Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable*

J. B. Wells
jbw@cs.bu.edu
Computer Science Dept.
Boston University
Boston, MA 02215, U.S.A.

Abstract

The problems of typability *and* type checking *exist for the Girard/Reynolds second-order polymorphic typed λ-calculus (also known as "system* **F***") when it is considered in the "Curry style" (where types are derived for pure λ-terms). Until now the decidability of these problems for* **F** *itself has remained unknown. We first prove that type checking in* **F** *is undecidable by a reduction from* semi-unification. *We then prove typability in* **F** *is undecidable by a reduction from type checking. Since the reduction from typability to type checking in* **F** *is already known, the two problems in* **F** *are equivalent (reducible to each other). The results hold for both the usual λK-calculus and the more restrictive λI-calculus.*

1 Introduction

Background and motivation. Girard [7] and Reynolds [21] independently formulated the type system of the second-order polymorphic typed λ-calculus about twenty years ago. This type system extended Curry's functionality theory, the simply-typed λ-calculus [3]. Girard developed his system (named by chance "system **F**") to prove properties of second-order propositional logic (hence **F**'s other name, "the second-order λ-calculus") while Reynolds wanted to express polymorphic typing in programming explicitly. Both Girard and Reynolds formulated **F** in "Church style", where types are embedded in terms and there is no such thing as an untypable term. When **F** is formulated in "Curry style", where types are given to pure terms of the λ-calculus, it becomes meaningful to ask for an arbitrary λ-term:

1. Is there any type that can be given to the λ-term?
2. Can the λ-term be given some particular type?

We call these problems Typ for *typability* and TC for *type checking*. Unless another type system is indicated, the names Typ and TC refer to these problems in the type system **F**.

Much research has been devoted to determining whether Typ and TC are decidable. Leivant first introduced **F** in the Curry style in 1983 and made the first attempt to answer the decidability for Typ [14]. The first interesting lower-bound for the computational complexity of Typ was given by Henglein who showed that Typ is DEXPTIME-hard (where DEXPTIME means DTIME($2^{n^{O(1)}}$)) [8]. Typ and TC have been considered for various restrictions and extensions of system **F** and related problems have been considered 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 [6], the number of generations of instantiation of quantifiers themselves introduced by instantiation [15], and the "rank" of polymorphic types (introduced in [14], further studied in [16, 11, 13]). Urzyczyn recently showed Typ to be undecidable for **F**'s powerful extension, system $\mathbf{F}_\omega$ [24]. For **F**, proofs of undecidability have been given for both the problem of *partial polymorphic type inference* in the Church style [18], which is related to Typ, and the problem of *conditional type inference* [4], which is related to TC. Only a small 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"[1] for system **F** itself.

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

*This work is partly supported by NSF grant CCR–9113196.

[1] Robin Milner quoted by Henk Barendregt [1].

with some combination of polymorphic, recursive, and intersection types and the equality and approximation rules, it is only for **F** that the decidability of TYP and TC has remained unknown until now [1]. Also, among the eight type systems in Barendregt's λ-cube [1], when they are considered in Curry style, there are only three distinct sets of typable pure λ-terms corresponding to the simply-typed λ-calculus, **F**, and $\mathbf{F}_\omega$ [4]. Since it has been known that TYP is decidable for the simply-typed λ-calculus [9] and (recently) undecidable for $\mathbf{F}_\omega$ [24], it is once again only for **F** that the answer has been unknown. Thus, right now, 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 decidability question for TYP has theoretical interest, a prime motivation for solving this problem has been its practical implications. Although the λ-calculus is the foundation of functional programming languages, in practice it is extended with a number of constants with special types and their own reduction rules, e.g. "**true**", "**and**", "**5**", "**+**". In the absence of strict typing enforced by the compiler, run-time type errors can occur after β-reduction, e.g. "**and true 5**". This can be seen in the side-effect-free fragment of the LISP programming language. To catch such type errors at compile-time, functional languages such as ML [17] and its successors (e.g. Miranda [22], Haskell [10]) use type systems whose bases include fragments of **F**. There are also programming languages which use type systems whose bases include fragments of $\mathbf{F}_\omega$, e.g. Quest [2] and LEAP [19]. Any type system whose typing power lies within that of $\mathbf{F}_\omega$ (such as **F**) will also have the benefit of guaranteeing termination, i.e. a program fragment will be guaranteed to halt if it is written without reference to extensions beyond $\mathbf{F}_\omega$ that introduce recursion.

The evidence shows that programming language designers feel that it is beneficial to implement strict typing using a type system based on some portion of **F** or $\mathbf{F}_\omega$. Two major considerations involved in picking the particular type system to use for a functional programming language are:

1. How flexibly may programs be written?
2. Is automatic type inference possible?

Regarding the first consideration, from a software engineering viewpoint, programmers wish to reuse program fragments rather than duplicating them. The type discipline of 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. The compiler will enforce this redundancy in a programming language based on this type system. One popular solution for this problem is *polymorphism*, which allows a polymorphic ("generic") type to be given to a function that can then be used as though it has any type that is an instance of the polymorphic type. ML has a weak form of polymorphism which allows a polymorphic function to be passed as a parameter, but only to a single predetermined non-polymorphic function. If ML were extended to use the full system **F** as its type system, a polymorphic function could be passed as a parameter to another non-predetermined polymorphic function, which itself could be passed as a parameter to other functions, and so on.

Regarding the second consideration, for a functional programming language with strict typing it is very desirable 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. Since it is also the nature of functional types that they grow to be very large, requiring the programmer to specify types is very unwieldy and user-unfriendly. These reasons contribute to the desire for *type inference*, a procedure that will provide a type for a λ-term if it is typable and will otherwise halt with an error.

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, 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 and a restricted version of the type system may have to be used instead. Therefore, using the full system **F** as the type system of a programming language would provide useful flexibility, but this will not be practical since we show TYP to be undecidable for **F**.

Contributions of this paper. The main contribution of this paper is answering the question of decidability for TYP and TC for **F**. We first prove that the problem of *semi-unification* can be reduced to TC using a simple encoding. We then reduce TC to TYP using a novel method of building λ-terms which force particular bound variables to be assigned particular types in any type derivation. Using this method, we can require that subterms in certain positions within a λ-term must be typable using a specific arbitrarily-picked *type assignment* in order for the entire term to be typable at all. Any desired type assignment may be simulated and this method works not only for the λK-calculus but within the λI-calculus as well. Since semi-unification is undecidable [12], and since the reduction from TYP to TC is already known [1], and since both reductions work within the λI-calculus, we conclude that TYP and TC are equivalent and that both are undecidable for both the λK and λI-calculi.

As explained above, our result has both theoretical and practical implications. By showing both problems to be undecidable, the question of decidability of TYP and TC for a variety of type systems is now completely answered. This can perhaps be best explained by pointing out that the table on page 183 of [1] may now be filled in completely (with the addition of the recent result that type inhabitation for intersection types is undecidable [23]). The practical implication of our result 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. Our result has other implications which we will develop in other papers. The methods used in this paper can also show the undecidability of TYP and TC for levels within the various stratifications of **F**, a result of both theoretical and practical interest. These methods can also be used to 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 typable in some extension of **F**.

Acknowledgements. Pawel Urzyczyn proofread an early handwritten draft and pointed out several improvements. Jerzy Tiuryn contributed valuable suggestions for notation and organization of the proofs. A. J. Kfoury provided vital support and encouragement.

2 Basic definitions and notation

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

The untyped λ-calculus. The reader should be familiar with standard notation for the untyped λ-calculus. The scope of "$\lambda x.$" extends as far to the right as possible. We assume at all times that every λ-term M obeys the restriction that no variable is λ-bound more than once and no variable occurs both λ-bound and free in M. K denotes the standard combinator $(\lambda x.\lambda y.x)$.

As usual, $\mathrm{FV}(M)$ and $\mathrm{BV}(M)$ denote the free and λ-bound variables of a λ-term M. $M[x_1 := N_1, \ldots, x_n := N_n]$ denotes the result of simultaneously substituting N_i for all free occurrences of x_i in M, renaming λ-bound variables in M as necessary to avoid capturing free variables of N_i. This may be abbreviated as $M[\vec{x} := \vec{N}]$. A context $C[\ \ ]$ is a λ-term with one hole and if M is a λ-term then $C[M]$ denotes the result of inserting M into the hole in $C[\ \ ]$, *including* the capture of free variables in M by the λ-bound variables of $C[\ \ ]$. If M and N are λ-terms, then $M \equiv N$ means that M and N are identical after allowing α-conversion. $N \subset M$ denotes that N is a proper subterm of M and $N \subseteq M$ includes the possibility that $N \equiv M$.

The λ-terms defined so far belong to the usual λK-calculus. The terms in the λI-calculus are defined as those λ-terms that do not have subterms of the form $(\lambda x.N)$ where $x \notin \mathrm{FV}(N)$.

System F. The set of all types $\mathbb{T}$ is built from the countably infinite set of type variables $\mathbb{V}$ using two type constructors specified by the grammar $\mathbb{T} ::= \mathbb{V} \mid (\mathbb{T} \to \mathbb{T}) \mid (\forall \mathbb{V}.\mathbb{T})$. A type is therefore either a type variable or a $\to$-*type* or a $\forall$-*type*. Small Greek letters from the beginning of the alphabet (e.g. α and β) are metavariables over $\mathbb{V}$ and small Greek letters towards the end of the alphabet (e.g. σ and τ) are metavariables over $\mathbb{T}$. Capital Roman letters in a "blackboard-bold" style (e.g. $\mathbb{X}$ and $\mathbb{Y}$) are metavariables over subsets of $\mathbb{T}$. When writing types, the arrows associate to the right so that $\sigma \to \tau \to \rho = \sigma \to (\tau \to \rho)$. The same scoping convention applies for "$\forall$" as for "λ". The notation $\sigma[\vec{\alpha} := \vec{\tau}]$ has the same meaning for types that it has for λ-terms. A *renaming* of free type variables is a one-to-one function $R\colon \mathbb{V} \to \mathbb{V}$ that extends naturally to a $\forall$-$\to$-homomorphism so that $R(\sigma \to \tau) = R(\sigma) \to (\tau)$ and $R(\forall\alpha.\sigma) = \forall R(\alpha).R(\sigma)$. The notation $\forall\vec{\alpha}.\sigma$ is shorthand for $\forall\alpha_1.\cdots.\forall\alpha_k.\sigma$. $\mathrm{FTV}(\tau)$ and $\mathrm{BTV}(\tau)$ denote the free and $\forall$-bound type variables of type τ, respectively. For a set of types $\mathbb{X}$, the notation $\mathrm{FTV}(\mathbb{X})$ denotes $\bigcup_{\tau\in\mathbb{X}} \mathrm{FTV}(\tau)$. The notation $\forall.\sigma$ means $\forall\vec{\alpha}.\sigma$ where $\vec{\alpha} = \mathrm{FTV}(\sigma)$. $\bot$ is shorthand for $\forall\alpha.\alpha$.

α-conversion of types and reordering of adjacent quantifiers is allowed at any time. For example, we consider the types $\forall\alpha.\forall\beta.\alpha \to \beta$, $\forall\beta.\forall\alpha.\beta \to \alpha$, and $\forall\beta.\forall\alpha.\alpha \to \beta$ to all be equal. Using α-conversion we assume that no variable is $\forall$-bound more than once in any type, that the $\forall$-bound type variables of any

$$\text{VAR}\quad A \vdash x : \sigma \qquad\qquad A(x) = \sigma$$

$$\text{APP}\quad \frac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\,N) : \tau}$$

$$\text{ABS}\quad \frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \to \tau}$$

$$\text{INST}\quad \frac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$$

$$\text{GEN}\quad \frac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma} \qquad\qquad \alpha \notin \text{FTV}(A)$$

Figure 1: **Inference Rules of System F**.

two type instances are disjoint, and that all $\forall$-bound type variables of any type instance are disjoint from the free type variables of another type instance. If $\sigma = \forall\alpha.\tau$ and $\alpha \notin \text{FTV}(\tau)$, we say that "$\forall\alpha$" is a *redundant* quantifier. We assume types do not contain redundant quantifiers.

A pair $x : \sigma$ where $x \in \mathcal{V}$ and $\sigma \in \mathbb{T}$ is called a *type assumption*. A finite set of type assumptions $A = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ which associates at most one type σ with each λ-term variable x is a *type assignment*. $A(x)$ denotes the unique type σ such that that $(x : \sigma) \in A$. $\text{FTV}(A)$ is the set of all free type variables in all of the types assigned by A.

An expression of the form $A \vdash M : \tau$, where A is a type assignment, M is a λ-term, and $\tau \in \mathbb{T}$, is a *sequent*. 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* $\mathcal{D}$ in system **F** is a finite sequence of sequents $\Delta_1, \ldots, \Delta_n$ for some $n \geq 1$, where each sequent Δ_j is obtained from one or two of the preceding sequents $\Delta_1, \ldots, \Delta_{j-1}$ according to the inference rules of **F** presented in Figure 1. For a derivation $\mathcal{D} = \vec{\Delta}^m$ where $\Delta_i = A_i \vdash M_i : \tau_i$, the *global type assignment* of $\mathcal{D}$ is $\mathcal{G}(\mathcal{D}) = \bigcup_{1 \leq i \leq m} A_i$. A *typing* of the λ-term M in **F** is a derivation in **F** whose last sequent is $A \vdash M : \tau$ for some type assignment A and type τ. A λ-term M is *typable* in **F** if and only if there is a typing of M in **F**.

Definition 2.1 (Typ) The typability problem: Given an arbitrary λ-term M, is M typable in **F**?

Definition 2.2 (TC) The type-checking problem: Given arbitrary λ-term M, type assignment A, and type $\tau \in \mathbb{T}$, is there a typing of M in **F** that ends with the sequent $A \vdash M : \tau$?

Semi-unification. For convenience, we define semi-unification (SUP) 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 proof that semi-unification is undecidable is actually for this special case [12]. The set of algebraic terms $\mathcal{T}$ is defined by the grammar $\mathcal{T} ::= \mathbb{V} \mid (\mathcal{T} \to \mathcal{T})$. An *instance* Γ of semi-unification is set of two pairs $\Gamma = \{\, \tau_1 \dot{\leq} \mu_1,\ \tau_2 \dot{\leq} \mu_2 \,\}$ where it is the case that $\tau_1, \tau_2, \mu_1, \mu_2 \in \mathcal{T}$. An *open substitution* is a function $S \colon \mathbb{V} \to \mathcal{T}$ that differs from the identity on only finitely many variables and which extends naturally to an $\to$-homomorphism $S \colon \mathcal{T} \to \mathcal{T}$ so that $S(\sigma \to \tau) = S(\sigma) \to S(\tau)$. 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 $S_2(S(\tau_2)) = S(\mu_2)$.

Definition 2.3 (SUP) The semi-unification problem: Given an arbitrary instance Γ of semi-unification, does Γ have a solution?

3 Reduction from SUP to TC

Definition 3.1 (INST before GEN property) Let the derivation $\mathcal{D}$ be the sequence of sequents $\vec{\Delta}$. Let c be a function so that if Δ_i is a sequent in $\mathcal{D}$ and if $c(i)$ is defined, then the sequent $\Delta_{c(i)}$ is immediately derived from Δ_i. $\mathcal{D}$ satisfies the *INST before GEN property* if it holds that for every subsequence $\Delta_{m_1}, \ldots, \Delta_{m_j}$ within $\mathcal{D}$, if $m_{i+1} = c(m_i)$ for $1 \leq i < j$, and if Δ_{m_1} is the result of the VAR, APP, or ABS rules, and if $\Delta_{m_2}, \ldots, \Delta_{m_j}$ are results of the INST and GEN rules, then there is a number k such that for $1 < i$ it is the case that Δ_{m_i} is a result of the INST rule if and only if $i < k$.

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

Proof: This is an immediate consequence (or restatement) of Theorem 3 in [5] which is itself a direct consequence of "the normalization property for second-order deductions given in [20]". ■

Theorem 3.3 (SUP $\leq$ TC) SUP *with a single binary function symbol and two pairs in each problem instance is reducible to* TC *in system* **F**.

Proof: Consider any instance Γ of SUP of the form $\Gamma = \{\, \tau_1 \dot{\leq} \mu_1,\ \tau_2 \dot{\leq} \mu_2 \,\}$. Let the free type variables in

τ_1, μ_1, τ_2, and μ_2 be contained within $\{\alpha_1, \ldots, \alpha_m\}$. Construct an instance of Typ from the SUP instance Γ. First, construct a λ-term M:

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

Then, construct a type assignment A:

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

We claim that Γ has a solution if and only if there is a typing in system **F** ending with the sequent $A \vdash M : \beta$. The two directions are proved separately.

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 $A \vdash M : \beta$ as follows. Let the type assignment $B = A \cup \{x : \forall.S(\tau_1) \rightarrow S(\tau_2)\}$. The following sequents are clearly derivable merely using the VAR and INST rules:

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

Now, since $S(\mu_1) = S_1(S(\tau_1))$ 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:

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

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:

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

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

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

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 $A \vdash M : \beta$. 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 assignment used in deriving a type for (cxx). Let σ be the type assigned to x by B, i.e. $B = A \cup \{x : \sigma\}$. The sequent that produces the final derived type for c must be the following sequent:

$$\begin{aligned} B \vdash c : &(T(\mu_1) \rightarrow T(\delta_1)) \\ &\rightarrow (T(\delta_2) \rightarrow T(\mu_2)) \\ &\rightarrow (T(\tau_1) \rightarrow T(\tau_2)) \end{aligned}$$

T is some function from $\mathbb{V}$ to $\mathbb{T}$ extended to an $\forall$-$\rightarrow$-homomorphism from $\mathbb{T}$ to $\mathbb{T}$. 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 $n \geq m$ and the bound type variables of τ are disjoint from this set, i.e. $\mathrm{FTV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}$ and $\mathrm{BTV}(\tau) \cap \{\alpha_1, \ldots, \alpha_n\} = \varnothing$.

Since 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. Since 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 $\rightarrow$-type. Thus, the sequent in $\mathcal{D}$ that produces the final derived type for the first occurrence of x must be exactly this:

(1) $$B \vdash x : T(\mu_1) \rightarrow T(\delta_1)$$

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

(2) $$B \vdash cx : (T(\delta_2) \rightarrow T(\mu_2)) \rightarrow (T(\tau_1) \rightarrow T(\tau_2))$$

Since (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:

(3) $$B \vdash x : T(\delta_2) \rightarrow T(\mu_2)$$

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

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

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

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

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:

$$(4) \qquad A \vdash \lambda x.cxx : \sigma \rightarrow \forall \vec{\varepsilon}.(T(\tau_1) \rightarrow T(\tau_2))$$

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

$$(5) \qquad A \vdash b : (\varphi \rightarrow \varphi) \rightarrow \beta$$

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:

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

In order for the APP rule to have been used this way, it must be the case that these types are equal:

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

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:

$$(6) \qquad B \vdash x : T_1(T(\tau_1)) \rightarrow T_1(T(\tau_2))$$

$$(7) \qquad B \vdash x : T_2(T(\tau_1)) \rightarrow T_2(T(\tau_2))$$

T_1 and T_2 are functions from $\mathbb{V}$ to $\mathbb{T}$ which must be the identity on any type variable not in $\{\alpha_1, \ldots, \alpha_n\}$ and which are extended to $\forall$-$\rightarrow$-homomorphisms from $\mathbb{T}$ to $\mathbb{T}$.

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

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

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

$$\begin{array}{rcl} erase(\alpha) & = & \alpha \\ erase(\sigma \rightarrow \tau) & = & erase(\sigma) \rightarrow erase(\tau) \\ erase(\forall \alpha.\sigma) & = & erase(\sigma) \end{array}$$

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. ■

4 Constrained types in derivations

Definition 4.1 (Invariant type assumption) Let M be a λ-term and A a type assignment such that $domain(A) = \mathrm{FV}(M)$. Let $\widehat{\mathcal{D}}$ be the set of all derivations such that for each $\mathcal{D} \in \widehat{\mathcal{D}}$ it holds that $\mathcal{D}$ satisfies the INST before GEN property of Definition 3.1 and there exists a type τ such that $\mathcal{D}$ ends with the sequent $A \vdash M : \tau$. Let $x \in \mathrm{BV}(M)$. Let ρ be an arbitrary type. Recall that $\mathcal{G}(\mathcal{D})$ denotes the global type assignment of $\mathcal{D}$. A and M induce the *invariant type assumption* $(x : \rho)$ if $\widehat{\mathcal{D}}$ is nonempty and for all $\mathcal{D} \in \widehat{\mathcal{D}}$ it is the case that there exists a renaming R of free type variables which is the identity on $\mathrm{FTV}(A)$ such that $(\mathcal{G}(\mathcal{D}))(x) = R(\rho)$.

Definition 4.2 ("$\blacktriangleright$") For two sets of types $\mathbb{X}, \mathbb{Y} \in \mathbb{T}$, write $\mathbb{X} \blacktriangleright \mathbb{Y}$ (spoken "the set of types $\mathbb{X}$ induces the set of types $\mathbb{Y}$") exactly when for every finite and non-empty subset $\mathbb{Z} \subseteq \mathbb{Y}$ there exist a context $C[\ \ ]$ and type assignments A and B such that all of the following listed requirements are satisfied. For a context $C[\ \ ]$ (with only one hole), define $\mathrm{BHV}(C[\ \ ])$ to be the subset of λ-bound variables in $\mathrm{BV}(C[\ \ ])$ whose scope includes the hole in $C[\ \ ]$.

1. $range(A) \subseteq \mathbb{X}$.
2. $domain(A) \supseteq \mathrm{FV}(C[\ \ ])$.
3. $A \subseteq B$.
4. $domain(B) = \mathrm{BHV}(C[\ \ ]) \cup domain(A)$.
5. $\mathbb{Z} \subseteq range(B)$.
6. For all M where $\mathrm{FV}(M) \subseteq domain(B)$ it is the case that there exists a type σ such that $B \vdash M : \sigma$ is derivable if and only if for every finite extension $A' \supseteq A$ such that $domain(A') \cap \mathrm{BHV}(C[\ \ ]) = \varnothing$ and $range(A') \subseteq \mathbb{X}$ there exists a type τ such that $A' \vdash C[M] : \tau$ is derivable.

Lemma 4.3 (Properties of "$\blacktriangleright$") *The "$\blacktriangleright$" relationship has the following properties. In these descriptions let $\mathbb{X}$, $\mathbb{Y}$, and $\mathbb{Z}$ be subsets of $\mathbb{T}$.*

1. *If $\mathbb{X} \blacktriangleright \mathbb{Y}$ then $\mathbb{X} \blacktriangleright \mathbb{X} \cup \mathbb{Y}$.*
2. *If $\mathbb{X} \blacktriangleright \mathbb{Y}$ and $\mathbb{Y} \blacktriangleright \mathbb{Z}$ then $\mathbb{X} \blacktriangleright R(\mathbb{Z})$ where R is a renaming that is the identity on $\mathrm{FTV}(\mathbb{Y})$ and*

maps the elements of $(\mathrm{FTV}(\mathbb{X}) - \mathrm{FTV}(\mathbb{Y}))$ *to fresh names.*

3. *If* $\mathbb{X} \blacktriangleright \mathbb{Y}$ *then* $\mathbb{X} \cup \mathbb{Z} \blacktriangleright R(\mathbb{Y})$ *where* R *is a renaming that is the identity on* $\mathrm{FTV}(\mathbb{X})$ *and maps the elements of* $(\mathrm{FTV}(\mathbb{Y}) - \mathrm{FTV}(\mathbb{X}))$ *to fresh names.*

4. *If* $\mathbb{X} \blacktriangleright \mathbb{Y}$ *then* $R(\mathbb{X}) \blacktriangleright R(\mathbb{Y})$ *for any renaming* R.

5. *If for all finite* $\mathbb{Y} \subseteq \mathbb{Z}$ *it is the case that* $\mathbb{X} \blacktriangleright \mathbb{Y}$, *then* $\mathbb{X} \blacktriangleright \mathbb{Z}$.

6. *If the type assignment* A *and* λ*-term* M *induce the invariant type assumption* $(x : \rho)$, *then it is the case that* $(\mathit{range}(A) \blacktriangleright \{\rho\})$.

The proof of Lemma 4.3 is omitted for lack of space.

Definition 4.4 (*height* **and** *parheight*) The auxiliary metric $\mathit{height}(\tau)$ simply 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 $k \geq 0$ and where $\beta \in \mathbb{V}$, define $\mathit{parheight}(\tau)$ to be the maximum of $\{\mathit{height}(\rho_1), \ldots, \mathit{height}(\rho_k), 0\}$.

Definition 4.5 ($\mathbb{B}$, $\mathbb{U}$, $\mathbb{C}$, $\mathbb{T}(k)$, $\mathbb{U}(k)$, $\mathbb{C}(k)$) Here are defined a number of sets of types within $\mathbb{T}$ which will be used throughout the rest of this section:

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

Lemma 4.6 ($\varnothing \blacktriangleright \mathbb{B}$) *There is a* λ*-term* J *such that* J *and the type assignment* $\varnothing$ *induce the invariant type assumptions* $(v : \forall\alpha.\alpha)$ *and* $(x : \alpha \to \alpha)$. *This implies that* $\varnothing \blacktriangleright \mathbb{B}$.

Proof: Throughout this proof, view a parse tree notation for types as being interchangeable with the regular notation, using the obvious correspondence. View the type symbol "$\to$" 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. 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:

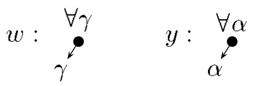
$$J \equiv (\lambda v.(\lambda y.\lambda z.v(yy)(yz))(\lambda x.\mathsf{K}x(x(xv)))(\lambda w.ww))$$

It is easy to check that J is typable with a derivation with the following global type assignment:

$$\{v\!:\!\bot, y\!:\!\forall\alpha.(\alpha\to\alpha)\to(\alpha\to\alpha), z\!:\!\bot\to\bot, x\!:\!\alpha\to\alpha, w\!:\!\bot\}$$

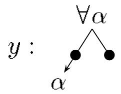
We now discover a very useful fact. It is the case that in all derivations for J, the λ-term variables x, y, and v must be assigned essentially the same types. In other words, there are invariant type assumptions for these variables. The analysis from here through the end of the proof leads to this conclusion.

Because of the subterm (ww), the type variable at the leaf at the end of the leftmost path in the type assumed for w is quantified at the root of the type. The same holds for y. 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 of length ≥ 0:



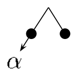
Since the abstraction over y is applied to the abstraction over x, the type assumed for y has a "$\to$" at its root, which is depicted like this:

Combining the two diagrams gives this result:



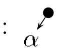
The type assumed for y must equal the final type derived for $(\lambda x.\mathsf{K}x(x(xv)))$. The only way there can be quantification at the root of the final derived type of this abstraction is if the GEN rule is applied, since the type derived for an abstraction by the ABS rule has no outermost quantifiers.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

(8) $(\lambda x.\mathsf{K}x(x(xv)))$:



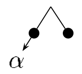
Now temporarily suppose that the type assumed for x did not match the following pattern, where the type variable α is free:

(9) x :

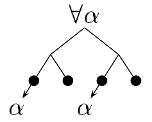


If the type assumption depicted in (9) did not hold, then the type variable at the end of the leftmost path in the assigned type of x would be closed within that type. If that were the case, then the derived type depicted in (8) could never happen. Hence, (9) must depict the type assumption for x in any derivation.

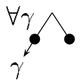
Due to the subterm (xv), the type assumed for x must have a "$\rightarrow$" at its root and cannot be simply α:

x : 

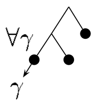
Thus, the type assumption for y matches this pattern:

y : 

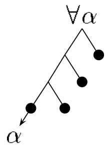
Considering again the type assumed for w and how this type must be embedded in the type assumed for z shows this:

(10) z : 

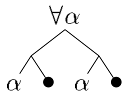
Consider the subterm (yz). The types assumed for y and z must be instantiated so that the left subtree of the instantiation of the type of y matches the instantiation of the type of z. Every instantiation of the type of z will match the pattern given in (10) for the type assumed for z. Thus, the type assumed for y must be instantiated to match this pattern:

(11) y : 

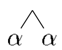
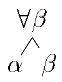
Now suppose the leftmost path in the type assumed for y is at least 3 edges long, in other words suppose this:

y : 

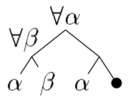
If this were the case, then the instantiation of the type assumed for y could never match the pattern in (11) because a quantifier owning the leftmost path cannot be inserted at the necessary spot in the type in (11) by instantiation. Thus, the leftmost path in the type of y must be exactly 2 edges long, 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 : 

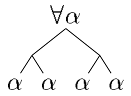
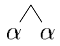
In the subterm $(x(xv))$, the final derived type for (xv) must be exactly α. Thus, the type assumed for x must match one of these two patterns, where α is free:

x :  or x : 

Suppose it were the latter, i.e. $x : \forall\beta.(\alpha\rightarrow\beta)$. Then the type assumed for y would have to match this pattern:

y : 

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

y :  x : 

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 $\rightarrow$-type. The only possible assumed type for v that can yield both of these derived types is $\bot$. This is the desired result. ■

Lemma 4.7 ($\varnothing \blacktriangleright \mathbb{B} \cup \{\forall.\alpha \rightarrow \beta \rightarrow \alpha\}$ in the λI-calculus) *There is a λ-term J_I of the λI-calculus such that the type assignment $\varnothing$ and J_I induce the invariant type assumptions $(v : \forall\alpha.\alpha)$, $(x : \alpha \rightarrow \alpha)$, and $(k : \forall.\alpha \rightarrow \beta \rightarrow \alpha)$.*

We omit the proof of Lemma 4.7 because it is similar to the proof of Lemma 4.6, because it is too long for this conference report, and because it is not necessary for proving that Typ is undecidable for system **F**. Lemma 4.7 is an alternative to Lemma 4.6 which can be used to perform all of the proofs in this section entirely within the λI-calculus. The important thing to note is that the type assigned to the term variable k allows k to be used to simulate the **K** combinator. By replacing every use of the combinator **K** in the proofs of the remaining lemmas by a free variable k which is assigned the type $\forall.\alpha \rightarrow \beta \rightarrow \alpha$, these proofs can be adapted to work in the λI-calculus.

Lemma 4.8 ($\mathbb{B} \cup \mathbb{U}(k) \blacktriangleright \mathbb{U}(k+1)$) *For any type τ in $\mathbb{U}(k+1)$, there exist a type assignment A such that $range(A) \subset \mathbb{B}\cup\mathbb{U}(k)$, a λ-term M such that $\mathrm{FV}(M) = domain(A)$, and a λ-term variable $x \in \mathrm{BV}(M)$ such that A and M induce the invariant type assumption $(x : \tau)$.*

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

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

Let the set of type variables $\{\alpha_1, \ldots, \alpha_n\}$ where $n \geq 2$ contain the $\forall$-bound type variables of τ, i.e. let $\mathrm{BTV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}$.

From τ, a type assignment A and a λ-term M having the desired properties will be defined. 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:

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

Let $\mu_1, \ldots, \mu_r$ be an enumeration of $\mathbb{W}$ possibly (probably) containing duplicates such that $n + c \le r$ and for $1 \le i \le n$ it holds that $\mu_i = \alpha_i$ and $\mu_{n+i} = \alpha_i \to \alpha_i$ and for $1 \le i \le c$ it holds that $\mu_{r-c+i} = \rho_i$.

Now define the type assignment A on the λ-term variables $\{a, b_1, \ldots, b_r\} \subset \mathcal{V}$. 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:

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

Now define the pieces of the λ-term M. First, for $1 \le i \le r$ 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 $m(i, h) = (((i + h) - 1) \bmod n) + 1$. ("mod" could be used more directly except that the set $\mathbb{S}$ is indexed starting with 1.)

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

Define the set $\vec{P}$ of λ-terms:

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

Let $P_1, \ldots, P_s$ be an arbitrary enumeration of $\vec{P}$.

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.

$$\begin{aligned}
M \equiv ((\lambda y.a(&Q_{n+g}^0(yQ_1^0 \cdots Q_r^0)) \\
&(Q_{n+g}^1(yQ_1^1 \cdots Q_r^1)) \\
&((\lambda x.a(Q_{n+g}^0(xQ_{r-c+1}^0 \cdots Q_r^0)) \\
&\qquad (Q_{n+g}^1(xQ_{r-c+1}^1 \cdots Q_r^1))) \\
&\ (y\overbrace{a \cdots a}^{r-c}))) \\
(\lambda z_1 \ldots z_r.&\mathsf{K} z_g(aP_1 \ldots P_s))).
\end{aligned}$$

It can be checked that the domain of A covers exactly the free variables of M and the range of A is restricted to $\mathbb{B} \cup \mathbb{U}(k)$. The end result of this construction is that the type assignment A and the λ-term M together induce the invariant type assumption $(x : \tau)$, which is the desired result. Because the full proof of this claim would add many more pages to this conference paper, we give only a tiny sketch here.

Consider some arbitrary typing $\mathcal{D}$ of the λ-term M using the type assignment A which satisfies the INST before GEN property of Definition 3.1. For any term variable w that occurs in the λ-term M, let the notation $[\![w]\!]$ stand for the type assumed for w in the derivation $\mathcal{D}$, i.e. let $[\![w]\!] = (\mathcal{G}(\mathcal{D}))(w)$. There is a renaming of free type variables R such that:

1. For $1 \le i \le r$, the type $[\![z_i]\!]$ is exactly $R(\mu_i)$.
2. For $\alpha_i \in \mathbb{S}$, it is the case that $R(\alpha_i) \notin \mathbb{S}$, i.e. $\mathbb{S} \cap R(\mathbb{S}) = \varnothing$.

After many steps, it is then obtained that the type $[\![y]\!]$ may be written as follows:

$$\forall \vec{\gamma_1}.\alpha_1 \to \cdots \to \forall \vec{\gamma_n}.\alpha_n \to \mu_{n+1} \to \cdots \to \mu_r \to \alpha_g$$

After several more steps, it is obtained that the type $[\![x]\!]$ may be written as follows:

$$[\![x]\!] = \forall.\rho_1 \to \cdots \to \rho_c \to \alpha_g = \tau$$

This is the desired result. ∎

Lemma 4.9 **($\mathbb{B} \cup \mathbb{U}(2) \cup \mathbb{C}(k) \blacktriangleright \mathbb{C}(k+1)$)** *For any type τ in $\mathbb{C}(k+1)$, there exist a type assignment A such that $range(A) \subset \mathbb{B} \cup \mathbb{U}(2) \cup \mathbb{C}(k)$, a λ-term M such that* $\mathrm{FV}(M) = domain(A)$*, and a λ-term variable $x \in$* $\mathrm{BV}(M)$ *such that A and M induce the invariant type assumption $(x : \tau)$.*

The proofs of Lemma 4.9 and Lemma 4.10 are omitted because they are too lengthy for this conference report, are similar to the proof of Lemma 4.8, and are not actually necessary to prove that Typ is undecidable for system **F**.

Lemma 4.10 **($\mathbb{B} \cup \mathbb{C} \blacktriangleright \mathbb{T}$)** *For any type τ in $\mathbb{T}$, there exist a type assignment A such that $range(A) \subset \mathbb{B} \cup \mathbb{C}$, a λ-term M such that* $\mathrm{FV}(M) = domain(A)$*, and a λ-term variable $x \in$* $\mathrm{BV}(M)$ *such that A and M induce the invariant type assumption $(x : \tau)$.*

Theorem 4.11 **($\varnothing \blacktriangleright \mathbb{T}$)** *The empty set of types $\varnothing$ induces the set of all types $\mathbb{T}$, i.e. $\varnothing \blacktriangleright \mathbb{T}$.*

Proof: This is a consequence of Lemmas 4.3, 4.6, 4.8, 4.9, and 4.10. ∎

5 Reduction from TC to Typ

Theorem 5.1 **(TC $\le$ Typ)** TC *in* **F** *is reducible to* Typ *in* **F**.

Proof: Consider an instance of TC in which it is asked whether the sequent $A \vdash M : \tau$ can be derived in **F**. By Theorem 4.11, there is a context $C[\ \ ]$ such that $A \vdash M : \tau$ is derivable if and only if $C[zM]$ is typable, where z is a fresh variable not occurring in the λ-term M and where the context $C[\ \ ]$ will force the λ-term variable z to be assigned the type $\tau \to \rho$ (where the type ρ is irrelevant). ∎

Theorem 5.2 (Main result) TYP *and* TC *in system* **F** *are of equivalent difficulty and are both undecidable.*

Proof: Recall that TYP easily reduces to TC [1] and that SUP is undecidable [12]. Then the result follows from Theorem 3.3 and Theorem 5.1 ∎

References

[1] H. 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.

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

[3] H. B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 1969.

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

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

[6] 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, Sept. 1991.

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

[8] F. Henglein and H. G. Mairson. The complexity of type inference for higher-order typed lambda calculi. In *Conf. Rec. 18th Ann. ACM Symp. Princ. Program. Lang.*, pp. 119–130, Jan. 1991. To appear in *J. Functional Programming*.

[9] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 1969.

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

[11] 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.

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

[13] 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.

[14] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Princ. Program. Lang.*, pp. 88–98, Austin, TX, U.S.A., Jan. 24–26, 1983.

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

[16] 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, Sophia-Antipolis, France, June 1984. Springer-Verlag.

[17] R. Milner. The standard ML core language. *Polymorphism*, 2(2), Oct. 1985.

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

[19] F. Pfenning and P. Lee. LEAP: a language with eval and polymorphism. In *Proc. Int'l Joint Conf. Theory Practice Softw. Develop.* Springer-Verlag, 1989.

[20] D. Prawitz. *Natural Deduction; A Proof-Theoretical Study*. Number 3 in Stockholm studies in philosophy. Almqvist and Wiksell, Stockholm, 1965.

[21] 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.

[22] 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.

[23] P. Urzyczyn. The emptiness problem for intersection types. Technical report, Institute of Informatics, University of Warsaw, Poland, Nov. 1993.

[24] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$ is undecidable. In *Int'l Conf. Typed Lambda Calculi and Applications*, vol. 664 of *LNCS*, pp. 418–432. Springer-Verlag, Mar. 1993.