Imperative Lambda Calculus Revisited

Hongseok Yang **Uday Reddy**
Department of Computer Science
The University of Illinois at Urbana-Champaign
{hyang,reddy}@cs.uiuc.edu

August 27, 1997

Abstract

Imperative Lambda Calculus is a type system designed to combine functional and imperative programming features in an orthogonal fashion without compromising the algebraic properties of functions. It has been noted that the original system is too restrictive and lacks the subject reduction property. We define a revised type system that solves these problems using ideas from Reynolds's Syntactic Control of Interference. We also extend it to handle Hindley-Milner style polymorphism and devise type reconstruction algorithms. A sophisticated constraint language is designed to formulate principal types for terms.

1 Introduction

The recent research in programming languages has greatly clarified the interaction between imperative and functional programming. The conventional notion that functional programming and imperative state-manipulation are in conflict has been dispelled. Several programming languages have now been designed which combine functions and assignments without destroying algebraic properties of either. These languages achieve "the best of both worlds", combining the usefulness of assignments with the semantic elegance of higher-order functional programming.

The languages presented in the literature fall into three classes:

- untyped languages [ORH93],
- typed languages with command types, but no state types [SRI91, CO94], and
- typed languages with typed state [LP95].

Our focus is on the second class of languages, which avoid the run-time overhead of the untyped languages as well as the conceptual overhead of typed state. These languages are also closer in conception to imperative languages based on Algol [Rey97]. Unfortunately, the type systems designed for these languages are not satisfactory. They have severe restrictions that disallow

```
factorial :: Int -> Int
factorial n = run (newref 1 >> \p ->
                   let loop n = if n = 0 then skip
                                else
                                  get p >> \x ->
                                  put p (x * n) $
                                  loop (n-1)
                   in loop n $ get p)
```

Figure 1: Example program for factorial

reasonable programs. They lack subject reduction property. And, writing polymorphic functions is cumbersome. In this paper, we define an improved type system that solves these problems.

An important feature of the *Imperative lambda calculus*(ILC) type system of [SRI91] is the notion of *encapsulated state threads* which carry out state-manipulation internally but appear as applicative values to the outside. The example program in Fig. 1 (in Haskell syntax) illustrates this. The *factorial* function returns an integer, but, internally, it computes the result by allocating state and destructively manipulating it. The body of the `run` operator in the definition is a *command* computing an integer. The `run` operator must encapsulate the command and ensure that no state information escape out of it.[1] The ILC type system ensures this property by using a rule of "promotion": a command that returns an applicative value can be regarded as an applicative value if all its free variables are of applicative types. The rationale is that if no imperative values, such as references and commands, are imported into the command then it has no visible side effects and every invocation must produce the same value. The soundness of the rule has been shown by denotational methods [SRI91].

Unfortunately, the ILC type system is too restrictive. Consider the modified factorial program in Fig. 2, where it is defined using a (library) function for "for loops". Since the type of the for loop combinator is an imperative type, the body of `run` violates the promotion rule. This program is prohibited by the ILC type system. Thus, the restrictions of ILC prohibit *modularity* which is meant to be the mainstay of functional programming languages.

Intuitively, it would seem that the for-loop combinator is an "applicative entity" as it merely composes several versions of its argument without adding any imperative effects of its own. But, there is no mechanism in the original ILC type system to recognize such applicative entities. An ad hoc fix is used in [CO94] where all let-bound identifiers are expanded out for the purpose of type checking. This handles the example in Fig. 2 because the binding of `for` has no free identifiers. But, naturally, this fix would be of no value for λ-bound identifiers.

[1] Note that programs like factorial can be written in traditional imperative languages as well as ML-like languages. The techniques of this paper are applicable to all such languages to ensure that functions do not have side effects.

```
for :: (Int, Int) -> (Int -> Cmd ()) -> Cmd ()
for (i,j) proc = if i > j then skip
                 else proc i $ for (i+1, j) proc
factorial :: Int -> Int
factorial n = run (newref 1 >> \p ->
                   for (1,n) (\i ->
                       get p >> \x ->
                       put p (x * i)) $
                   get p)
```

Figure 2: A problematic program for factorial

The second problem is that the ILC type system lacks the *subject reduction* property. Both [SRI91] and [CO94] claim subject reduction without adequate proof. But, it is now widely recognized that type systems with promotion constructs have subtle subject reduction issues [Rey78, O'H91, Wad93, BBdPH92]. Suppose `run`(M) is a promoted term with a free variable x of type `Int`. During reduction, we can substitute for x a term that has imperative-typed free variables. The resulting term `run`$(M[N/x])$ is not well-typed because the promoted command has free variables of imperative types. For example, with free variables `size :: Array t -> Int` and `a :: Array (Ref Int)`, we can form a term that has the following reduction:

`(\x-> run (return x)) (size a)` $\longrightarrow$ `run (return (size a))`

While the left hand side term type checks, the right hand side term does not. Thus, subject reduction fails. While this problem may not be fatal, it points to the limitations of the type system. In particular, it is unable to cover the reducts of terms that are already accepted by the system.

Finally, the ILC type system is cumbersome for writing polymorphic functions. Since the rule of promotion involves a strict separation between applicative types and imperative types, there are two "kinds" of types involved. Thus, one needs two kinds of type variables and most polymorphic functions need to come in (at least) two versions which differ only in the type variables used.

Our solutions for these problems are based on the pioneering work of Reynolds [Rey78] on "syntactic control of interference" and the recent advances on the topic [Rey89, Red94, Red93, OPTT95]. While the main thrust of this work is the control of interference (to facilitate reasoning), embedded in it is a comprehensive treatment of "passivity" which addresses the issues of concern to us. We believe the separation of the "passivity" issues from interference control is valuable because they are of more fundamental importance in language design.

The contributions of this paper include the reformulation of the ILC type system by relaxing its restrictions and ensuring subject reduction property, its extension to ML-style polymorphism and type inference algorithms for the polymorphic type system. This last part of the work poses significant challenges. As will be seen, ensuring subject reduction property using Reynolds's ideas involves quite unorthodox features in the type system. In particular, there will be two "kinds" of types corresponding to the two layers of types, and these kinds will have structural implications for the allowed type derivations. The principal type property is obtained by enriching the type system with "type constraints" which characterize the possible kinds for type variables in order for a term to be well-typed.

The revised type system of ILC has been included in the revised version of [SRI91] appearing as [SRI97]. The formal properties and semantic issues have been studied there. However, the polymorphic type system presented here is new.

This summary is organized for the program committee readership. In Sec. 2, we review the issues in ILC type system. Our revised type system is presented in Sec. 3. Sections 4 and 5 define polymorphic type systems, and section 6 sketches the type reconstruction algorithm.

1.1 Related Work

Peyton Jones and Launchbury [LP95] propose a type system for extending Haskell with imperative features (called "state threads"). Their system differs from ILC in that they eschew the division of types into imperative and applicative types, but use additional type parameters for the imperative type constructors that express state-dependence. These additional type parameters represent a conceptual overhead which we would like to avoid. Rabin [Rab96] attempts to combine the features of ILC and state threads though he did not consider type reconstruction issues.

Huang and Reddy [HR96] present a type reconstruction system for syntactic control of interference. The promotion and dereliction operations are explicitly represented in the term syntax. In contrast, we make them implicit which gives rise to a harder type inference problem.

Some features of our work resemble linear logic-based type systems [Wad91, GH90], and effect systems [LG88, TJ94]. There are also close connections to subtyping systems [Mit91, FM90].

Reynolds [Rey89] has devised an alternative type system for syntactic control of interference using conjunctive types. These ideas could also be used in the context of ILC, though we do not consider them here.

2 Issues in the ILC type system

Consider a typed lambda calculus with the following types:

$$\theta \quad ::= \quad \texttt{Int} \mid \texttt{Bool} \mid \texttt{Ref}\ \theta \mid \texttt{Cmd}\ \theta \mid () \mid \theta \times \theta' \mid \theta \to \theta'$$

The term syntax is that of typed lambda calculus. (We continue to use Haskell syntax in examples.) The following (families of) constants are used for supporting imperative computations:

$$
\begin{array}{lcl}
\texttt{return} & : & \theta \to \texttt{Cmd}\ \theta \\
(\texttt{>>}) & : & \texttt{Cmd}\ \theta \to (\theta \to \texttt{Cmd}\ \theta') \to \texttt{Cmd}\ \theta' \\
\texttt{get} & : & \texttt{Ref}\ \theta \to \texttt{Cmd}\ \theta \\
\texttt{put} & : & \texttt{Ref}\ \theta \to \theta \to \texttt{Cmd}\ () \\
\texttt{newref} & : & \theta \to \texttt{Cmd Ref}\ \theta
\end{array} \tag{1}
$$

Briefly, `return` x is a trivial command that makes no change to state and returns x. If $c : \texttt{Cmd}\ \theta$ and $f : \theta \to \texttt{Cmd}\ \theta'$ then $c \mathbin{>\!>} f$ is the command that carries out the state transformer c, obtaining a θ-typed value x in the process, and then carries out the state transformer $f(x)$. The constants `get` and `put` are used for reading and writing references, while `newref` x is a command that produces a new reference initialized to x. We also use the following constants as abbreviations:

$$
\begin{array}{lcl@{\qquad}lcl}
\texttt{skip} & : & \texttt{Cmd}\ () & \texttt{skip} & \equiv & \texttt{return}\ () \\
(\$) & : & \texttt{Cmd}\ \theta \to \texttt{Cmd}\ \theta' \to \texttt{Cmd}\ \theta' & c \,\$\, c' & \equiv & c \mathbin{>\!>} \lambda x.\, c'
\end{array}
$$

The example in Fig. 1 illustrates these constants. Programming examples of more practical interest may be found in [SRI91, ORH93, LP95].

To add a `run` operator to the language, we would like to designate a *subclass* of types as *applicative types*. These types should at least include:

$$\tau ::= \texttt{Int} \mid \texttt{Bool} \mid () \mid \tau \times \tau' \mid \theta \to \tau'$$

But, as mentioned in Introduction, the type of the `for`-loop combinator is not among them. Intuitively, the reason we expect the for-loop combinator to be applicative is that it is formed without using any imperative-typed free variables, and, hence, its meaning is independent of state. Let us agree to call such values "passive values." To capture this property in the type system, we add a new type constructor denoted "!". (Our notation is inspired by linear logic [Gir87, Wad91], but, unlike linear logic, there are no restrictions on weakening and contraction of !-typed values.) For any type θ, $!\theta$ is a type whose values are passive values of type θ. Since the for-loop combinator is passive it can be given the type:

```
for :: !((Int, Int) -> (Int -> Cmd ()) -> Cmd ())
```

We classify all types of the form $!\theta$ as applicative types. So, free variables of such types can appear in promoted terms.

Our ! types are closely related to Reynolds's [Rey78] passive function types constructed by $\to_P$. His type $\theta \to_P \theta'$ corresponds to $!(\theta \to \theta')$ in our notation.[2] We allow passivity annotations

[2] In Reynolds's setting, passive functions can access global references for reading, though not for writing. Our

to non-function types as well. The ! types also correspond to universally quantified types (over state types) in Launchbury-Peyton Jones type system [LP95]. If θ corresponds to a parameterized type $T(s)$ in their system, where the parameter s denotes the state type, then $!\theta$ corresponds to $\forall s.\, T(s)$. However, !-types in our system are first-class types whose values can be inputs and outputs of functions. This is in contrast to the restricted fashion in which quantified types are treated in the Launchbury-Peyton Jones system. (They are forced to treat them in a restricted fashion to avoid the undecidability issues in type inference with quantified types [Wel94].)

We can now add a constant:

$$\texttt{run} \quad : \quad !(\texttt{Cmd}\ \tau) \to \tau \tag{2}$$

representing the intuition that a passive command that returns an applicative value is just a computation of an applicative value.

The solution for the subject reduction problem is more subtle. If a term $M[x]$ with a free variable x is promotable then any substitution instance $M[N]$ ought to be promotable, even if N has free variables of imperative types. Suppose x and N are of an applicative type τ. Any computation of type τ is expected to be state-independent. Hence, if the language lives up to its promise, any imperative information in the free variables of M can never be "used" in computing the value of M. Therefore, all such free variables can be treated as if they were of applicative types.

More formally, using ideas from [Rey78], we classify the *occurrences* of free variables as "imperative" or "applicative." Any free occurrence of x in an applicative-typed term is an *applicative occurrence*. All other occurrences are *imperative occurrences*. To promote a command, we require that all its free variables have applicative occurrences, irrespective of their actual types. For instance, the command `return (size a)` is promotable because the occurrence of `a` is an applicative occurrence. It occurs in the applicative-typed subterm `(size a)`. The fact that `a` is of an imperative type is of no consequence.

We first formalize these ideas as a monomorphic type system and then consider the issue of extending it with polymorphism.

3 Type System of ILC Revisited

The type system of "ILC revisited" (ILCR) is as follows. Let β range over primitive type constructors (such as `Int`, `Bool`, `Ref` and `Cmd`) and β_a over the applicative type constructors among these. Types are given by the context-free syntax:

$$\theta \ ::= \ \beta(\theta_1, \ldots, \theta_n) \mid \theta \times \theta' \mid \theta \to \theta' \mid !\theta$$

A subclass of these types are designated as applicative types:

$$\tau ::= \beta_a(\tau_1, \ldots, \tau_n) \mid \tau \times \tau' \mid \theta \to \tau \mid !\theta$$

notion of "passive" is stronger, but still similar to Reynolds's notion.

$$\frac{}{\mid x{:}\theta \vdash x : \theta}\ \text{Axiom} \qquad \frac{}{\mid\ \vdash k : \theta}\ \text{Const} \quad \text{if } k : \theta \text{ is a constant}$$

$$\frac{\Pi \mid \Gamma \vdash M : \theta}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash M : \theta}\ \text{Weakening}$$

$$\frac{\Pi \mid \Gamma,\, x{:}\theta \vdash M : \tau}{\Pi,\, x{:}\theta \mid \Gamma \vdash M : \tau}\ \text{Co-promotion} \qquad \frac{\Pi,\, x{:}\theta \mid \Gamma \vdash M : \theta'}{\Pi \mid \Gamma,\, x{:}\theta \vdash M : \theta'}\ \text{Co-dereliction}$$

$$\frac{\Pi \mid \Gamma \vdash M : \theta_1 \quad \Pi \mid \Gamma \vdash N : \theta_2}{\Pi \mid \Gamma \vdash \langle M, N \rangle : \theta_1 \times \theta_2}\ \times I \qquad \frac{\Pi \mid \Gamma \vdash M : \theta_1 \times \theta_2}{\Pi \mid \Gamma \vdash \pi_i M : \theta_i}\ \times E_i\ (i = 1, 2)$$

$$\frac{\Pi \mid \Gamma,\, x{:}\theta_1 \vdash M : \theta_2}{\Pi \mid \Gamma \vdash \lambda x{:}\theta_1.M : \theta_1 \to \theta_2} \to I \qquad \frac{\Pi \mid \Gamma \vdash M : \theta_1 \to \theta_2 \quad \Pi \mid \Gamma \vdash N : \theta_1}{\Pi \mid \Gamma \vdash MN : \theta_2} \to E$$

$$\frac{\Pi \mid\ \vdash M : \theta}{\Pi \mid\ \vdash M : !\theta}\ \text{Promotion} \qquad \frac{\Pi \mid \Gamma \vdash M : !\theta}{\Pi \mid \Gamma \vdash M : \theta}\ \text{Dereliction}$$

Figure 3: ILC Revisited - Typing Rules

Note that all ! types are considered applicative. Moreover, a function type $\theta \to \tau$ is applicative if its result type is applicative, even if the argument type is not. This uses the intuition mentioned in the previous section that an applicative computation cannot use any imperative information from its inputs.

The essential syntax of the language is defined by the type rules shown in Fig. 3, which is adapted from the "SCI Revisited" type system of [OPTT95].

Typing judgements are of the form $\Pi \mid \Gamma \vdash M{:}\theta$ where M is a term, θ its type and Π and Γ are collections of typing assumptions for the free identifiers of M. The two partitions Π and Γ of the typing context are called the *applicative zone* and the *imperative zone* respectively, and the free identifiers mentioned in them are called the *applicative free identifiers* and the *imperative free identifiers* of M. The applicative free identifiers can only be used in applicative-typed subterms. Note that there is no restriction on the types that can appear in Π.

The Promotion rule converts a term of type θ to type $!\theta$, provided all its free identifiers are in the applicative zone, i.e., they are applicative free identifiers. The free identifiers make it to the applicative zone via the Co-promotion rule which permits this only if all their occurrences are in applicative-typed terms. For instance, here is a derivation for the term `run (return (size a))` mentioned in Introduction:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mid \texttt{size} : \texttt{Array (Ref Int)} \to \texttt{Int},\ \texttt{a} : \texttt{Array (Ref Int)} \vdash \texttt{size a} : \texttt{Int}}}{\texttt{size} : \texttt{Array (Ref Int)} \to \texttt{Int},\ \texttt{a} : \texttt{Array (Ref Int)} \mid\ \vdash \texttt{size a} : \texttt{Int}}\ \text{Co-promotion}}{\texttt{size} : \texttt{Array (Ref Int)} \to \texttt{Int},\ \texttt{a} : \texttt{Array (Ref Int)} \mid\ \vdash \texttt{return (size a)} : \texttt{Cmd Int}}}{\texttt{size} : \texttt{Array (Ref Int)} \to \texttt{Int},\ \texttt{a} : \texttt{Array (Ref Int)} \mid\ \vdash \texttt{return (size a)} : !\,\texttt{(Cmd Int)}}\ \text{Promotion}}{\texttt{size} : \texttt{Array (Ref Int)} \to \texttt{Int},\ \texttt{a} : \texttt{Array (Ref Int)} \mid\ \vdash \texttt{run (return (size a))} : \texttt{Int}}
$$

Even though `size` and `a` have imperative types, the fact that they are in the applicative zone is enough to allow promotion.

The requisite properties of the type system have been proved in [SRI97]:

- The system has the *subject reduction* property: well-typed terms reduce to well-typed terms.
- The system is *coherent*: the meaning of well-typed terms is independent of their type derivations.

3.1 Standard types

The type constructor ! serves the purpose of identifying values that are built using only applicative information. So, for any applicative τ, $!\tau$ carries the same information as τ. In fact, there are well-typed terms $\mid x{:}\tau \vdash x{:}\,!\tau$ and $\mid z{:}\,!\tau \vdash z{:}\,\tau$. So, the isomorphism

$$!\tau \;\cong\; \tau$$

holds in all coherent semantic models of ILCR. In particular, since $!\theta$ is an applicative type, we have

$$!!\theta \;\cong\; !\theta$$

Thus, types of the form $!!\theta$ are redundant and can be prohibited without loss of expressiveness. Types that do not contain consecutive ! constructors will be called *standard types*.

Let ρ range over type terms without an outermost ! constructor. A standard type is either of the form $!\rho$ or of the form ρ (with or without an outermost !). We can cover both cases uniformly by using annotated constructor notation $!^n$, where $n \in \{0, 1\}$ is called an *annotation*. The notation $!^1$ means ! whereas $!^0$ corresponds to having no ! constructor. (Similar notation is used by Wadler in his linear type system [Wad91].) So, standard types will be given by the syntax:

$$
\begin{array}{lcl}
\theta & ::= & !^n\rho \\
\rho & ::= & \beta(\theta_1, \ldots, \theta_n) \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \\
n & ::= & 0 \mid 1
\end{array}
$$

$$\frac{}{\mid z{:}\theta \vdash z{:}\theta}\ \text{Axiom} \qquad \frac{}{\mid\ \vdash k{:}\theta}\ \text{Const} \qquad k{:}\theta \text{ is a constant}$$

$$\frac{\Pi \mid \Gamma \vdash M{:}\theta}{\Pi,\, \Pi' \mid \Gamma,\, \Gamma' \vdash M{:}\theta}\ \text{Weakening}$$

$$\frac{\Pi \mid z{:}\theta,\, \Gamma \vdash M{:}\tau}{\Pi,\, z{:}\theta \mid \Gamma \vdash M{:}\tau}\ \text{Co-promotion} \qquad \frac{\Pi,\, z{:}\theta \mid \Gamma \vdash M{:}\theta'}{\Pi \mid z{:}\theta,\, \Gamma \vdash M{:}\theta'}\ \text{Co-dereliction}$$

$$\frac{\Pi \mid \Gamma,\, x{:}\theta \vdash M{:}\theta'}{\Pi \mid \Gamma \vdash (\lambda x{:}\theta.M){:}\,!^0(\theta \to \theta')}\ \to\text{-Intro}$$

$$\frac{\Pi \mid \Gamma \vdash M_1{:}\,!^0(\theta \to \theta') \qquad \Pi \mid \Gamma \vdash M_2{:}\theta}{\Pi \mid \Gamma \vdash M_1(M_2){:}\theta'}\ \to\text{-Elim}$$

$$\frac{\Pi \mid \Gamma \vdash M_1{:}\theta_1 \qquad \Pi \mid \Gamma \vdash M_2{:}\theta_2}{\Pi \mid \Gamma \vdash \langle M_1,\, M_2\rangle{:}\,!^0(\theta_1 \times \theta_2)}\ \times\text{-Intro} \qquad \frac{\Pi \mid \Gamma \vdash M{:}\,!^0(\theta_1 \times \theta_2)}{\Pi \mid \Gamma \vdash \pi_i\, M{:}\theta_i}\ \times\text{-Elim}\ (i = 1, 2)$$

$$\frac{\Pi \mid\ \vdash M{:}\,!^0\rho}{\Pi \mid\ \vdash \textbf{promote}\ M{:}\,!^1\rho}\ \text{Promotion} \qquad \frac{\Pi \mid \Gamma \vdash M{:}\,!^1\rho}{\Pi \mid \Gamma \vdash \textbf{derelict}\ M{:}\,!^0\rho}\ \text{Dereliction}$$

Figure 4: ILCR$_S$ - Type rules for standard types

$$
\begin{array}{lcl}
ap(!^n\rho) & = & (n = 1 \vee ap(\rho)) \\
ap(\beta(\theta_1, \ldots, \theta_n)) & = & \begin{cases} \bigwedge_i ap(\theta_i) & \text{if } \beta \text{ is an applicative type constructor} \\ \text{false} & \text{otherwise} \end{cases} \\
ap(\theta_1 \times \theta_2) & = & ap(\theta_1) \wedge ap(\theta_2) \\
ap(\theta_1 \to \theta_2) & = & ap(\theta_2)
\end{array}
$$

Figure 5: Definition of "applicative type" predicate

Fig. 4 shows the ILCR type system adapted to standard types (called ILCR_S). We will use this system as a reference for devising type inference algorithms. So, the promotion and dereliction operations are explicitly represented in the term syntax. They will be filled in automatically by the type inference algorithm.

4 Polymorphism

To extend the ILCR type system with Hindley-Milner polymorphism, we must add type variables. Since there are two "kinds" of types (imperative and applicative), an immediate idea is to postulate separate classes of type variables to range over them [CO94]. However, this would mean that terms would have multiple types which differ in only the type variables used. Principal typing property would be lost. So, we use instead a single class of type variables and represent the kind information by a predicate $ap(\alpha)$ meaning "α is an applicative type."

The type syntax is extended with type variables as follows:

$$
\begin{array}{rcl}
\theta & ::= & !^n\rho \\
\rho & ::= & \beta(\theta_1, \ldots, \theta_n) \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \mid \alpha \\
n & ::= & 0 \mid 1 \mid i
\end{array}
$$

Here α is a *ρ-type variable*, which ranges over ρ-types, and i is an *annotation variable*, which ranges over annotations $\{0, 1\}$. The *ap* predicate is extended to type terms by the definition in Fig. 5.

A *type constraint* is a boolean formula given by the following syntax:

$$
C ::= \text{true} \mid \text{false} \mid ap(\theta) \mid ap(\rho) \mid n_1 \leq n_2 \mid C_1 \wedge C_2 \mid C_1 \vee C_2
$$

We feel free to write $n = 0$ for $n \leq 0$ and $n = 1$ for $1 \leq n$. We regard two constraints as equal, $C_1 = C_2$, if for all type substitutions σ, $\sigma(C_1) \Leftrightarrow \sigma(C_2)$.

An ILCR term can type check under different kind assignments for its type variables. For example, consider the preterm

$$\textbf{promote}\ (f\ x)$$

For it to type check, the free variables f and x must be used applicatively, i.e., either

1. both f and x have applicative types, or
2. $(f\ x)$ has an applicative type.

Type checking considerations of $(f\ x)$ suggest the following type schemes for f and x:

$$\begin{array}{l} f\colon !^0(!^j\alpha \to !^k\beta) \\ x\colon !^j\alpha \end{array}$$

Then one of the following conditions must hold for **promote** $(f\ x)$ to type check:

1. $ap(!^j\alpha \to !^k\beta) \wedge ap(!^j\alpha)$
2. $ap(!^k\beta)$

Note that if we analyze terms inside out, we cannot tell that these constraints are necessary until the **promote** operator is seen. This suggests that we must associate type constraints with free variables as well as the term. To formalize these ideas, we use judgements of the form:

$$x_1\colon\theta_1\ [A_1],\ \ldots,\ x_n\colon\theta_n\ [A_n] \vdash e\colon\theta\ [G]$$

where $A_1, \ldots, A_n$ and G are type constraints. The constraints $A_1, \ldots, A_n$ are "applicative constraints" that indicate the conditions under which the respective free variables can be regarded as applicative, and G is the "global constraint" which indicates the condition under which the term type checks. We call such judgements *polymorphic judgements*. Formally, the semantics of polymorphic judgements is defined as follows:

Definition 1 1. Let e be a preterm without type declarations for λ-bound variables. Then a judgement $\Pi\,|\,\Gamma \vdash e\colon\theta$ is said to be *implicitly derivable* if there is a derivable ILCR$_S$ term $\Pi\,|\,\Gamma \vdash M\colon\theta$ such that $e = \mathit{Erase}(M)$ is obtained by erasing type declarations from M.

2. A judgement $\Pi\,|\,\Gamma \vdash e\colon\phi_0$ is an *instance* of a polymorphic judgement $\vec{x}\colon\vec{\theta}\ [\vec{A}] \vdash e\colon\phi\ [G]$ iff there is a substitution σ for the type variables in the latter such that
   (a) the judgement $\sigma(\vec{x}\colon\vec{\theta} \vdash e\colon\phi)$ is the same as $\Pi, \Gamma \vdash e\colon\phi_0$,
   (b) $\sigma(G)$ is true, and
   (c) the constraint $\sigma(A_i)$ is true for all $x_i\colon\sigma(\theta_i)$ in Π.

3. A polymorphic judgement is said to be *valid* if all its ground instances are implicitly derivable in ILCR$_S$ (where "ground" means with no type variables).

□

Examples of valid polymorphic judgements are

$$f{:}\,!^0(!^j\alpha \to !^k\beta)\ [ap(!^k\beta)],\ x{:}\,!^j\alpha\ [ap(!^j\alpha) \vee ap(!^k\beta)] \vdash f\ x{:}\,!^k\beta\ [\text{true}]$$

$$f{:}\,!^0(!^j\alpha \to !^0\beta)\ [\text{true}],\ x{:}\,!^j\alpha\ [\text{true}] \vdash \textbf{promote}\ (f\ x){:}\,!^1\beta\ [ap(!^0\beta) \wedge (ap(!^j\alpha) \vee ap(!^0\beta))]$$

They have the following example instances in ILCR$_S$:

$$\begin{array}{rcl} f{:}\,(\texttt{Cmd Int} \to \texttt{Int}) \mid x{:}\,\texttt{Cmd Int} & \vdash & f\ x{:}\,\texttt{Int} \\ f{:}\,(\texttt{Cmd Int} \to \texttt{Int}),\ x{:}\,\texttt{Cmd Int} \mid & \vdash & \textbf{promote}\ (f\ x){:}\,!\,\texttt{Int} \end{array}$$

While subtyping systems [Mit91, FM90] as well as Wadler's linear type systems [Wad91] use type constraints associated with terms, the novel feature of our judgements is that they also have type constraints associated with free variables. The need for these constraints is clear from the definition of the instance relation. They keep track of the conditions under which the free variables can be put in the applicative zone.

An inference system for valid polymorphic judgements is shown in Fig. 6. Note that there are no rules for Co-promotion and Co-dereliction. Their effect is implicitly represented in the type constraints for free variables. Recall that a free variable is regarded as applicative if all its occurrences are within applicative subterms. In other words, for every occurrence of the free variable, *some* subterm enclosing the occurrence must be applicative. Thus, every term formation rule potentially adds a disjunct to the A-constraints of its free variables. Notice this in *Axiom* and the elimination rules. For the introduction rules, such addition turns out to be redundant. Since a **promote** -term can only have applicative free variables, all the constraints of the free variables are transferred to the global constraint.

Lemma 2 If a polymorphic judgement $\Gamma\,[\vec{A}] \vdash e{:}\,\theta\,[G]$ is derivable, then $ap(\theta) \Rightarrow A_i$ is true for all A_i in $\vec{A}$. □

The requisite properties of the polymorphic system are as follows.

Theorem 3 (Soundness) Every derivable polymorphic judgement is valid. □

Theorem 4 (Completeness) Every judgement implicitly derivable in ILCR$_S$ is an instance of a derivable polymorphic judgement. □

Proof. If the ILCR$_S$ judgement is $\Pi \mid \Gamma \vdash M{:}\,\phi$, then the corresponding polymorphic judgement is of the form $\Pi\,[\overrightarrow{\text{true}}],\ \Gamma\,[\vec{A}] \vdash \mathit{Erase}(M){:}\,\phi\ [\text{true}]$. (It has no type variables.) The proof is by induction on the derivation of the ILCR$_S$ judgement and proceeds by case analysis on the last step. We show key cases.

- *Co-promotion* : Consider a step of the form

$$\frac{\Pi \mid z{:}\,\theta,\ \Gamma \vdash M{:}\,\tau}{\Pi,\ z{:}\,\theta \mid \Gamma \vdash M{:}\,\tau}\ \text{Co-promotion}$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash e{:}\theta'\ [G]}{\vec{z}{:}\vec{\theta}\ [\vec{A}],\ \vec{x}{:}\vec{\theta''}\ [\overrightarrow{\mathrm{true}}] \vdash e{:}\theta'\ [G]}\ \text{Weakening}$$

$$\frac{}{\vdash k{:}\theta\ [\mathrm{true}]}\ \text{Const} \quad \text{where } k \text{ is a constant of type } \theta$$

$$\frac{}{z{:}\theta\ [ap(\theta)] \vdash z{:}\theta\ [\mathrm{true}]}\ \text{Axiom}$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}],\ x{:}\theta'\ [A'] \vdash e{:}\theta''\ [G]}{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash (\lambda x.\, e){:}\,!^0(\theta' \to \theta'')\ [G]}\ \to\text{-Intro}$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash e_1{:}\,!^0(\theta' \to \theta'')\ [G] \qquad \vec{z}{:}\vec{\theta}\ [\vec{A'}] \vdash e_2{:}\theta'\ [G']}{\vec{z}{:}\vec{\theta}\ [(\vec{A} \wedge \vec{A'}) \vee ap(\theta'')] \vdash e_1(e_2){:}\theta''\ [G \wedge G']}\ \to\text{-Elim}$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash e_1{:}\theta'\ [G] \qquad \vec{z}{:}\vec{\theta}\ [\vec{A'}] \vdash e_2{:}\theta''\ [G']}{\vec{z}{:}\vec{\theta}\ [(\vec{A} \wedge \vec{A'})] \vdash \langle e_1,\, e_2\rangle{:}\,!^0(\theta' \times \theta'')\ [G \wedge G']}\ \times\text{-Intro}$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash e{:}\,!^0(\theta_1 \times \theta_2)\ [G]}{\vec{z}{:}\vec{\theta}\ [\vec{A} \vee ap(\theta_i)] \vdash \pi_i\, e{:}\theta_i\ [G]}\ \times\text{-Elim}_i\ (i = 1, 2)$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash e{:}\,!^0\rho\ [G]}{\vec{z}{:}\vec{\theta}\ [\overrightarrow{\mathrm{true}}] \vdash \mathbf{promote}\ e{:}\,!^1\rho\ [G \wedge (\bigwedge\vec{A})]}\ \text{Promotion}$$

$$\frac{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash e{:}\,!^1\rho\ [G]}{\vec{z}{:}\vec{\theta}\ [\vec{A}] \vdash \mathbf{derelict}\ e{:}\,!^0\rho\ [G]}\ \text{Dereliction}$$

Figure 6: Polymorphic type rules

If the polymorphic judgement corresponding to the antecedent is $\Pi\,[\overrightarrow{\text{true}}],\ z\!:\theta\,[A],\ \Gamma\,[\vec{A}] \vdash \mathit{Erase}(M)\!:\tau\ [\text{true}]$ then, by Lemma 2, $A = \text{true}$. So the consequent is also an instance.

- *Co-dereliction* : Consider a step of the form

$$\frac{\Pi,\ z\!:\theta \mid,\ \Gamma \vdash M\!:\tau}{\Pi \mid z\!:\theta,\ \Gamma \vdash M\!:\tau}\ \text{Co-promotion}$$

If the polymorphic judgement corresponding to the antecedent is $\Pi\,[\overrightarrow{\text{true}}],\ z\!:\theta\,[\text{true}],\ \Gamma\,[\vec{A}] \vdash \mathit{Erase}(M)\!:\tau\ [\text{true}]$, then the consequent is also clearly an instance of it.

- *Promotion* : Consider a step of the form.

$$\frac{\Pi \mid\ \vdash M\!:!^0\rho}{\Pi \mid\ \vdash \textbf{promote}\ M\!:!^1\rho}\ \text{Promotion}$$

The polymorphic judgement for the antecedent is $\Pi\,[\overrightarrow{\text{true}}] \vdash \mathit{Erase}(M)\!:!^0\rho\ [\text{true}]$. By the rule Promotion, we obtain, $\Pi\,[\overrightarrow{\text{true}}] \vdash \textbf{promote}\ (\mathit{Erase}(M))\!:!^1\rho\ [\text{true}]$.

The other cases are similar. □

5 Implicit promotion and dereliction

In this section, we modify the polymorphic type system to deal with promotion and dereliction implicitly. To appreciate the issues, consider the sample term

$$(f\ x)$$

and assume the type schemes

$$\begin{array}{l} f\!:!^i(!^j\alpha \rightarrow !^k\beta) \\ x\!:!^l\alpha \end{array}$$

The possibilities for promotion include promoting x and promoting $(f\ x)$. (There is no need to consider the promotion of f since it is immediately derelicted.) In addition, there are three possibilities for dereliction. These possibilities give rise to the following constraints, assuming $!^m\beta$ for the result type:

1. $j \leq l \vee ap(!^l\alpha)$, denoting dereliction and promotion possibilities for x.
2. $m \leq k \vee (ap(!^i(!^j\alpha \rightarrow !^k\beta)) \wedge ap(!^l\alpha)) \vee ap(!^k\beta)$, denoting dereliction and promotion possibilities for $(f\ x)$.

Simplifying the constraints, we obtain

$$\frac{\vec{z}\!:\vec{\theta}\ [\vec{A}] \vdash e\!:\theta'\ [G]}{\vec{z}\!:\vec{\theta}\ [\vec{A}],\ \vec{x}\!:\vec{\theta''}\ [\overrightarrow{\text{true}}] \vdash e\!:\theta'\ [G]}\ \text{Weakening}$$

$$\frac{}{\vdash k\!:!^m\rho\ [\text{true}]}\ \text{Const} \qquad k \text{ is a constant of type } !^n\rho$$

$$\frac{}{z\!:!^n\rho\ [ap(!^n\rho)] \vdash z\!:!^m\rho\ [m \le n \vee ap(!^n\rho)]}\ \text{Axiom}$$

$$\frac{\vec{z}\!:\vec{\theta}\ [\vec{A}],\ x\!:\theta'\ [A'] \vdash e\!:\theta''\ [G]}{\vec{z}\!:\vec{\theta}\ [\vec{A}] \vdash (\lambda x.\, e)\!:!^m(\theta' \to \theta'')\ [G \wedge (m = 0 \vee (\bigwedge \vec{A}))]}\ \to\text{-Intro}$$

$$\frac{\vec{z}\!:\vec{\theta}\ [\vec{A}] \vdash e_1\!:!^0(\theta' \to !^n\rho)\ [G] \qquad \vec{z}\!:\vec{\theta}\ [\vec{A'}] \vdash e_2\!:\theta'\ [G']}{\vec{z}\!:\vec{\theta}\ [(\vec{A} \wedge \vec{A'}) \vee ap(!^n\rho)] \vdash e_1(e_2)\!:!^m\rho\ [G \wedge G' \wedge (m \le n \vee ((\bigwedge \vec{A}) \wedge (\bigwedge \vec{A'})) \vee ap(!^n\rho))]}\ \to\text{-Elim}$$

$$\frac{\vec{z}\!:\vec{\theta}\ [\vec{A}] \vdash e_1\!:\theta\ [G] \qquad \vec{z}\!:\vec{\theta}\ [\vec{A'}] \vdash e_2\!:\theta'\ [G']}{\vec{z}\!:\vec{\theta}\ [(\vec{A} \wedge \vec{A'})] \vdash \langle e_1,\, e_2\rangle\!:!^m(\theta \times \theta')\ [G \wedge G' \wedge (m = 0 \vee ((\bigwedge \vec{A}) \wedge (\bigwedge \vec{A'})))]}\ \times\text{-Intro}$$

$$\frac{\vec{z}\!:\vec{\theta}\ [\vec{A}] \vdash e\!:!^0(!^n\rho \times \theta')\ [G]}{\vec{z}\!:\vec{\theta}\ [\vec{A} \vee ap(!^n\rho)] \vdash \pi_1\, e\!:!^m\rho\ [G \wedge (m \le n \vee (\bigwedge \vec{A}) \vee ap(!^n\rho))]}\ \times\text{-Elim}_1$$

$$\frac{\vec{z}\!:\vec{\theta}\ [\vec{A}] \vdash e\!:!^0(\theta' \times !^n\rho)\ [G]}{\vec{z}\!:\vec{\theta}\ [\vec{A} \vee ap(!^n\rho)] \vdash \pi_2\, e\!:!^m\rho\ [G \wedge (m \le n \vee (\bigwedge \vec{A}) \vee ap(!^n\rho))]}\ \times\text{-Elim}_2$$

Figure 7: Polymorphic type rules with implicit promotion and dereliction

1. $j \le l \vee ap(\alpha)$
2. $m \le k \vee ap(\beta) \vee (i = 1 \wedge (l = 1 \vee ap(\alpha)))$

This gives the most general typing for $(f\ x)$.

Fig. 7 gives the inference system for implicit promotion and dereliction. The judgements are of the same form as in Sec. 4. Their meaning in terms of "instances" is also the same except that the *Erase* operator also erases **derelict** and **promote** operators, in addition to type declarations. The type rules incorporate, after each operation, the possibility of implicit promotion and dereliction operations. The only exceptions are based on the following property:

Lemma 5 In any ILCR$_S$ derivation, every dereliction step occurs after either Axiom, Const, Promotion, or elimination step followed by zero or more structural rule steps. □

Hence, we need not consider the possibility of dereliction after an introduction step, only promotion. In all other cases, both dereliction and promotion are possible. Consider the *Axiom* rule in detail.

The global constraint covers three possibilities: $m = n$, which represents a simple use of the variable, $m < n$, which represents an implicit dereliction, and $m > n$, which represents an implicit promotion. In the last case, the type of z must be applicative, represented by the constraint $ap(!^n\rho)$. The other rules are similar.

The requisite properties of the type system are as follows:

Theorem 6 (Soundness) All derivable polymorphic judgements are valid. □

Lemma 7 If a polymorphic judgement $\Gamma\,[\vec{A}] \vdash e\colon\phi\,[G]$ is derivable, then $ap(\phi) \wedge G \Rightarrow A_i$ holds for all A_i in $\vec{A}$. □

Theorem 8 (Completeness) Every judgement implicitly derivable in ILCR$_S$ is an instance of a derivable polymorphic judgement. □

Proof. If $\Pi\,|\,\Gamma \vdash M\colon\phi$ is the ILCR$_S$ judgement and $\mathit{Erase}(M) = e$, then the corresponding polymorphic judgement is of the form $\Pi\,[\overrightarrow{\text{true}}],\, \Gamma\,[\vec{A}] \vdash e\colon\phi$ [true]. The proof is by induction on the height of the ILCR$_S$ derivation and proceeds by case analysis on the last step. We show selected cases.

- *Axiom*

$$\frac{}{|\; x\colon !^n\rho \vdash x\colon !^n\rho}\ \text{Axiom}$$

  The corresponding polymorphic judgement $x\colon !^n\rho\ [ap(!^n\rho)] \vdash x\colon !^n\rho$ [true] is derivable by Axiom.

- *Dereliction* : Consider also the step preceding the dereliction step. (Use Lemma 5.)

  – *Axiom*

$$\frac{\dfrac{}{|\; z\colon !^1\rho \vdash z\colon !^1\rho}\ \text{Axiom}}{|\; z\colon !^1\rho \vdash \textbf{derelict}\ z\colon !^0\rho}\ \text{Dereliction}$$

  We can derive $z\colon !^1\rho\ [ap(!^1\rho)] \vdash z\colon !^0\rho$ [true] by Axiom.

  – *Weakening*

$$\frac{\dfrac{\Pi \mid \Gamma \vdash M\colon !^1\rho}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash M\colon !^1\rho}\ \text{Weakening}}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash \textbf{derelict}\ M\colon !^0\rho}\ \text{Dereliction}$$

  Then reorder the two steps.

$$\frac{\dfrac{\Pi \mid \Gamma \vdash M\colon !^1\rho}{\Pi \mid \Gamma \vdash \textbf{derelict}\ M\colon !^0\rho}\ \text{Dereliction}}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash \textbf{derelict}\ M\colon !^0\rho}\ \text{Weakening}$$

By induction hypothesis, the consequent of the dereliction step is an instance of a derivable polymorphic judgement. The Weakening step can be simulated by a Weakening step of the implicit dereliction system.

- *Promotion*

  Consider the previous step other than Co-promotion or Co-dereliction.

  - *Dereliction*

$$\dfrac{\dfrac{\Pi \mid \Gamma \vdash M\colon !^1\rho}{\Pi \mid \Gamma \vdash \textbf{derelict } M\colon !^0\rho}\text{Dereliction}}{\dfrac{\vdots}{\dfrac{\Pi, \Gamma \mid \ \vdash \textbf{derelict } M\colon !^0\rho}{\Pi, \Gamma \mid \ \vdash \textbf{promote } (\textbf{derelict } M)\colon !^1\rho}\text{Promotion}}}$$

    By inductive hypothesis, $\Pi\,[\overrightarrow{\text{true}}], \Gamma\,[\vec{A}] \vdash e\colon !^1\rho$ [true] is a derivable polymorphic judgement. And $ap(!^1\rho)$ is true. By Lemma 7, $\vec{A} = \overrightarrow{\text{true}}$. This gives the conclusion.

  - $\to$-Intro

$$\dfrac{\dfrac{\Pi \mid \Gamma, x\colon\phi \vdash M\colon\phi'}{\Pi \mid \Gamma \vdash \lambda x\colon\phi.\, M\colon !^0(\phi \to \phi')}\to\text{-Intro}}{\dfrac{\vdots}{\dfrac{\Pi, \Gamma \mid \ \vdash \lambda x\colon\phi.\, M\colon !^0(\phi \to \phi')}{\Pi, \Gamma \mid \ \vdash \textbf{promote } (\lambda x\colon\phi.\, M)\colon !^1(\phi \to \phi')}\text{Promotion}}}$$

    By inductive hypothesis, $\Pi\,[\overrightarrow{\text{true}}], \Gamma\,[\vec{A}], x\colon\phi\ [A] \vdash e\colon\phi'$ [true] is a derivable polymorphic judgement, where $e = \mathit{Erase}(M)$. If Γ is nonempty (and its variables are co-promoted between the two steps), then $ap(!^0(\phi \to \phi')) = ap(\phi') = \text{true}$. By Lemma 7, $\vec{A} = \overrightarrow{\text{true}}$. The polymorphic judgement $\Pi\,[\overrightarrow{\text{true}}], \Gamma\,[\overrightarrow{\text{true}}] \vdash \lambda x.\, e\colon !^1(\phi \to \phi')$ [true] is obtained by $\to$-Intro.

□

5.1 Practical Considerations

Fig. 8 shows examples of most general types derived in the implicit promotion system. It is apparent that even simple terms have long constraints in their types. Making promotion implicit results in numerous possibilities for promotion which the type system must account for. Since the constraints arise mainly from promotion operations, the most general typings tend to get very large.

We suggest some techniques to reduce the size of constraints. Recall that we have the isomorphisms:

$$\begin{array}{rcl} !\tau & \cong & \tau \\ !(\theta_1 \times \theta_2) & \cong & !\theta_1 \times !\theta_2 \end{array}$$

$$
\begin{array}{lcl}
\lambda x.\, x & : & !^i(!^j\alpha \rightarrow !^k\alpha)\ [k \leq j \vee ap(\alpha)] \\
\lambda x.\, \lambda y.\, x & : & !^i(!^j\alpha \rightarrow !^k(!^l\beta \rightarrow !^m\alpha))\ [(m \leq j \vee ap(\alpha)) \wedge (k \leq j \vee ap(\alpha))] \\
\lambda x.\, \texttt{put}\ x\ 0 & : & !^i(!^j\, \texttt{Ref}(\,!^l\texttt{Int}) \rightarrow !^k\, \texttt{Cmd}\,())\ [k \leq j] \\
\lambda x.\, \texttt{run}\ (\texttt{get}\ x) & : & !^i(!^j\, \texttt{Ref}(\,!^l\alpha) \rightarrow !^k\alpha)\ [j = 1 \wedge ap(!^l\alpha)]
\end{array}
$$

Figure 8: Examples of implicit promotion types

Thus, types of the form $!\tau$ and $!(\theta_1 \times \theta_2)$ are redundant. There is no loss in prohibiting them. Secondly, types of the form $!\,\texttt{Ref}\,\alpha$ are quite useless because there are no values of such types (other than undefined values). Similarly, types of the form $!\,(\texttt{Cmd}\ ())$ serve no useful purpose. So, we can prohibit such types as well. Then the only useful ! constructors are those applied to $\rightarrow$ types, Cmd types and type variables.

Using this fact, we formulate the following abbreviated notation:

$$
\begin{array}{lcl}
\theta \rightarrow^n \theta' & = & !^n(\theta \rightarrow \theta') \\
\texttt{Cmd}^n\ \tau & = & !^n(\texttt{Cmd}\ \tau) \\
\alpha^n & = & !^n\alpha
\end{array}
$$

Using these simplifications and notations, the types of Fig. 8 can be expressed as follows:

$$
\begin{array}{lcll}
\lambda x.\, x & : & \alpha^j \rightarrow^i \alpha^k & [k \leq j \vee ap(\alpha)] \\
\lambda x.\, \lambda y.\, x & : & \alpha^j \rightarrow^i \beta^l \rightarrow^k \alpha^m & [(m \leq j \vee ap(\alpha)) \wedge (k \leq j \vee ap(\alpha))] \\
\lambda x.\, \texttt{put}\ x\ 0 & : & \texttt{Ref Int} \rightarrow^i \texttt{Cmd}\ () & [\text{true}] \\
\lambda x.\, \texttt{run}\ (\texttt{get}\ x) & : & \texttt{Ref}\ \alpha^l \rightarrow^i \alpha^k & [\text{false}]
\end{array}
$$

Note that the last term $\lambda x.\,\texttt{run}\ (\texttt{get}\ x)$ is not typeable any more because $!\texttt{Ref}\ \alpha$ is prohibited. The example programs in Fig. 2 have the following principal types:

$$
\begin{array}{lcl}
\texttt{for} & : & (\texttt{Int},\ \texttt{Int}) \rightarrow^i (\texttt{Int} \rightarrow^j \texttt{Cmd}\ ()) \rightarrow^l \texttt{Cmd}\ ()\ [\text{true}] \\
\texttt{factorial} & : & \texttt{Int} \rightarrow^i \texttt{Int}\ [\text{true}]
\end{array}
$$

6 Type reconstruction

The type reconstruction algorithm is directly based on the rules of Figure 7. In fact, the type rules have been designed so that they are essentially form a "logic program" for type reconstruction. They are syntax-directed (except for the Weakening rule), and are closed under type substitution.

$$
\begin{array}{rcll}
T\colon & & & \text{Preterm} \times \text{Env} \to \text{Judgement} \\[6pt]
T(x,\, E) & = & \text{if} & \Sigma \vdash x\colon \phi\ [G] \in E \\
 & & \text{then} & \Sigma \vdash x\colon \phi\ [G] \\
 & & \text{if} & \Sigma \vdash x\colon \phi\ [G] \notin E \\
 & & \text{then} & x\colon !^i\alpha\ [ap(!^i\alpha)] \vdash x\colon !^j\alpha\ [j \leq i \vee ap(!^j\alpha)] \\[6pt]
T(\lambda x.\, e,\, E) & = & \text{if} & T(e,\, E) = (\Sigma,\, x\colon \phi\ [A''] \vdash e\colon \phi'\ [G]) \\
 & & \text{then} & \Sigma \vdash \lambda x.\, e\colon !^j(\phi \to \phi')\ [G \wedge (j = 0 \vee \text{AND}(\Sigma))] \\
 & & \text{if} & T(e,\, E) = (\Sigma \vdash e\colon \phi'\ [G]) \text{ and } x \text{ doesn't occur in } \Sigma \\
 & & \text{then} & \Sigma \vdash \lambda x.\, e\colon !^j(\alpha \to \phi')\ [G \wedge (j = 0 \vee \text{AND}(\Sigma))] \\[6pt]
T(e_1(e_2),\, E) & = & \text{let} & (\Sigma_1 \vdash e_1\colon \phi_1\ [G_1]) = T(e_1,\, E) \\
 & & & (\Sigma_2 \vdash e_2\colon \phi_2\ [G_2]) = T(e_2,\, E) \\
 & & & \quad \text{with type variables renamed to be disjoint} \\
 & & & \quad \text{from those in } T(e_2,\, E) \\
 & & & \sigma = \text{unify}(\{\phi_1 = !^0(\phi_2 \to !^i\alpha)\} \\
 & & & \quad \cup \{(\theta_1 = \theta_2) \mid x\colon \theta_1\ [A_1] \in \Sigma_1,\, x\colon \theta_2\ [A_2] \in \Sigma_2\}) \\
 & & & \Sigma = \text{DISJOIN}(\text{MERGE}(\sigma(\Sigma_1),\, \sigma(\Sigma_2)), ap(!^i\alpha)) \\
 & & \text{in} & \Sigma \vdash e_1(e_2)\colon \sigma(!^j\alpha)\ [\sigma(G_1 \wedge G_2) \wedge (\sigma(j \leq i) \vee \text{AND}(\Sigma))] \\[6pt]
T(\texttt{let}\ x = e_1\ \texttt{in}\ e_2,\, E) & = & \text{let} & (\Sigma \vdash e_1\colon \phi\ [G_1]) = T(e_1,\, E) \\
 & & & E' = E \cup \{\Sigma \vdash e_1\colon \phi\ [G_1]\} \\
 & & & (\Sigma' \vdash e_2\colon \phi'\ [G_2]) = T(e_2,\, E') \\
 & & \text{in} & \text{MERGE}(\Sigma,\, \Sigma') \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\colon \phi'\ [G_1 \wedge G_2] \\[12pt]
\text{MERGE}(\Sigma_1,\, \Sigma_2) & = & & \{(x\colon \theta\ [A_1 \wedge A_2]) \mid x\colon \theta\ [A_1] \in \Sigma_1,\, x\colon \theta\ [A_2] \in \Sigma_2\} \\
 & & \cup & \{(x\colon \theta\ [A_1]) \mid x\colon \theta\ [A_1] \in \Sigma_1,\, x \notin \text{dom}(\Sigma_2)\} \\
 & & \cup & \{(x\colon \theta\ [A_2]) \mid x \notin \text{dom}(\Sigma_1),\, x\colon \theta\ [A_2] \in \Sigma_2\} \\
\text{DISJOIN}(\Sigma,\, C) & = & & \{x\colon \theta\ [A \vee C] \mid x\colon \theta\ [A] \in \Sigma\} \\
\text{AND}(\Sigma) & = & & \bigwedge \{A \mid (x\colon \theta\ [A]) \in \Sigma\}
\end{array}
$$

Figure 9: Type reconstruction algorithm

Sample clauses of the reconstruction algorithm for implicit promotion and dereliction are shown in Fig. 9

We also add a let-construct whose effect is given by the type rule:

$$
\frac{\vec{z}\colon \vec{\theta}\ [\vec{A_1}] \vdash e_1\colon \phi_1\ [G_1] \qquad \vec{z}\colon \vec{\theta}\ [\vec{A_2}] \vdash [e_1/x]e_2\colon \phi_2\ [G_2]}{\vec{z}\colon \vec{\theta}\ [\vec{A_1} \wedge \vec{A_2}] \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\colon \phi_2\ [G_1 \wedge G_2]}\ \text{Let}
$$

Clearly, each occurrence of the let-bound variable in the body can have a different instance of the type of e_1. The type rule is not compositional. However, the reconstruction algorithm handles it in a compositional fashion.

The reconstruction algorithm T takes a term and an "environment" which is a collection of typings for distinct let-bound variables. Every occurrence of a let-bound variable will receive a separate instance of its typing in the environment. (The technique is from [KMM91].)

Theorem 9 $T(e, E) = (\Sigma \vdash e : \theta\,[G])$ if and only if the judgement is the principal typing for e in the following sense: If $E = \{\Sigma'_i \vdash x_i : \theta_i\,[G_i]\}_i$ then, for all terms with most general typings $\{\Sigma'_i \vdash e_i : \theta_i\,[G_i]\}_i$, $\Sigma \vdash e[\vec{e}/\vec{x}] : \theta\,[G]$ is the most general typing derivable for $e[\vec{e}/\vec{x}]$. □

7 Conclusion

We have reformulated the Imperative Lambda Calculus type system using ideas from Reynolds's Syntactic Control of Interference, and extended it to handle Hindley-Milner style polymorphism. The new type system is more flexible than the original one and it possesses the subject reduction property. The extension to polymorphism involves novel techniques of type constraints and the association of constraints with free variables in typing judgements.

The techniques designed here are useful in all situations where "freedom from side effects" or "access control" of some form needs to be guaranteed. For instance, one of the applications we are studying is a language that has both stack and heap storage (like modern object-oriented languages) and must ensure that no references to stack storage are present in the objects created on the heap. Another application is ensuring secure flow of information in security-critical systems [VSI96]. The aspects of Syntactic Control of Interference that are retained in the Imperative Lambda Calculus type system have a wide range of applications.

We have implemented a prototype type reconstruction system in Prolog. We are experimenting with various strategies for constraint simplification and satisfaction. The examples in the paper have been checked on the system.

References

[BBdPH92] P. N. Benton, G. M. Bierman, V. C. V. de Paiva, and J. M. E. Hyland. Term assignment for intuitionistic linear logic. Technical Report 262, Computer Laboratory, University of Cambridge, August 1992.

[BMMM95] S. Brookes, M. Main, A. Melton, and M. Mislove, editors. *Mathematical Foundations of Programming Semanatics: Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995.

[Bro94] S. Brookes, editor. *Math. Foundations of Program. Semantics: 9th Intern. Conf., 1991*, volume 802 of *LNCS*. Springer-Verlag, 1994.

[CO94] K. Chen and M. Odersky. A type system for a lambda calculus with assignments. In Hagiya and Mitchell [HM94], pages 347–364.

[FM90] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Comput. Sci.*, 73:155–175, 1990.

[GH90] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science* [IEE90], pages 333–343.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Comput. Sci.*, 50:1–102, 1987.

[HM94] M. Hagiya and J. C. Mitchell, editors. *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*. Springer-Verlag, 1994.

[HR96] H. Huang and U. S. Reddy. Type reconstruction for SCI. In D. N. Turner, editor, *Functional Programming, Glasgow 1995*, Electronic Workshops in Computing. Springer-Verlag, 1996.

[IEE90] IEEE Computer Society Press. *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[IEE94] IEEE Computer Society Press. *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, July 1994.

[KMM91] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. D. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *ACM Symp. on Princ. of Program. Lang.*, pages 47–57, 1988.

[LP95] J. Launchbury and S. L. Peyton Jones. State in Haskell. *J. Lisp and Symbolic Comput.*, 8(4):293–341, 1995.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *J. Functional Program.*, 1(3):245–285, July 1991.

[O'H91] P. W. O'Hearn. Linear logic and interference control. In *Category Theory and Computer Science*, volume 350 of *LNCS*, pages 74–93. Springer-Verlag, 1991.

[OPTT95] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In Brookes et al. [BMMM95]. (Chapter 18 of [OT97b]).

[ORH93] M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment and the lambda calculus. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.

[OT97a] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages, Vol. 1.* Birkhäuser, Boston, 1997.

[OT97b] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages, Vol. 2*. Birkhäuser, Boston, 1997.

[Rab96] D. Rabin. *Calculi for Functional Programming Languages with Assignment.* PhD thesis, Yale University, 1996.

[Red93] U. S. Reddy. Global state considered unnecessary: Semantics of interference-free imperative programming. In *ACM SIGPLAN Workshop on State in Program. Lang.*, pages 120–135. Technical Report YALEU/DCS/RR-968, June 1993.

[Red94] U. S. Reddy. Passivity and independence. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science* [IEE94], pages 342–352.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978. (Chapter 10 of [OT97a]).

[Rey89] J. C. Reynolds. Syntactic control of interference, Part II. In *Intern. Colloq. Aut., Lang. and Program.*, volume 372 of *LNCS*, pages 704–722. Springer-Verlag, 1989.

[Rey97] J. C. Reynolds. Design of the programming language Forsythe. In *Algol-like Languages, Vol. 1* [OT97a], chapter 8, pages 173–234.

[SRI91] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, volume 523 of *LNCS*, pages 192–214. Springer-Verlag, 1991.

[SRI97] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In *Algol-like Languages, Vol. 1* [OT97a], chapter 9, pages 235–272.

[TJ94] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, Jun 1994.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[Wad91] P. Wadler. Is there a use for linear logic? In *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273. ACM, 1991. (SIGPLAN Notices, Sep. 1991).

[Wad93] P. Wadler. A syntax for linear logic. In Brookes [Bro94].

[Wel94] J. B. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science* [IEE94], pages 176–185.