but only to a single predetermined non-polymorphic function. Also, a polymorphic function can have only monomorphic arguments. In the full System F, a polymorphic function can be passed as a parameter to another non-predetermined polymorphic function, which itself can be passed as a parameter to other such functions, and so on. This flexibility is reflected in the fact that System F types more λ-terms than does the Hindley/Milner type system. In fact, every computable function that is provably total using second-order Peano arithmetic is representable by a λ-term typable in System F [GLT89]. Another reason for using System F (or some related system) is that all computations on typable λ-terms are guaranteed to halt. In practice, general-purpose programming languages add features that allow non-termination, but this property can still be helpful to the compiler for handling subexpressions which do not use the non-terminating features. Some special-purpose programming languages do take advantage of this property of System F.

A very important criterion for choosing a type system for a functional programming language is the desire to have all typing done automatically by the compiler rather than by the programmer. Because function application is the central construct of a functional programming language, strict typing involves assigning a type not only to every identifier but also to every fragment of the program. Because it is also the nature of functional types that they grow to be very large, requiring the programmer to specify more than a tiny fraction of the types in a λ-term is very unwieldy and user-unfriendly. These reasons contribute to the desire for type inference, a procedure that will provide a typing for a λ-term if it is typable and will otherwise halt and return an error message. As part of ensuring that programs will be portable between different versions of the language's compiler, the type inference algorithm used by the compiler should always find a type (ideally, a most-general type) for a program fragment if that is possible. However, the type inference algorithm should also be guaranteed to halt if a program fragment is untypable. Otherwise, the compiler will have to choose to stop arbitrarily at some point (or will be halted by the impatient user), which will impede program portability among different compiler versions. Thus, unless Typ is decidable for a type system, both automatic type inference and portability of programs can not be achieved. System F has many of the nice properties desired in a type system for a functional programming language, but would be problematic to use in a language with automatic type inference unless Typ were decidable. Unfortunately, this article proves that Typ is undecidable for System F.

1.4 What does this article contribute?

The main contribution of this report is proving the undecidability of Typ and TC for System F.

1. We first prove that the problem of *semi-unification* can be reduced to TC using a simple encoding. Because semi-unification is undecidable [KTU93], so is TC.

2. We then reduce TC to Typ using a novel method of building λ-terms which simulate arbitrarily chosen *type environments*. The proof begins by showing that there exists a typable λ-term J such that in every typing of J, its bound variable x is assigned the type $\alpha \to \alpha$:

$$J \equiv \lambda v.(\lambda y.\lambda z.v(yy)(yz))(\lambda x.\mathsf{K}x(x(xv)))(\lambda w.ww)$$

   Then, building upon the term J, contexts (terms with holes) are constructed which simulate more and more complex type environments. These λ-terms force particular bound variables to be assigned particular types in every type derivation. Using this method, we can require that subterms in certain positions within a λ-term must be typable using a specific arbitrarily chosen type environment in order for the entire term to be typable at all. Any desired type environment may be simulated, thus allowing any instance of TC to be simulated by an instance of Typ. This establishes the undecidability of Typ.

All of the results work not only for the λK-calculus but within the λI-calculus as well (where every bound variable occurs at least once in the scope of its binding). Because the reduction from Typ to TC is already known [Bar92], and because both reductions work within the λI-calculus, we conclude for System F that Typ and TC are equivalent and that both are undecidable for both the λK and λI-calculi. By showing both problems to be undecidable, as mentioned earlier, the question of decidability of Typ and TC for a variety of type systems is now completely answered. As a result, the table in figure 1 has now been filled in completely (with the addition of the result that type inhabitation for intersection types is undecidable [Urz93a, Urz94]).

The most important practical implication of the results presented here is that no functional programming language will use F as its type system with fully automatic type inference. This will focus future research on restrictions of F and other entirely different approaches. These results have other implications of both practical and theoretical interest.

1. The methods of this article can be used to show the undecidability of TC and Typ for restrictions and extensions of System F. The symbol Λ_k is our name for the type system that results from restricting System F to types of finite "rank" k, a notion introduced by Leivant [Lei83]. Our methods have been used to show the undecidability of Typ and TC for Λ_k where $k \geq 3$ [KW94]. These methods have also been extended to show the undecidability of Typ for System F+η, the result of extending F with a rule for subject η-reduction (equivalent to extending F with Mitchell's subtyping) [Wel96a, Wel96b].

2. The methods of this article can also be used to show the differences in the sets of terms typable by various restrictions and extensions of F. They can easily exhibit λ-terms that are typable in F but not in some subsystem of F or that are not typable in F but are

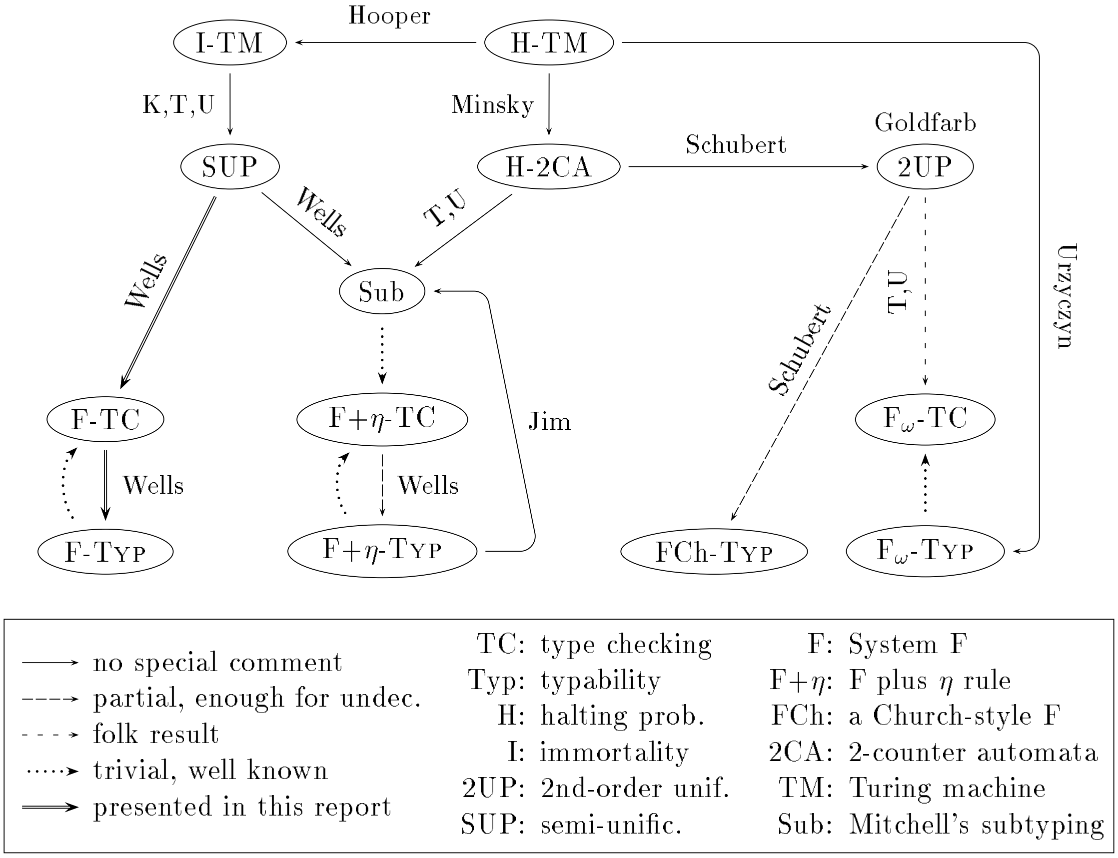


Figure 3: Undecidability Reductions Related to System F.

typable in some extension of F. For example, we can show that Λ_{k+1} types more terms than Λ_k for every k, because for any arbitrarily chosen rank, a λ-term can be constructed whose typing must contain a type of that rank. Similarly, because we can force typings to contain types of arbitrarily chosen height, every level of the stratification of System F in [GRDR91] types a larger set of terms than the previous level. As a final example, the following slight modification of the λ-term J is strongly β-normalizing but not typable in System F (the first such term was given in [GHRDR87]):

$$\lambda v.(\lambda y.\lambda z.v(yy)(yz))(\lambda x.\mathsf{K}x(x(xv)v))(\lambda w.ww)$$

Figure 3 diagrams the relationships between various undecidable problems both for F as well as for the related systems F_ω and F+η, with each arrow indicating a problem reduction. Dotted arrows are trivial reductions, the light dashed arrow is a folk result reported by the indicated authors, and double lines indicate results presented in this report. The heavy dashed line indicates that only part of the problem is reduced, enough to show undecidability. In the diagram, the problems H-TM, I-TM, H-2CA, 2UP, SUP, and Sub are respectively, the halting problem, the immortality problem for Turing machines [Hoo66], the halting problem for two-counter automata [Min61], the second-order unification problem [Gol81], the semi-unification problem, and Mitchell's subtyping relation [Mit88, TU96, Wel95, Jim95]. In the diagram, the system FCh is the Church-style presentation of System F where *only binding occurrences of variables are annotated with types*. Schubert recently discovered the surprising result that typability in that system is undecidable [Sch98]. Note that the reduction from H-2CA to Sub [TU96] does not prove undecidable a fragment of Sub that matches with the partial reduction from F+η-TC to

F+η-Typ [Wel96a, Wel96b], so it does not contribute to proving F+η-Typ undecidable. Similarly, Goldfarb's original proof of undecidability for second-order unification does not contribute to the proving typability undecidable for Church-style System F.

1.5 Who helped this happen?

This author did this work while a Ph.D. student at Boston University, where he was supported as a research assistant by NSF grants CCR–9113196 and CCR–9417382. A final revision was done while the author was a researcher at the University of Glasgow and at Heriot-Watt University, supported by EPSRC grant GR/L 36963. During this time, the author's advisor Assaf Kfoury provided vital support and encouragement. After the results were achieved, Jerzy Tiuryn made valuable suggestions for notation and organization of the proofs and Paweł Urzyczyn proofread several drafts and pointed out numerous improvements. The anonymous referees also made many helpful suggestions.

2 Basic Definitions and Notation

This section presents basic definitions, notation, and nomenclature for standard concepts. We define the untyped λ-calculus and System F in the Curry style. We state precisely the problems of *typability* (Typ), *type checking* (TC), and *semi-unification* (SUP).

2.1 General Notation

Small Roman letters from the middle of the alphabet (from d through t) are used as metavariables ranging over $\mathbb{N}$ (the natural numbers: 0, 1, 2, ...). If we say that some character is used as a metavariable for some set, then this also applies for superscripted or subscripted variations.

For any kind of entity X, the notation $\vec{X}^n$ denotes the sequence $X_1 X_2 \cdots X_n$ and the notation $\vec{X}$ denotes $\vec{X}^n$ for some natural number n that is either unspecified or clear from the context. The notation $\vec{X}$ may also be used to stand for either the set $\{X_1, X_2, \ldots, X_n\}$ or the comma-separated sequence X_1, X_2, ..., X_n, depending on the context.

For any function f (which may be partial), the expression $\mathrm{DOM}(f)$ denotes the domain of definition of f and $\mathrm{RAN}(f)$ denotes the range of f. For any injective function f, the expression f^{-1} denotes the inverse of f. For any functions f and g, the expression $f \circ g$ denotes the least defined function such that $(f \circ g)(x) = z$ if there exists y such that $f(x) = y$ and $g(y) = z$. For any set S and function f, the application $f(S)$ denotes the set $\{\, f(s) \mid s \in S \,\}$ and $f \downarrow S$ denotes the restriction of the function f defined only on $\mathrm{DOM}(f) \cap S$. For any sequence $\vec{X}^n$, writing $f(\vec{X}^n)$ denotes the sequence $f(X_1) \cdots f(X_n)$. For any set S, the expression $|S|$ gives the cardinality of S (written ω if the same as the size of $\mathbb{N}$).

2.2 $\boldsymbol{\lambda}$-Terms

Our notation for the λ-calculus generally follows Barendregt's [Bar84]. The set of all λ-terms Λ is built from the countably infinite set of λ-term variables $\mathcal{V}$ using application and abstraction as specified by this grammar:

$$\Lambda ::= \mathcal{V} \mid (\Lambda\,\Lambda) \mid (\lambda\mathcal{V}.\Lambda)$$

Small Roman letters from the beginning or end of the alphabet (e.g., a, b, c, x, y, z) are used as metavariables ranging over $\mathcal{V}$ and capital Roman letters from the middle of the alphabet (e.g., M, N) as metavariables ranging over Λ. When writing λ-terms, omitted parentheses are to be inferred by the rule that application associates to the left and the scope of the binding "$\lambda x.$" extends as far to the right as possible, i.e., $MNP \equiv ((MN)P)$ and $\lambda x.MN \equiv (\lambda x.(MN))$. The notation $\lambda\vec{x}.M$ stands for $\lambda x_1. \cdots .\lambda x_k.M$ for some appropriate k. We allow only λ-terms which obey the restriction that no variable is λ-bound more than once and no variable occurs both λ-bound and free. The symbol $\mathbf{K}$ denotes the standard combinator $(\lambda x.\lambda y.x)$.

The expressions $\mathrm{FV}(M)$ and $\mathrm{BV}(M)$ denote respectively the free and bound variables of a λ-term M and $\mathrm{V}(M) = \mathrm{FV}(M) \cup \mathrm{BV}(M)$. A λ-term is *open* if it has no λ-bound variables and is *closed* if it has no free variables.

If M and N are λ-terms, then $M \equiv N$ means that M and N are syntactically identical (we have no need for α-conversion on terms in this article). The statement $N \subset M$ denotes that N is a proper subterm of M and $N \subseteq M$ includes the possibility that $N \equiv M$.

The grammar for terms is extended with the possibility of a distinguished symbol $\Box$, called a *hole*, to yield the notion of a *context*. Only contexts with one hole are allowed. Let C range over contexts. If M is a λ-term, then $C[M]$ denotes the result of replacing the hole in C by M, *including* the possible capture of free variables in M by λ-bindings in C. For a context C, define $\mathrm{BHV}(C)$ to be the subset of λ-bound variables in $\mathrm{BV}(C)$ whose scope includes the hole in C. All other notions defined on terms are automatically extended to contexts. The hole of a context is considered to contain no variables, either free or bound.

A *λ-term-variable renaming* is a bijective mapping from $\mathcal{V}$ to $\mathcal{V}$. A renaming which differs from the identity on only finitely many variables may be specified as a finite mapping R such that $\mathrm{DOM}(R) = \mathrm{RAN}(R)$; it is automatically extended to be the identity on any $x \notin \mathrm{DOM}(R)$. The application $R(M)$ of a renaming R to a λ-term M is carried out by replacing each variable x (either bound or free) in M by $R(x)$. (Note that substitution and α-conversion are not defined on λ-terms in this report.)

The λ-terms defined so far belong to the usual λ-calculus, which is sometimes called the λK-calculus. The λI-calculus is similar but forbids the binding of variables with no λ-bound occurrences. The set $\Lambda_I \subset \Lambda$ of terms in the λI-calculus is defined as those λ-terms that do not have subterms of the form $(\lambda x.N)$ where $x \notin \mathrm{FV}(N)$.

2.3 Types

The set of type expressions $\mathbb{T}_{\mathrm{exp}}$ is built from the countably infinite set of type variables $\mathbb{V}$ using the "$\rightarrow$" and "$\forall$" type constructors as specified by this grammar:

$$\mathbb{T}_{\mathrm{exp}} ::= \mathbb{V} \mid (\mathbb{T}_{\mathrm{exp}} \rightarrow \mathbb{T}_{\mathrm{exp}}) \mid (\forall\, \mathbb{V}.\mathbb{T}_{\mathrm{exp}})$$

Small Greek letters from the beginning of the alphabet (e.g., α, β, γ, δ) are metavariables over $\mathbb{V}$. Small Greek letters towards the end of the alphabet (e.g., σ, τ, ρ) are metavariables over $\mathbb{T}_{\mathrm{exp}}$. When writing type expressions, omitted parentheses are to be inferred by the rule that arrows associate to the right and the scope of the binding "$\forall\alpha.$" extends as far to the right as possible, i.e., $\sigma \rightarrow \tau \rightarrow \rho = (\sigma \rightarrow (\tau \rightarrow \rho))$ and $\forall\alpha.\sigma \rightarrow \tau = (\forall\alpha.(\sigma \rightarrow \tau))$.

The expression $\mathrm{FTV}(\tau)$ denotes the free type variables of type expression τ. A type expression is *open* if it has no bound type variables and is *closed* if it has no free type variables. In the type expression $\forall\alpha.\forall\beta.\tau$, the quantifiers "$\forall\alpha$" and "$\forall\beta$" are *adjacent*. In the type expression $\forall\alpha.\tau$, if $\alpha \notin \mathrm{FTV}(\tau)$, then "$\forall\alpha$" is a *redundant* quantifier.

The set of types $\mathbb{T}$ is obtained from $\mathbb{T}_{\mathrm{exp}}$ by quotienting $\mathbb{T}_{\mathrm{exp}}$ by the operations of α-conversion, reordering of adjacent quantifiers, and inserting/deleting redundant quantifiers. The set $\mathbb{T}_\alpha$ is obtained from $\mathbb{T}_{\mathrm{exp}}$ by quotienting $\mathbb{T}_{\mathrm{exp}}$ by α-conversion without considering the other operations. For example, by reordering and α-conversion, the type expressions $\sigma = \forall\alpha.\forall\beta.\alpha \rightarrow \beta$, $\tau = \forall\beta.\forall\alpha.\beta \rightarrow \alpha$, and $\rho = \forall\beta.\forall\alpha.\alpha \rightarrow \beta$ all denote the same member of $\mathbb{T}$, but only σ and τ denote the same member of $\mathbb{T}_\alpha$. As another example, by insertion/deletion of redundant quantifiers, the type expressions $\forall\beta.\forall\alpha.\alpha$ and $\forall\alpha.\alpha$ denote the same member of $\mathbb{T}$. Unless specified otherwise, type expressions given in this report denote types. Operations on types will be specified using type expressions which represent the types. Type expressions without quantifiers are identified with the corresponding type. The usual definition of System F uses $\mathbb{T}_\alpha$ instead of $\mathbb{T}$; see section 3.2 for a proof this makes no significant difference.

A type is either a type variable or a $\rightarrow$-*type* or a $\forall$-*type*. Capital Roman letters in a "blackboard-bold" style (e.g., $\mathbb{X}$ and $\mathbb{Y}$) are metavariables over subsets of $\mathbb{T}$. For a set of types $\mathbb{X}$, the notation $\mathrm{FTV}(\mathbb{X})$ denotes $\bigcup_{\tau\in\mathbb{X}} \mathrm{FTV}(\tau)$. The notation $\forall\vec{\alpha}.\sigma$ stands for $\forall\alpha_1.\cdots.\forall\alpha_k.\sigma$, which in turn stands for $\forall\alpha_1.(\cdots(\forall\alpha_k.\sigma)\cdots)$. The symbol "$\perp$" is shorthand for $\forall\alpha.\alpha$. The notation $\forall.\sigma$ means $\forall\vec{\alpha}.\sigma$ where $\vec{\alpha} = \mathrm{FTV}(\sigma)$.

Convention 2.1. When type expressions are written to specify types (the default meaning of a type expression in this report), using α-conversion it is assumed that no variable is $\forall$-bound more than once, that the $\forall$-bound type variables of any two type expressions are distinct, and that the $\forall$-bound type variables of any type expression are distinct from the free type variables of another type expression. □

A *substitution* is a partial function from $\mathbb{V}$ to $\mathbb{T}$. Let R, S, and T range over substitutions. When used on a type σ, a substitution S is automatically extended into a function from $\mathbb{T}$ to $\mathbb{T}$ in the standard way, i.e., S is extended to the identity on variables not in its domain of definition, and then it simultaneously substitutes $S(\alpha)$ for all free occurrences of α in σ, renaming bound variables in σ as necessary to avoid capturing free variables of $S(\alpha)$. A substitution may be specified in any of the usual ways for writing partial functions. In addition, the notation $[\alpha_1{:=}\tau_1, \ldots, \alpha_n{:=}\tau_n]$, which may be abbreviated as $[\vec{\alpha}{:=}\vec{\tau}]$, denotes the substitution $\{\, \alpha_i \mapsto \tau_i \mid 1 \le i \le n \,\}$. The application of a nameless substitution is written with the substitution on the right, e.g., $\sigma[\vec{\alpha}{:=}\vec{\tau}]$, while a named substitution S is written on the left, e.g., $S(\sigma)$.

A *type-variable renaming* R is a bijective substitution such that $\mathrm{DOM}(R) = \mathrm{RAN}(R)$ (which guarantees that when R is extended to be the identity on variables not in $\mathrm{DOM}(R)$, the extended function is a bijection on $\mathbb{V}$). A *renaming* is a partial function R from $\mathcal{V} \cup \mathbb{V}$ to $\mathcal{V} \cup \mathbb{V}$ such that $R\downarrow\mathcal{V}$ is a renaming of λ-term variables and $R\downarrow\mathbb{V}$ is a renaming of type variables. When R is applied to a term, $R\downarrow\mathcal{V}$ is used, when applied to a type, $R\downarrow\mathbb{V}$ is used.

The notation $\sigma \subset \tau$ means that the type σ is properly *embedded* in the type τ. This relation is the transitive closure of the smallest relation such that $\sigma \subset \tau$ if there exist both $\vec{\alpha}$ and ρ such that $\tau = \forall\vec{\alpha}.\sigma \rightarrow \rho$ or $\tau = \forall\vec{\alpha}.\rho \rightarrow \sigma$. This definition is a bit unusual because $\forall\beta.\sigma \not\subset \forall\alpha.\forall\beta.\sigma$. The notation $\sigma \subseteq \tau$ includes the possibility that $\sigma = \tau$.

2.4 Type Inference

A pair $x{:}\sigma$ where $x \in \mathcal{V}$ and $\sigma \in \mathbb{T}$ is called a *type assumption*.[5] A finite set of type assumptions $A = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ which associates at most one type σ with any λ-term variable x is a *type environment*.[6] (Because we assume that no variable is λ-bound twice in a λ-term, viewing a type environment as a set causes no problems.) A type environment can also be viewed as a function so that $A(x)$ denotes the unique type σ such that $(x{:}\sigma) \in A$. The expression $\mathrm{FTV}(A)$ denotes $\mathrm{FTV}(\mathrm{RAN}(A))$. For a substition S and type environment A, define $S(A) = \{\, x{:}S(\tau) \mid (x{:}\tau) \in A \,\}$. For a renaming R, define $R(A) = \{\, R(x){:}R(\tau) \mid (x{:}\tau) \in A \,\}$.

An expression $A \vdash M : \tau$, where A is a type environment, M is a λ-term, and $\tau \in \mathbb{T}$, is a *sequent*.[7] We assume that throughout a sequent it is the case that all $\forall$-bound type variables are named distinctly from each other and that the $\forall$-bound and free type variables do not overlap (satisfied by α-conversion).

A *derivation* $\mathcal{D}$ consists of a (final) sequent, zero or more subderivations $\vec{\mathcal{D}}$, and a rule r such that the final sequent of $\mathcal{D}$ is a valid conclusion of the rule r using the final sequents of

[5] A type assumption has also been called by others a *declaration*, a *type assignment*, or a *type statement*.

[6] A type environment has also been called by others a *type assignment*, a *basis*, an *environment*, or a *context*, although the term *context* usually indicates it is ordered.

[7] A sequent has also been called by others an *assertion*, a *type assignment formula*, a *judgement*, or a *typing*.

VAR	$A \vdash x : A(x)$
APP	$\dfrac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$
ABS	$\dfrac{A \cup \{x{:}\sigma\} \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \to \tau} \qquad x \notin \mathrm{DOM}(A)$
INST	$\dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha{:=}\tau]}$
GEN_α	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma} \qquad \alpha \notin \mathrm{FTV}(A)$

Figure 4: Type Inference Rules of System F.

the subderivations $\vec{\mathcal{D}}$ as premises. We may write this as $\mathcal{D} = \langle r, A \vdash M : \tau, \vec{\mathcal{D}}\rangle$ or

$$\mathcal{D} = \frac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{A \vdash M : \tau}\, r$$

We extend renamings to derivations by $R(\langle r, A \vdash M : \tau, \vec{\mathcal{D}}\rangle) = \langle R(r), R(A) \vdash R(M) : R(\tau), R(\vec{\mathcal{D}})\rangle$. (The $R(r)$ part only does anything when $r = \mathrm{GEN}_\alpha$.) The *global type environment* of $\mathcal{D}$ is $\mathcal{G}(\mathcal{D}) = \bigcup_{1 \le i \le n} A_i$. (Due to our assumption that no variable is λ-bound more than once in a term, $\mathcal{G}(\mathcal{D})$ is a well defined type environment.) The *final derived type* of a subterm occurrence N in $\mathcal{D}$ is $\mathrm{FDT}(\mathcal{D}, N) = \tau$ where $A \vdash N : \tau$ is the outermost sequent in $\mathcal{D}$ to mention N as its subject.[8] When the derivation $\mathcal{D}$ is clear from the context of discussion, $\mathcal{G}(x) = (\mathcal{G}(\mathcal{D}))(x)$ and $\mathrm{FDT}(N) = \mathrm{FDT}(\mathcal{D}, N)$.

The inference rules of System F are given in figure 4. Unless specified otherwise, all derivations in this report are of System F. The name GEN stands for the collection of GEN_α for all type variables α.

A *typing* of the term M is a derivation whose final sequent is $A \vdash M : \tau$ for some type environment A and type τ. A λ-term M is *typable* if and only if there is a typing for M.

Definition 2.2 (Typ). The typability problem: Given an arbitrarily chosen λ-term M, is M typable? □

Definition 2.3 (TC). The type-checking problem: Given arbitrarily chosen λ-term M, type environment A, and type $\tau \in \mathbb{T}$, is there a derivation whose final sequent is $A \vdash M : \tau$? □

2.5 Semi-Unification

For convenience, we define semi-unification using a first-order signature containing the single infix binary function symbol "$\to$" and for the case where there are only two pairs of terms. (The general definition of semi-unification is reducible to this special case [Pud88, KTU93] and the proof that semi-unification is undecidable is actually for this special case [KTU93].)

[8] This notion is not completely defined because there may be multiple occurrences of a subterm $N \subset M$ with distinct typings. This does not pose a problem in this report because it is always clear from the context of discussion which subterm occurrence is meant.

The set of algebraic terms $\mathcal{T}$ is defined by the grammar $\mathcal{T} ::= \mathbb{V} \mid (\mathcal{T} \to \mathcal{T})$. (This definition is chosen because it allows mapping terms onto types. In fact, $\mathcal{T} \subset \mathbb{T}$.) An *instance* Γ of semi-unification is a set of two pairs

$$\Gamma = \{ (\tau_1, \mu_1), (\tau_2, \mu_2) \}$$

where $\tau_1, \tau_2, \mu_1, \mu_2 \in \mathcal{T}$.[9] An *open substitution* is a substitution S such that $\mathrm{RAN}(S)$ contains only open types. An open substitution S is a *solution* for an instance Γ of semi-unification if and only if there also exist open substitutions S_1, S_2 such that $S_1(S(\tau_1)) = S(\mu_1)$ and $S_2(S(\tau_2)) = S(\mu_2)$.[10]

Definition 2.4 (SUP). The semi-unification problem: Given an arbitrarily chosen instance Γ of semi-unification, does Γ have a solution? □

3 Properties of Derivations in System F

This section proves some useful properties about derivations in System F. These properties make it easier to reason about and manipulate derivations.

3.1 Basic Properties

Definition 3.1 (INST-Before-GEN Property). A derivation $\mathcal{D}$ satisfies the INST-*before*-GEN *property* if in $\mathcal{D}$ the premise of a use of the INST rule is never the conclusion of a use of the GEN rule. □

Lemma 3.2. *If $\mathcal{D}$ is a derivation ending with the sequent $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ ending with the same sequent that satisfies the* INST-*before*-GEN *property.* □

Proof. This is an immediate consequence (or restatement) of theorem 3 in [GRDR88] which is itself a direct consequence of "the normalization property for second-order deductions given in [Pra65]". □

Definition 3.3 (GEN-Distinct Derivations). A derivation $\mathcal{D}$ is GEN-*distinct* if for every subderivation ending with

$$\frac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}\ \mathrm{GEN}_\alpha$$

it is the case that the type variable α does not appear free later in $\mathcal{D}$ or in any other subderivation of $\mathcal{D}$. □

Lemma 3.4. *For any derivation $\mathcal{D}$, there is a* GEN-*distinct derivation $\mathcal{D}'$ which has the same final sequent and rule-tree structure as $\mathcal{D}$.* □

Proof. Let $\mathcal{P} = \mathbb{N}^*$ be the set of strings over $\mathbb{N}$. Let p and q range over $\mathcal{P}$. Let ε denote the empty string and $p \cdot q$ denote concatenation. Let Ψ be an injective function from $\mathcal{P}$ to type variables which do not appear in $\mathcal{D}$. Define the function Φ on derivations as follows:

$$\begin{aligned}
\Phi(p, \langle \mathrm{GEN}_\alpha, A \vdash M : \tau, \mathcal{D} \rangle) &= \langle \mathrm{GEN}_{\Psi(p)}, A \vdash M : \tau, \Phi(p \cdot 1, R_{\alpha,p}(\vec{\mathcal{D}})) \rangle \\
\Phi(p, \langle r, A \vdash M : \tau, \vec{\mathcal{D}}^n \rangle) &= \langle r, A \vdash M : \tau, \Phi(p \cdot 1, \mathcal{D}_1), \ldots, \Phi(p \cdot n, \mathcal{D}_n) \rangle \\
&\qquad \text{if } r \in \{\mathrm{VAR}, \mathrm{ABS}, \mathrm{APP}, \mathrm{INST}\}
\end{aligned}$$

[9] Sometimes the pairs are written with the symbol "$\leq$". We avoid that symbol because of its frequent use for denoting subtyping relationships.

[10] It is possible to define semi-unification on $\mathbb{T}$ rather than on $\mathcal{T}$. Then the requirement that the solution be *open* could be dropped. However, this is not the definition used in the proof of undecidability [KTU93], so it would be necessary to prove the two problems equivalent.

where $R_{\alpha,p} = \{\alpha \mapsto \Psi(p), \Psi(p) \mapsto \alpha\}$ is a type-variable renaming that swaps α with a fresh type variable and is the identity on other variables appearing in $\mathcal{D}$. It can be checked that $\Phi(\varepsilon, \mathcal{D})$ is a GEN-distinct derivation with the same final sequent as $\mathcal{D}$ and the same rule-tree structure. □

Using lemma 3.4, henceforth it is assumed (unless specified otherwise) that all derivations are GEN-distinct.

Remark 3.5. In a Church-style presentation of System F, the INST-before-GEN property would be expressed as the notion of being in normal form with respect to type-level β-reduction. The GEN-distinct property would be expressed as the notion of distinct type-level λ-abstractions binding distinct type-variable names. □

The following lemma is a specialized version of the standard *weakening* lemma. In this version, we are interested not only in the final result (the weakened derivation), but how it is necessary to alter the internal structure of the derivation to get this result.

Lemma 3.6 (Weakening). *Let $\mathcal{D}$ be a derivation whose sequents are $A_i \vdash M_i : \tau_i$ for $1 \le i \le n$. Let B be a type environment such that $\mathrm{DOM}(B) \cap \mathrm{DOM}(\mathcal{G}(\mathcal{D})) = \varnothing$. Let $\mathbb{X}$ be the set of type variables which are generalized in $\mathcal{D}$. Let R be a type-variable renaming such that (1) for every type variable α occurring in $\mathcal{D}$, either $R(\alpha) = \alpha$ or $R(\alpha)$ is fresh (not mentioned in $\mathcal{D}$) and (2) R maps $\mathrm{FTV}(B) \cap \mathbb{X}$ to fresh names. Then there must be a derivation $\mathcal{D}'$ whose sequents are $R(A_i) \cup B \vdash M_i : R(\tau_i)$ for $1 \le i \le n$.* □

Proof. By induction on the length of $\mathcal{D}$. □

Definition 3.7 (“$\preceq$” and “$\preccurlyeq$”). In this definition, let X range over either types, sets of types, or type environments. Let “$\preceq_X$” be the least relation such that $\forall\vec{\alpha}.\sigma \preceq_X \forall\vec{\gamma}.S(\sigma)$ if $\mathrm{DOM}(S) = \{\vec{\alpha}\} \cup (\mathrm{FTV}(\sigma) - \mathrm{FTV}(X))$ and $\{\vec{\gamma}\} \cap \mathrm{FTV}(X) = \varnothing$. Let “$\preccurlyeq_X$” be the least relation such that $\forall\vec{\alpha}.\sigma \preccurlyeq_X \forall\vec{\gamma}.S(\sigma)$ if $\mathrm{DOM}(S) = \{\vec{\alpha}\}$ and $\{\vec{\gamma}\} \cap \mathrm{FTV}(X) = \varnothing$. □

Lemma 3.8 (Initial/Final Derived Types). *Let $\mathcal{D}$ be a typing for M and let $N \subset M$ be a subterm occurrence. Let $\mathcal{D}'$ be the subderivation of $\mathcal{D}$ for N. Let $\mathcal{D}'$ end with $A \vdash N : \tau$. Then* $\mathrm{IDT}(\mathcal{D}, N) \preceq_A \mathrm{FDT}(\mathcal{D}, N)$. *Furthermore, if $\mathcal{D}$ satisfies the* INST-*before*-GEN *property, then* $\mathrm{IDT}(\mathcal{D}, N) \preccurlyeq_A \mathrm{FDT}(\mathcal{D}, N)$. □

Proof. By induction on derivations. □

Lemma 3.9 (INST-before-GEN Generation). *Let $\mathcal{D}$ be a derivation satisfying the* INST-*before*-GEN *property.*

1. *If the final sequent of $\mathcal{D}$ is $A \vdash x : \tau$, then $A(x) \preccurlyeq_A \tau$.*
2. *If the final sequent of $\mathcal{D}$ is $A \vdash MN : \tau$, then $\mathcal{D}$ contains a subderivation of the following form where $\tau' \preccurlyeq_A \tau$:*

$$\dfrac{\dfrac{\vdots}{A \vdash M : \sigma \to \tau'}\,r \qquad \dfrac{\vdots}{A \vdash N : \sigma}\,r'}{A \vdash MN : \tau'}\,\mathrm{APP}$$

3. *If the final sequent of $\mathcal{D}$ is $A \vdash \lambda x.M : \tau$, then $\tau = \forall\vec{\alpha}.\sigma \to \rho$ where $\{\vec{\alpha}\} \cap \mathrm{FTV}(A) = \varnothing$ and $\mathcal{D}$ contains a subderivation of this form:*

$$\dfrac{\dfrac{\vdots}{A \cup \{x{:}\sigma\} \vdash M : \rho}\,r}{A \vdash \lambda x.M : \sigma \to \rho}\,\mathrm{ABS}$$

□

Proof. Using lemma 3.8 and the rules of System F in figure 4. □

Lemmas 3.8 and 3.9 will be used implicitly in proofs throughout the rest of the paper, so the reader needs to remember them.

3.2 Adjacent and Redundant Quantifiers

This section shows that this report's convention of freely allowing reordering of adjacent quantifiers in types and adding or removing redundant quantifiers to/from types does not affect the set of terms typable in System F and has only a negligible effect on the derivable typings.

Definition 3.10 (Canonical Type Expressions). Let Υ be the function from $\mathbb{T}$ to $\mathbb{T}_\alpha$ defined as follows:

$\Upsilon(\forall\vec{\alpha}.\beta)$	$=\beta$	if $\beta \notin \{\vec{\alpha}\}$
$\Upsilon(\forall\vec{\alpha}^n.\alpha_i)$	$=\bot$	if $1 \leq i \leq n$
$\Upsilon(\forall\vec{\alpha}.\sigma \to \tau)$	$=\forall\vec{\beta}.\Upsilon(\sigma) \to \Upsilon(\tau)$	where $\{\vec{\beta}\} = \{\vec{\alpha}\} \cap \mathrm{FTV}(\sigma \to \tau)$ and the members of $\vec{\beta}$ are in order from left-to-right of their leftmost occurrences in $\sigma \to \tau$

Extend Υ to type environments so that $(\Upsilon(A))(x) = \Upsilon(A(x))$. A type $\sigma \in \mathbb{T}_\alpha$ is *canonical* iff $\sigma \in \mathrm{RAN}(\Upsilon)$. Let F_α be defined in the same way as System F except using $\mathbb{T}_\alpha$ wherever $\mathbb{T}$ is used in the definition of System F.[11] A derivation $\mathcal{D}$ of F_α is canonical iff all types appearing in $\mathcal{D}$ are canonical. □

Lemma 3.11. *If $A \vdash M : \tau$ is derivable in System F, then there is a canonical derivation $\mathcal{D}$ in F_α ending with $\Upsilon(A) \vdash M : \Upsilon(\tau)$.* □

Proof. By induction on the structure of derivations. Let $\mathcal{D} = \langle r, A \vdash M : \tau, \vec{\mathcal{D}}^n \rangle$ be a derivation of System F. By cases on r, the last rule used. The cases of VAR, ABS, and APP are omitted because they are simple. The interesting cases are GEN and INST:

1. Suppose r is GEN_α. Then the last step in $\mathcal{D}$ looks like this:

$$\frac{A \vdash M : \tau}{A \vdash M : \forall\alpha.\tau}\ \mathrm{GEN}_\alpha$$

   By the induction hypothesis, there is a canonical derivation $\mathcal{D}'$ of F_α ending with $\Upsilon(A) \vdash M : \Upsilon(\tau)$. There are two cases.

   (a) Suppose $\alpha \notin \mathrm{FTV}(\tau)$. Then $\Upsilon(\forall\alpha.\tau) = \Upsilon(\tau)$. Thus, $\mathcal{D}'$ is the desired derivation.

   (b) Suppose $\alpha \in \mathrm{FTV}(\tau)$. Let $\Upsilon(\forall\alpha.\tau) = \forall\vec{\gamma}.\forall\alpha.\forall\vec{\delta}.\sigma$ where σ is not a $\forall$-type. Observe that $\Upsilon(\tau) = \forall\vec{\gamma}.\forall\vec{\delta}.\sigma$. Observe that $\alpha \notin \mathrm{FTV}(\Upsilon(A))$. By α-conversion, take $\vec{\gamma}$ to be such that $\vec{\gamma} \notin \mathrm{FTV}(\Upsilon(A))$. Thus, by multiple uses of INST starting from $\mathcal{D}'$, the sequent $\Upsilon(A) \vdash M : \forall\vec{\delta}.\sigma$ is derivable in F_α. Multiple uses of GEN then derive $\Upsilon(A) \vdash M : \Upsilon(\forall\alpha.\tau)$ in F_α. The resulting derivation satisfies the claim of the lemma.

2. Suppose r is INST. Then the last step in $\mathcal{D}$ looks like this:

$$\frac{A \vdash M : \forall\alpha.\tau}{A \vdash M : \tau[\alpha := \rho]}\ \mathrm{INST}$$

   By the induction hypothesis, there is a canonical derivation $\mathcal{D}'$ in F_α ending with $\Upsilon(A) \vdash M : \Upsilon(\forall\alpha.\tau)$. There are two cases.

[11] The usual definition of System F is in fact F_α.

(a) Suppose $\alpha \notin \mathrm{FTV}(\tau)$. Then $\Upsilon(\forall\alpha.\tau) = \Upsilon(\tau)$. Thus, $\mathcal{D}'$ is the desired derivation.

(b) Suppose $\alpha \in \mathrm{FTV}(\tau)$. Let $\Upsilon(\forall\alpha.\tau) = \forall\vec{\gamma}.\forall\alpha.\forall\vec{\delta}.\sigma$ where σ is not a $\forall$-type. By α-conversion, take $\vec{\gamma}$, α, and $\vec{\delta}$ to be such that $\vec{\gamma} \notin \mathrm{FTV}(\Upsilon(A))$, $\vec{\gamma}, \vec{\delta} \notin \mathrm{FTV}(\rho)$, and all of $\vec{\gamma}$, α, and $\vec{\delta}$ are distinct. Observe that $\Upsilon(\tau) = \forall\vec{\gamma}.\forall\vec{\delta}.\sigma$. By multiple uses of INST starting from $\mathcal{D}'$, the sequent $\Upsilon(A) \vdash M : \forall\alpha.\forall\vec{\delta}.\sigma$ is derivable in F_α. By INST, the sequent $\Upsilon(A) \vdash M : (\forall\vec{\delta}.\sigma)[\alpha := \Upsilon(\rho)]$ is derivable in F_α. By multiple uses of GEN, the sequent $\Upsilon(A) \vdash M : \forall\vec{\gamma}.((\forall\vec{\delta}.\sigma)[\alpha := \Upsilon(\rho)])$ is derivable in F_α. Observe that any occurrence of α in σ must either be at the root of σ or immediately below a "$\rightarrow$". It can be checked that $\forall\vec{\gamma}.((\forall\vec{\delta}.\sigma)[\alpha := \Upsilon(\rho)]) = (\forall\vec{\gamma}.\forall\vec{\delta}.\sigma)[\alpha := \Upsilon(\rho)] = \Upsilon(\tau)[\alpha := \Upsilon(\rho)] = \Upsilon(\tau[\alpha := \rho])$ in F_α. Thus, the resulting derivation satisfies the claim of the lemma. □

4 Undecidability of Type Checking for System F

This section proves that TC is undecidable for System F. The proof works by reducing the SUP problem to the TC problem.

Theorem 4.1 (SUP $\leq$ TC). SUP *with a single binary function symbol and two pairs in each problem instance is reducible to* TC *in System F. Furthermore, this reduction works for System F restricted to terms of the* λI*-calculus.* □

Proof. Consider any instance Γ of SUP of the form $\Gamma = \{\,(\tau_1, \mu_1), (\tau_2, \mu_2)\,\}$. Let the free type variables in τ_1, μ_1, τ_2, and μ_2 be contained within $\{\alpha_1, \ldots, \alpha_m\}$. Pick fresh type variables δ_1 and δ_2 We construct an instance of TC from the SUP instance Γ. First, construct a λ-term M (which does not depend on Γ):

$$M \equiv b(\lambda x.cxx)$$

Then, construct a type environment A_Γ (which depends on Γ):

$$\begin{aligned} A_\Gamma(b) &= \forall\gamma.(\gamma \rightarrow \gamma) \rightarrow \beta \\ A_\Gamma(c) &= \forall.(\mu_1 \rightarrow \delta_1) \rightarrow (\delta_2 \rightarrow \mu_2) \rightarrow (\tau_1 \rightarrow \tau_2) \end{aligned}$$

The free type variable β will be the type of M if it is typable at all with the type environment A_Γ. The $\forall$-bound type variables $\alpha_1, \ldots, \alpha_m$ correspond directly to the variables occurring in the semi-unification instance Γ. The $\forall$-bound type variables δ_1 and δ_2 are used where we do not care to what types they are instantiated so long as they match whatever needs to be matched. The $\forall$-bound type variable γ is used to require the final derived type for (cxx) to be equal to the assigned type for x.

It is easy to see that the λ-term M is a term of the λI-calculus. What is left to show is that Γ has a solution if and only if there is a derivation in System F with final sequent $A_\Gamma \vdash M : \beta$. The two directions are proved separately in the rest of this proof.

Suppose Γ has a solution, i.e., there are open substitutions S, S_1, and S_2 so that $S_1(S(\tau_1)) = S(\mu_1)$ and $S_2(S(\tau_2)) = S(\mu_2)$. Then there is a derivation of $A_\Gamma \vdash M : \beta$ as follows. Let B be the following type environment:

$$B = A_\Gamma \cup \{x{:}\forall.S(\tau_1) \rightarrow S(\tau_2)\}$$

The following sequents are clearly derivable merely using the VAR and INST rules:

$$\begin{aligned} &B \vdash c : (S(\mu_1) \rightarrow S_1(S(\tau_2))) \rightarrow (S_2(S(\tau_1)) \rightarrow S(\mu_2)) \rightarrow (S(\tau_1) \rightarrow S(\tau_2)) \\ &B \vdash x : S_1(S(\tau_1)) \rightarrow S_1(S(\tau_2)) \\ &B \vdash x : S_2(S(\tau_1)) \rightarrow S_2(S(\tau_2)) \end{aligned}$$

Now, because $S(\mu_1) = S_1(S(\tau_1))$ and $S(\mu_2) = S_2(S(\tau_2))$ (because S, S_1, and S_2 form a solution for Γ), these sequents are derivable from the preceding sequents using the APP and ABS rules:

$$\begin{aligned} B &\vdash cx : (S_2(S(\tau_1)) \to S(\mu_2)) \to (S(\tau_1) \to S(\tau_2)) \\ B &\vdash cxx : (S(\tau_1) \to S(\tau_2)) \end{aligned}$$

Assume with no loss of generality that the range of the open substitutions S, S_1, and S_2 do not mention the type variable β. So this sequent is derivable using the GEN rule:

$$B \vdash cxx : \forall.S(\tau_1) \to S(\tau_2)$$

These uses of the ABS, VAR, INST, and APP rules are straightforward:

$$\begin{aligned} A_\Gamma &\vdash \lambda x.cxx : (\forall.S(\tau_1) \to S(\tau_2)) \to (\forall.S(\tau_1) \to S(\tau_2)) \\ A_\Gamma &\vdash b : ((\forall.S(\tau_1) \to S(\tau_2)) \to (\forall.S(\tau_1) \to S(\tau_2))) \to \beta \\ A_\Gamma &\vdash b(\lambda x.cxx) : \beta \end{aligned}$$

The final sequent in this derivation is the desired one.

The proof of the other direction is more complicated. Suppose that there is a derivation $\mathcal{D}$ that ends with the sequent $A_\Gamma \vdash M : \beta$. By lemma 3.2 we assume that $\mathcal{D}$ satisfies the INST-before-GEN property of definition 3.1. The following analysis shows there is a solution for the semi-unification instance Γ.

In this derivation $\mathcal{D}$, let B be the type environment used in deriving a type for (cxx). Let σ be the type assumed for x by B, i.e., $B = A_\Gamma \cup \{x{:}\sigma\}$. The sequent that produces the final derived type for c must be the following sequent:

$$B \vdash c : (T(\mu_1) \to T(\delta_1)) \to (T(\delta_2) \to T(\mu_2)) \to (T(\tau_1) \to T(\tau_2))$$

The substitution T represents the effects of using the INST rule for each type variable in $\{\alpha_1, \ldots, \alpha_m, \delta_1, \delta_2\}$. With no loss of generality, assume that for each type τ in the range of T, the free type variables of τ are contained in the set $\{\alpha_1, \ldots, \alpha_n\}$ for some $n \geq m$, i.e., $\mathrm{FTV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}$.

Because c will be applied to x, the final derived type of c must have no outermost quantifiers. By the INST-before-GEN property, there are no uses of the GEN rule for c. Because the shape of the final derived type for c lacks embedded quantifiers in particular positions, the final derived type for each occurrence of x must be an $\to$-type. Thus, the sequent in $\mathcal{D}$ that produces the final derived type for the first occurrence of x must be exactly this:

$$B \vdash x : T(\mu_1) \to T(\delta_1) \tag{1}$$

Then, the APP rule must be used to produce this sequent in $\mathcal{D}$:

$$B \vdash cx : (T(\delta_2) \to T(\mu_2)) \to (T(\tau_1) \to T(\tau_2)) \tag{2}$$

Because (cx) will be applied to x, the final derived type for (cx) must have no outermost quantifiers. If some quantifiers were introduced by GEN, then they would have to be removed again by INST. Thus, by invoking the INST-before-GEN property, there will be no use of the GEN rule using sequent (2) as its premise. Thus, for the second occurrence of x, the sequent producing its final derived type must be exactly this:

$$B \vdash x : T(\delta_2) \to T(\mu_2) \tag{3}$$

At this point the APP rule must be used to produce this sequent:

$$B \vdash cxx : T(\tau_1) \to T(\tau_2)$$

Now some number of uses of the GEN rule will result in a sequent that looks like this:

$$B \vdash cxx : \forall\vec{\varepsilon}.(T(\tau_1) \to T(\tau_2))$$

where $\vec{\varepsilon}$ is a subset of $\{\alpha_1, \ldots, \alpha_n\}$. By the INST-before-GEN property, there are no uses of the INST rule at this point. The next step in the derivation must be to use ABS to produce this:

$$A_\Gamma \vdash \lambda x.cxx : \sigma \to \forall\vec{\varepsilon}.(T(\tau_1) \to T(\tau_2)) \tag{4}$$

Consider now the sequent producing the final derived type for b, which must look like this:

$$A_\Gamma \vdash b : (\varphi \to \varphi) \to \beta \tag{5}$$

The derivation must now type the application $(b(\lambda x.cxx))$. Observing the shape of the final derived type for b shows that the final derived type for $(\lambda x.cxx)$ must not have any outermost quantifiers. Using the INST-before-GEN property, there is no use of the GEN rule with the sequent (4) as its premise. Thus, the sequents (4) and (5) must be combined using the APP rule to produce this sequent:

$$A_\Gamma \vdash b(\lambda x.cxx) : \beta$$

As expected, the type derived for $(b(\lambda x.cxx))$ must be exactly β. In order for the APP rule to have been used this way, it must be the case that these types are equal:

$$\varphi = \sigma = \forall\vec{\varepsilon}.(T(\tau_1) \to T(\tau_2))$$

Based on our new knowledge of the type σ which was assigned to x, the sequents producing the final derived types of the first and second occurrences of x must be exactly these:

$$B \vdash x : T_1(T(\tau_1)) \to T_1(T(\tau_2)) \tag{6}$$
$$B \vdash x : T_2(T(\tau_1)) \to T_2(T(\tau_2)) \tag{7}$$

In these equations, T_1 and T_2 are substitutions from $\mathbb{V}$ to $\mathbb{T}$ which must be the identity on any type variable not in $\{\alpha_1, \ldots, \alpha_n\}$. The substitutions T_1 and T_2 represent the effects of using the INST rule once for each $\varepsilon_i \in \vec{\varepsilon}$.

Recall now the sequents (1) and (3) from above. Because the sequents (1) and (6) must actually be identical and likewise the sequents (3) and (7), the following equalities must hold:

$$\begin{array}{rclcrcl} T_1(T(\tau_1)) & = & T(\mu_1) & \quad & T_1(T(\tau_2)) & = & T(\delta_1) \\ T_2(T(\tau_1)) & = & T(\delta_2) & \quad & T_2(T(\tau_2)) & = & T(\mu_2) \end{array}$$

This is almost, but not quite, a solution for the semi-unification instance Γ. The difference is that the ranges of T, T_1, and T_2 may include types with bound type variables.

Using T, T_1, and T_2, define the open substitutions S, S_1, and S_2 which will form a solution for Γ. Pick some type variable β arbitrarily. Define the function *erase* to erase quantifiers from a type as follows:

$$\begin{array}{rcl} erase(\alpha) & = & \alpha \\ erase(\sigma \to \tau) & = & erase(\sigma) \to erase(\tau) \\ erase(\forall\alpha.\sigma) & = & erase(\sigma[\alpha{:=}\beta]) \end{array}$$

Now define the behavior of S, S_1, and S_2 for each $\alpha_i \in \{\alpha_1, \ldots, \alpha_n\}$ so that $S(\alpha_i) = erase(T(\alpha_i))$, $S_1(\alpha_i) = erase(T_1(\alpha_i))$, and $S_2(\alpha_i) = erase(T_2(\alpha_i))$. Let S, S_1, and S_2 be the identity on any other variables. The observation that for $i, j \in \{1, 2\}$ it is the case that $erase(T_i(T(\tau_j))) = erase(T_i(erase(T(\tau_j))))$ shows that the equalities $S_1(S(\tau_1)) = S(\mu_1)$ and $S_2(S(\tau_2)) = S(\mu_2)$ both hold. Thus, S is a solution for Γ. $\square$

5 Derivations with Invariant Type Assumptions

This section establishes machinery that will be used in section 6 to chain together a number of results. We define the notion of an *invariant type assumption* and what it means for one set of types to *induce* another set of types.

Definition 5.1 (Invariant Type Assumption). An *invariant type assumption* is a type assumption for a bound variable that must be essentially the same in all type derivations for a particular term. This notion is defined more precisely as follows. Let M be a λ-term, x a bound variable of M, ρ a type, and A a type environment such that $\mathrm{DOM}(A) = \mathrm{FV}(M)$. We say that A and M induce the *invariant type assumption* $(x{:}\rho)$ if and only if both of the following properties hold:

1. There is a derivation $\mathcal{D}$ ending with $A \vdash M : \tau$ for some type τ where $\mathcal{G}(x) = \rho$.
2. For every derivation $\mathcal{D}$ ending with $A \vdash M : \tau$ for some type τ, there exists a type-variable renaming R such that $R(A) = A$ and $\mathcal{G}(x) = R(\rho)$.

(Remember that $\mathcal{G}$ denotes the global type environment of a derivation.) $\square$

Section 6 will prove that type environments and λ-terms that induce various desired invariant type assumptions can be constructed. However, achieving the main result will need a way to chain these results together and to express that a number of invariant type assumptions can be induced simultaneously. To accomplish this, definition 5.2 provides a similar notion, but in terms of contexts and type environments. The difference between the complexity of definition 5.1 and definition 5.2 is due to the latter definition's need to impose the invariant type assumptions on arbitrary terms, to handle simultaneous constraints, and to deal with extra term variables. For simultaneous constraints, one of the requirements is that contexts must be chained together and that the invariant type assumptions imposed by the outer context must be successfully passed through the inner context. Extra term variables are used in the construction of an invariant type assumption, but the types of these variables might not be constrained and will not be used.

Definition 5.2 ($A, C \rhd B$). Let $A \cup B$ be a type environment and let C be a context such that $\mathrm{FV}(C) \subseteq \mathrm{DOM}(A)$, $\mathrm{BV}(C) \cap \mathrm{DOM}(A) = \varnothing$, and $\mathrm{DOM}(B) \subseteq \mathrm{BHV}(C) \cup \mathrm{DOM}(A)$. The statement $A, C \rhd B$ ("the type environment A and context C together induce the type environment B") holds iff for any term M and type environment A' such that $C[M]$ is a term, $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A \cup B \cup A')$, $\mathrm{DOM}(A') \cap (\mathrm{DOM}(A) \cup \mathrm{BV}(C[M])) = \varnothing$, and $\mathrm{FTV}(A') \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$, both of the following properties hold:

1. If σ is a type such that
$$A' \cup A \cup B \vdash M : \sigma$$
is derivable, then there exists a type τ, a type environment E such that $\mathrm{DOM}(E) = \mathrm{BHV}(C) - \mathrm{DOM}(B)$, and a derivation $\mathcal{D}$ containing these sequents:
$$A' \cup A \vdash C[M] : \tau$$
$$A' \cup A \cup B \cup E \vdash M : \sigma$$

2. If τ is a type and $\mathcal{D}$ is a derivation containing
$$A \cup A' \vdash C[M] : \tau,$$
then there exists a type σ, a type environment E where $\mathrm{DOM}(E) = \mathrm{BHV}(C) - \mathrm{DOM}(B)$, and a type-variable renaming R satisfying $R(A \cup A') = A \cup A'$ such that $\mathcal{D}$ also contains this sequent:
$$A \cup A' \cup R(B) \cup E \vdash M : \sigma$$

(The type environment A' covers extra free variables of M that are not handled by either A or the bindings in C whose types should be invariant. This is necessary to support chaining contexts to induce simultaneous constraints. Type variables in the difference $\mathrm{FTV}(B) - \mathrm{FTV}(A)$ will always need to be discharged by the GEN rule in the course of typing C, so they must not be mentioned by A'. The type environment E covers the extra variable bindings of C whose scope includes the hole and whose types need not be invariant (i.e., bindings which do not have types given for them by B).) □

Often, the particular choice of the context and the term variables mentioned in the type environments is unimportant. The following definition abstracts away these details.

Definition 5.3 ($\mathbb{X} \blacktriangleright \mathbb{Y}$). Let $\mathbb{X}$ and $\mathbb{Y}$ be type sets. An algorithm Ψ taking a type environment as input and returning a type environment and a context as output is a *witness* for the statement $\mathbb{X} \blacktriangleright \mathbb{Y}$ ("the set of types $\mathbb{X}$ induces the set of types $\mathbb{Y}$") iff for every non-empty type environment B such that $\mathrm{RAN}(B) \subseteq \mathbb{Y}$ it holds that $\Psi(B) = (A, C)$ where $\mathrm{RAN}(A) \subseteq \mathbb{X}$, $\mathrm{FTV}(\mathrm{RAN}(A)) \supseteq \mathrm{FTV}(\mathbb{X}) \cap \mathrm{FTV}(\mathrm{RAN}(B))$, and $A, C \rhd B$. The statement $\mathbb{X} \blacktriangleright \mathbb{Y}$ holds iff there exists a witness for it. □

Lemma 5.4 (Properties of $\rhd$ and $\blacktriangleright$). *Let $\mathbb{X}$, $\mathbb{Y}$, and $\mathbb{Z}$ range over subsets of $\mathbb{T}$.*

1. *If $A, C \rhd B$ then $R(A), R(C) \rhd R(B)$ for any renaming R.*
2. *If $A, C \rhd B$, then $A, C \rhd A \cup B$.*
3. *If $A, C \rhd B_1 \cup B_2$, then $A, C \rhd B_1$.*
4. *If $A, C \rhd B$, $\mathrm{FTV}(A) \supseteq \mathrm{FTV}(A') \cap \mathrm{FTV}(B)$, and $\mathrm{DOM}(A') \cap (\mathrm{DOM}(A) \cup \mathrm{BV}(C)) = \varnothing$, then $A \cup A', C \rhd B$.*
5. *If the type environment A and the term M induce the invariant type assumption $(x{:}\tau)$ and either τ is not a $\forall$-type or $\mathrm{FTV}(\tau) \subseteq \mathrm{FTV}(A)$, then $\mathrm{RAN}(A) \blacktriangleright \{\tau\}$.*
6. *If $\mathbb{X} \blacktriangleright \mathbb{Y}$, then $R(\mathbb{X}) \blacktriangleright R(\mathbb{Y})$ for any type-variable renaming R.*
7. *If $\mathbb{X} \blacktriangleright \mathbb{Y}$ then $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$.*
8. *If $\mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2$ then $\mathbb{X} \blacktriangleright \mathbb{Y}_1$.*
9. *If $\mathbb{X}_1 \blacktriangleright \mathbb{Y}$ and $\mathrm{FTV}(\mathbb{X}_1) \supseteq \mathrm{FTV}(\mathbb{X}_2) \cap \mathrm{FTV}(\mathbb{Y})$, then $\mathbb{X}_1 \cup \mathbb{X}_2 \blacktriangleright \mathbb{Y}$.*
10. *If $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$ and $\mathbb{X} \cup \mathbb{Y} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$, then $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$.*
11. *If $\mathbb{X} \blacktriangleright \mathbb{Y}_1$ and $\mathbb{X} \blacktriangleright \mathbb{Y}_2$ and $\mathrm{FTV}(\mathbb{Y}_1) \cap \mathrm{FTV}(\mathbb{Y}_2) \subseteq \mathrm{FTV}(\mathbb{X})$, then $\mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2$.*

□

Proof. Given decidable sets $X, Y, Z \subset \mathcal{V}$ of λ-term variables, let $R_{X,Y,Z}$ be a term-variable renaming which swaps the members of $(X \cap Y) - Z$ with variables outside of $X \cup Y \cup Z$ chosen in some deterministic manner and which is otherwise the identity. Let $R_{X,Y} = R_{X,Y,\varnothing}$. Each property is proved separately.

1. The statement $A, C \rhd B$ does not depend on the particular variable names involved, but rather on which variable names are the same and which are different. Applying a renaming does not affect this, so $A, C \rhd B$ holds iff $R(A), R(C) \rhd R(B)$ holds.
2. By inspection of definition 5.2.
3. By inspection of definition 5.2.

4. It is given that $A, C \rhd B$, $\mathrm{FTV}(A) \supseteq \mathrm{FTV}(A') \cap \mathrm{FTV}(B)$, and $\mathrm{DOM}(A') \cap (\mathrm{DOM}(A) \cup \mathrm{BV}(C)) = \varnothing$

   It must now be checked that $A \cup A', C \rhd B$. We prove that it satisfies each property of definition 5.2. Let M and $\hat{A}$ be such that $C'[M]$ is a term, $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A \cup A' \cup \hat{A} \cup B)$, $\mathrm{DOM}(\hat{A}) \cap (\mathrm{DOM}(A \cup A') \cup \mathrm{BV}(C'[M])) = \varnothing$, and $\mathrm{FTV}(\hat{A}) \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A \cup A')$.

   (a) Let σ be such that $A \cup A' \cup \hat{A} \cup B \vdash M : \sigma$ is derivable. By $A, C \rhd B$ and definition 5.2 (1), using $A' \cup \hat{A}$ as the extra type environment such that $\mathrm{FTV}(A' \cup \hat{A}) \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$, there is a type τ, a type environment E and a derivation $\mathcal{D}$ containing the sequents $A \cup A' \cup \hat{A} \vdash C[M] : \tau$ and $A \cup A' \cup \hat{A} \cup B \cup E \vdash M : \sigma$. This derivation $\mathcal{D}$ is exactly what is necessary.

   (b) Let τ and $\mathcal{D}$ be some type and derivation such that $\mathcal{D}$ contains $A \cup A' \cup \hat{A} \vdash C[M] : \tau$. By $A, C \rhd B$ and definition 5.2 (2), interpreting $A' \cup \hat{A}$ as the extra type environment such that $\mathrm{FTV}(A' \cup \hat{A}) \cap \mathrm{FTV}(B) \subseteq \mathrm{FTV}(A)$, there must exist some type σ, type environment E, and type-variable renaming R satisfying $R(A \cup A' \cup \hat{A}) = A \cup A' \cup \hat{A}$ such that $\mathcal{D}$ contains $A \cup A' \cup \hat{A} \cup R(B) \cup E \vdash M : \sigma$. This subderivation is exactly what is needed.

5. It is given that A and the term M induce the invariant type assumption $(x{:}\tau)$. It holds that $M \equiv C[\lambda x.N]$ for some context C and term N. Let $C' \equiv C[\lambda x.\mathsf{K} N \Box]$. It can be checked that $A, C' \rhd \{x{:}\tau\}$.

   The witness Ψ for $\mathrm{RAN}(A) \blacktriangleright \{\tau\}$ is defined in one of two different ways as follows.

   (a) Suppose τ is not a $\forall$-type. Then define Ψ so that

   $$\Psi(B) = (R(A), R(C')[(\lambda y_1. \cdots ((\lambda y_n.\Box) R(x)) \cdots) R(x)])$$

   where $\mathrm{DOM}(B) = \{\vec{y}\}$ and $R = R_{(\mathrm{DOM}(A) \cup \mathrm{V}(C')), \mathrm{DOM}(B)}$. It can be checked that Ψ is a witness for $\mathrm{RAN}(A) \blacktriangleright \{\tau\}$.

   (b) Suppose $\mathrm{FTV}(\tau) \subseteq \mathrm{FTV}(A)$. Then define Ψ so that

   $$\Psi(B) = (R(A), C_1[C_2[\cdots C_n \cdots]])$$

   where $\mathrm{DOM}(B) = \{\vec{y}\}$, $R = R_{(\mathrm{DOM}(A) \cup \mathrm{V}(C')), \mathrm{DOM}(B)}$, R_i renames $\mathrm{BV}(R(C'))$ to fresh names except for $R(x)$ which is renamed to y_i, and $C_i = R_i(R(C'))$. It can be checked that Ψ is a witness for $\mathrm{RAN}(A) \blacktriangleright \{\tau\}$.

6. Let Ψ' be a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}$. Let $\Psi(B) = \text{let } (A, C) = \Psi'(R^{-1}(B)) \text{ in } (R(A), R(C))$. It can be checked that Ψ is a witness for $R(\mathbb{X}) \blacktriangleright R(\mathbb{Y})$.

7. Let Ψ be a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}$. Let $\Xi(A) = \{\, x{:}\tau \mid (x{:}\tau) \in A, \tau \in \mathbb{Y} \,\}$. Define Ψ' like this:

   $$\begin{aligned} \Psi'(B) = {} & \text{let } (A, C) = \Psi(\Xi(B)) \\ & \quad \text{in let } B' = B - \Xi(B) \\ & \quad\quad \text{in let } (Y, Z) = (\mathrm{DOM}(A) \cup \mathrm{V}(C), \mathrm{DOM}(B')) \\ & \quad\quad\quad \text{in let } (A', C') = (R_{Y,Z}(A), R_{Y,Z}(C)) \\ & \quad\quad\quad\quad \text{in let } A'' = A' \cup B' \\ & \quad\quad\quad\quad\quad \text{in } (A'', C') \end{aligned}$$

   The following argument shows that Ψ' is a witness for $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$. Let $\mathrm{RAN}(B) \subseteq \mathbb{X} \cup \mathbb{Y}$. Let A, C, B', Y, Z, A', C', and A'' be defined as in the above algorithm for computing $\Psi'(B)$. It is not hard to check that $\mathrm{RAN}(A'') \subseteq \mathbb{X}$ and $\mathrm{FTV}(\mathrm{RAN}(A'')) \supseteq \mathrm{FTV}(\mathbb{X}) \cap \mathrm{FTV}(\mathrm{RAN}(B))$. Thus, it only remains to show that $A'', C' \rhd B$.

It holds that $A, C \rhd \Xi(B)$. By property 1 proved above and the fact that $R_{Y,Z}(\Xi(B)) = \Xi(B)$, it holds that $A', C' \rhd \Xi(B)$. It can be checked that $\mathrm{FTV}(\mathrm{RAN}(A)) \supseteq \mathrm{FTV}(\mathrm{RAN}(B')) \cap \mathrm{FTV}(\mathrm{RAN}(\Xi(B)))$, using the fact that $\mathrm{RAN}(B') \subseteq \mathbb{X}$ and definition 5.3. By property 4 proved above, $A' \cup B', C' \rhd \Xi(B)$. By property 2 proved above, $A' \cup B', C' \rhd \Xi(B) \cup B'$. This is exactly $A'', C' \rhd B$, the desired result.

8. Let Ψ be a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2$. It holds trivially that Ψ is also a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}_1$.

9. Let Ψ be a witness for $\mathbb{X}_1 \blacktriangleright \mathbb{Y}$. Because $\mathrm{FTV}(\mathbb{X}_1) \supseteq \mathrm{FTV}(\mathbb{X}_2) \cap \mathrm{FTV}(\mathbb{Y})$, it can be checked that Ψ is also a witness for $\mathbb{X}_1 \cup \mathbb{X}_2 \blacktriangleright \mathbb{Y}$.

10. Let Ψ_1 be a witness for $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$ and let Ψ_2 be a witness for $\mathbb{X} \cup \mathbb{Y} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$. Let Ψ be the algorithm defined as follows:

$$\begin{aligned}\Psi(B) = {} & \text{let } (A'_2, C_2) = \Psi_2(B) \\ & \quad \text{in let } (A'_1, C'_1) = \Psi_1(A'_2) \\ & \quad\quad \text{in let } (X, Y, Z) = (\mathrm{DOM}(A'_1) \cup \mathrm{BV}(C'_1), \mathrm{BV}(C_2), \mathrm{DOM}(A'_2)) \\ & \quad\quad\quad \text{in let } (A_1, C_1) = (R_{X,Y,Z}(A'_1), R_{X,Y,Z}(C'_1)) \\ & \quad\quad\quad\quad \text{in } (A_1, C_1[C_2])\end{aligned}$$

The following argument shows that Ψ is a witness for $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$. Let $\mathrm{RAN}(B) \subseteq \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}$. Let A'_1, A_1, A'_2, C'_1, C_1, C_2, X, Y, and Z be defined as in the above algorithm for computing $\Psi(B)$. Let $A_2 = A_1 \cup A'_2$, let $A_3 = A_2 \cup B$, and let $C = C_1[C_2]$. It can be checked that $\mathrm{RAN}(A_1) \subseteq \mathbb{X}$ and $\mathrm{FTV}(\mathrm{RAN}(A_1)) \supseteq \mathrm{FTV}(\mathbb{X}) \cap \mathrm{FTV}(\mathrm{RAN}(B))$. Thus, the only thing remaining to be shown is that $A_1, C \rhd B$.

By the definition of Ψ, it holds that $A'_1, C'_1 \rhd A'_2$. By property 1 proved above and the fact that $R_{X,Y,Z}(A'_2) = A'_2$, it holds that $A_1, C_1 \rhd A'_2$. By property 2 proved above, $A_1, C_1 \rhd A_2$. By the definition of Ψ, it holds that $A'_2, C_2 \rhd B$. Using property 4 proved above, it can be checked that $A_2, C_2 \rhd B$. By property 2 proved above, $A_2, C_2 \rhd A_3$. It is now sufficient to show that $A_1, C_1[C_2] \rhd A_3$, because, by property 3 proved above, that implies $A_1, C \rhd B$.

The following argument now shows that $A_1, C_1[C_2] \rhd A_3$. Observe that $A_1 \subseteq A_2 \subseteq A_3$. It can be checked that $A_1 \cup A_3$ is a type environment, that $\mathrm{FV}(C) \subseteq \mathrm{DOM}(A_1)$, that $\mathrm{BV}(C) \cap \mathrm{DOM}(A_1) = \varnothing$, and that $\mathrm{DOM}(A_3) \subseteq \mathrm{BHV}(C) \cup \mathrm{DOM}(A_1)$. Let M and A' be any term and type environment such that $C[M]$ is a term, $\mathrm{FV}(M) \subseteq \mathrm{DOM}(A_3 \cup A')$, $\mathrm{DOM}(A') \cap (\mathrm{DOM}(A_1) \cup \mathrm{BV}(C[M])) = \varnothing$, and $\mathrm{FTV}(A') \cap \mathrm{FTV}(A_3) \subseteq \mathrm{FTV}(A_1)$. Observe that $\mathrm{FTV}(A') \cap \mathrm{FTV}(A_3) \subseteq \mathrm{FTV}(A_2)$, because $\mathrm{FTV}(A_1) \subseteq \mathrm{FTV}(A_2)$. Observe also that $\mathrm{FTV}(A') \cap \mathrm{FTV}(A_2) \subseteq \mathrm{FTV}(A') \cap \mathrm{FTV}(A_3) \subseteq \mathrm{FTV}(A_1)$. Each property of definition 5.2 is now shown separately.

(a) This part proves that def. 5.2 prop. 1 holds. Let σ be such that $A_3 \cup A' \vdash M : \sigma$ is derivable.

By $A_2, C_2 \rhd A_3$ and def. 5.2 prop. 1, there is a type τ_2, a type environment E_2 and a derivation $\mathcal{D}_2$ containing the sequents $A_2 \cup A' \vdash C_2[M] : \tau_2$ and $A_3 \cup A' \cup E_2 \vdash M : \sigma$. By $A_1, C_1 \rhd A_2$ and def. 5.2 prop. 1, there is a type τ_1, a type environment E_1 and a derivation $\mathcal{D}_1$ containing the sequents $A_1 \cup A' \vdash C_1[C_2[M]] : \tau_1$ and $A_2 \cup A' \cup E_1 \vdash C_2[M] : \tau_2$.

The existence of $\mathcal{D}_1$ does not quite show the desired result, because the subderivation inside $\mathcal{D}_1$ for the subterm M does not necessarily have the type environment which satisfies def. 5.2 prop. 1. We can not merely use (the standard notion of) weakening on $\mathcal{D}_2$ and then splice the result of weakening into $\mathcal{D}_1$, because the standard notion of weakening does not preserve the internal structure of the derivation properly. Instead, we will use the more sophisticated notion of weakening of lemma 3.6. To do that, we must first establish the preconditions of lemma 3.6.

Pick some type-variable renaming R that swaps $\mathrm{FTV}(E_1)-(\mathrm{FTV}(A_2 \cup A') \cup \mathrm{FTV}(\tau_2))$ with fresh names and is otherwise the identity. The derivation $R(\mathcal{D}_1)$ contains the sequents $A_1 \cup A' \vdash C_1[C_2[M]] : R(\tau_1)$ and $A_2 \cup A' \cup R(E_1) \vdash C_2[M] : \tau_2$.

Now, we will construct a variation on $\mathcal{D}_2$ through weakening which can be spliced into $R(\mathcal{D}_1)$. Let $\mathbb{W}$ be the set of type variables generalized in $\mathcal{D}_2$. We want to show that $\mathrm{FTV}(R(E_1)) \cap \mathbb{W} = \varnothing$. Suppose for $\alpha \in \mathrm{FTV}(R(E_1))$ that $\alpha = R(\beta)$ where $\alpha \neq \beta$. Then α is fresh and not in $\mathbb{W}$. Suppose otherwise, i.e., for $\alpha \in \mathrm{FTV}(R(E_1))$ that $\alpha = R(\alpha)$. Then $\alpha \in (\mathrm{FTV}(A_2 \cup A') \cup \mathrm{FTV}(\tau_2))$. Because $\mathcal{D}_2$ is GEN-distinct (standard assumption), this means that $\alpha \notin \mathbb{W}$. Thus, by lemma 3.6, there is a derivation $\mathcal{D}_2'$ containing the sequents $A_2 \cup A' \cup R(E_1) \vdash C_2[M] : \tau_2$ and $A_3 \cup A' \cup R(E_1) \cup A_3 \cup E_2 \vdash M : \sigma$.

Form the derivation $\mathcal{D}$ by splicing the derivation $\mathcal{D}_2'$ into $R(\mathcal{D}_1)$ at the appropriate place. Let $\tau = R(\tau_1)$ and $E = R(E_1) \cup E_2$. It holds that $\mathcal{D}$ contains $A_1 \cup A' \vdash C_1[C_2[M]] : \tau$ and $A_3 \cup A' \cup E \vdash M : \sigma$, which is the desired result.

(b) This part proves that def. 5.2 prop. 2 holds. Let τ be some type and let $\mathcal{D}$ be a derivation containing $A_1 \cup A' \vdash C_1[C_2[M]] : \tau$.

By $A_1, C_1 \rhd A_2$ and def. 5.2 prop. 2, there exist a type σ_1, a type environment E_1, and a type-variable renaming R_1 satisfying $R_1(A_1 \cup A') = A_1 \cup A'$ such that $\mathcal{D}$ contains this sequent:

$$A_1 \cup A' \cup R_1(A_2) \cup E_1 \vdash C_2[M] : \sigma_1 \tag{8}$$

The fact that $A_2, C_2 \rhd A_3$ can not be directly used. Some renaming must be applied first to match the type variables used in sequent (8). Let R_2 be a type-variable renaming that swaps $(\mathrm{FTV}(R_1(A_3)) - \mathrm{FTV}(R_1(A_2)))$ with fresh type variables and is the identity on other type variables in $\mathcal{D}$. By lemma 5.4 prop. 1 it holds that $R_2(R_1(A_2)), C_2 \rhd R_2(R_1(A_3))$. Because $R_2(\mathrm{FTV}(R_1(A_3)) - \mathrm{FTV}(R_1(A_2)))$ are all fresh variables, it holds that $\mathrm{FTV}(A' \cup E_1) \cap \mathrm{FTV}(R_2(R_1(A_3))) = \mathrm{FTV}(A' \cup E_1) \cap \mathrm{FTV}(R_1(A_2)) \subseteq \mathrm{FTV}(R_1(A_2))$. Observe that $R_2(R_1(A_2)) = R_1(A_2) = A_1 \cup R_1(A_2)$. Thus, by def. 5.2 prop. 2, interpreting $A' \cup E_1$ as the extra type environment, there exist a type σ_2, a type environment E_2, and a type-variable renaming R_3 satisfying $R_3(A' \cup E_1 \cup R_2(R_1(A_2))) = A' \cup E_1 \cup R_2(R_1(A_2))$ such that $\mathcal{D}$ contains this sequent:

$$A' \cup E_1 \cup R_2(R_1(A_2)) \cup R_3(R_2(R_1(A_3))) \cup E_2 \vdash M : \sigma_2 \tag{9}$$

Observe that $R_2(R_1(A_1)) = A_1$. Thus, $A_1 \subseteq R_2(R_1(A_2))$. Note that $R_3(R_2(R_1(A_2))) = R_2(R_1(A_2))$. Thus, $R_2(R_1(A_2)) \subseteq R_3(R_2(R_1(A_3)))$. Let $R = R_1 \circ R_2 \circ R_3$ and let $E = E_1 \cup E_2$. The sequent (9) may be rewritten as:

$$A_1 \cup A' \cup R(A_3) \cup E \vdash M : \sigma_2$$

This is the desired result.

11. It is given that $\mathbb{X} \blacktriangleright \mathbb{Y}_1$ and $\mathbb{X} \blacktriangleright \mathbb{Y}_2$ and $\mathrm{FTV}(\mathbb{Y}_1) \cap \mathrm{FTV}(\mathbb{Y}_2) \subseteq \mathrm{FTV}(\mathbb{X})$.

By lemma 5.4 (9), it holds that $\mathbb{X} \cup \mathbb{Y}_1 \blacktriangleright \mathbb{Y}_2$. By lemma 5.4 (7), it holds that $\mathbb{X} \cup \mathbb{Y}_1 \blacktriangleright \mathbb{X} \cup \mathbb{Y}_1 \cup \mathbb{Y}_2$. By lemma 5.4 (7), it holds that $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}_1$. By lemma 5.4 (10), it holds that $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}_1 \cup \mathbb{Y}_2$. By lemma 5.4 (8), it holds that $\mathbb{X} \blacktriangleright \mathbb{Y}_1 \cup \mathbb{Y}_2$. □

Definition 5.5. Inspection of the proof of lemma 5.4 reveals that the proof of each property has constructive content, i.e., algorithms that take witnesses (themselves algorithms) for various statements and return witnesses for new statements. To support building larger algorithms, the following algorithms are identified:

1. Let Π_{alt} be the algorithm (from proof of property 11) such that if Ψ'_i is a witness of $\mathbb{X}' \blacktriangleright \mathbb{Y}'_i$ for $i \in \{1, 2\}$ and $\mathrm{FTV}(\mathbb{Y}'_1) \cap \mathrm{FTV}(\mathbb{Y}'_2) \subseteq \mathrm{FTV}(\mathbb{X}')$, then $\Pi_{\mathrm{alt}}(\Psi'_1, \Psi'_2)$ is a witness for $\mathbb{X}' \blacktriangleright \mathbb{Y}'_1 \cup \mathbb{Y}'_2$.

2. Let Π_R be the algorithm (from proof of property 6) such that if Ψ' is a witness of $\mathbb{X}' \blacktriangleright \mathbb{Y}'$ and R is a type-variable renaming, then $\Pi_R(\Psi')$ is a witness of $R(\mathbb{X}') \blacktriangleright R(\mathbb{Y}')$.

3. Let Π_{self} be the algorithm (from proof of property 7) such that if Ψ' is a witness of $\mathbb{X}' \blacktriangleright \mathbb{Y}'$, then $\Pi_{\mathrm{self}}(\Psi')$ is a witness for $\mathbb{X}' \blacktriangleright \mathbb{X}' \cup \mathbb{Y}'$.

4. Let Π_{chain} be the algorithm (from proof of property 10) such that if Ψ' is a witness of $\mathbb{X}' \blacktriangleright \mathbb{X}' \cup \mathbb{Y}'$ and Ψ'' is a witness of $\mathbb{X}' \cup \mathbb{Y}' \blacktriangleright \mathbb{X}' \cup \mathbb{Y}' \cup \mathbb{Z}'$, then $\Pi_{\mathrm{chain}}(\Psi', \Psi'')$ is a witness for $\mathbb{X}' \blacktriangleright \mathbb{X}' \cup \mathbb{Y}' \cup \mathbb{Z}'$. □

Lemma 5.6. *Let $\mathbb{X}, \mathbb{Y} \subseteq \mathbb{T}$. Let $\mathbb{Y}_1, \mathbb{Y}_2, \ldots$ be an infinite sequence of finite decidable type sets approximating $\mathbb{Y}$ such that $\mathbb{Y}_i \subseteq \mathbb{Y}_{i+1}$ and $\bigcup_{1 \le i \le \omega} \mathbb{Y}_i = \mathbb{Y}$. Let Θ be a computable function such that if Ψ is a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}_i$ then $\Theta(\Psi, i)$ is a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}_{i+1}$. Let $\mathbb{X} \blacktriangleright \mathbb{Y}_1$. Then it follows that $\mathbb{X} \blacktriangleright \mathbb{Y}$.* □

Proof. Let Ψ_1 be a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}_1$. Let $\Psi(B) = \Psi'(\Psi_1, 1, B)$ where Ψ' is:

$$\Psi'(\Psi'', i, B) = \text{if } \mathrm{RAN}(B) \subseteq \mathbb{Y}_i \text{ then } \Psi''(B) \text{ else } \Psi'(\Theta(\Psi'', i), i+1, B)$$

It can be checked that Ψ is a witness for $\mathbb{X} \blacktriangleright \mathbb{Y}$. □

6 Undecidability of Typability

This section uses the machinery of section 5 to prove that TC is reducible to Typ, thus proving Typ to be undecidable.

Definition 6.1 (*height* **and** *parheight*)**.** The auxiliary metric $height(\tau)$ measures the height of the type τ viewing τ as a tree, ignoring quantifiers, and letting a single type variable be of height 1. The metric *parheight* measures the heights of the parameter subtypes of a type. For a type τ such that $\tau = \forall\overrightarrow{\alpha_1}.\rho_1 \to \cdots \forall\overrightarrow{\alpha_k}.\rho_k \to \forall\overrightarrow{\alpha_{k+1}}.\beta$ for $k \ge 0$ and where $\beta \in \mathbb{V}$, define $parheight(\tau)$ to be the maximum of $\{height(\rho_1), \ldots, height(\rho_k), 0\}$. □

Definition 6.2 ($\mathbb{B}$, $\mathbb{U}$, $\mathbb{C}$, $\mathbb{T}(k)$, $\mathbb{U}(k)$, $\mathbb{C}(k)$)**.** Here are definitions of a number of sets of types within $\mathbb{T}$ which will be used throughout the rest of this section.

$$\begin{array}{rcl}
\mathbb{B} & = & \{\bot\} \cup \{\, \alpha \to \alpha,\ \alpha \mid \alpha \in \mathbb{V}\,\} \cup \mathbb{V} \\
\mathbb{U} & = & \{\, \forall.\tau \mid \tau \in \mathbb{T} \text{ and } \tau \text{ is open}\,\} \\
\mathbb{C} & = & \{\, \forall.\tau \mid \tau \in \mathbb{T}\,\} \\
\mathbb{T}(k) & = & \{\, \tau \mid \tau \in \mathbb{T} \text{ and } parheight(\tau) \le k\,\} \\
\mathbb{U}(k) & = & \mathbb{U} \cap \mathbb{T}(k) \\
\mathbb{C}(k) & = & \mathbb{C} \cap \mathbb{T}(k)
\end{array}$$

The set $\mathbb{B}$ will be used as the basis of the a long chain of results culminating in theorem 6.14. $\mathbb{U}$ is the set of universal types and $\mathbb{C}$ is the set of closed types. $\mathbb{T}(k)$, $\mathbb{U}(k)$, and $\mathbb{C}(k)$ are the subsets respectively of $\mathbb{T}$, $\mathbb{U}$, and $\mathbb{C}$ which contain only types of *parheight* $\le k$. Observe that $\{\bot\} = \mathbb{U}(0) = \mathbb{C}(0) \subset \mathbb{B}$. □

Lemma 6.3. *There is a λ-term J such that J and the type environment $\varnothing$ induce the invariant type assumptions $(v{:}\forall\alpha.\alpha)$ and $(x{:}\alpha \to \alpha)$.* □

Proof. Because the techniques used in proving this lemma are also required for the later lemmas, a thorough explanation is given here to aid in later understanding.

Throughout this proof, view a parse tree notation for types as being interchangeable with the regular notation, using the following simple correspondence. View the type symbol " $\rightarrow$ " as an internal tree node with two children and a quantification "$\forall\alpha$" as a node label. A *left-going path* in a tree is a path containing no branches that descend to the right. The *left-most path of a type* is the unique left-going path from the root of the type to a leaf. A quantifier labelling a tree node *owns a path* if the type reached by following that path from the node with the quantifier is exactly the quantified variable. A "$\bullet$" is used to indicate the presence of a node without specifying whether it is an internal node or a leaf.

Let the λ-term J be as follows:

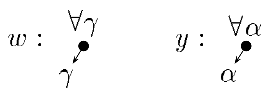
$$J \equiv \lambda v.(\lambda y.\lambda z.v(yy)(yz))(\lambda x.\mathbf{K}x(x(xv)))(\lambda w.ww)$$

It is easy to check that there is a typing of J with the following global type environment:

$$A = \{v{:}\bot, y{:}\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha), z{:}\bot \rightarrow \bot, x{:}\alpha \rightarrow \alpha, w{:}\bot\}$$

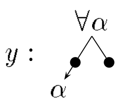
We now discover a very useful fact. It is the case that in all derivations for J, the λ-term variables x, y, and v must be assigned essentially the same types. In other words, there are invariant type assumptions for these variables. The analysis from here through the end of the proof leads to this conclusion. Assume $\mathcal{D}$ is a typing for J and we will prove that it has the desired properties.

In order for the subterm (ww) to be typed, the left-most path in the final derived type of the first occurrence of w must be one edge longer than the left-most path in the final derived type of the second occurrence of w. If the left-most path in the assumed type for w is of length k and the type variable at the leaf at the end of this path is free or quantified at some position other than the root of the type, then the length of the leftmost path in any derived type for w must also be k. Thus, the left-most path in the assumed type for w must be owned by a quantifier at the root of this type. The same holds for y, due to the subterm (yy). This is depicted as follows, where the arrow indicates the existence of the variable to which it points at the end of a left-going path:

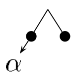


Because the abstraction over y is applied to the abstraction over x, the type assumed for y must be of height larger than 1, which is depicted like this:

Combining the two diagrams gives this result:



The type assumed for y must equal the final type derived for $(\lambda x.\mathbf{K}x(x(xv)))$. The only way there can be quantification at the root of the final derived type of this abstraction is if the GEN rule is applied, because the type derived for an abstraction by the ABS rule has no outermost quantifiers. Thus, an earlier type derived for the abstraction over x looks like this, where α is a free variable:

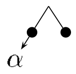
$(\lambda x.\mathbf{K}x(x(xv))) :$  (10)

Now temporarily suppose that the type assumed for x did not match the following pattern, where the type variable α is free:

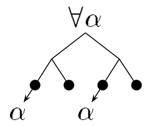
x :  (11)

If the type assumption depicted in (11) did not hold, then the type variable at the end of the leftmost path in the assigned type of x would be closed within that type. If that were the case, then the derived type depicted in (10) could never happen. Hence, (11) must depict the type assumption for x in any derivation.

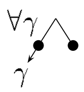
Due to the subterm (xv), the type assumed for x must be of height larger than 1:

x : 

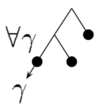
Thus, the type assumption for y matches this pattern:

y : 

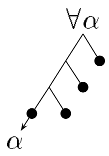
Considering again the type assumed for w and how this type must be embedded in the type assumed for z shows this:

z :  (12)

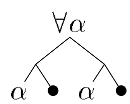
Consider the subterm (yz). The types assumed for y and z must be instantiated so that the left subtree of the instantiation of the type of y matches the instantiation of the type of z. Every instantiation of the type of z will match the pattern given in (12) for the type assumed for z. Thus, the type assumed for y must be instantiated to match this pattern:

y :  (13)

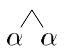
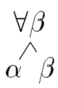
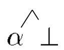
Now suppose the leftmost path in the type assumed for y is at least 3 edges long, in other words suppose this:

y : 

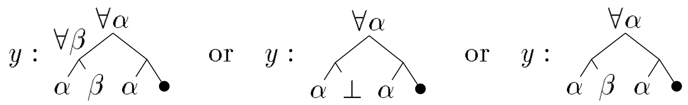
If this were the case, then the instantiation of the type assumed for y could never match the pattern in (13) because a quantifier owning the leftmost path can not be inserted at the necessary spot in the type in (13) by instantiation. Thus, the leftmost path in the type of y must be exactly 2 edges long, and thus the leftmost path in the type of x must be 1 edge long. Thus, both of these type assumptions must hold, where α in the type of x is free:

y :  x : 

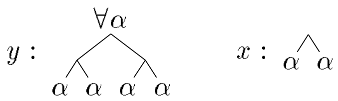
In the subterm $(x(xv))$, the final derived type for (xv) must be exactly α. Thus, the type assumed for x must match one of these patterns, where α is free:

x :  or x :  or x : 

Suppose it were the second or third case, i.e., $x\colon \forall\beta.(\alpha \to \beta)$ or $x\colon \alpha \to \bot$. Then the type assumed for y would have to match one of these patterns:



However, then the subterm (yy) could not be typed. Thus, both of these type assumptions hold exactly in all typings of J:



All that is left is to show that the type $\bot$ (i.e., the type $\forall\alpha.\alpha$) must be assumed for v. In the subterm (xv), the final derived type of v must be exactly α. In the subterm $(v(yy))$, the final derived type of v must be an $\to$-type. The only possible assumed type for v that can yield both of these derived types is $\bot$. This is the desired result. $\square$

Lemma 6.4 ($\varnothing \blacktriangleright \{\bot, \alpha \to \alpha, \forall.\alpha \to \beta \to \alpha\}$ in the $\boldsymbol{\lambda}I$-calculus). *There is a λ-term J_I of the λI-calculus such that the type environment $\varnothing$ and J_I induce the invariant type assumptions $(r{:}\forall\alpha.\alpha)$, $(x_1{:}\alpha \to \alpha)$, and $(k{:}\forall.\alpha \to \beta \to \alpha)$.* $\square$

Proof. Let the λ-term J_I be defined as follows:

$$\begin{array}{rcl}
C_i & \equiv & ((\lambda y_i.r(y_i y_i)(y_i(\lambda w_i.w_i w_i))(y_i(\lambda z_i.\lambda v_i.z_i v_i))) \\
 & & (\lambda x_i.r(x_i(x_i r))\square)) \\
H & \equiv & ((\lambda h.r(x_3(h x_1 x_2(x_1 r)(x_2 r)))(x_1(h x_2 x_3(x_2 r)(x_3 r)))\square) \\
 & & (\lambda a_1.\lambda b_1.\lambda c_1.\lambda d_1.r(a_1(a_1 c_1))(b_1(b_1 d_1)))) \\
G & \equiv & ((\lambda g.r(x_1(g x_1 x_2(x_1 r)(x_2 r)))(x_2(g x_2 x_1(x_2 r)(x_1 r)))\square) \\
 & & (\lambda a_2.\lambda b_2.\lambda c_2.\lambda d_2.a_2(h a_2 b_2 c_2 d_2))) \\
L & \equiv & ((\lambda k.r(x_1(k(x_1 r)(x_2 r)))(x_2(k(x_2 r)(x_1 r)))\square) \\
 & & (grr)) \\
J_I & \equiv & (\lambda r.C_1[C_2[C_3[H[G[L[r]]]]]])
\end{array}$$

Inspection reveals that J_I is a term of the λI-calculus, because every variable that is λ-bound in J_I occurs at least once as a subterm. Inspection also reveals that J_I is a closed term and thus all of its free variables (none) occur within the domain of the type environment $\varnothing$. It can be checked that the λ-term J_I is typable using the following global type environment:

$$\begin{array}{llll}
r:\bot, & y_i:\ \forall\alpha.((\alpha \to \alpha) \to (\alpha \to \alpha)), & & x_i:\alpha_i \to \alpha_i \\
w_i:\bot, & z_i:\alpha \to \alpha, & v_i:\alpha & \\
a_1:\beta_1 \to \beta_1, & b_1:\beta_2 \to \beta_2, & c_1:\beta_1, & d_1:\beta_2 \\
a_2:\beta_3 \to \beta_3, & b_2:\beta_4 \to \beta_4, & c_2:\beta_3, & d_2:\beta_4 \\
\multicolumn{4}{l}{h:\ \forall\alpha.(\alpha \to \alpha) \to \forall\beta.(\beta \to \beta) \to \alpha \to \beta \to \forall\gamma.\gamma} \\
\multicolumn{4}{l}{g:\ \forall\alpha.(\alpha \to \alpha) \to \forall\beta.(\beta \to \beta) \to \alpha \to \beta \to \alpha} \\
\multicolumn{4}{l}{k:\ \forall.\alpha \to \beta \to \alpha}
\end{array}$$

It turns out that little variation is possible from the types that were just given above. Except for the types assigned to the variables v_i, w_i, y_i, and z_i, which can vary, the only other possible variations in this type environment are in the naming of the free variables, which may be renamed in a one-to-one manner, and in the types assigned to the λ-bound variables h and

g, where the quantifiers over the type variables β and γ may occur in any position whose scope encloses their depicted position. Thus, the term J_I and the empty type environment induce the desired invariant type assumption $k{:}\forall.\alpha \to \beta \to \alpha$.

The explanation for the limited range of possible variation proceeds as follows. At the base of the construction, the context C_i is simply a variation on the λ-term J from lemma 6.3 which works in the λI-calculus. The triple nesting of contexts $C_1[C_2[C_3]]$ gives three invariant type assumptions $x_i{:}\alpha_i \to \alpha_i$ for $i \in \{1, 2, 3\}$ where the type variables α_1, α_2, and α_3 must be distinct because the GEN rule must be used for each of them in three separate places. The context H uses the types of x_1, x_2, and x_3 to constrain the size of the type of h and to force certain leaves in the type of h to mention the same type variable. The context G further refines the type of h into the type of g which does not have a troublesome quantifier in the rightmost position. Finally, the context L fixes the desired type for the variable k. Note that at each stage, the types of x_1, x_2, and x_3 must be used to ensure the type being constructed does not go out of control through unwanted used of INST. The proofs of the various facts just mentioned are similar to the reasoning used in the proof of lemma 6.3 and are left to the reader.

It is worth observing that under β-reduction, an application kMN inserted inside the above-constructed context will be eventually replaced by the λ-term $(r(r(r(rM))(r(rN))))$. Because the type of r is $\perp$, no abstraction can ever be substituted for r in reduction. Thus, the application kMN will have the type of M, but can never reduce to M. □

Remark 6.5 (Adapting Proofs for λI-calculus). Lemma 6.4 is an alternative to lemma 6.3 which can be used to perform all of the proofs entirely within the λI-calculus. The important thing to note is that the invariant type assumption $(k : \forall.\alpha \to \beta \to \alpha)$ allows k to be used to simulate the K combinator, at least as far as typability is concerned. By replacing every use of the combinator K in the proofs by a free variable k which must be assigned the type $\forall.\alpha \to \beta \to \alpha$, the proofs can be adapted to work in the λI-calculus. This adaptation, which is left to the reader to perform, involves the following steps. Let $C_J \equiv (\lambda r.C_1[C_2[C_3[H[G[L]]]]])$ using the contexts defined in the proof of lemma 6.4. This differs from J_I only in the presence of a hole in C_J where there is an occurrence of r in J_I. It can be shown that $\varnothing, C_J \triangleright \{k{:}\forall.\alpha \to \beta \to \alpha\}$ and that $\varnothing \blacktriangleright \{\forall.\alpha \to \beta \to \alpha\}$. Then the entire sequence of proofs in sections 5 and 6 ending in theorem 6.14 which shows $\varnothing \blacktriangleright \mathbb{T}$ can be replayed, using a free variable k of type $\forall.\alpha \to \beta \to \alpha$ in place of K, to show that $\{\forall.\alpha \to \beta \to \alpha\} \blacktriangleright \mathbb{T}$ and hence that $\varnothing \blacktriangleright \mathbb{T}$. The key point is that this will yield a witness for $\varnothing \blacktriangleright \mathbb{T}$ which produces contexts in which every bound variable has at least one occurrence, regardless of what is later placed in the hole. □

Lemma 6.6. *The type environment $A = \{x{:}\alpha \to \alpha, v{:}\perp\}$ and the λ-term $M \equiv ((\lambda y.y)(xv))$ induce the invariant type assumption $(y{:}\alpha)$.* □

Proof. Trivial. □

Lemma 6.7 ($\varnothing \blacktriangleright \mathbb{B}$). *The empty set of types $\varnothing$ induces the set of types $\mathbb{B}$.* □

Proof. Let α_1, α_2, ... be an enumeration of $\mathbb{V}$. Let $\mathbb{B}_i = \{\perp\} \cup \{\, \alpha_j \to \alpha_j, \alpha_j \mid 1 \leq j \leq i \,\}$. Observe the following chain of reasoning:

$\varnothing \blacktriangleright \{\perp\}$	lemma 6.3, lemma 5.4 (5)
$\varnothing \blacktriangleright \{\alpha_1 \to \alpha_1\}$	lemma 6.3, lemma 5.4 (5)
$\varnothing \blacktriangleright \{\perp, \alpha_1 \to \alpha_1\}$	lemma 5.4 (11)
$\{\perp, \alpha_1 \to \alpha_1\} \blacktriangleright \{\alpha_1\}$	lemma 6.6, lemma 5.4 (5)
$\{\perp, \alpha_1 \to \alpha_1\} \blacktriangleright \{\perp, \alpha_1 \to \alpha_1, \alpha_1\}$	lemma 6.6, lemma 5.4 (7)
$\varnothing \blacktriangleright \{\perp, \alpha_1 \to \alpha_1, \alpha_1\} = \mathbb{B}_1$	lemma 5.4 (10)

Let Ψ_1 be a witness for $\varnothing \blacktriangleright \mathbb{B}_1$. Let $R_{i,j}$ be a type-variable renaming which swaps α_i with α_j. Let $\Theta(\Psi', i) = \Pi_{\mathrm{alt}}(\Psi', \Pi_{R_{1,i}}(\Psi_1))$. Using lemma 5.4 properties 6 and 11, it holds that if Ψ' is a witness for $\varnothing \blacktriangleright \mathbb{B}_i$, then $\Theta(\Psi', i)$ is a witness for $\varnothing \blacktriangleright \mathbb{B}_{i+1}$. Invoking lemma 5.6 with Ψ_1 and Θ yields a witness Ψ for $\varnothing \blacktriangleright \mathbb{B}$. □

Lemma 6.8. *For any type τ in $\mathbb{U}(k+1)$, there exist a type environment A such that* $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{U}(k)$*, a λ-term M such that* $\mathrm{FV}(M) = \mathrm{DOM}(A)$*, and a λ-term variable $x \in \mathrm{BV}(M)$ such that A and M induce the invariant type assumption $(x{:}\tau)$.* □

Proof. Because this proof is long enough to take many pages, we will set off a number of nested claims with somewhat self-contained arguments and interleave them with the main course of argument.

Let τ be the following type in $\mathbb{U}(k+1)$:

$$\tau = \forall.\rho_1 \to \cdots \to \rho_c \to \alpha_g$$

Let $\mathbb{S} = \{\alpha_1, \ldots, \alpha_n\}$ be a set of type variables such that $n \geq 2$, $1 \leq g \leq n$, and $\mathbb{S} \supseteq \mathrm{FTV}(\rho_i)$ for $1 \leq i \leq c$. (There may need to be an extra type variable in $\mathbb{S}$ to meet the requirement that $n \geq 2$.) Observe that τ is a closed type (because $\tau \in \mathbb{U}(k+1)$) and $\rho_1, \ldots, \rho_c$ are open types.

From τ, a type environment A and a λ-term M having the desired properties will be constructed. In the course of this definition, a number of types, sets of types, λ-terms, and sets of λ-terms will be defined.

Define the following sets of types:

$$\begin{aligned}
\mathbb{R} &= \{\rho_1, \ldots, \rho_c\} \\
\mathbb{X} &= \{\, \pi \mid \exists \rho \in \mathbb{R}.\, \pi \subseteq \rho \,\} \\
\mathbb{Y} &= \{\, \varphi \to \alpha_1 \mid \exists \pi.\, (\pi \to \varphi) \in \mathbb{X} \,\} \\
\mathbb{Z} &= \{\, \alpha_i \to \alpha_i \mid 1 \leq i \leq n \,\} \\
\mathbb{W} &= \mathbb{R} \cup \mathbb{S} \cup \mathbb{X} \cup \mathbb{Y} \cup \mathbb{Z}
\end{aligned}$$

Let $\mu_1, \ldots, \mu_r$ be a sequence of members of $\mathbb{W}$ with the following properties. First, $\vec{\mu}$ mentions every member of $\mathbb{W}$ one or two times; it may mention a type μ more than once only if $\mu \in (\mathbb{S} \cup \mathbb{Z}) \cap \mathbb{R}$. Second, the subsequence $\mu_1, \ldots, \mu_n$ is exactly $\alpha_1, \ldots, \alpha_n$, the subsequence $\mu_{n+1}, \ldots, \mu_{2n}$ is exactly $\alpha_1 \to \alpha_1, \ldots, \alpha_n \to \alpha_n$, and the subsequence $\mu_{r-c+1}, \ldots, \mu_r$ is exactly $\rho_1, \ldots, \rho_c$. Third, it must hold that $n + c \leq r$. (This ensures that the initial subsequence $\alpha_1, \ldots, \alpha_n$ does not overlap with the final subsequence $\mu_{r-c+1}, \ldots, \mu_r$.)

Claim 6.8.1. *For $\sigma \in \mathbb{R} \cup \mathbb{X} \cup \mathbb{Y}$, it holds that $\mathit{parheight}(\sigma) \leq k$.*

Proof of claim 6.8.1. Remember the definitions of *parheight* and *height* from definition 6.1. For arbitrarily chosen type σ, it hold that $\mathit{parheight}(\sigma) \leq \mathit{height}(\sigma) - 1$. Thus, it is sufficient to show for $\sigma \in \mathbb{R} \cup \mathbb{X} \cup \mathbb{Y}$ that $\mathit{height}(\sigma) \leq k+1$. Because $\tau \in \mathbb{U}(k+1)$, it holds that $\mathit{parheight}(\tau) \leq k+1$. By the definition of *parheight*, for each $\rho_i \in \mathbb{R}$, it holds that $\mathit{height}(\rho_i) \leq k+1$. Because every member of $\mathbb{X}$ occurs within some member of $\mathbb{R}$, for every $\psi \in \mathbb{X}$ it holds that $\mathit{height}(\psi) \leq k+1$. By the definition of $\mathbb{Y}$, for every type $\theta \in \mathbb{Y}$, it then holds that $\mathit{height}(\theta) \leq k+1$. □

The type environment A is now defined on the λ-term variables $a, b_1, \ldots, b_r$. First, let $A(a) = \bot$. Then, for each $i \in \{1, \ldots, r\}$, define $A(b_i)$ in terms of the type $\mu_i \in \mathbb{W}$ as follows:

$$A(b_i) = \begin{cases} \mu_i & \text{if } \mu_i \in (\mathbb{S} \cup \mathbb{Z}), \\ \forall.\alpha_1 \to \cdots \to \alpha_n \to \mu_i & \text{otherwise.} \end{cases}$$

Claim 6.8.2. *It holds that* $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{U}(k)$.

Proof of claim 6.8.2. Observe that $A(a) = \bot \in \mathbb{B}$. The two cases in the definition of $A(b_i)$ for $1 \leq i \leq r$ are handled separately. In the first case, $A(b_i) = \mu_i \in (\mathbb{S} \cup \mathbb{Z}) \subset \mathbb{B}$. In the second case, $A(b_i) = \forall.\alpha_1 \to \cdots \to \alpha_n \to \mu_i$. By the definition of $\mathbb{W}$ and the fact that $\mu_i \in \mathbb{W} - (\mathbb{S} \cup \mathbb{Z})$, it holds that $\mu_i \in (\mathbb{R} \cup \mathbb{X} \cup \mathbb{Y})$. By claim 6.8.1, it holds that $\mathit{parheight}(\mu_i) \leq k$. If $k = 0$, then $\mu_i \in \mathbb{S}$, a contradiction. If $k \geq 1$, then it holds that $\mathit{parheight}(\forall.\alpha_1 \to \cdots \to \alpha_n \to \mu_i) \leq k$, which in turn implies $A(b_i) \in \mathbb{U}(k)$. □

Now define the pieces of the λ-term M. First, for $1 \leq i \leq r$ and $h \in \{0,1\}$ the λ-term Q_i^h will be defined. In this definition, let the function m on natural numbers be defined so that $m(i,h) = (((i+h)-1) \mod n) + 1$. ("mod" could be used more directly except that the set $\mathbb{S}$ is indexed starting with 1.)

$$Q_i^h \equiv \begin{cases} b_{m(j,h)} & \text{if } \mu_i = \alpha_j, \\ b_{n+m(j,h)} & \text{if } \mu_i = \alpha_j \to \alpha_j, \\ (b_i b_{m(1,h)} \cdots b_{m(n,h)}) & \text{otherwise.} \end{cases}$$

Claim 6.8.3. *Using any extension of the type environment A, for $1 \leq i \leq r$ and $h \in \{0,1\}$, it holds that* $\mathrm{FDT}(Q_i^h) = R^h(\mu_i)$ *where* $R^h = \{\, \alpha_j \mapsto \alpha_{m(j,h)} \mid 1 \leq j \leq n \,\}$. *(Note that R^0 is the identity substitution.)*

Proof of claim 6.8.3. Recall that the first n elements of $\vec{\mu}$ are $\alpha_1, \ldots, \alpha_n$ (the elements of $\mathbb{S}$) and that the second n elements of $\vec{\mu}$ are $\alpha_1 \to \alpha_1, \ldots, \alpha_n \to \alpha_n$ (the elements of $\mathbb{Z}$). First, consider the case where $Q_i^h \equiv b_j$ for some $j \leq 2n$. It is clear that the GEN and INST rules can not be used because $A(b_j)$ is open so that $\mathrm{FDT}(Q_i^h) = A(b_j)$. It can be checked that this is the type specified by the claim. Second, consider the case where $Q_i^h \equiv (b_i b_{m(1,h)} \cdots b_{m(n,h)})$. Remember that $A(b_i) = \forall.\alpha_1 \to \cdots \to \alpha_n \to \mu_i$. Above we have just shown that $\mathrm{FDT}(b_{m(j,h)}) = \alpha_{m(j,h)}$. Without loss of generality, assume that the subderivation for the subterm Q_i^h obeys the INST-before-GEN property. Thus, in typing b_i, the INST rule must be used to eliminate all quantifiers. The GEN rule can not be used after that, because it would prevent typing the application $(b_i b_{m(1,h)})$. There can not be any subsequent uses of GEN or INST without violating the INST-before-GEN property. Thus, it must hold that $\mathrm{FDT}(b_i) = \alpha_{m(1,h)} \to \cdots \to \alpha_{m(n,h)} \to S(\mu_i)$ for some substitution S. The only possible S to achieve this result is R^h. $\square$

Now, define the set $\vec{P}$ of λ-terms:

$$\vec{P} = \{\, (z_h(z_i z_j)) \mid \exists \theta, \pi \in \mathbb{W}.\ \mu_i = \mu_j \to \theta \ \text{ and } \ \mu_h = \theta \to \pi \,\}$$

Let $P_1, \ldots, P_s$ be an arbitrarily chosen enumeration of $\vec{P}$. Most of the pieces of the λ-term M are now present.

Define the λ-term M using the subterms defined above. The variables $z_1, \ldots, z_r$ which are free in the subterms $P_1, \ldots, P_s$ will be captured by bindings within M. The free variables of the subterms $Q_1^0, \ldots, Q_r^0$ and $Q_1^1, \ldots, Q_r^1$ will be free in M.

$$\begin{aligned}
M^j &\equiv Q^j_{n+g}(yQ^j_1 \cdots Q^j_r) \\
N^j &\equiv Q^j_{n+g}(xQ^j_{r-c+1} \cdots Q^j_r) \\
N &\equiv (\lambda z_1 \ldots z_r.\mathbf{K} z_g(aP_1 \ldots P_s)) \\
N' &\equiv ((\lambda u.a(ub_{n+1})(u(x\overbrace{a \cdots a}^{c-1}))) \\
&\qquad (\lambda v.a(b_{n+1}(va)))) \\
M &\equiv ((\lambda y.aM^0M^1 \\
&\qquad\qquad ((\lambda x.aN^0N^1N') \\
&\qquad\qquad\ (y\overbrace{a \cdots a}^{r-c}))) \\
&\qquad N)
\end{aligned}$$

Inspection reveals that $\mathrm{FV}(M) = \{a, b_1, \ldots, b_r\} = \mathrm{DOM}(A)$.

To help readers (including the author!) better remember the above definitions, we list for reference the definitions of the natural numbers c, g, k, n, r, and s, which might otherwise be hard to find in the details:

c The largest index of the components $\vec{\rho}$ of τ mentioned at the beginning of the proof which all belong to the set $\mathbb{R}$.

g The index of $\alpha_g \in \mathbb{S}$ where α_g is the rightmost component of τ.

k The parameter in $\mathbb{U}(k)$ which limits the value of *parheight* for members of the range of the type environment A which do not belong to $\mathbb{B}$.

n The largest index of the set $\mathbb{S}$ of $\forall$-bound type variables in τ.

r The largest index in the special enumeration $\vec{\mu}$ of types in $\mathbb{W}$.

s The largest index in the enumeration of $\vec{P}$.

The type environment A and the λ-term M have now been defined and it has been shown that $\mathrm{FV}(M) = \mathrm{DOM}(A)$ and $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{U}(k)$. Let R' be a type-variable renaming which maps the members of $\mathbb{S}$ to fresh names. It can be easily checked that M is typable under the type environment A with a global type environment including the following additional type assumptions:

$$\begin{array}{ll} x : \tau & \\ y : \forall.\mu_1 \to \cdots \to \mu_r \to \mu_g & \\ z_i : R'(\mu_i) & \text{for } 1 \leq i \leq r \\ u : \forall\beta.\forall\gamma.(\beta \to \alpha_1) \to \gamma & \\ v : \beta \to \alpha_1 & \end{array}$$

The rest of the proof of this lemma shows the final condition necessary for A and M to induce the invariant type assumption $(x{:}\tau)$: for any typing $\mathcal{D}$ for M with final type environment A it holds that $(\mathcal{G}(\mathcal{D}))(x) = R(\tau)$ where R is a type-variable renaming such that $R(A) = A$. In fact, no renaming is necessary because the type τ is closed.

Throughout the rest of the proof, consider some arbitrarily chosen typing $\mathcal{D}$ of the λ-term M using the type environment A. For any term variable w that occurs in the λ-term M, remember that $\mathcal{G}(w)$ stands for the type assumed for w in the derivation $\mathcal{D}$, i.e., $\mathcal{G}(w) = (\mathcal{G}(\mathcal{D}))(w)$. If we refer to a subterm's "typability", we mean whether it can be typed using the type assumptions of $\mathcal{G}(\mathcal{D})$.

Claim 6.8.4. *The types $\mathcal{G}(x)$ and $\mathcal{G}(y)$ are closed. Furthermore, no quantifiers are embedded in these types to the left of an arrow or to the right of more than j arrows, where $j = c$ in the case of $\mathcal{G}(x)$ and $j = r$ in the case of $\mathcal{G}(y)$ and where a quantifier is "to the right" (resp. left) of an arrow if it is inside the right (left) argument of the arrow.*

Proof of claim 6.8.4. We consider only $\mathcal{G}(y)$ as the case of $\mathcal{G}(x)$ uses exactly the same reasoning.

It is impossible for there to be a free type variable α in the type $\mathcal{G}(y)$ at a position exactly to the right of i arrows where $i < r$, because then $(yQ_1^0 \ldots Q_i^0)$ would have type α, which would make $(yQ_1^0 \ldots Q_i^0 Q_{i+1}^0)$ untypable.

Now, observe that for $i \in \{1, \ldots, r\}$, by claim 6.8.3 the final derived types for Q_i^0 and Q_i^1 are both open (have zero bound variables) and yet do not match in any position.

Suppose there were a free type variable in the component of $\mathcal{G}(y)$ to the right of r arrows. Then this free type variable would still be present in $\mathrm{FDT}(yQ_1^h \ldots Q_r^h)$ for $h \in \{1, 2\}$, and at most one of M^0 or M^1 would be typable, a contradiction.

Suppose there were a quantifier properly inside (not at the root of) the component of $\mathcal{G}(y)$ to the right of r arrows. Then this quantifier would still be present in $\mathrm{FDT}(yQ_1^h \ldots Q_r^h)$ for $h \in \{1, 2\}$, and neither M^0 nor M^1 would be typable, a contradiction.

Suppose there were either a quantifier or a free type variable in the component of $\mathcal{G}(y)$ reached by going to the right of i arrows and then to the left of one arrow where $i < r$. Then at most one of the application subterms $(yQ_1^h \ldots Q_i^h Q_{i+1}^h)$ for $h \in \{1, 2\}$ could be typed, a contradiction. □

Without loss of generality, assume that the subderivation of $\mathcal{D}$ for the subterm N satisfies the INST-before-GEN property of definition 3.1. This does not harm the proof because the final sequent of the new subderivation is the same as for the old subderivation. The only type that must be proved to be constrained in $\mathcal{D}$ is the type $\mathcal{G}(x)$, which this assumption does not affect because the λ-binding of x occurs in a separate part of the λ-term M.

Claim 6.8.5. *It holds that*

$$\mathcal{G}(y) = \mathrm{FDT}(N) = \forall\vec{\beta_1}.\mathcal{G}(z_1) \to \cdots \to \cdots \to \forall\vec{\beta_r}.\mathcal{G}(z_r) \to \forall\vec{\beta_{r+1}}.\tau'$$

for some τ' and some sequences of type variables $\vec{\beta_1}, \ldots, \vec{\beta_{r+1}}$.

Proof of claim 6.8.5. Because $\mathcal{G}(y)$ is closed, the assumed type $\mathcal{G}(y)$ must be the same as the final derived type for the subterm N. The type inference rules of System F and the shape of N allow us to impose some general constraints on the shape of the type $\mathrm{FDT}(N)$. Using the INST-before-GEN property assumed for the subderivation for N, we can deduce that the INST rule is not used between typing the subterm $\lambda z_r.\mathbf{K} z_g(aP_1 \ldots P_s))$ and finishing the subderivation for N. The claimed shape of $\mathcal{G}(y)$ follows. $\square$

Let R be the substitution $[\alpha_1 := \mathcal{G}(z_1), \ldots, \alpha_n := \mathcal{G}(z_n)]$.

Claim 6.8.6. *It holds that* $\mathrm{RAN}(R) \subset \mathbb{V}$ *and that $\mathcal{G}(z_i) = R(\mu_i)$ for $1 \leq i \leq r$.*

Proof of claim 6.8.6. First, observe that typing the subterm M^0 requires that the tree of the type $\mathcal{G}(y)$ can not be larger than the tree of the type $\mu_1 \to \cdots \to \mu_r \to \mu_g$. Thus, by claim 6.8.5, the type $\mathcal{G}(z_i)$ does not have a larger tree than μ_i for $1 \leq i \leq r$. By claims 6.8.4 and 6.8.5, the type $\mathcal{G}(z_i)$ must be open for $1 \leq i \leq r$. Thus, it is clear that $R(\alpha_i)$ is a variable for $1 \leq i \leq n$.

The rest of the proof proceeds by induction on the size of μ_i. In the base case, consider $\mu_i = \alpha_j$ for some $j \in \{1, \ldots, n\}$. By definition, $\mathcal{G}(z_j) = R(\alpha_j)$. By definition, there are subterms $(z_{j+n} z_j)$ and $(z_{j+n} z_i)$ (identical if $i = j$) among the members of $\vec{P}$. Because all of the types in question are open, we know there are no uses of INST or GEN in typing these subterms. Hence, $\mathcal{G}(z_j) = \mathcal{G}(z_i)$ and the base case is done. For the induction case, consider $\mu_i = \mu_j \to \mu_f$. (The type μ_i can be decomposed that way because $\mathbb{W}$ is closed under type components.) There is a type $\mu_h \in \mathbb{W}$ such that $\mu_h = \mu_f \to \varphi$ for some φ. There are subterms $(z_h(z_i z_j))$ and $(z_h z_f)$ in $\vec{P}$. By the induction hypothesis, $\mathcal{G}(z_j) = R(\mu_j)$ and $\mathcal{G}(z_f) = R(\mu_f)$. Thus, because INST and GEN can not be used in typing these subterms, $\mathcal{G}(z_i) = R(\mu_i)$. $\square$

Claim 6.8.7. *It holds that:*

$$\mathcal{G}(y) = \mathrm{FDT}(N) = \forall\vec{\beta_1}.\mathcal{G}(z_1) \to \cdots \to \forall\vec{\beta_n}.\mathcal{G}(z_n) \to \mathcal{G}(z_{n+1}) \to \cdots \to \mathcal{G}(z_r) \to \mathcal{G}(z_g)$$

Proof of claim 6.8.7. We show that the GEN and INST rules may not be used in certain positions in the subderivation of $\mathcal{D}$ for the subterm N. Let B be the type environment

$$B = \{z_1 : \mathcal{G}(z_1), \ldots, z_n : \mathcal{G}(z_n)\}$$

The initial sequent in the derivation $\mathcal{D}$ for the subterm z_g must look like this for some type environment C:

$$A \cup B \cup C \vdash z_g : \mathcal{G}(z_g) \tag{14}$$

Because $1 \leq g \leq n$, it is clear that $\mathrm{FTV}(\mathcal{G}(z_g)) \subseteq \mathrm{FTV}(B)$, and thus the GEN rule can not be used with sequent (14). The INST rule also can not be used because $\mathcal{G}(z_g)$ is an open type. Thus, (14) is the final sequent for the subterm z_g. The final derived type for the subterm $\mathbf{K}$ must be $\mathcal{G}(z_g) \to \varphi \to \mathcal{G}(z_g)$ for some type φ which does not matter. By similar reasoning as above, the GEN and INST can not affect the portions of this type that are equal to $\mathcal{G}(z_g)$.

Thus, the initial and final derived type for the subterm $(\mathbf{K} z_g(aP_1 \ldots P_s))$ must be exactly $\mathcal{G}(z_g)$. Now consider each subterm $(\lambda z_i \ldots z_r.\mathbf{K} z_g(aP_1 \ldots P_s))$ for $n+1 \leq i \leq r+1$. A simple induction shows that the initial and final derived types for each of the subterms are the same, i.e., that the GEN and INST rules are not used for these subterms. (The GEN rule can begin to be used above this point, and all free type variables must be quantified by the end of the subderivation for N.) The base case of $i = r+1$ is already shown. Consider the case where $n+1 \leq i \leq r$. By induction and one use of the ABS rule, the initial sequent for the subterm must look like this for some type environment C:

$$A \cup B \cup C \vdash (\lambda z_i \ldots z_r.\mathbf{K} z_g(aP_1 \ldots P_s)) : \mathcal{G}(z_i) \to \cdots \to \mathcal{G}(z_r) \to \mathcal{G}(z_g)$$

For $1 \leq h \leq r$, because $\mathcal{G}(z_h) = R(\mu_h)$, it holds that $\mathrm{FTV}(\mathcal{G}(z_h)) = \mathrm{FTV}(R(\mu_h)) = R(\mathrm{FTV}(\mu_h)) \subseteq R(\mathbb{S}) = \{\mathcal{G}(z_1), \ldots, \mathcal{G}(z_n)\} = \mathrm{FTV}(B)$. Thus, the free type variables

$$\mathrm{FTV}(\mathcal{G}(z_i) \to \cdots \to \mathcal{G}(z_r) \to \mathcal{G}(z_g))$$

are also free in the type environment B. Because this type is also completely open, neither the GEN nor the INST rules can be used for this subterm, and the induction is complete. The rest of the proof follows from earlier claims. □

Claim 6.8.8. *If $1 \leq i < j \leq n$, then $\mathcal{G}(z_i) \neq \mathcal{G}(z_j)$. Therefore, the substitution R is one-to-one.*

Proof of claim 6.8.8. Suppose $i \neq j$ and $\mathcal{G}(z_i) = \mathcal{G}(z_j)$. Then the corresponding positions in $\mathcal{G}(y)$ would have the same type variable. Hence, in the typing of $(yQ_1^0 \ldots Q_n^0)$, the subterms Q_i^0 and Q_j^0 would have to have the same type, contradicting the fact that $\mathrm{FDT}(Q_i^0) = \alpha_i \neq \alpha_j = \mathrm{FDT}(Q_j^0)$ (by claim 6.8.3). □

We have now shown that there is a substitution R such that $R(\mu_i) = \mathcal{G}(z_i)$ for $1 \leq i \leq r$. Thus, we can conclude that the type $\mathcal{G}(y)$ must have the following shape:

$$\mathcal{G}(y) = \forall \vec{\beta}_1.R(\alpha_1) \to \cdots \to \forall \vec{\beta}_n.R(\alpha_n) \to R(\mu_{n+1}) \to \cdots \to R(\mu_r) \to R(\alpha_g)$$

Because the type $\mathcal{G}(y)$ must be closed, the set of type variables $\bigcup_{1 \leq i \leq n} \vec{\beta}_i$ must contain every member of the set of type variables $R(\mathbb{S})$. By α-conversion on types and claim 6.8.8, we obtain that the type $\mathcal{G}(y)$ may also be written as follows, where for $1 \leq i \leq n$ it holds that $R(\vec{\gamma}_i) = \vec{\beta}_i$:

$$\mathcal{G}(y) = \forall \vec{\gamma}_1.\alpha_1 \to \cdots \to \forall \vec{\gamma}_n.\alpha_n \to \mu_{n+1} \to \cdots \to \mu_r \to \alpha_g$$

At this point, we know as much about the type $\mathcal{G}(y)$ as is possible. It is time to consider the type $\mathcal{G}(x)$, which we desire to show to be exactly the type τ.

Recall the subterm $(ya \cdots a)$ of the λ-term M, in which there are $r - c$ copies of a in the subterm. Remember also that $n + c \leq r$. Recall that the type assumed for a is $\perp$, so each subterm a can take on any derived type. Therefore, the final derived type of the subterm $(ya \cdots a)$, which equals the type $\mathcal{G}(x)$, must look like this:

$$\mathcal{G}(x) = \forall \vec{\delta}.T(\mu_{r-c+1}) \to \cdots \to T(\mu_r) \to T(\alpha_g) \tag{15}$$

where T is a substitution such that $\mathrm{DOM}(T) = \mathrm{FTV}(\mathbb{R}) \cup \{\alpha_g\}$ and $\{\vec{\delta}\} = \mathrm{FTV}(\mathrm{RAN}(T))$ (because $\mathcal{G}(x)$ must be closed by claim 6.8.4). Recall that for $1 \leq i \leq c$ it is the case that $\mu_{r-c+i} = \rho_i$. Thus, the type (15) is the same as this:

$$\mathcal{G}(x) = \forall \vec{\delta}.T(\rho_1) \to \cdots \to T(\rho_c) \to T(\alpha_g) \tag{16}$$

Claim 6.8.9. *In the subderivation for the subterm $(Q_{n+g}^0(xQ_{r-c+1}^0 \cdots Q_r^0))$, it holds that* $\mathrm{FDT}(x) = \rho_1 \to \cdots \to \rho_r \to S(T(\alpha_g))$ *for some substitution S such that $S(T(\rho_h)) = \rho_h$ for $1 \leq h \leq c$.*

Proof of claim 6.8.9. Without loss of generality, assume that the subderivation for this subterm satisfies the INST-before-GEN property. Because x is applied to Q^0_{r-c+1}, the subterm x must be typed by VAR followed by one or more uses of INST, without using GEN at all. Therefore $\mathrm{FDT}(x) = S(T(\rho_1)) \to \cdots \to S(T(\rho_c)) \to S(T(\alpha_g))$ for some substitution S by the known shape of $\mathcal{G}(x)$ in (16). By claim 6.8.3, it holds that $\mathrm{FDT}(Q_{r-c+h}) = \rho_h$ for $1 \leq h \leq c$. We prove by induction on $h \in \{1, \ldots, c\}$ that $\mathrm{FDT}(xQ^0_{r-c+1} \cdots Q^0_{r-c+h-1}) = S(T(\rho_h)) \to \cdots \to S(T(\rho_c)) \to S(T(\alpha_g))$ and that $S(T(\rho_h)) = \rho_h$. In the base case where $h = 1$, the subterm $(xQ^0_{r-c+1} \cdots Q^0_{r-c+h-1})$ is simply x, and the result about $\mathrm{FDT}(x)$ has been presented above. It is clear that typing the application to Q_{r-c+1} requires that $S(T(\rho_1)) = \rho_1$. For the induction case where $h \geq 2$, by the induction hypothesis and the APP rule, $\mathrm{IDT}(xQ^0_{r-c+1} \cdots Q^0_{r-c+h-1}) = S(T(\rho_h)) \to \cdots \to S(T(\rho_c)) \to S(T(\alpha_g))$. Because this type has no outermost quantifiers, there can be no uses of INST and GEN and hence the final derived type is the same. It holds that $S(T(\rho_h)) = \rho_h$ for the same reason as in the base case. □

Claim 6.8.10. *It holds that* $\mathrm{RAN}(T) \subset \mathbb{V}$.

Proof of claim 6.8.10. By claim 6.8.9, it holds for x in the subterm $(Q^0_{n+g}(xQ^0_{r-c+1} \cdots Q^0_r))$ that $\mathrm{FDT}(x) = \rho_1 \to \cdots \to \rho_r \to S(T(\alpha_g))$ for some substitution S. This would be impossible if $T(\alpha_j) \notin \mathbb{V}$ where $\alpha_j \in \mathrm{FTV}(\mathbb{R})$. This shows that $T(\alpha_i) \in \mathbb{V}$ for $i \in (\{1, \ldots, n\} - \{g\})$.

Consider now the subderivation for the subterm N':

$$N' \equiv ((\lambda u.a(ub_{n+1})(u(x\overbrace{a \cdots a}^{c-1})))(\lambda v.a(b_{n+1}(va))))$$

It holds that $\mathcal{G}(v)$ is open, because otherwise there would be one or more quantifiers embedded in $\mathcal{G}(u)$ at positions preventing the typing of the subterm (ub_{n+1}). Furthermore, the tree of $\mathcal{G}(v)$ may not be larger than $\beta \to \beta'$. To type the subterm $(b_{n+1}(va))$, it must hold that $\mathrm{FDT}(va) = \alpha_1$. Thus, $\mathcal{G}(v) = \beta_1 \to \alpha_1$ for some type variable β. Thus, $\mathcal{G}(u) = \forall\vec{\delta}.(\beta \to \alpha_1) \to \tau'$ for some type τ' and some sequence of type variables $\vec{\delta}$ such that $\alpha_1 \notin \{\vec{\delta}\}$. In order to type the application $(u(xa \cdots a))$, by considering $\mathcal{G}(u)$ it must hold that $\mathrm{FDT}(xa \cdots a) = \tau' \to \alpha_1$ for some τ'. By considering the subderivation for $(xa \cdots a)$ and using similar reasoning to that in the proof of claim 6.8.9, it must hold that $\mathrm{FDT}(xa \cdots a) = S(T(\rho_c) \to T(\alpha_g))$ for some S. This can only happen if $T(\alpha_g) \in \mathbb{V}$. □

Claim 6.8.11. *If* $1 \leq i < j \leq n$, *then* $T(\alpha_i) \neq T(\alpha_j)$.

Proof of claim 6.8.11. By claim 6.8.9, it holds for x in the subterm $(Q^0_{n+g}(xQ^0_{r-c+1} \cdots Q^0_r))$ that $\mathrm{FDT}(x) = \rho_1 \to \cdots \to \rho_r \to S(T(\alpha_g))$ for some substitution S such that $S(T(\rho_h)) = \rho_h$ for $1 \leq h \leq c$. Suppose for $1 \leq i < j \leq n$ that $T(\alpha_i) = T(\alpha_j)$. We consider two cases.

Consider first the case where $g \in \{i, j\}$. Without loss of generality, consider only the case where $i = g$, as the reasoning where $j = g$ is identical. Let $\beta = T(\alpha_i) = T(\alpha_j)$ (using 6.8.10). Because $j \neq g$, it holds that $\alpha_j \in \mathrm{FTV}(\rho_h)$ for some $h \in \{1, \ldots, c\}$. Furthermore, $S(T(\rho_h)) = \rho_h$ implies $T \circ S$ is the identity on $\mathrm{FTV}(\rho_h)$. Thus, $S(\beta) = \alpha_j$, implying that $S(T(\alpha_g)) = \alpha_j$. However, this implies that $\mathrm{FDT}(xQ^0_{r-c+1} \cdots Q^0_r) = \alpha_j$ which conflicts with $\mathrm{FDT}(Q_{n+g}) = \alpha_g \to \alpha_g$ (by claim 6.8.3), a contradiction.

The case where $g \notin \{i, j\}$ uses similar reasoning, but is simpler. □

Using the results we have so far, we obtain by α-conversion on types that the type $\mathcal{G}(x)$ in (16) may be written as follows, where $T(\vec{\epsilon}) = \vec{\delta}$:

$$\mathcal{G}(x) = \forall\vec{\epsilon}.\rho_1 \to \cdots \to \rho_c \to \alpha_g$$

Comparing this with the definition of τ reveals that $\mathcal{G}(x) = \tau$. This is the desired result which finishes the proof of lemma 6.8. □

Lemma 6.9 ($\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}$). *The set of types $\mathbb{B}$ induces the set of types $\mathbb{B} \cup \mathbb{U}$.* $\square$

Proof. The proof of lemma 6.8 yields a method for constructing from any τ such that $\tau \in \mathbb{U}(k+1)$ a term and a type environment with types from $\mathbb{B} \cup \mathbb{U}(k)$ which together induce the invariant type assumption $(x{:}\tau)$. The proof of lemma 5.4 prop. 5 together with the method already mentioned allows constructing an algorithm Ω such that $\Omega(\tau)$ is a witness of $\mathbb{B} \cup \mathbb{U}(k) \blacktriangleright \{\tau\}$ for any k such that $\tau \in \mathbb{U}(k+1)$.

Let $\tau_1, \tau_2, \ldots$ be an enumeration of $\mathbb{U}$ such that $\mathit{parheight}(\tau_i) \leq \mathit{parheight}(\tau_{i+1})$ for any i. Let $\mathbb{U}'_i = \{\tau_1, \ldots, \tau_i\}$. Observe that $\Omega(\tau_{i+1})$ is a witness for $\mathbb{B} \cup \mathbb{U}_i \blacktriangleright \{\tau_{i+1}\}$. Observe that $\mathbb{U}_1 = \{\bot\} \subset \mathbb{B}$ and let Ψ_1 be a witness for $\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}_1$. Let $\Theta(\Psi', i) = \Pi_{\mathrm{chain}}(\Psi', \Pi_{\mathrm{self}}(\Omega(\tau_{i+1})))$. Invoking lemma 5.6 with Ψ_1 and Θ yields a witness Ψ for $\mathbb{B} \blacktriangleright \mathbb{B} \cup \mathbb{U}$. $\square$

Lemma 6.10. *For any type τ in $\mathbb{C}(k+1)$, there exist a type environment A such that* $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{U}(2) \cup \mathbb{C}(k)$*, a λ-term M such that* $\mathrm{FV}(M) = \mathrm{DOM}(A)$*, and a λ-term variable $x \in$* $\mathrm{BV}(M)$ *such that A and M induce the invariant type assumption $(x{:}\tau)$.* $\square$

Proof. Let τ be the following type in $\mathbb{C}(k+1)$:

$$\tau = \forall \overrightarrow{\epsilon_1}.\rho_1 \to \forall \overrightarrow{\epsilon_2}.\rho_2 \to \cdots \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s$$

where ρ_s is a type variable and each occurrence of "$\overrightarrow{\epsilon_i}$" for $1 \leq i \leq s$ is a sequence of type variables. Let $\{\alpha_1, \ldots, \alpha_n\}$ be a set of type variables, let f be a non-decreasing function, let $f'(i) = f(i+1) - 1$, and let g be a natural number such that all of the following hold:

1. $n \geq 2$.
2. $f(1) = 1$.
3. For $1 \leq i \leq s$, it holds that $\overrightarrow{\epsilon_i}$ is $\alpha_{f(i)}, \ldots, \alpha_{f'(i)}$.
4. $\rho_s = \alpha_g$ where $1 \leq g \leq n$.

Note that $f'(s)$ may be less than n to allow $n \geq 2$ to hold. The set $\{\vec{\alpha}\}$ and function f can be found by choosing an appropriate representative of the α-equivalence class of τ. Because $\tau \in \mathbb{C}(k+1)$, it is the case that τ has no free type variables, i.e., $\mathrm{FTV}(\tau) = \varnothing$.

From τ, a type environment A and a λ-term M having the desired properties will be constructed. In the course of this definition, functions, sets of type variables, and λ-terms will be defined.

Define the type environment A as follows. Let the function m on natural numbers be defined so that $m(i,h) = (((i+h) - 1) \mod n) + 1$. For $1 \leq i \leq s$, add the following type assumptions to A:

$$\begin{aligned}
A(e_i) &= \begin{cases} \forall.(\gamma \to \delta) \to \alpha_1 \to \cdots \to \alpha_{f'(i)} \to \beta \to \gamma & \text{if } \rho_i = \bot, \\ \forall.\alpha_1 \to \cdots \to \alpha_{f'(i)} \to \beta \to \rho_i & \text{otherwise.} \end{cases} \\
A(c_i) &= \forall.(\alpha_{f(i)} \to \cdots \to \alpha_{f'(i)} \to \beta \to \gamma) \to \gamma \\
A(d_i) &= \forall.\alpha_{f(i)} \to \cdots \to \alpha_{f'(i)} \to (\alpha_{f(i)} \to \cdots \to \alpha_{f'(i)} \to \beta \to \delta_1) \to \delta_2
\end{aligned}$$

(The first case of $A(e_i)$ above and O_i below is only needed when $k = 0$ to avoid requiring an environment type from $\mathbb{C}(k+1) = \mathbb{C}(1)$.) For $1 \leq i \leq n$, add the following type assumptions to A:

$$\begin{aligned}
A(b_i) &= \alpha_i \\
A(b_{n+i}) &= \alpha_i \to \alpha_i
\end{aligned}$$

Finally, add these type assumptions to A:

$$
\begin{aligned}
A(a) &= \bot \\
A(q) &= \forall.\gamma \to (\gamma \to \delta_1) \to \delta_2 \to \beta \to \gamma \\
A(p) &= \forall.(\beta_1 \to \gamma_1) \to (\beta_2 \to \gamma_2) \to \beta_3 \to (\gamma_1 \to \gamma_2)
\end{aligned}
$$

It can be checked that the range of the type environment A lies within the set $\mathbb{B} \cup \mathbb{U}(2) \cup \mathbb{C}(k)$.

The following definitions are pieces that will be assembled to form the λ-term M. Define the following λ-terms for $1 \leq i \leq s$ and for $h \in \{0, 1\}$:

$$
\begin{aligned}
O_i &\equiv \begin{cases} (e_i(\lambda w_i.a(b_1 w_i)(b_2 w_i))) & \text{if } \rho_i = \bot, \\ e_i & \text{otherwise.} \end{cases} \\
U^h_{i,j} &\equiv O_j z_1 \ldots z_{f'(i)} b_{m(f(i+1),h)} \ldots b_{m(f'(j),h)} \\
T_i &\equiv U^0_{i,i} \\
P_i &\equiv \begin{cases} T_i & \text{if } i = s, \\ (pT_i N_{i+1}) & \text{for } 1 \leq i < s. \end{cases} \\
W^h_i &\equiv \begin{cases} b_{n+m(g,h)} & \text{if } f(i) \leq g \leq f'(s), \\ a & \text{otherwise.} \end{cases} \\
Q^h_i &\equiv W^h_i(v_i(U^h_{i-1,i}a) \ldots (U^h_{i-1,s-1}a)) \\
V^h_i &\equiv (d_i b_{m(f(i),h)} \cdots b_{m(f'(i),h)} u_i) \\
N_i &\equiv ((\lambda u_i.q(c_i u_i)(\lambda v_i.aQ^0_i Q^1_i)(aV^0_i V^1_i)) \\
&\qquad (\lambda z_{f(i)} \cdots z_{f'(i)}.P_i))
\end{aligned}
$$

Finally, define $M \equiv N_1$ and $x = v_1$. Inspection reveals that $\mathrm{FV}(M) = \mathrm{DOM}(A)$.

The type environment A and the λ-term M have now been defined and it has been shown that $\mathrm{FV}(M) = \mathrm{DOM}(A)$ and $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{U}(k)$. Let R' be a type-variable renaming which maps the members of $\{\vec{\alpha}\}$ to fresh names. It can be easily checked that M is typable under the type environment A with a global type environment including the following additional type assumptions where $1 \leq i \leq s$:

$$
\begin{aligned}
&w_i : \bot \qquad \text{if } \rho_i = \bot \\
&z_j : R'(\alpha_j) \qquad \text{for } 1 \leq j \leq n \\
&u_i : R'(\forall \overrightarrow{\epsilon_i}.\alpha_{f(i)} \to \cdots \to \alpha_{f'(i)} \to \bot \to \rho_i \to \forall \overrightarrow{\epsilon_{i+1}}.\rho_{i+1} \to \cdots \to \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s) \\
&v_i : R'(\forall \overrightarrow{\epsilon_i}.\rho_i \to \cdots \to \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s)
\end{aligned}
$$

To prove that the type environment A and the λ-term M together induce the invariant type assumption $(x{:}\tau)$, what remains to be shown is for any particular derivation $\mathcal{D}$ that ends with a sequent $A \vdash M : \sigma$ for some type σ, that there is a type-variable renaming R such that $(\mathcal{G}(\mathcal{D}))(x) = R(\tau)$. Because τ is closed, $R(\tau) = \tau$ for any R.

Let $\mathcal{D}$ be some derivation ending with $A \vdash M : \sigma$ for some type σ. Without loss of generality, assume that every subderivation of $\mathcal{D}$ which does not contain the binding "λv_1" satisfies the INST-before-GEN property. There is a type-variable renaming R such that the following type assumptions and derived types must occur in $\mathcal{D}$ for $1 \leq i \leq s$ (where $\Box$ denotes some unknown type which does not matter):

$$
\begin{aligned}
\mathcal{G}(w_i) &= \bot \qquad \text{if } \rho_i = \bot \\
\mathrm{FDT}(U^h_{j,i}) &= \Box \to S(\rho_i) \quad \text{for } 0 \leq j < i \text{ and } h \in \{0,1\} \\
&\ \text{where } S(\alpha_k) = \begin{cases} R(\alpha_k) & \text{if } 1 \leq k \leq f'(i), \\ \alpha_{m(k,h)} & \text{if } f(i+1) \leq k \leq n. \end{cases} \\
\mathrm{FDT}(T_i) &= \Box \to R(\rho_i) \\
\mathrm{FDT}(P_i) &= \begin{cases} R(\Box \to \rho_i \to \forall \overrightarrow{\epsilon_{i+1}}.\rho_{i+1} \to \cdots \to \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s) & \text{if } i < s, \\ R(\Box \to \rho_s) & \text{if } i = s. \end{cases} \\
\mathcal{G}(z_j) &= R(\alpha_j) \qquad \text{for } 1 \leq j \leq n \\
\mathcal{G}(u_i) &= R(\forall \vec{\beta}.\forall \overrightarrow{\epsilon_i}.\alpha_{f(i)} \to \cdots \to \alpha_{f'(i)} \to \Box \to \rho_i \to \forall \overrightarrow{\epsilon_{i+1}}.\rho_{i+1} \to \cdots \to \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s) \\
&\ \text{where } \vec{\beta} \text{ can occur only in } \Box \\
\mathcal{G}(v_i) &= R(\forall \overrightarrow{\epsilon_i}.\rho_i \to \cdots \to \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s) \\
\mathrm{FDT}(N_i) &= R(\Box \to \forall \overrightarrow{\epsilon_i}.\rho_i \to \cdots \to \forall \overrightarrow{\epsilon_{s-1}}.\rho_{s-1} \to \forall \overrightarrow{\epsilon_s}.\rho_s)
\end{aligned}
$$

The explanation for these facts proceeds as follows. First, in many places that types of the shape $\Box \to \sigma$ are formed where σ is the type of interest. The construction does this to prevent the outermost quantifiers of σ from being altered. Second, the subterm $U^h_{i,j}$ is used to provide the type ρ_j, subject to a particular renaming. It is parameterized because differently renamed versions of ρ_j are used in different places. Third, the application of p is used to join two types whose outermost quantifiers are protected by the "$\Box \to$" technique. Fourth, the constraints imposed by $(aV^0_iV^1_i)$ for $1 \leq i \leq s$ are used to ensure that the types $\mathcal{G}(z_j)$ for $1 \leq j \leq n$ are all distinct type variables. Fifth, the application (c_iu_i) temporarily exposes the root of the type being constructed to allow the GEN rule to be used. The constraints imposed by $(aQ^0_iQ^1_i)$ ensure that all of the necessary uses of GEN occur and the only uses of INST that occur merely rename the variables distinctly. The application of q is used to move the types around appropriately for this.

The details of why these various technique work are omitted because the techniques are similar to lemma 6.8. Also, the result of this lemma is not necessary for the main result (theorem 6.16), because it is only necessary to simulate instances of TC with universal types to prove the undecidability of TYP.

It is simple to check at the end that the type $\mathcal{G}(v_i)$ is exactly τ, which is the desired result. $\Box$

Lemma 6.11 ($\mathbb{B} \cup \mathbb{U} \blacktriangleright \mathbb{B} \cup \mathbb{U} \cup \mathbb{C}$). *The set of types $\mathbb{B} \cup \mathbb{U}$ induces the set of types $\mathbb{B} \cup \mathbb{U} \cup \mathbb{C}$.* $\Box$

Proof. The proof proceeds as in lemma 6.9, except that lemma 6.10 is used instead of lemma 6.8. $\Box$

Lemma 6.12. *For type τ in $\mathbb{T}$, there exist a type environment A such that* $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{C}$, *a λ-term M such that* $\mathrm{FV}(M) = \mathrm{DOM}(A)$, *and a λ-term variable $x \in \mathrm{BV}(M)$ such that A and M induce the invariant type assumption $(x{:}\tau)$.* $\Box$

Proof. Let τ be an arbitrarily chosen type in $\mathbb{T}$. Let $\mathrm{FTV}(\tau) = \{\alpha_1, \ldots, \alpha_n\}$. Define σ so that

$$\sigma = \forall.\alpha_1 \to \cdots \to \alpha_n \to \beta \to \tau$$

Define the type environment A as follows. For $1 \leq i \leq n$ define $A(b_i) = \alpha_i$. Define $A(c) = \sigma$. Define $A(d) = \forall.(\gamma \to \delta_1) \to (\delta_2 \to \gamma) \to \delta_3$. It is clear that $\mathrm{RAN}(A) \subset \mathbb{B} \cup \mathbb{C}$. Define M as the λ-term $M \equiv (d(\lambda x.x)(cb_1 \ldots b_n))$. It can be checked that the type environment A and the λ-term M together induce the invariant type assumption $(x{:}\tau)$. $\Box$

Lemma 6.13 ($\mathbb{B} \cup \mathbb{C} \blacktriangleright \mathbb{T}$). *The set of types $\mathbb{B} \cup \mathbb{C}$ induces the set of types $\mathbb{T}$.* $\Box$

Proof. The proof proceeds as in lemma 6.9, except that lemma 6.12 is used instead of lemma 6.8 and that the proof is a bit simpler. □

Theorem 6.14 ($\varnothing \blacktriangleright \mathbb{T}$). *The empty set of types $\varnothing$ induces the set of all types $\mathbb{T}$.* □

Proof. By lemmas 6.9, 6.11, and 6.13 together with lemma 5.4 prop. 10. □

Theorem 6.15 (TC $\leq$ Typ). TC *in F is reducible to* Typ *in F.* □

Proof. The following sentences describe a method for constructing an instance of Typ from an instance of TC. Consider an instance of TC in which it is asked whether the sequent $A \vdash M : \tau$ can be derived in F. Let the type environment $B = A \cup \{z{:}\tau \to \bot\}$ where z is a fresh variable. Using theorem 6.14 and definition 5.3, construct (effectively) a context C such that $\varnothing, C \rhd B$. Without loss of generality, assume that $\mathrm{BV}(M) \cap \mathrm{V}(C) = \varnothing$. To show that TC has been reduced to Typ, it is sufficient to show that $A \vdash M : \tau$ is derivable if and only if $C[zM]$ is typable. Each direction is checked separately.

($\Rightarrow$) Suppose $A \vdash M : \tau$ is derivable. Then $B \vdash zM : \bot$ is derivable. By definition 5.2 prop. 1, $\varnothing \vdash C[zM] : \sigma$ is derivable for some type σ.

($\Leftarrow$) Suppose $A' \vdash C[zM] : \sigma$ is derivable for some type environment A' and type σ. Then $\varnothing \vdash C[zM] : \sigma$ is derivable, because $\mathrm{FV}(C[zM]) = \varnothing$. By definition 5.2 prop. 2, it holds that $R(B) \cup E \vdash zM : \rho$ is derivable for some type-variable renaming R, type environment E, and type ρ. Because $\mathrm{FV}(zM) \subseteq \mathrm{DOM}(B)$, there must be a derivation $\mathcal{D}$ ending with $R(B) \vdash zM : \rho$. A step in $\mathcal{D}$ must be:

$$\frac{R(B) \vdash z : R(\tau \to \bot) \qquad R(B) \vdash M : R(\tau)}{R(B) \vdash zM : \bot}\ \mathrm{APP}$$

Because $z \notin \mathrm{FV}(M)$, there is a derivation $\mathcal{D}'$ ending with $R(A) \vdash M : R(\tau)$. The result of applying R^{-1} to $\mathcal{D}'$ is also a valid derivation and it ends with $A \vdash M : \tau$. □

Theorem 6.16 (Main result). Typ *and* TC *in System F are of equivalent difficulty and are both undecidable.* □

Proof. Recall that Typ easily reduces to TC [Bar92] and that SUP is undecidable [KTU93]. Then the result follows from theorem 4.1 and theorem 6.15 □

References

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*, vol. 2, chapter 2, pp. 117–309. Oxford University Press, 1992.

[Boe85] H.-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Ann. Symp. Foundations Comput. Sci.*, pp. 339–345. IEEE, Oct. 1985.

[Car91] L. Cardelli. Typeful programming. In *Formal Description of Programming Concepts*. Springer-Verlag, 1991. Also DEC SRC Research Report 45.

[GHRDR87] P. Giannini, F. Honsell, and S. Ronchi Della Rocca. A strongly normalizable term having no type in the System F (second order λ-calculus). Rapporto interno, Univ. di Torino, 1987.

[GHRDR93] P. Giannini, F. Honsell, and S. Ronchi Della Rocca. Type inference: Some results, some problems. *Fund. Informa.*, 19(1/2):87–125, Sept./Oct. 1993.

[Gir72] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'Etat, Université de Paris VII, 1972.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Number 7 in Cambridge Tracts in Theor. Comp. Sci. Cambridge Univ. Press, 1989.

[Gol81] W. D. Goldfarb. Undecidability of the second-order unification problem. *Theor. Comp. Sc.*, 13:225–230, 1981.

[GRDR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. 3rd Ann. Symp. Logic in Computer Sci.*, pp. 61–70, Edinburgh, Scotland, July 5–8, 1988.

[GRDR91] P. Giannini and S. Ronchi Della Rocca. Type inference in polymorphic type discipline. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, vol. 526 of *LNCS*, pp. 18–37. Springer-Verlag, 1991.

[Hin69] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[HM91] F. Henglein and H. G. Mairson. The complexity of type inference for higher-order typed lambda calculi. In *Conf. Rec. 18th Ann. ACM Symp. Principles of Programming Languages*, pp. 119–130, Jan. 1991. An expanded version appeared as [HM94].

[HM94] F. Henglein and H. G. Mairson. The complexity of type inference for higher-order typed lambda calculi. *J. Funct. Prog.*, 4(4):435–477, 1994. First appeared as [HM91].

[Hoo66] P. K. Hooper. The undecidability of the Turing machine immortality problem. *J. Symbolic Logic*, 31(2):219–234, 1966.

[HW88] P. Hudak and P. L. Wadler. Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR656, Yale University, 1988.

[Jim95] T. Jim. Type inference in System F plus subtyping. Manuscript., Dec. 1995.

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the second-order λ-calculus. *Inf. & Comput.*, 98(2):228–257, June 1992.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Inf. & Comput.*, 102(1):83–101, Jan. 1993.

[KW94] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, 1994.

[Lei83] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Principles of Programming Languages*, pp. 88–98, 1983.

[Lei91] D. Leivant. Finitely stratified polymorphism. *Inf. & Comput.*, 93(1):93–113, July 1991.

[LICS94] *Proc. 9th Ann. IEEE Symp. Logic in Computer Sci.*, 1994.

[McC84] N. McCracken. The typechecking of programs with implicit type structure. In *Semantics of Data Types : Int'l Symp.*, vol. 173 of *LNCS*, pp. 301–315. Springer-Verlag, 1984.

[Min61] M. Minsky. Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines. *Annals of Math.*, 74(3):437–455, 1961.

[Mit88] J. C. Mitchell. Polymorphic type inference and containment. *Inf. & Comput.*, 76(2/3):211–249, Feb./Mar. 1988.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pfe93] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fund. Informa.*, 19(1,2):185–199, Sept./Oct. 1993.

[PL89] F. Pfenning and P. Lee. LEAP: a language with eval and polymorphism. In *Proc. 3rd Int'l Joint Conf. Theory & Practice of Software Development*. Springer-Verlag, 1989.

[Pra65] D. Prawitz. *Natural Deduction; A Proof-Theoretical Study*. Number 3 in Stockholm Studies in Philosophy. Almqvist and Wiksell, Stockholm, 1965.

[Pud88] P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988. Prague, Czechoslovakia.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, vol. 19 of *LNCS*, pp. 408–425, Paris, France, 1974. Springer-Verlag.

[Sch98] A. Schubert. Second-order unification and type inference for Church-style polymorphism. In *Conf. Rec. POPL '98: 25th ACM Symp. Principles of Prog. Languages*, 1998.

[TU95] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. Tech. rep., Univ. of Warsaw, Nov. 1995.

[TU96] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proc. 11th Ann. IEEE Symp. Logic in Computer Sci.*, 1996. Shorter proceedings version of [TU95].

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *IFIP Int'l Conf. Funct. Program. Comput. Arch.*, vol. 201 of *LNCS*. Springer-Verlag, 1985.

[Urz93a] P. Urzyczyn. The emptiness problem for intersection types. Technical report, Inst. of Informatics, Univ. of Warsaw, Poland, Nov. 1993.

[Urz93b] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$ is undecidable. In *Proc. Int'l Conf. Typed Lambda Calculi and Applications*, pp. 418–432, 1993.

[Urz94] P. Urzyczyn. The emptiness problem for intersection types. In LICS '94 [LICS94], pp. 300–309.

[Urz97] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Struc. in Comp. Sc.*, 7(4):329–358, 1997.

[Wel94] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In LICS '94 [LICS94].

[Wel95] J. B. Wells. The undecidability of Mitchell's subtyping relation. Tech. rep., Comp. Sci. Dept., Boston Univ., Dec. 1995.

[Wel96a] J. B. Wells. Typability is undecidable for F+eta. Tech. rep., Comp. Sci. Dept., Boston Univ., Mar. 1996.

[Wel96b] J. B. Wells. *Type Inference for System F with and without the Eta Rule.* Ph.D. thesis, Boston Univ., 1996.