

Type Reconstruction for SCI, Part 2

Hongseok Yang Howard Huang

Department of Computer Science

University of Illinois at Urbana-Champaign

{hyang,hhuang}@cs.uiuc.edu

October 1, 1997

Keywords: type systems, Algol-like languages, interference, SCI, type inference

Type Reconstruction for SCI, Part 2

Hongseok Yang Howard Huang

Department of Computer Science
University of Illinois at Urbana-Champaign
{hyang,hhuang}@cs.uiuc.edu

October 1, 1997

Abstract

Syntactic Control of Interference (**SCI**) [Rey78] has long been studied as a basis for interference-free programming, with cleaner reasoning properties and semantics than traditional imperative languages. This paper makes **SCI**-based languages more practical by introducing a revised version of Huang and Reddy's type reconstruction system [HR96] with two significant improvements. First, we eliminate the need for explicit coercion operators in terms. Although this can lead to more complex principal typings, we introduce some minor restrictions on the inference system to keep the typings manageable. Second, we consider adding let-bound polymorphism. Such a modification appears to be nontrivial. We propose a new approach which addresses issues of both polymorphism and interference control. Some examples show the utility of our system. **SCI** can be adapted to a wide variety of languages, and our techniques should be applicable to any such language with **SCI**-based interference control.

1 Introduction

One of the main challenges in reasoning about imperative programs is dealing with interference, which occurs when the execution of one program phrase affects the outcome of another. Interference can appear in many forms, all of which invalidate traditional reasoning techniques. It poses a particular challenge for parallel systems, since the concurrent execution of interfering program phrases would yield indeterminate results. Even worse, it can be quite difficult for both users and machines to detect interference. For example, parallelizing compilers often implement many complex static analyses to detect the possibility of interference.

The example program in Figure 1 (adopted from [Rey78]) illustrates some of the problems. (The program syntax is nearly the same as that of **SCIR**, as discussed in Section 2.) The procedure `map` applies its argument `p` to each element of a linked list of integers `data`. Procedure `reclaim` removes a node from `data` and inserts it into `free`, a list of free nodes. For simplicity, we represent list nodes using arrays `prev` and `next`, which contain the indices of the previous and next nodes. The global variables `data` and `free` store the indices of the first nodes of their lists.

The call `(map reclaim data)` may appear to be a reasonable way to delete all elements of `list`. But, since `reclaim` modifies `next(i)`, the second node that `map` tries to reclaim will belong to list

<pre> map ≡ λp. λi. if (i ≠ 0) then p i; map p next(i) else skip </pre>	<pre> reclaim ≡ λi. if (prev(i) ≠ 0) then next(prev(i)) := next(i) else data := next(i); if (next(i) ≠ 0) then prev(next(i)) := prev(i) else skip next(i) := free; free := i </pre>
---	---

Figure 1: Example procedures `map` and `reclaim`

`free`, not `data`! To ensure correctness, the arguments to a function should not interfere with the function itself. But unfortunately it is not always obvious when this condition holds.

In 1978, Reynolds proposed Syntactic Control of Interference, or **SCI** [Rey78], which syntactically restricts a language to prevent undesired interference. The principles of **SCI** suggest guidelines for structuring programs so that it is clear where interference might occur, making reasoning about programs much easier. O’Hearn *et al.* later developed **SCIR** [OPTT95], an elegant type system and semantics for **SCI**. Interference-free Algol also became the foundation for Reddy’s object-based semantics [Red96].

The techniques of **SCI** can be adapted to a variety of languages. We believe that it can serve as a foundation for the design and study of more advanced languages, such as those which address references or synchronization. But for **SCI** to gain wider acceptance, practical type inference algorithms must be designed to allow static typechecking of programs which do not provide any explicit type information. Such algorithms have long been available for functional languages with state such as ML and Haskell, but in the absence of any interference control.

Huang and Reddy first studied the problem of type reconstruction for **SCIR** [HR96]. Because of the interaction between the Passification and Contraction rules of **SCIR**, there may exist many possible type derivations for any term. It can be shown that **SCIR** does not have principal typings. Huang and Reddy were able to infer principal typings in their **SCIR_K** system by extending **SCIR**

judgements with *kind constraints* that specify the conditions under which a term is well-typed.

Our work improves upon this system in two important ways. First, the **promote** and **derelict** operators are explicit in the term syntax of **SCIR_K**. They are used frequently, but since they serve only to provide type information, they should be made implicit for programming convenience. We show how these coercion terms can be removed from the language.

Second, there is no provision for polymorphism in **SCIR_K**. Adding **let** terms in the Hindley-Milner style is non-trivial, because type quantification is complicated by the need to discharge the kind constraints in judgements. Furthermore, the substitution-based approach fails in the presence of interference. We discuss a third approach which can be used to support polymorphism along with interference control.

The next section of this paper reviews the principles of **SCI** and presents the **SCIR** type system. Section 3.1 introduces our extended system with implicit coercions and let-based polymorphism. In Section 4, we outline a reconstruction algorithm and illustrate the system with several examples. Concluding remarks are given in Section 5.

1.1 Related Work

Our work is unique in that it is the only one to address interference control, but there are other lines of research which are partially related. In effect systems [LG88, TJ94], types are used to specify (but not control) the side effects that may occur in program phrases. This information can aid a compiler in determining constraints on the order of execution of terms in a parallel system [Luc87].

Wadler presents a set of linear type systems aimed at deriving information about identifier sharing in programs [Wad90, Wad91]. Unlike **SCIR**, the languages considered are purely functional and do not deal with side effects. Much research has focused on extending functional languages with state [LP95, PW93, SRI91, SRI97, CO94]. In these systems, the state is single-threaded and references cannot escape the thread, but interference may still occur within a thread.

Our work incorporates some ideas from the type inference algorithm designed for **ILCR**, an imperative lambda calculus [YR97]. In particular, their system supports implicit coercion of terms between state-dependent and state-independent types, which corresponds closely to the **promote** and **derelict** operators of **SCIR**.

2 The SCIR Type System

We introduce the concepts of **SCI** in the context of an Algol-like functional language with state and call-by-name evaluation. The types of the language are divided into two levels. The *data types* δ represent storable values, such as **int** and **bool**. *Phrase types* θ , given by the following abstract syntax, are the types of arbitrary terms:

$$\theta ::= \delta \mid \mathbf{var}[\delta] \mid \mathbf{comm} \mid \theta_1 \times \theta_2 \mid \theta_1 \rightarrow \theta_2$$

The phrase types δ (sometimes written $\mathbf{exp}[\delta]$) are the types of state-dependent expressions which yield δ -typed values. Variables are constrained so that only values of types δ may be stored. The base type **comm** is the type of values that modify the global state. Type constructors \times and \rightarrow are as in typed lambda calculus.

Some typical terms of Algol languages are given by the following syntax. We assume c ranges over a set of constants and i ranges over $\{1, 2\}$:

$$\begin{aligned} M ::= & c \mid x \mid \lambda x. M \mid M_1 M_2 \mid \\ & \langle M_1, M_2 \rangle \mid \pi_i M \mid \mathbf{if} M_1 \mathbf{then} M_2 \mathbf{else} M_3 \mid \mathbf{rec} M \mid \mathbf{do}[\delta] \lambda x. M \mid \\ & \mathbf{skip} \mid M_1 := M_2 \mid \mathbf{get}(M) \mid M_1; M_2 \mid \mathbf{new}[\delta] \lambda x. M \end{aligned}$$

The basic commands include **skip**, assignments and variable dereferencing. The “;” operator sequences commands. Finally, $\mathbf{new}[\delta] \lambda x. M$ executes the command M after binding x to a newly allocated variable of type $\mathbf{var}[\delta]$. The scope of x is the term M .

The functional side of the language is λ -calculus extended with pairs, conditionals and the recursive Y-combinator. The term $\mathbf{do}[\delta] \lambda x. M$ allows computations which only manipulate local state to be embedded in a δ -typed term. The command M is executed with x bound to a new variable of type $\mathbf{var}[\delta]$. The result of the term is the δ -typed value stored in x upon completion of M . In the recommended usage, M should not cause any “side effects” (changes to non-local variables other than x).

2.1 SCIR

The basic approach of **SCI** is to focus on the free identifiers of terms. As a first approximation, two terms interfere if they have any common free identifiers. For example, the terms $x := \mathbf{get}(x) + 1$ and $y := \mathbf{get}(x)$ interfere because x is free in both terms, and the assignment to x can obviously affect the result of the assignment to y .

Terms with disjoint sets of free identifiers will not interfere only if we can guarantee that *distinct* identifiers do not interfere. In our example language, the only ways to bind identifiers are via `new`[δ] and function applications. `new`[δ] always creates a fresh storage location, so there can be no aliasing. For applications, **SCI** requires that M and N do not interfere in any term $(M N)$, so that the lambda-bound identifier of the function M cannot be aliased to any free identifiers in M . For example, in the term

$$(\lambda x. x := \mathbf{get}(x) + 1; y := \mathbf{get}(y) + 1) y$$

the function and argument interfere, and variables x and y are aliases within the abstraction body.

This basic definition of interference is too restrictive. Reynolds designates a subset of the types as *passive* types. Terms with passive types are guaranteed not to have any “side effects” and cannot interfere with other passive terms.¹ Furthermore, all free identifiers in a passive term are said to *occur passively*. A free identifier x is a *passive identifier* if each of its occurrences is within some passive subterm. Other identifiers are *active identifiers*.

In our basic language, the types δ are passive. For function types, $\theta_1 \rightarrow \theta_2$ is passive iff θ_2 is passive. The argument type θ_1 is inconsequential, because if the body of the function has a passive type, then it cannot have any “side effects.”

Functions are often regarded as free from “side effects” if they do not change global variables, even if they can cause state changes via their arguments. The `dotwice` function $\lambda c. c; c$ is an example of this. It is possible to regard such terms as passive terms, but we must capture in their types the information that they do not change global variables. For this purpose, we add a new type constructor “!”. The type $!\theta$ represents values of type θ that do not change global variables. The type of `dotwice`, then, would be $!(\mathbf{comm} \rightarrow \mathbf{comm})$.²

The type syntax of our language is extended with “!”, and passive types can be formally defined.

$$\begin{array}{ll} \text{(types)} & \theta ::= \delta \mid \mathbf{var}[\delta] \mid \mathbf{comm} \mid \theta_1 \times \theta_2 \mid \theta_1 \rightarrow \theta_2 \mid !\theta \\ \text{(passive types)} & \phi ::= \delta \mid \phi_1 \times \phi_2 \mid \theta \rightarrow \phi \mid !\theta \end{array}$$

We formalize the above intuitions in **SCIR**, whose type rules are shown in Figure 2. The system we discuss is similar to the one presented in [OPTT95], but extended with the new type constructor

¹This intuition is formalized by the semantic models of [OPTT95, Red96].

²Our types $!(\theta \rightarrow \theta')$ correspond to the passive function types $\theta \rightarrow_p \theta'$ in Reynolds’s presentation [Rey78]. Adding a generic “!” constructor is a useful notational convenience.

$$\begin{array}{c}
\frac{}{| x:\theta \vdash x:\theta} \text{Axiom} \\
\frac{\Pi \mid \Gamma, x:\theta \vdash M:\phi}{\Pi, x:\theta \mid \Gamma \vdash M:\phi} \text{Passification} \qquad \frac{\Pi, x:\theta \mid \Gamma \vdash M:\theta'}{\Pi \mid \Gamma, x:\theta \vdash M:\theta'} \text{Activation} \\
\frac{\Pi \mid \Gamma \vdash M:\theta}{\Pi, \Pi' \mid \Gamma, \Gamma' \vdash M:\theta} \text{Weakening} \\
\frac{\Pi, x:\theta, x':\theta \mid \Gamma \vdash M:\theta'}{\Pi, x:\theta \mid \Gamma \vdash M[x/x']:\theta'} \text{Contraction} \\
\frac{\Pi \mid \Gamma \vdash M:\theta_1 \quad \Pi \mid \Gamma \vdash N:\theta_2}{\Pi \mid \Gamma \vdash \langle M, N \rangle:\theta_1 \times \theta_2} \times I \qquad \frac{\Pi \mid \Gamma \vdash M:\theta_1 \times \theta_2}{\Pi \mid \Gamma \vdash \pi_i M:\theta_i} \times E_i \ (i = 1, 2) \\
\frac{\Pi \mid \Gamma, x:\theta_1 \vdash M:\theta_2}{\Pi \mid \Gamma \vdash \lambda x:\theta_1. M:\theta_1 \rightarrow \theta_2} \rightarrow I \qquad \frac{\Pi_1 \mid \Gamma_1 \vdash M:\theta_1 \rightarrow \theta_2 \quad \Pi_2 \mid \Gamma_2 \vdash N:\theta_1}{\Pi_1, \Pi_2 \mid \Gamma_1, \Gamma_2 \vdash MN:\theta_2} \rightarrow E \\
\frac{\Pi \mid \vdash M:\theta}{\Pi \mid \vdash \mathbf{promote} M:!\theta} \text{Promotion} \qquad \frac{\Pi \mid \Gamma \vdash M:!\theta}{\Pi \mid \Gamma \vdash \mathbf{derelict} M:\theta} \text{Dereliction}
\end{array}$$

Figure 2: Type rules for **SCIR**

“!”. Typing judgements are of the form $\Pi \mid \Gamma \vdash M:\theta$ where M is a term, and θ is its type. The context is partitioned into the *passive zone* Π , which contains passive identifiers, and the *active zone* Γ , which may contain any kind of identifier.

The Passification rule says that all free identifiers in a passive term occur passively. Activation allows us to ignore the fact that x is a passive identifier, which is necessary for abstracting x with $\rightarrow I$. Weakening allows free identifiers to be added to the context. Identifier sharing is achieved via Contraction, which allows two *passive* free identifiers x and x' to be shared. As a consequence, it is possible for passive free identifiers to occur in both the operator and operand of an application.

The rule $\times I$ allows unrestricted identifier sharing between the components of the pair. But for $\rightarrow E$, the zones $\Pi_1, \Pi_2, \Gamma_1, \Gamma_2$ must all be disjoint to prevent interference between distinct identifiers. Any term may be explicitly promoted to a passive type using Promotion if it contains no active free identifiers; thus Γ should be empty. The Dereliction rule is necessary in order to apply the elimination rules to a promoted term.

It is worth noting that currying does not hold in **SCIR**. In most functional languages, we would expect that the application $(M \langle N, L \rangle)$ is equivalent to $(\mathbf{curry}(M) N L)$. But if subterms N and

$:=_{\delta} : \mathbf{var}[\delta] \times \delta \rightarrow \mathbf{comm}$ $\mathbf{if}_{\theta} : \mathbf{bool} \times \theta \times \theta \rightarrow \theta$ $\mathbf{do}[\delta] : !(\mathbf{var}[\delta] \rightarrow \mathbf{comm}) \rightarrow \delta$ $\parallel : \mathbf{comm} \rightarrow \mathbf{comm} \rightarrow \mathbf{comm}$ $\mathbf{skip} : \mathbf{comm}$	$\mathbf{get}_{\delta} : \mathbf{var}[\delta] \rightarrow \delta$ $\mathbf{rec}_{\theta} : !(\theta \rightarrow \theta) \rightarrow \theta$ $\mathbf{new}[\delta] : (\mathbf{var}[\delta] \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm}$ $; : \mathbf{comm} \times \mathbf{comm} \rightarrow \mathbf{comm}$
---	--

Figure 3: Constants for **SCIR**

L interfere in **SCIR**, then the former application may be typable even when the latter is not.

This basic type system is enough to define many Algol constructs as constants, as shown in Figure 3. With interference control, we can now add a determinate parallel composition operator “ \parallel ”, which is modelled by function application.

To illustrate **SCIR**, we explain how to derive a typing for the `map` procedure from Figure 1. It is simple enough to translate the example program into **SCIR** syntax. An array of integer variables, such as `next`, can be treated as a function of type $\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}]$.

$$\mathbf{map} \equiv \mathbf{rec}(\mathbf{promote} (\lambda \mathbf{map}. \lambda p. \lambda i. \\ \mathbf{if} (i = 0) \mathbf{then} \mathbf{skip} \\ \mathbf{else} p \ i; \mathbf{map} \ p \ \mathbf{get}(\mathbf{next}(i))))$$

Even though `next` does not have a passive type, it is only used passively and can appear in the passive zone:

$$\frac{i:\mathbf{int} \mid \mathbf{next}:\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}] \mid \vdash \mathbf{get}(\mathbf{next}(i)):\mathbf{int}}{i:\mathbf{int}, \mathbf{next}:\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}] \mid \vdash \mathbf{get}(\mathbf{next}(i)):\mathbf{int}} \text{Passification}$$

The rest of the derivation is straightforward. Promotion can be applied after $\rightarrow I$, because `next` is the only free variable, and it is in the passive zone.

$$\begin{array}{c} \vdots \\ \frac{\mathbf{next}:\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}] \mid \vdash \\ \lambda \mathbf{map}. \lambda p. \lambda i. \dots : ((\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}) \rightarrow (\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}}{\mathbf{next}:\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}] \mid \vdash \\ \mathbf{promote} (\dots) : !(((\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}) \rightarrow (\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm})} \text{Promotion} \\ \frac{\mathbf{next}:\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}] \mid \vdash \\ \mathbf{promote} (\dots) : !(((\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}) \rightarrow (\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm})}{\mathbf{next}:\mathbf{int} \rightarrow \mathbf{var}[\mathbf{int}] \mid \vdash \mathbf{rec}(\mathbf{promote} (\dots)) : (\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}} \text{Rec} \end{array}$$

3 Type Inference

In this section, we consider the issue of type inference for **SCIR** with implicit dereliction and polymorphism. We first present a sound and complete reconstruction system where only `derelict` is

implicit. Then, extensions which allow **promote** to be made implicit in certain places and polymorphism are discussed.

For clarity, we omit cross products from our discussion, since they are not immediately relevant to the issue of implicit promotion and dereliction. It is straightforward to add products into our system. The reader is referred to [HR96] for further details.

3.1 Implicit Dereliction

The input to the type inference algorithm is a preterm in which the type declarations of lambda-bound identifiers and **derelict** operations are omitted. The context-free grammar for preterms is as follows:

$$e ::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \mathbf{promote} e$$

(We use e to represent terms in the implicit language, while M represents terms of the explicit language of Section 2.)

The algorithm succeeds if there is some way to fill in the missing information to obtain a well-typed **SCIR** term. The algorithm will produce a *principal typing* which represents all possible ways that the missing information can be filled in.

Since principal typings will involve type variables, we must extend the type syntax presented in the last section. In addition, we will restrict our attention to *standard types*, where the “!” constructor may be applied at most once to any type. There is no loss of expressiveness, since for any passive type ϕ , we can show that $!\phi \cong \phi$. Since $!\theta$ is a passive type, we have $!\theta \cong !!\theta$.

$$\begin{aligned} \theta & ::= !^n \rho \\ \rho & ::= \delta \mid \mathbf{var}[\delta] \mid \mathbf{comm} \mid \theta_1 \rightarrow \theta_2 \mid \alpha \\ n & ::= 0 \mid 1 \mid i \end{aligned}$$

The types ρ range over types without an outermost “!” constructor. Standard types θ may or may not contain an outermost “!”. We cover both cases uniformly in the type syntax by using the notation $!^n$, where $n \in \{0, 1\}$ is an *annotation*. The type $!^1 \rho$ denotes $!\rho$, while $!^0 \rho$ denotes ρ . Type variables α range over ρ -type variables, while *annotation variables*, ranged over by i , stand for the 0 and 1 annotations of “!”.

The subset of passive types ϕ may be redefined as follows:

$$\phi ::= !^0 \delta \mid !^0 (\theta \rightarrow \phi) \mid !^1 \rho$$

$$\begin{aligned}
p(!^n \rho) &= (n = 1 \vee p(\rho)) \\
p(\delta) &= \text{true} \\
p(\mathbf{comm}) &= \text{false} \\
p(\mathbf{var}[\delta]) &= \text{false} \\
p(\theta_1 \rightarrow \theta_2) &= p(\theta_2)
\end{aligned}$$

Figure 4: The definition of p

We represent the distinction between passive and general types by a predicate $p(\theta)$ that is true iff θ is passive. The definition of the predicate (other than for type variables) is given in Figure 4. To determine the passivity of type variables, we must rely on a *kind assignment*. However, a **SCIR** term can type check under different kind assignments for its type variables. For example, consider the preterm

promote ($f x$)

For it to type check, the free variables f and x must be used passively, so either

1. both f and x have passive types, or
2. $(f x)$ has a passive type.

The reconstruction algorithm uses *kind constraints* to represent the class of all kind assignments under which a term is typable. The constraints are boolean predicates given by the following syntax:

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

A constraint is *satisfiable* if there is some kind assignment under which it can be simplified to true. We regard two constraints as equal, $C_1 = C_2$ if for all type substitutions σ , $\sigma(C_1) \Leftrightarrow \sigma(C_2)$. We feel free to write $n = 0$ for $n \leq 0$ and $n = 1$ for $1 \leq n$.

Returning to the above example, type checking considerations of $(f x)$ suggest the following standard types for f and x :

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

Then the constraint under which **promote** ($f x$) typechecks in **SCIR** is $(p(!^i(!^j \alpha \rightarrow !^k \beta)) \wedge p(!^l \alpha)) \vee p(!^k \beta)$, which simplifies to $(i = 1 \wedge p(!^l \alpha)) \vee p(!^k \beta)$.

Note that if we type check a term from the bottom up, we cannot tell that this constraint is necessary until the **promote** operator is seen. This suggests that we must associate kind constraints with free identifiers as well as the term.

The judgements in our new system, called *inference judgements*, are of the form:

$$x_1:\theta_1 [P_1; C_1], \dots, x_n:\theta_n [P_n; C_n] \vdash e:\theta [G]$$

Sometimes we represent the assumptions using vector notation, such as $\vec{x}:\vec{\theta} [\vec{P}; \vec{C}]$, or $\Gamma [\vec{P}; \vec{C}]$ when we are only interested in \vec{P} and \vec{C} .

With each free identifier x_i is associated a *passification constraint* P_i and a *contraction constraint* C_i . The constraint G is called the *global constraint*. We also refer to these as the P-constraint, C-constraint, and G-constraint respectively. The P-constraint P_i of a free identifier x_i indicates the conditions under which x_i can be regarded as passive; i.e., if all occurrences of x_i are in some passive subterm. Recall that in a well-typed term (MN) , all free identifiers common to M and N should be passive, so that Contraction could have been applied in **SCIR**. The constraint C_i indicates the condition under which all occurrences of the free identifier x_i can be contracted. So, we would expect to have $P_i \Rightarrow C_i$ as an invariant. Finally, the G-constraint contains conditions arising from promotions, implicit derelictions, and C-constraints of bound variables.

A judgement $A \vdash e:\theta [G]$ can be read as “The term e has type θ under the assignment A , as long as the constraint G and all contraction constraints in A hold. Further, all free identifiers x whose passification constraints in A hold are passive free identifiers.” Formally, the semantics of inference judgements is defined as follows:

- Definition 1**
1. Let e be a preterm without **derelict** and type declarations for λ -bound identifiers. Then a judgement $\Pi | \Gamma \vdash e:\theta$ is *implicitly derivable* if there is a derivable **SCIR** term $\Pi | \Gamma \vdash M:\theta$ such that e is related to M by the *Erase* relation; that is, e is obtained by erasing type declarations and **derelict** from M .
 2. A judgement $\Pi | \Gamma \vdash e:\theta'_0$ is an *instance* of an inference judgement $\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash e:\theta' [G]$ iff there is a substitution σ for the type variables in the latter such that
 - (a) the judgement $\sigma(\vec{x}:\vec{\theta}, \vec{z}:\vec{\theta}'' \vdash e:\theta')$ is the same as $\Pi, \Gamma \vdash e:\theta'_0$ for some \vec{z} and $\vec{\theta}''$ (which represent weakening),

$$\begin{array}{c}
\frac{}{x: !^n \rho [p(!^n \rho); \text{true}] \vdash x: !^m \rho [(m \leq n)]} \text{Axiom} \\
\\
\frac{\vec{z}: \vec{\theta} [\vec{P}; \vec{C}], x: \theta_1 [P_1; C_1] \vdash e: \theta_2 [G]}{\vec{z}: \vec{\theta} [\vec{P}; \vec{C}] \vdash \lambda x. e: !^0(\theta_1 \rightarrow \theta_2) [G \wedge C_1]} \rightarrow I \\
\\
\frac{\vec{z}: \vec{\theta} [\vec{P}; \vec{C}] \vdash e: \theta_2 [G]}{\vec{z}: \vec{\theta} [\vec{P}; \vec{C}] \vdash \lambda x. e: !^0(\theta_1 \rightarrow \theta_2) [G]} \rightarrow I' \quad (x \text{ not in } \vec{z}) \\
\\
\frac{\vec{x}: \vec{\theta} [\vec{P}; \vec{C}], \vec{y}: \vec{\theta}_1 [\vec{P}_1; \vec{C}_1] \vdash e_1: !^0(\theta' \rightarrow !^n \rho) [G_1] \quad \vec{x}: \vec{\theta} [\vec{P}'; \vec{C}'], \vec{z}: \vec{\theta}_2 [\vec{P}_2; \vec{C}_2] \vdash e_2: \theta' [G_2]}{\vec{x}: \vec{\theta} [(\vec{P} \wedge \vec{P}') \vee p(!^n \rho); (\vec{P} \wedge \vec{P}') \vee p(!^n \rho)], \vec{y}: \vec{\theta}_1 [\vec{P}_1 \vee p(!^n \rho); \vec{C}_1 \vee p(!^n \rho)], \\ \vec{z}: \vec{\theta}_2 [\vec{P}_2 \vee p(!^n \rho); \vec{C}_2 \vee p(!^n \rho)] \vdash e_1 e_2: !^m \rho [G_1 \wedge G_2 \wedge (m \leq n)]} \rightarrow E \\
\\
\frac{\vec{x}: \vec{\theta} [\vec{P}; \vec{C}] \vdash e: !^0 \rho [G]}{\vec{x}: \vec{\theta} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}] \vdash \mathbf{promote} \ e: !^m \rho [G \wedge (\bigwedge \vec{P})]} \text{Promotion}
\end{array}$$

Figure 5: Type Inference Rule

- (b) $\sigma(G)$ and $\sigma(C_i)$ are true for all x_i , and
- (c) the constraint $\sigma(P_i)$ is true for all $x_i: \sigma(\theta_i)$ in Π .

3. An inference judgement is *valid* if all its ground instances (where “ground” means with no type variables) are implicitly derivable in **SCIR**.

□

An inference system for valid inference judgements is shown in Figure 5. For $\rightarrow E$, the identifier lists \vec{y} and \vec{z} are disjoint. Identifiers common to both hypotheses are represented by \vec{x} , but the constraint information may differ in each hypothesis. Boolean operations on constraint vectors can be defined straightforwardly. If $\vec{P} = P_1, \dots, P_n$ and $\vec{P}' = P'_1, \dots, P'_n$, then

$$\begin{aligned}
\vec{P} \vee p(\mu) &= P_1 \vee p(\mu), \dots, P_n \vee p(\mu) \\
\vec{P} \wedge \vec{P}' &= P_1 \wedge P'_1, \dots, P_n \wedge P'_n \\
\bigwedge \vec{P} &= P_1 \wedge P_2 \wedge \dots \wedge P_n
\end{aligned}$$

Also, $\overrightarrow{\text{true}}$ is a vector of trivially satisfiable constraints.

There are two important aspects of the system: interference control and implicit dereliction. For interference control, rules Axiom and $\rightarrow E$ weaken the constraints in the assumptions. This

ensures that if the terms x or $(e_1 e_2)$ have passive types, then the P- and C-constraints for all free identifiers of the terms will hold. This constraint weakening is redundant for $\rightarrow I$ and $\rightarrow I'$.

Since the function and argument of an application term should not interfere, the rule $\rightarrow E$ modifies the C-constraints of all shared identifiers \vec{x} . The C-constraint holds (and Contraction is possible) only if \vec{x} are all passive identifiers. The Promotion rule ensures that all free identifiers of e occur passively by adding all P-constraints of the assumption list to the G-constraint. The rule $\rightarrow I$ also modifies the G-constraint. Since x is removed from the assumptions, its C-constraint cannot be further weakened.

Regarding dereliction, note that there is no Dereliction rule in the new system. It has been merged with the rules Axiom, $\rightarrow E$ and Promotion, which allow a type of the form $!^m \rho$ in the conclusion (instead of $!^n \rho$ or $!^1 \rho$) to indicate an implicit dereliction step. We need not consider the possibility of dereliction after the other steps, because of the following property:

Lemma 2 In any **SCIR** derivation, every dereliction step occurs after either an Axiom, Promotion, or elimination step followed by zero or more structural rule steps (Weakening, Contraction, Passification, or Activation). \square

Examples of derivable (and valid) inference judgements are

$$\begin{aligned} & f: !^i (!^j \alpha \rightarrow !^k \beta) [i = 1 \vee p(!^k \beta); \text{true}], x: !^l \alpha [p(!^l \alpha) \vee p(!^k \beta); \text{true}] \\ & \vdash f x: !^m \beta [m \leq k \wedge j \leq l] \end{aligned}$$

$$\begin{aligned} & f: !^i (!^j \alpha \rightarrow !^k \beta) [\text{true}; \text{true}], x: !^l \alpha [\text{true}; \text{true}] \\ & \vdash \mathbf{promote} (f x): !^n \beta [((i = 1 \wedge p(!^l \alpha)) \vee p(!^k \beta)) \wedge (j \leq l)] \end{aligned}$$

The latter typing is derivable from the former by Promotion. Note the additional conjunct $j \leq l$ in the G-constraints of these judgements, which was not mentioned in the previous discussion of **promote** ($f x$). This constraint arises from the possibility of derelicting x before f is applied.

Two example instances of these judgements in **SCIR** are:

$$f: !((\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow (\mathbf{int} \rightarrow \mathbf{comm})) \mid x: (\mathbf{int} \rightarrow \mathbf{comm}) \vdash f x: \mathbf{int} \rightarrow \mathbf{comm}$$

$$f: !((\mathbf{int} \rightarrow \mathbf{comm}) \rightarrow (\mathbf{int} \rightarrow \mathbf{comm})), x: !(\mathbf{int} \rightarrow \mathbf{comm}) \mid \vdash \mathbf{promote} (f x): !(\mathbf{int} \rightarrow \mathbf{comm})$$

The requisite properties of the implicit dereliction system are as follows.

Theorem 3 (Soundness) All derivable inference judgements are valid. \square

Lemma 4 If an inference judgement $\Gamma [\vec{P}; \vec{C}] \vdash e: \theta [G]$ is derivable, then $p(\theta) \wedge G \Rightarrow P_i$ holds for all P_i in \vec{P} . \square

Lemma 5 If an inference judgement $\Gamma [\vec{P}; \vec{C}] \vdash e: \theta [G]$ is derivable, then $P_i \Rightarrow C_i$ for all P_i, C_i in \vec{P} and \vec{C} . \square

Lemma 6 If $x: \theta [P; C]$, $x': \theta [P'; C']$, $A \vdash M: \theta' [G]$ is derivable, then $x: \theta [P \wedge P'; C'']$, $A \vdash e[x/x']: \theta' [G]$ is derivable, for some C -constraint C'' . \square

Theorem 7 (Completeness) Every judgement implicitly derivable in **SCIR** is an instance of a derivable inference judgement. \square

Proof. If $\Pi \mid \Gamma \vdash M: \theta$ and $Eraser(e, M)$, then the corresponding inference judgement is of the form $\Pi|_{fv(e)} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}], \Gamma|_{fv(e)} [\vec{P}; \overrightarrow{\text{true}}] \vdash e: \theta [\text{true}]$, where $\Pi|_{fv(e)}$ denotes the context Π restricted so that its domain contains only the free variables of e . The proof is by induction on the height of **SCIR** derivation and proceeds by case analysis on the last step. We show selected cases.

- *Passification*

$$\frac{\Pi \mid \Gamma, x: \theta \vdash M: \phi}{\Pi, x: \theta \mid \Gamma \vdash M: \phi} \text{Passification}$$

By induction hypothesis, $\Pi|_{fv(e)} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}], (\Gamma, x: \theta)|_{fv(e)} [\vec{P}; \overrightarrow{\text{true}}] \vdash e: \phi [\text{true}]$. By Lemma 4, $\vec{P} = \overrightarrow{\text{true}}$, which gives the conclusion.

- *Contraction*

This follows from Lemma 6 and Lemma 5.

- $\rightarrow E$

$$\frac{\Pi_1 \mid \Gamma_1 \vdash M: !^0(\theta \rightarrow !^n \rho) \quad \Pi_2 \mid \Gamma_2 \vdash N: \theta}{\Pi_1, \Pi_2 \mid \Gamma_1, \Gamma_2 \vdash M N: !^n \rho} \rightarrow E$$

By induction,

$$\begin{aligned} & \Pi_1|_{fv(e_1)} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}], \Gamma_1|_{fv(e_1)} [\vec{P}_1; \overrightarrow{\text{true}}] \vdash e_1: !^0(\theta \rightarrow !^n \rho) [\text{true}] \\ & \Pi_2|_{fv(e_2)} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}], \Gamma_2|_{fv(e_2)} [\vec{P}_2; \overrightarrow{\text{true}}] \vdash e_2: \theta [\text{true}] \end{aligned}$$

We can derive the following inference judgement from those two.

$$\begin{array}{c} \Pi_1 |_{fv(e_1)} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}], \Pi_2 |_{fv(e_2)} [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}], \\ \Gamma_1 |_{fv(e_1)} [\vec{P}_1 \vee p(!^n \rho); \overrightarrow{\text{true}}], \Gamma_2 |_{fv(e_2)} [\vec{P}_2 \vee p(!^n \rho); \overrightarrow{\text{true}}] \\ \vdash (e_1 \ e_2) : !^m \rho \ [m \leq n] \end{array}$$

From this, we can get the conclusion.

- *Dereliction*

Use Lemma 2, and consider the preceding step other than Passification.

- *Axiom*

$$\frac{\overline{| z : !^1 \rho \vdash z : !^1 \rho \text{ Axiom}}}{| z : !^1 \rho \vdash \mathbf{derelict} \ z : !^0 \rho \text{ Dereliction}}$$

or

$$\frac{\frac{\overline{| z : !^1 \rho \vdash z : !^1 \rho \text{ Axiom}}}{z : !^1 \rho \mid \vdash z : !^1 \rho \text{ Passification}}}{z : !^1 \rho \mid \vdash \mathbf{derelict} \ z : !^0 \rho \text{ Dereliction}}$$

Since $p(!^1 \rho) = \text{true}$, we can derive $z : !^1 \rho [\text{true}; \text{true}] \vdash z : !^0 \rho [\text{true}]$ by Axiom.

- *Weakening*

$$\frac{\frac{\frac{\Pi \mid \Gamma_1, \Gamma_2 \vdash M : !^1 \rho}{\Pi, \Pi' \mid \Gamma_1, \Gamma_2, \Gamma'_1, \Gamma'_2 \vdash M : !^1 \rho} \text{Weakening}}{\Pi, \Pi', \Gamma_1, \Gamma'_1 \mid \Gamma_2, \Gamma'_2 \vdash M : !^1 \rho} \text{Passification}}{\Pi, \Pi', \Gamma_1, \Gamma'_1 \mid \Gamma_2, \Gamma'_2 \vdash \mathbf{derelict} \ M : !^0 \rho \text{ Dereliction}}$$

Then reorder the steps.

$$\frac{\frac{\frac{\Pi \mid \Gamma_1, \Gamma_2 \vdash M : !^1 \rho}{\Pi, \Gamma_1 \mid \Gamma_2 \vdash M : !^1 \rho} \text{Passification}}{\Pi, \Gamma_1 \mid \Gamma_2 \vdash \mathbf{derelict} \ M : !^0 \rho \text{ Dereliction}}}{\Pi, \Pi', \Gamma_1, \Gamma'_1 \mid \Gamma_2, \Gamma'_2 \vdash \mathbf{derelict} \ M : !^0 \rho \text{ Weakening}}$$

By induction hypothesis, the consequent of the dereliction step is an instance of a derivable inference judgement. This gives us the conclusion.

□

3.2 Implicit Promotion

In this section, we consider the issues of making **promote** implicit. Note that to apply **promote**, all the free identifiers must be passive. This leads to a large number of constraints (see Figure 5). If **promote** operators are to be left implicit, then the inference algorithm must account for the possibility of promotion at every term position. This leads to excessively large constraints.

A practical solution is to allow implicit promotion for a few selected operators. For example, **do** $[\delta]$ and **rec** are good choices because they take arguments of “!” types and, in practice, these operators often occur in combination with **promote**. The following type rules allow one to write **do** $[\delta]$ e instead of **do** $[\delta]$ (**promote** e) and **rec** e instead of **rec** (**promote** e):

$$\frac{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash e: !^n (!^k \rho \rightarrow !^k \rho) [G]}{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash \mathbf{rec} e: !^m \rho [G \wedge (n = 1 \vee (\bigwedge \vec{P})) \wedge (m \leq k)]} \text{Rec}$$

$$\frac{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash e: !^n (!^0 \mathbf{var}[\delta] \rightarrow !^0 \mathbf{comm}) [G]}{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash \mathbf{do}[\delta] e: !^0 \delta [G \wedge (n = 1 \vee (\bigwedge \vec{P}))]} \mathbf{do}[\delta]$$

The basic idea of these rules is to combine Promotion with rules for applying **rec** and **do** $[\delta]$ respectively. The constraint $(n = 1 \vee (\bigwedge \vec{P}))$ represents the implicit promotion of e . That is, if e is not already of a promoted function type, we promote it implicitly.

By changing the definition of *Erase*, the soundness and completeness theorems in Section 3.1 can be carried over to this extended system.

Definition 8 For any term M and preterm e , *Erase* is a relation such that *Erase*(e, M) iff e is obtained by erasing not only **derelict** and type declarations from M but also *selectively* erasing **promote** when it is used in **do** $[\delta]$ (**promote** M') or **rec**(**promote** M'). \square

Theorem 9 (Soundness, Completeness) All derivable inference judgements are valid and every judgement implicitly derivable in **SCIR** is an instance of a derivable inference judgement. \square

As an example of the new **rec** rule, consider erasing the explicit **promote** operator from the

map definition given in Section 2.1. The last step in the derivation of this term appears below: ³

$$\frac{\begin{array}{l} \text{next}: !^i(\mathbf{int} \rightarrow !^j \mathbf{var}[\mathbf{int}]) [\text{true}; \text{true}] \vdash \lambda \text{map}. \lambda p. \lambda i. \dots \\ : ((\mathbf{int} \rightarrow !^k \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}) \rightarrow ((\mathbf{int} \rightarrow !^k \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm}) [\text{true}] \end{array}}{\text{next}: !^i(\mathbf{int} \rightarrow !^j \mathbf{var}[\mathbf{int}]) [\text{true}; \text{true}] \vdash \mathbf{rec}(\lambda \text{map} \dots): (\mathbf{int} \rightarrow !^k \mathbf{comm}) \rightarrow \mathbf{int} \rightarrow \mathbf{comm} [\text{true}]} \text{Rec}$$

Similarly, the following inference judgement for the **reclaim** function of Figure 1 can be derived:

$$\begin{array}{l} \text{next}: !^l(\mathbf{int} \rightarrow !^m \mathbf{var}[\mathbf{int}]) [l = 1 \vee m = 1; \text{true}], \\ \text{prev}: !^n(\mathbf{int} \rightarrow !^p \mathbf{var}[\mathbf{int}]) [n = 1 \vee p = 1; \text{true}], \\ \text{free}: !^q \mathbf{var}[\mathbf{int}] [q = 1; \text{true}], \\ \text{head}: !^r \mathbf{var}[\mathbf{int}] [r = 1; \text{true}] \\ \vdash \mathbf{reclaim}: (!^s \mathbf{int} \rightarrow \mathbf{comm}) \end{array}$$

3.3 Polymorphism

In ML-like languages, polymorphism is made available via the **let** construct. A term of the form

$$\mathbf{let} \ z = e_1 \ \mathbf{in} \ e_2$$

is type checked by allowing z to take a quantified type such as $\forall \alpha. \theta$. It can be shown that the effect of such quantification is the same as regarding the above **let** term as being equivalent to $e_2[e_1/z]$ (cf. [Mit97]).

Unfortunately, neither of these solutions is directly applicable to **SCIR**. The type quantification idea runs into the problem that the type variable to be quantified (α) might occur in the kind constraints. For example, the principal typing of e_1 might involve a kind constraint of the form $p(\alpha) \vee p(\beta)$. No single kind assignment to α covers all possibilities. The substitution idea is unsound for **SCIR** because desugaring of $\mathbf{let} \ z = e_1 \ \mathbf{in} \ e_2$ as

$$(\lambda z. e_2) e_1$$

requires that $\lambda z. e_2$ and e_1 should not interfere. Therefore, $\mathbf{let} \ z = e_1 \ \mathbf{in} \ e_2$ is not equivalent to $e_2[e_1/z]$ for type checking purposes.

Here, we suggest a third approach as a tentative solution. Replace all n occurrences of z in e_2 by $\pi_1 z', \dots, \pi_n z'$ and call the resulting term e_2' . Then $\mathbf{let} \ z = e_1 \ \mathbf{in} \ e_2$ is equivalent to

$$\mathbf{let} \ z' = \langle e_1, \dots, e_1 \rangle \ \mathbf{in} \ e_2'$$

³To make examples easy to read, we will remove $!^0$ from all of the examples.

which is in turn equivalent to

$$(\lambda z'. e'_2) \langle e_1, \dots, e_1 \rangle \quad (1)$$

Type checking $\mathbf{let} z = e_1 \mathbf{in} e_2$ is equivalent to type checking (1). More precisely, we use the type rule:⁴

$$\frac{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash (\lambda z'. e'_2) \langle e_1, \dots, e_1 \rangle : \theta' [G]}{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash \mathbf{let} z = e_1 \mathbf{in} e_2 : \theta' [G]} \mathbf{let}_1 \quad (z \text{ occurs free in } e_2)$$

(for $i = 1, \dots, n$, each $\pi_i z'$ occurs precisely once in e'_2)

We still need to type check e_1 and prevent interference between e_1 and e_2 even if z is not free in e_2 .

$$\frac{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash (\lambda z. e_2) e_1 : \theta [G]}{\vec{x}:\vec{\theta} [\vec{P}; \vec{C}] \vdash \mathbf{let} z = e_1 \mathbf{in} e_2 : \theta' [G]} \mathbf{let}_2 \quad (z \text{ doesn't occur free in } e_2)$$

For example, $\mathbf{let} id = \lambda x. x \mathbf{in} \langle id \ 3, id \ (y:=3) \rangle$ can be type checked as follows:

$$\frac{y: !^i \mathbf{var}[\mathbf{int}] [i = 1; \mathbf{true}] \vdash (\lambda z'. \langle (\pi_1 z') \ 3, (\pi_2 z') \ (y:=3) \rangle) \langle \lambda x. x, \lambda x. x \rangle : !^j \mathbf{int} \times \mathbf{comm} [\mathbf{true}]}{y: !^i \mathbf{var}[\mathbf{int}] [i = 1; \mathbf{true}] \vdash \mathbf{let} id = \lambda x. x \mathbf{in} \langle id \ 3, id \ (y:=3) \rangle : !^j \mathbf{int} \times \mathbf{comm} [\mathbf{true}]}$$

The type reconstruction algorithm in Section 4 implements the effect of this rule in a compositional fashion.

4 Reconstruction Algorithm

The type reconstruction algorithm is directly based on the rules which are described in Section 3. In fact, the type rules have been designed so that they essentially form a “logic program” for type reconstruction. They are syntax-directed and are closed under type substitution. Sample clauses of the reconstruction algorithm are shown in Figure 6 and Figure 7 using implicit pattern matching style. We assume bound identifiers are different from one another and free identifiers.

Recall in the rule \mathbf{let}_1 , we need to infer a typing for the n -tuple whose components represent different instantiations of a \mathbf{let} -bound term. Inferring a typing for the n -tuple and applying the application rule is inefficient. Instead, we keep track of type informations associated with each instantiation in the second component of the return value of \mathbf{T} .

The reconstruction algorithm \mathbf{T} takes a term and an “environment,” which is a collection of typings for distinct \mathbf{let} -bound identifiers. It returns a typing judgement and a set of “instantiations,” which are (identifier z , type assignment, G-constraint) tuples for each \mathbf{let} -bound identifier

⁴In our inference system, there is no rule for n -tuple. But the rules for cross product can easily be generalized for n -tuples.

$$\mathbf{T} : \text{Preterm} \times \text{Env} \rightarrow \text{Judgement} \times \text{“Instantiations”}$$

$$\begin{aligned} \mathbf{T}(x, E) = & \\ \text{if } & \Sigma \vdash x: !^i \rho [G] \in E \\ \text{then } & \langle (x: !^i \rho [p(!^k \rho)]; \text{true}] \vdash x: !^j \rho' [j \leq k], \{(x, \Sigma', G')\} \rangle \\ & \text{where } \rho', \Sigma', G' \text{ are } \rho, \Sigma, G \text{ respectively} \\ & \text{with all type variables substituted by new type variables} \\ \text{else } & \langle (x: !^i \alpha [p(!^i \alpha)]; \text{true}] \vdash x: !^j \alpha [j \leq i], \emptyset \rangle \\ \\ \mathbf{T}(\lambda x. e, E) = & \\ \text{if } & \mathbf{T}(e, E) = \langle (\Sigma, x: \theta [P''; C''] \vdash e: \theta' [G']), I \rangle \\ \text{then } & \langle (\Sigma \vdash \lambda x. e: !^0(\theta \rightarrow \theta') [G \wedge C'']), I \rangle \\ \text{else if } & \mathbf{T}(e, E) = \langle (\Sigma \vdash e: \theta' [G']), I \rangle \text{ where } x \notin \text{Dom}(\Sigma) \\ \text{then } & \langle (\Sigma \vdash \lambda x. e: !^0(!^i \alpha \rightarrow \theta') [G']), I \rangle \\ \\ \mathbf{T}((e_1 e_2), E) = & \\ \text{let } & \mathbf{T}(e_1, E) = \langle (\Sigma_1 \vdash e_1: \theta [G_1]), I_1 \rangle \\ & \mathbf{T}(e_2, E) = \langle (\Sigma_2 \vdash e_2: !^0(\theta \rightarrow !^i \alpha) [G_2]), I_2 \rangle \\ & \sigma = \text{unify}(\{\theta_1 = \theta_2 \mid x: \theta_1 [P_1; C_1] \in \Sigma_1, x: \theta_2 [P_2; C_2] \in \Sigma_2\}) \\ & I = \text{join}(I_1, I_2) \\ & \Sigma = \text{disjunct}(\text{merge}(\sigma(\Sigma_1), \sigma(\Sigma_2)), p(\sigma(!^i \alpha))) \\ \text{in } & \langle (\Sigma \vdash (e_1 e_2): \sigma(!^j \alpha) [\sigma(G_1 \wedge G_2) \wedge \sigma(j \leq i)]), \sigma(I) \rangle \\ \\ \mathbf{T}(\text{rec } e, E) = & \\ \text{let } & \mathbf{T}(e, E) = \langle (\Sigma \vdash e: !^i(!^j \rho \rightarrow !^j \rho) [G]), I \rangle \\ \text{in } & \langle (\text{setTrue}(\Sigma) \vdash \text{rec } e: !^k \rho [G \wedge (k \leq j) \wedge (i = 1 \vee \text{and}(\Sigma))]), I \rangle \\ \\ \mathbf{T}(\text{let } x = e_1 \text{ in } e_2, E) = & \\ \text{if } & x \text{ isn't a free variable of } e_2 \\ \text{then } & \mathbf{T}(((\lambda x. e_2) e_1), E) \\ \text{else } & \\ \text{let } & \mathbf{T}(e_1, E) = \langle (\Sigma_1 \vdash e_1: \theta [G_1]), I_1 \rangle \\ & E' = E \cup \{(\Sigma_1 \vdash x: \theta [G_1])\} \\ & \mathbf{T}(e_2, E') = \langle (\Sigma_2, x: \theta [P; C] \vdash e_2: \theta' [G_2]), I_2 \rangle \\ & (x, \Sigma_0, G_0) \in I_2 \\ & I = \text{join}((I_2 - \{(x, \Sigma_0, G_0)\}), I_1) \\ & \sigma = \text{unify}\{\theta_0 = \theta_2 \mid x: \theta_0 [P_0; C_0] \in \Sigma_0, x: \theta_2 [P_2; C_2] \in \Sigma_2\} \\ & \Sigma = \text{disjunct}(\text{merge}(\sigma(\Sigma_0), \sigma(\Sigma_2)), p(\sigma(\theta'))) \\ \text{in } & \langle (\Sigma \vdash \text{let } x = e_1 \text{ in } e_2: \sigma(\theta) [\sigma(G_2 \wedge C \wedge G_0)]), \sigma(I) \rangle \end{aligned}$$

Figure 6: Type reconstruction algorithm

$$\begin{aligned}
\mathbf{merge}(\Sigma_1, \Sigma_2) &= \{(x:\theta [P_1 \wedge P_2; P_1 \wedge P_2]) \mid x:\theta [P_1; C_1] \in \Sigma_1, x:\theta [P_2; C_2] \in \Sigma_2\} \\
&\cup \{(x:\theta [P_1; C_1]) \mid x:\theta [P_1; C_1] \in \Sigma_1, x \notin \text{Dom}(\Sigma_2)\} \\
&\cup \{(x:\theta [P_2; C_2]) \mid x \notin \text{Dom}(\Sigma_1), x:\theta [P_2; C_2] \in \Sigma_2\} \\
\mathbf{UniMerge}(\Sigma_1, \Sigma_2) &= \\
\quad \mathbf{let} \quad \sigma &= \mathbf{unify}(\theta_1 = \theta_2 \mid x:\theta_1 [G_1; P_1] \in \Sigma_1, x:\theta_2 [G_2; P_2] \in \Sigma_2) \\
\quad \mathbf{in} \quad &\{\sigma(x:\theta [P_1 \wedge P_2; C_1 \wedge C_2]) \mid x:\theta [P_1; C_1] \in \Sigma_1, x:\theta' [P_2; C_2] \in \Sigma_2\} \\
&\cup \{\sigma(x:\theta [P_1; C_1]) \mid x:\theta [P_1; C_1] \in \Sigma_1, x \notin \text{Dom}(\Sigma_2)\} \\
&\cup \{\sigma(x:\theta [P_2; C_2]) \mid x \notin \text{Dom}(\Sigma_1), x:\theta [P_2; C_2] \in \Sigma_2\} \\
\mathbf{disjunct}(\Sigma, A) &= \{(x:\theta [P \vee A; C \vee A]) \mid x:\theta [P; C] \in \Sigma\} \\
\mathbf{join}(I_1, I_2) &= \{(x, \mathbf{UniMerge}(\Sigma_1, \Sigma_2), G_1 \wedge G_2) \mid (x, \Sigma_1, G_1) \in I_1, (x, \Sigma_2, G_2) \in I_2\} \\
&\cup \{(x, \Sigma_1, G_1) \mid (x, \Sigma_1, G_1) \in I_1, (x, \Sigma, G) \notin I_2 \text{ (for any } \Sigma, G)\} \\
&\cup \{(x, \Sigma_2, G_2) \mid (x, \Sigma_2, G_2) \in I_2, (x, \Sigma, G) \notin I_1 \text{ (for any } \Sigma, G)\} \\
\mathbf{and}(\Sigma) &= \bigwedge \{P \mid (x:\theta [P; C]) \in \Sigma\} \\
\mathbf{setTrue}(\Sigma) &= \{x:\theta [\text{true}; \text{true}] \mid (x:\theta [P; C]) \in \Sigma\}
\end{aligned}$$

Figure 7: Type reconstruction algorithm (Continued)

z . Every occurrence of a **let**-bound variable will receive a separate instance of its typing in the environment. (The technique is from [KMM91].)

Theorem 10 $\mathbf{T}(e, \emptyset) = (\Sigma \vdash e : \theta [G])$ if and only if the judgement is the principal typing for e . \square

4.1 Practical Considerations

Figure 8 shows examples of most general types produced by the algorithm. Because of implicit promotion and dereliction, the resulting type and constraints becomes complicated quite often. We suggest some techniques to reduce the size of constraints.

We have the following isomorphisms:

$$\begin{aligned}
! \delta &\cong \delta \\
!(\theta_1 \times \theta_2) &\cong !\theta_1 \times !\theta_2 \\
!(\theta_1 \otimes \theta_2) &\cong !\theta_1 \otimes !\theta_2
\end{aligned}$$

Thus, types of the form $!\phi$, $!(\theta_1 \times \theta_2)$, and $!(\theta_1 \otimes \theta_2)$ are redundant. There is no loss in prohibiting them. Types of the form $!\mathbf{var}[\delta]$ are also quite useless because there are no values of such types (other than undefined values). Similarly, types of the form $!\mathbf{comm}$ serves no useful purpose. So,

$$\begin{aligned} \vdash \lambda f. \lambda x. f(f x) & : \ !^i(!^j \alpha \rightarrow !^k \alpha) \rightarrow !^l \alpha \rightarrow !^m \alpha \\ & [(j \leq l) \wedge (j \leq k) \wedge (m \leq k) \wedge (k = 1 \vee i = 1 \vee p(\alpha))] \\ f: !^i(!^j \mathbf{int} \times \mathbf{comm}) \rightarrow !^k(!^j \mathbf{int} \times \mathbf{comm}) & [\mathbf{true}; \mathbf{true}], y: !^l \mathbf{var}[\mathbf{int}] [\mathbf{true}; \mathbf{true}] \vdash \\ \mathbf{let} \ \mathit{twice} = \lambda f. \lambda x. f(f x) \ \mathbf{in} \ \pi_1(\mathit{twice} \ f \ \langle 3, y := 1 \rangle) & \\ : \ !^m \mathbf{int} [(p \leq i) \wedge (q \leq j) \wedge (r \leq k) \wedge (k = 1 \vee p = 1) \wedge (m \leq q)] & \\ x: !^i \mathbf{var}[\mathbf{int}] [\mathbf{true}; \mathbf{true}] \vdash \ \mathbf{do}[\mathbf{int}] \ \lambda \mathit{result}. \mathit{result} := 8; x := 10 & : \ \mathbf{int} [i = 1] \end{aligned}$$

Figure 8: Examples

we can prohibit such types as well. Then the only useful $!$ constructors are those applied to \rightarrow types, and type variables.

Using this fact, we formulate the following abbreviated notation:

$$\begin{aligned} \theta \rightarrow^n \theta' & = \ !^n(\theta \rightarrow \theta') \\ \alpha^n & = \ !^n \alpha \end{aligned}$$

With these simplifications and notations, the types of Figure 8 can be expressed as follows:

$$\begin{aligned} \vdash \lambda f. \lambda x. f(f x) & : \ (\alpha^j \rightarrow^i \alpha^k) \rightarrow \alpha^l \rightarrow \alpha^m \\ & [(j \leq l) \wedge (j \leq k) \wedge (m \leq k) \wedge (k = 1 \vee i = 1 \vee p(\alpha))] \\ f: \mathbf{int} \times \mathbf{comm} \rightarrow^i \mathbf{int} \times \mathbf{comm} & [\mathbf{true}; \mathbf{true}], y: \mathbf{var}[\mathbf{int}] [\mathbf{true}; \mathbf{true}] \vdash \\ \mathbf{let} \ \mathit{twice} = \lambda f. \lambda x. f(f x) \ \mathbf{in} \ \pi_1(\mathit{twice} \ f \ \langle 3, y := 1 \rangle) & : \ \mathbf{int} [i = 1] \\ x: \mathbf{var}[\mathbf{int}] [\mathbf{true}; \mathbf{true}] \vdash \ \mathbf{do}[\mathbf{int}] \ \lambda \mathit{result}. \mathit{result} := 8; x := 10 & : \ \mathbf{int} [\mathbf{false}] \end{aligned}$$

Note that the last term is not typable any more because the G-constraint is false.

5 Conclusion

This paper extends the previous research on type inference for **SCIR**-based languages. We have shown how the coercion operators **promote** and **derelict** may be removed from the term syntax, and introduced a technique for adding let-based polymorphism to the language.

Possible areas for future work include investigating how type quantification may be introduced into the system. It is unclear how to account for the different possible kinds of quantified type

variables. It would also be interesting to study extensions of **SCI** to include references, as in **ILC** [SRI91, SRI97].

Acknowledgements We would like to thank Uday Reddy for offering many valuable suggestions.

References

- [CO94] K. Chen and M. Odersky. A type system for a lambda calculus with assignments. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 347–364. 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.
- [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.
- [Luc87] John M. Lucassen. *Types and Effects—Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT Laboratory for Computer Science, August 1987. Technical Report LCS TR-408.
- [Mit97] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1997.
- [OPTT95] P. W. O’Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics: Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995. (Chapter 18 of [OT97b]).
- [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.
- [PW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.
- [Red96] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation*, 9:7–76, 1996.
- [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]).
- [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.
- [Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North-Holland, Amsterdam, 1990. (Proc. IFIP TC 2 Working Conf., Sea of Galilee, Israel).
- [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).
- [YR97] H. Yang and U. S. Reddy. Imperative lambda calculus revisited. University of Illinois at Urbana-Champaign, 1997.