The Essence of Principal Typings

J. B. Wells[1]*

Heriot-Watt University
http://www.cee.hw.ac.uk/~jbw/

Abstract. Let S be some type system. A *typing* in S for a typable term M is the collection of all of the information other than M which appears in the final judgement of a proof derivation showing that M is typable. For example, suppose there is a derivation in S ending with the judgement $A \vdash M : \tau$ meaning that M has result type τ when assuming the types of free variables are given by A. Then (A, τ) is a typing for M.
A *principal typing* in S for a term M is a typing for M which somehow represents all other possible typings in S for M. It is important not to confuse this with a weaker notion in connection with the Hindley/Milner type system often called "principal types". Previous definitions of principal typings for specific type systems have involved various syntactic operations on typings such as *substitution* of types for type variables, *expansion*, *lifting*, etc.
This paper presents a new general definition of principal typings which does not depend on the details of any particular type system. This paper shows that the new general definition correctly generalizes previous system-dependent definitions. This paper explains why the new definition is the right one. Furthermore, the new definition is used to prove that certain polymorphic type systems using $\forall$-quantifiers, namely System F and the Hindley/Milner system, do not have principal typings.

All proofs can be found in a longer version available at the author's home page.

1 Introduction

1.1 Background and Motivation

Why Principal Typings? A *term* represents a fragment of a program or other system represented in some calculus. In this paper, the examples will be drawn from the λ-calculus, but much of the discussion is independent of it. A typing t for a term in a specific type system for a calculus is *principal* if and only if all other typings for the same term can be derived from t by some set of semantically sensible operations. It is important not to confuse the *principal typings* property of a type system with the property of the Hindley/Milner system and the ML programming language often referred to (erroneously) as "principal types".

Principal typings allow *compositional* type inference, where the procedure of finding types for a term uses only the analysis *results* for its immediate subfragments, which can be analyzed independently in any order. Compositionality helps with such things as performing separate analysis of program modules

* This work was partly supported by EC FP5 grant IST-2001-33477, EPSRC grants GR/L 36963 and GR/R 41545, NATO grant CRG 971607, NSF grants 9806745, 9988529, and 0113193, and Sun Microsystems grant EDUD-7826-990410-US.

(and hence helps with separate compilation) and also helps in making a complete/terminating type inference algorithm. For a system lacking principal typings, any type inference algorithm must either be incomplete (i.e., sometimes not finding a typing even though one exists), be noncompositional, or use something other than typings of the system to represent intermediate results. An example of a noncompositional type inference algorithm is the $\mathcal{W}$ algorithm of Damas and Milner [6] for the Hindley/Milner (HM) type system which is used in programming languages like Haskell and Standard ML (SML). For an SML program fragment of the form `(let val x =` e_1 `in` e_2 `end)`, the $\mathcal{W}$ algorithm first analyzes e_1 and then uses the result while analyzing e_2.

Why Automated Type Inference? Principal typings help with automated type inference in general. For higher-order languages, the necessary types can become quite complex and requiring all of the types to be supplied in advance is burdensome. It is desirable to have as much *implicit typing* as possible, where types are omitted by humans who write terms. With type inference, the compiler takes an untyped or partially typed term, and it either completes the typing of the term or reports an error if the term is untypable.

Algorithm $\mathcal{W}$ for HM is the most widely used type inference algorithm. HM supports polymorphism with quite restricted uses of $\forall$ quantifiers. In practice, HM's limitations on polymorphic types make some kinds of code reuse more difficult [19]. Programmers are sometimes forced into contortions to provide code for which the compiler can find typings. This has motivated a long search for more flexible type systems with good type inference algorithms. Candidates have included extensions of the HM system such as System F [7, 25], $F_{\leq}$, F+η, or F_{ω}.

This search has yielded many negative results. For quite some time it seemed that HM was as good as a system could get and still have a complete/terminating type inference algorithm. Indeed, for many systems (F, F_{ω}, etc.), *typability* (whether an untyped program fragment can be given a type) has been proven undecidable, which means no type inference algorithm can be both complete and terminating for all program fragments. Wells proved this for System F [32, 34], the finite-rank restrictions of F above 3 [16], and F+η [33]. Urzyczyn proved it for F_{ω} [29], Pottinger proved it for unrestricted intersection types [24], and Pierce proved that even the subtyping relation of $F_{\leq}$ is undecidable [22]. Even worse, for System F it seems hard to find an amount of type information less than total that is enough to obtain terminating type inference [28].

Along the way have been a few positive results, some extensions of the HM system, and some with restricted intersection types (cf. recent work on intersection types of arbitrarily high finite ranks [17, 15]).

A New Principal Typing Definition vs. $\forall$ Quantifiers For many years it was not known whether some type systems with $\forall$ quantifiers could have principal typings. The first difficulty is simply in finding a sufficiently general system-independent definition of principal typings. The first such definition is given in this paper. A *typing* t is defined to be a pair $(A \vdash \tau)$ of a set A of *type*

assumptions together with a *result type* τ. The meaning $\mathsf{Terms}(t)$ of a typing $t = (A \vdash \tau)$ in a particular type system is defined to be the set of all the program fragments M such that $A \vdash M : \tau$ is provable in the system (meaning "M is well typed with result type τ under the type assumptions A"). A typing t_1 is defined to be *stronger* than typing t_2 if and only if $\mathsf{Terms}(t_1) \subset \mathsf{Terms}(t_2)$. This is "stronger" because t_1 is a stronger predicate on terms than t_2 and provides more information about $M \in \mathsf{Terms}(t_1)$, allowing M to be used in more contexts. A typing t is defined to be *principal* in some system for program fragment M if and only if t is at least as strong as all other typings for M in that system.

Comparison with prior type-system-specific principal typing definitions shows the new definition either exactly matches the old definitions, or is slightly more liberal, admitting some additional principal typings. The new definition seems to be the best possible system-independent definition.

Importantly, the new definition can be used to show that various systems with $\forall$ quantifiers do not have principal typings. Using this definition, a this paper proves that HM and System F do not have principal typings. Because the definition used here is liberal, the failure of HM and System F to have principal typings by this definition can be taken to mean that there is no reasonable definition of "principal typings" such that these systems have them. The proof for System F can be adapted for related systems such as F+η, and System F's finite-rank restrictions Λ_k for $k \geq 3$.

If polymorphism is to be based only on $\forall$ quantifiers, it is not clear how to design a type system with the principal typings property. Even systems with extremely restricted uses of $\forall$ quantifiers such as the HM system do not have principal typings. The lack of principal typings has manifested itself as a difficulty in making type systems more flexible than the restrictive HM system that also have convenient type inference algorithms. But this difficulty appears to have been due to the use of the $\forall$ quantifier for supporting type polymorphism, because many type systems with intersection types have principal typings.

1.2 Summary of this Paper's Contributions

1. An explanation is given of the motivations behind the notion of "principal typing" and how this has affected type inference in widely used type systems such as the simply typed λ-calculus and HM.
2. A new, system-independent definition of principal typings is presented. It is shown that this definition correctly generalizes existing definitions.
3. The new definition is used to finally prove that HM and System F do *not* have principal typings. Proving this was impossible before.

2 Definitions

This paper restricts attention to the pure λ-calculus extended with one constant c of ground type. The type systems considered here derive judgements traditionally of the form $A \vdash M : \tau$. Many interesting type systems derive judgements with

more information. To extend the machinery here to such type systems, the extra information should be considered part of the *typing* which is defined below.

Types Each type system S will have a set of types $\mathsf{Types}(S)$. Let σ, τ, and ρ range over types. Included in the set of types are an infinite set of *type variables*, ranged over by α, β, and γ. There will also be one ground type, o, added to help illustrate certain typing issues. For a type system S considered in this paper, the set $\mathsf{Types}(S)$ will be some subset of the set Types of types given by the following pseudo-grammar:

$$\sigma, \tau, \rho ::= \alpha \mid \mathsf{o} \mid (\sigma \to \tau) \mid (\sigma \cap \tau) \mid (\forall \alpha\, \tau)$$

The *free type variables* $\mathsf{FTV}(\tau)$ of the type τ are those variables not bound by $\forall$. Types are identified which differ only by α-conversion. Let s range over *type substitutions*, finite maps from a set of type variables to types.

Typings A pair $x{:}\sigma$ is a *type assumption*. A finite set of type assumptions is a *type environment*. Let A range over type environments. In this paper, type environments are required to mention each term variable at most once. Let $A(x)$ be undefined if x is not mentioned in A and otherwise be the unique type τ such that $(x{:}\tau) \in A$. Let $A_x = \{\, y{:}\tau \mid (y{:}\tau) \in A \text{ and } y \neq x \,\}$. Let $\mathsf{FTV}(A) = \bigcup_{(x:\tau)\in A} \mathsf{FTV}(\tau)$. Let $s(A) = \{\, (x{:}s(\tau)) \mid (x{:}\tau) \in A \,\}$.

Let a *typing judgement* be a triple of a type environment A, a λ-term M, and a result type τ with the meaning "M is well typed with result type τ under the type assumptions A". Rather than the traditional notation $A \vdash M : \tau$, this paper instead writes judgements in the form $M : (A \vdash \tau)$. Although this notation change simplifies the presentation, its real importance is that the new perspective will help dispel widespread misunderstanding of principal typings in the research community. The pair $A \vdash \tau$ is called a *typing*. Let t range over typings. If $t = (A \vdash \tau)$, let $\mathsf{FTV}(t) = \mathsf{FTV}(A) \cup \mathsf{FTV}(\tau)$ and let $s(t) = (s(A) \vdash s(\tau))$.

For a type system S, let the statement $S \rhd M : t$ hold iff the judgement $M : t$ is derivable using the typing rules of S. A type system S *assigns* typing t to term M iff $S \rhd M : t$. Let $\mathsf{Terms}_S(t) = \{\, M \mid S \rhd M : t \,\}$ and let $\mathsf{Typings}_S(M) = \{\, t \mid S \rhd M : t \,\}$.

Ordering Typings This paper introduces a new ordering on typings in a type system S according to how much information they provide about terms to which they can be assigned, with typings that are lower in the order providing more information. Let $t_1 \leq_S t_2$ iff $\mathsf{Terms}_S(t_1) \subseteq \mathsf{Terms}_S(t_2)$.

Remark 1. Suppose $t_1 \leq_S t_2$. Then typing t_1 can be viewed as providing more information about any $M \in \mathsf{Terms}(t_1)$. If t_1 and t_2 are viewed as predicates on terms, then $t_1(M)$ implies $t_2(M)$ for any M. In practice, if all that is known about a term is whether t_1 or t_2 can be assigned to it, then knowing that $M \in \mathsf{Terms}(t_1)$ represents an increase in knowledge over knowing that $M \in \mathsf{Terms}(t_2)$. This increased knowledge enlarges the set of contexts in which it is *known* that it is safe to use M. □

Some Specific Type Systems The formulation of the simply typed lambda calculus (STLC) presented here is in *Curry style*, meaning that type information is assigned to pure λ-terms [5]. The set Types(STLC) is the subset of Types containing all types which do not mention $\forall$ or $\cap$. The typing rules of STLC are CON, VAR, APP, and ABS:

$$\begin{array}{lll} \mathsf{CON} & & \Rightarrow \quad \mathsf{c} : (A \vdash \mathsf{o}) \\ \mathsf{VAR} & A(x) = \tau & \Rightarrow \quad x : (A \vdash \tau) \\ \mathsf{APP} & (M : (A \vdash \sigma \to \tau) \text{ and } N : (A \vdash \sigma)) & \Rightarrow \quad (MN) : (A \vdash \tau) \\ \mathsf{ABS} & M : (A_x \cup \{x{:}\sigma\} \vdash \tau) & \Rightarrow \quad (\lambda x.M) : (A \vdash \sigma \to \tau) \end{array}$$

The Hindley/Milner type system (HM) is an extension of STLC which was introduced by Milner for use in the ML programming language [19, 20]. The HM system introduces a syntactic form ($\mathsf{let}\ x = M\ \mathsf{in}\ N$) for allowing definitions with polymorphic types. The set Types(HM) is the subset of Types containing all types which do not mention $\cap$ and which do not mention $\forall$ inside either argument of a function type constructor ("$\to$"). In a typing $(A \vdash \tau)$, the type τ must not mention $\forall$. The typing rules of HM are CON, APP, and ABS from STLC and the new typing rules $\mathsf{VAR}_{\mathsf{HM}}$ and LET:

$$\begin{array}{ll} \mathsf{VAR}_{\mathsf{HM}} & (A(x) = \forall\vec{\alpha}.\tau \text{ and } \mathsf{dom}(s) = \{\vec{\alpha}\}) \quad \Rightarrow \quad x : (A \vdash s(\tau)) \\ \mathsf{LET} & \begin{array}{l} \quad (M : (A \vdash \tau) \text{ and } \{\vec{\alpha}\} = \mathsf{FTV}(\tau) \setminus \mathsf{FTV}(A) \text{ and } N : (A_x \cup \{x{:}\forall\vec{\alpha}.\tau\} \vdash \sigma)) \\ \Rightarrow\ (\mathsf{let}\ x = M\ \mathsf{in}\ N) : (A \vdash \sigma) \end{array} \end{array}$$

Girard formulated System F [7] (independently invented by Reynolds [25]) in the *Church style*, with explicitly typed terms. The Curry style presentation of F which is given here was first published by Leivant [18]. The set Types(F) is the subset of Types containing all types which do not mention $\cap$. The typing rules of F are CON, VAR, APP, and ABS from STLC and the new typing rules INST and GEN:

$$\begin{array}{lll} \mathsf{INST} & (M : (A \vdash \forall\alpha\,\sigma) \text{ and } s = \{\alpha \mapsto \tau\}) & \Rightarrow \quad M : (A \vdash s(\sigma)) \\ \mathsf{GEN} & (M : (A \vdash \sigma) \text{ and } \alpha \notin \mathsf{FTV}(A)) & \Rightarrow \quad M : (A \vdash \forall\alpha\,\sigma) \end{array}$$

3 History of Principal Typings

3.1 Basic Motivations and STLC

The notions of *principal type* and *principal typing* (which has also been called *principal pair*) first occurred in the context of type assignment systems for the λ-calculus or combinatory logic using simple types. The motivation was determining whether a term is typable and finding types for the term if it is typable. The key idea is to define the typing algorithm by structural recursion on terms. This means that in calculating types for a term M, the algorithm will invoke itself recursively on the immediate subterms of M. For this to work, the result returned by the recursive invocations must be sufficiently informative.

Example 1. This example illustrates the need for some sort of "most general" typing in the process of inferring type information. Consider these λ-terms:

$$M = \lambda z.NP \qquad N = \lambda w.w(wz) \qquad P = \lambda yx.x$$

A type inference algorithm Inf for STLC defined using structural recursion would generate a call tree like this:

$$\mathsf{Inf}(M) = \mathsf{Cmb}_\lambda(z, \mathsf{Inf}(NP)) = \mathsf{Cmb}_\lambda(z, \mathsf{Cmb}_@(\mathsf{Inf}(N), \mathsf{Inf}(P)))$$

Here the algorithm Inf uses two subalgorithms Cmb_λ and $\mathsf{Cmb}_@$ to combine the results from recursively processing the subterms.

Suppose the recursive call to $\mathsf{Inf}(P)$ were to return the typing $t_1 = (\emptyset \vdash \alpha \to \alpha \to \alpha)$, which is a derivable STLC typing for P. Unfortunately, there would be no way that $\mathsf{Cmb}_@$ could combine this result with any typing for N to yield a typing for NP. The application $(w(wz))$ inside N needs w to have a type of the shape $\sigma \to \sigma$ for some σ. This could be solved by using a typing like $t_2 = (\emptyset \vdash (\alpha \to \alpha) \to \alpha \to \alpha)$ for P.

However, the only thing that $\mathsf{Cmb}_@$ knows about the subterm P is the typing t_1, which does not imply the typing t_2. This can be seen from the following example of a term $P' \in \mathsf{Terms}(t_1) \setminus \mathsf{Terms}(t_2)$:

$$P' = \lambda xy.(\lambda z.x)(\lambda w.wx(wyx))$$

To see more precisely why $P' \in \mathsf{Terms}(t_1)$, here is a type-annotated version:

$$P' = \lambda x^\alpha.\lambda y^\alpha.(\lambda z^{(\alpha\to\alpha\to\alpha)\to\alpha}.x)(\lambda w^{\alpha\to\alpha\to\alpha}.wx(wyx))$$

It should also be clear why $P' \notin \mathsf{Terms}(t_2)$ — the types of x and y are forced to be the same by the applications (wx) and (wy), and this prevents P' from having a result type of the shape $\sigma \to \sigma$. Thus, $t_1 \not\leq t_2$.

The problem here is that the result that $\mathsf{Inf}(P)$ returned is not the most general result. It would have been more useful to have returned the result $t_3 = (\emptyset \vdash \beta \to \alpha \to \alpha)$. In fact, t_3 is in a certain sense the *best* possible result, because it can be checked that $t_3 \leq t$ for every $t \in \mathsf{Typings}(P)$. □

To avoid the kind of problems mentioned in example 1, type inference algorithms have been designed so that their intermediate results for subterms are, in some sense, most general. There have been several different ways of characterizing the needed notions of "most general". Hindley [9] gives the following definitions which were intended for use with STLC.

Definition 1 (Hindley's Principal Type). *A* principal type *in system S of a term M is a type τ such that*
1. there exists a type environment A such that $S \rhd M : (A \vdash \tau)$, and
2. if $S \rhd M : (A' \vdash \tau')$, then there exists some s such that $\tau' = s(\tau)$. □

Notice that definition 1 completely ignores the type environments! Hindley used the name "principal pair" for what is called here "principal typing".

Definition 2 (Hindley's Principal Typing). *A* principal typing *in system S of a term M is a typing $t = (A \vdash \tau)$ such that*
1. *$S \rhd M : t$, and*
2. *if $S \rhd M : t'$ for some typing $t' = (A' \vdash \tau')$, then there exists some s such that $A' \supseteq s(A)$ and $\tau' = s(\tau)$.* □

Clearly, if $t = (A \vdash \tau)$ is a principal typing for a term M, then the result type τ is a principal type. The key property satisfied by STLC w.r.t. these definitions is the following. (See [9] for the history of this discovery.)

Theorem 1 (Principality for STLC). *Every term typable in* STLC *has a principal typing and a principal type. Also, there is an algorithm that decides if a term is typable in* STLC *and if the answer is "yes" outputs the principal typing.* □

Hindley gave a further definition of a *principal derivation* (called by Hindley "deduction") which is not needed for this discussion. These definitions of Hindley essentially represent the earlier approaches of Curry and Feys [5] and Morris [21]. There are two important aspects of this approach:
1. The notion of "more general" is tied to substitution and weakening. For STLC, this exactly captures what is needed, but this fails for more sophisticated type systems.
2. The literature using these definitions freely switches between "principal type" and "principal typing" (or "principal pair"). The algorithms for STLC which are described as having the goal of calculating the "principal type" in fact are designed to calculate principal typings. Because for STLC every term has both a principal typing and a principal type, many people did not pay much attention to the difference. For more sophisticated type systems the difference becomes important.

3.2 Type Polymorphism and HM

Although STLC is well behaved, in practice it is quite insufficient for programming languages. To overcome the limitations of STLC, various approaches to adding type polymorphism have been explored and for each approach efforts have been directed to the problem of type inference.

One approach to adding type polymorphism is System F, which was discovered around the beginning of the early 1970s. Towards the end of that decade people were thinking about Curry-style presentations of F and how to perform type inference for it [18]. In the mid-1990s, I proved that typability for F is undecidable and that therefore there is no complete and always terminating type inference algorithm for F [32, 34]. Later in this paper, the further result that F does not have principal typings is proven.

So far, the most successful and widely used approach to adding type polymorphism is the Hindley/Milner (HM) system, an extension of STLC and also a restriction of F. The approach to type inference for HM differs from that for STLC, because Hindley's notion of principal typing (needed by the type inference algorithms used for STLC) quite clearly does not hold for HM.

Example 2. This example illustrates why definition 2 is not useful for HM. Consider the λ-terms $M = (\mathsf{let}\ x = (\lambda y.y)\ \mathsf{in}\ N)$ and $N = (xx)$. The term M is typable in HM. For example, the judgement $M : t$ where $t = (\emptyset \vdash \alpha \to \alpha)$ can be derived in HM. A derivation of this typing might use as an intermediate step the assignment of the typing $t_1 = (\{x{:}\forall\beta.\beta \to \beta\} \vdash \alpha \to \alpha)$ to the subterm N. Given any σ, let σ^0 stand for σ and let σ^{i+1} stand for $\sigma^i \to \sigma^i$. Thus, using the new abbreviation, $t_1 = (\{x{:}\forall\beta.\beta^1\} \vdash \alpha^1)$. The subterm N can in fact be assigned for any $i \geq 0$ the typing $t_i = (\{x{:}\forall\beta.\beta^i\} \vdash \alpha^i)$. And for distinct $i, j \geq 0$, there is no substitution s such that $t_i = s(t_j)$. Note that the type $\forall\beta.\beta^i$ is closed and $s(\tau) = \tau$ for any closed type τ and substitution s. Furthermore, it is not hard to check for $i \geq 0$ that for any other typing t' assignable to N, that there is no substitution s such that $t_i = s(t')$. So N has no principal typing using definition 2. In contrast, the term M does have a principal typing by that definition, namely t. Although some HM-typable terms (e.g., N) have no principal typings by definition 2, it turns out that any HM-typable term with no free variables does. □

Until this paper, it was not known whether we were simply not clever enough to conceive of a set of operations which would yield all other HM typings from (hypothetical) HM principal typings. Later in this paper, it is shown that there is no reasonable replacement definition of principal typing that will work for HM.

Milner's cleverness was in finding a way around this problem. The key lies in in the following definition (a clear statement of which can be found in [8]).

Definition 3 (A-Typable and A-Principal).

1. *A term M is* A-typable *in* HM *iff there is some A' mentioning only monotypes (types without any occurrence of "$\forall$") and some type τ such that $M : (A \cup A' \vdash \tau)$ is derivable in* HM.
2. *A typing $(A \cup A' \vdash \tau)$ is* A-principal *for term M in* HM *iff*
   (a) *A' mentions only monotypes,*
   (b) *$M : (A \cup A' \vdash \tau)$ is derivable in* HM*, and*
   (c) *whenever $M : ((s(A)) \cup A'' \vdash \tau')$ is derivable for A'' mentioning only monotypes, there is a substitution s' such that $A'' \supseteq (s'(A'))$, $\tau' = s'(\tau)$, and $s'(\alpha) = s(\alpha)$ for $\alpha \in \mathsf{FTV}(A)$.* □

The property that HM satisfies w.r.t. the above definition is the following, due to Damas and Milner [6].

Theorem 2 (Principality for HM). *If a term is A-typable in* HM*, then it has a A-principal typing in* HM*. Also, there is an algorithm that decides if a term is A-typable in* HM *and if the answer is "yes" outputs its A-principal typing.* □

It is not hard to see that a closed term (with no free variables) is A-typable iff it is $\emptyset$-typable. So theorem 2 implies that typability is decidable for closed programs and that closed programs have $\emptyset$-principal typings. This is good enough for use in a type inference algorithm for a programming language implementation and, indeed, the HM type system has been very widely used as a result. There are some drawbacks that should be noticed:

1. In order to take advantage of the notion of A-principality, any polymorphic types to be used in a term M must be determined before analyzing M. So in analyzing a subterm of the shape ($\mathsf{let}\ x = N\ \mathsf{in}\ M$), the subterm N must be completely analyzed before M and the result of analyzing N is used in analyzing M. This is the behavior of Milner's algorithm $\mathcal{W}$ [19].
2. Because the only notion of principality requires fixing the part of the type environment containing polytypes, for an arbitrarily chosen HM-typable term there does not seem to be an HM typing which represents *all* possible typings for that term. Later in this paper it is shown in fact that this is not mere appearance — individual HM typings can not be used to represent all possible HM typings for a term. This makes it more difficult to use the HM type system for approaches that involve *incremental* or *separate* computation of type information. So HM may not be right for some applications.

3.3 Principal Typings with Intersection Types

At the present time, for a type system to support both type polymorphism and principal typings, the most promising approaches rely on the use of intersection types. There is not room in this paper to go into much detail about intersection types, so the discussion here gives only the highlights.

The first system of intersection types for which principal typings was proved was presented by Coppo, Dezani, and Venneri [2] (a later version is [3]). The same general approach has been followed by Ronchi Della Rocca and Venneri [27] and van Bakel [30] for other systems of intersection types. In this approach, finding a principal typing algorithm for a term M involves

- finding a normal form (or *approximate* normal form) M' for M,
- assigning a typing t to M',
- proving that any typing for the normal form M' is also a typing for the original term M, and
- proving that any other typing t' for the normal form M' can be obtained from t by a sequence of operations each of which is one of *expansion* (sometimes called *duplication*), *lifting* (implementing subtyping, sometimes called *rise*), or *substitution*.

This general approach is summarized nicely in [31, §5.3]. This is intrinsically an impractical method and hence is primarily of theoretical interest. The definitions of the operations on typings are sometimes quite complicated, so they will not be discussed in this paper.

The first unification-based approach to principal typing with intersection types is by Ronchi Della Rocca [26]. An always-terminating restriction is presented which bounds the height of types. Unfortunately, this approach uses a complicated approach to expansion and is thus quite difficult to understand.

The first potentially practical approaches to principal typing with intersection types were subsequent unification-based methods which focused on the rank-2 restriction of intersection types. Van Bakel presented a unification algorithm for principal typing for a rank-2 system [30]. Later independent work by Jim also attacks the same problem, but with more emphasis on handling

practical programming language issues such as recursive definitions, separate compilation, and accurate error messages [14]. Successors to Jim's method include Banerjee's [1], which integrates flow analysis, and Jensen's [12], which integrates strictness analysis. Other approaches to principal typings and type inference with intersection types include [4] and [11].

The most recent development in this area is the introduction of the notion of *expansion variables* [17]. The key idea is that with expansion variables, the earlier notions of expansion and substitution can be integrated in a single notion of substitution called β-substitution. This results in a great simplification over earlier approaches beyond the rank-2 restriction. However, there are still many technical issues needing to be overcome before this is ready for general use.

3.4 An Observation about all Previous Definitions

The following holds for each system S with principal typings that I know about. In S, each typable term M can be assigned a typing t which is *principal for* M in the sense that for every other typing t' assignable to M, there exist operations $O_1, \ldots, O_n$ such that

- $t' = O_n(\cdots(O_1(t))\cdots)$, and
- the operations are *sound* in the sense that for any term N, if $S \rhd N : t$, then $S \rhd N : t_i$ where $t_i = O_i(\cdots(O_1(t))\cdots)$ for $1 \leq i \leq n$.

For some (but not all) systems a stronger statement about the soundness of the operations holds:

For $0 \leq i < n$ and any term N, if $S \rhd N : t'$, then $S \rhd N : O_i(t')$.

For STLC, these operations are substitution and weakening, which are sound in the stronger sense (provided weakening is defined to do nothing on typings already mentioning the term variable in question). For various systems with intersection types, these operations are expansion, lifting, rise, as well as substitution and weakening. In some systems with intersection types, these operations are sound in the stronger sense, and in others, they are sound in the weaker sense.

Observation 3 *In each such system S, if t is principal for M, then $t \leq_S t'$ for every $t' \in \mathsf{Typings}_S(M)$.* □

4 A New Definition of Principal Typing

In designing a general definition of "principal typing", the important issue seems to be the following:

> A *principal typing* in S for a term M should be a typing for M which somehow represents all other possible typings in S for M.

This paper has already introduced the new technical machinery necessary to capture this notion in the information order $\leq_S$ on typings. This suggests that the following new definition is the right one.

Definition 4 (Principal Typing). *A typing t is* principal *in system S for term M iff $S \rhd M : t$ and $S \rhd M : t'$ implies $t \leq_S t'$.* □

4.1 Positive Results

The first thing to check about the new definition is whether existing definitions for specific type systems are instances of the new definition. In all cases, every typing that is principal by one of the old definitions is principal by the new one. This is justified by observation 3.

It remains to be considered whether every typing in a system S that is principal by the new definition is also principal by the old one for S. For STLC, the new definition corresponds exactly to the old definition.

Theorem 4. *A typing t is principal in* STLC *for M according to definition 2 iff t is principal in* STLC *for M according to definition 4.* □

For some other type systems, the new definition will be slightly more liberal, accepting some additional typings as being principal. For example, consider the term $\omega = (\lambda x.xx)$. The usual principal typing of ω in a system with intersection types is $t_1 = (\emptyset \vdash ((\alpha \to \beta) \cap \alpha) \to \beta)$. With the new definition, $t_2 = (\emptyset \vdash ((\alpha \to \beta) \cap \alpha \cap \gamma) \to \beta)$ is also a principal typing, because in some systems with intersection types a term can be assigned t_1 iff it can be assigned t_2. This is merely a harmless quirk. The old definitions ruled out unneeded intersections with type variables because they were inconvenient.

4.2 Type Systems without Principal Typings

The new definition 4 of principal typings can be used to finally prove that certain type systems do *not* have principal typings. These results are significant, as clarified by this statement by Jim in [13] about the best previous knowledge:

> "This imprecision [in the definition of principal typings] makes it impossible for us to *prove* that a given type system lacks the principal type property."

Theorem 5. *The* HM *system does not have principal typings for all terms.* □

It is quite important that the research community is made aware of Theorem 5, because as Jim points out in [13], "a number of authors have published offhand claims that ML possesses the principal typings property".

Theorem 6. *System* F *does not have principal typings for all terms.* □

References

[1] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*. ACM Press, 1997.

[2] M. Coppo, M. Dezani-Ciancaglini, B. Venneri. Principal type schemes and λ-calculus semantics. In Hindley and Seldin [10].

[3] M. Coppo, M. Dezani-Ciancaglini, B. Venneri. Functional characters of solvable terms. *Z. Math. Logik Grundlag. Math.*, 27(1), 1981.

[4] M. Coppo, P. Giannini. A complete type inference algorithm for simple intersection types. In *17th Colloq. Trees in Algebra and Programming*, vol. 581 of *LNCS*. Springer-Verlag, 1992.

[5] H. B. Curry, R. Feys. *Combinatory Logic I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.
[6] L. Damas, R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, 1982.
[7] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'Etat, Université de Paris VII, 1972.
[8] R. Harper, J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. on Prog. Langs. & Systs.*, 15(2), 1993.
[9] J. R. Hindley. *Basic Simple Type Theory*, vol. 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1997.
[10] J. R. Hindley, J. P. Seldin, eds. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.
[11] B. Jacobs, I. Margaria, M. Zacchi. Filter models with polymorphic types. *Theoret. Comput. Sci.*, 95, 1992.
[12] T. Jensen. Inference of polymorphic and conditional strictness properties. In POPL '98 [23].
[13] T. Jim. What are principal typings and what are they good for? Tech. memo. MIT/LCS/TM-532, MIT, 1995.
[14] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.
[15] A. J. Kfoury, H. G. Mairson, F. A. Turbak, J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming.* ACM Press, 1999.
[16] A. J. Kfoury, 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.
[17] A. J. Kfoury, J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999.
[18] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Princ. of Prog. Langs.*, 1983.
[19] R. Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17, 1978.
[20] R. Milner, M. Tofte, R. Harper, D. B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
[21] J. H. Morris. *Lambda-calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Mass., U.S.A., 1968.
[22] B. Pierce. Bounded quantification is undecidable. *Inform. & Comput.*, 112, 1994.
[23] *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.
[24] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In Hindley and Seldin [10].
[25] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, vol. 19 of *LNCS*, Paris, France, 1974. Springer-Verlag.
[26] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2), 1988.
[27] S. Ronchi Della Rocca, B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2), 1984.
[28] A. Schubert. Second-order unification and type inference for Church-style polymorphism. In POPL '98 [23].
[29] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Structures Comput. Sci.*, 7(4), 1997.
[30] S. J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Catholic University of Nijmegen, 1993.
[31] S. J. van Bakel. Intersection type assignment systems. *Theoret. Comput. Sci.*, 151(2), 1995.
[32] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. Logic in Comput. Sci.*, 1994. Superseded by [34].
[33] J. B. Wells. Typability is undecidable for F+eta. Tech. Rep. 96-022, Comp. Sci. Dept., Boston Univ., 1996.
[34] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3), 1999. Supersedes [32].