Type Reconstruction for Syntactic Control of Interference, Part 2

Hongseok Yang
Department of Computer Science
University of Illinois
Urbana, IL 61801

Howard Huang
Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

*Syntactic Control of Interference (***SCI***) [17] has long been studied as a basis for interference-free programming, with cleaner reasoning properties and semantics than traditional imperative languages. This paper improves upon Huang and Reddy's type inference system [7] for* **SCI***-based languages in two significant ways. First, we eliminate the need for explicit coercion operators in terms. Second, we consider adding let-bound polymorphism, which appears to be nontrivial in the presence of interference control.* **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, such as aliasing, which invalidate traditional reasoning techniques. Even worse, it can be quite difficult for both users and machines to detect interference.

The example program in Figure 1, adopted from [17], illustrates some of the problems. The procedure `map` applies its argument p to each element of a linked list of integers, represented by i. Procedure `reclaim` inserts a node i into a list of free nodes, represented by the global variable `free` of type var[int]. For simplicity, we represent lists using arrays of integer variables: `next` holds the indices of the next nodes, and another array would contain the integer values of the list.

The call (`map reclaim data`) may appear to be a reasonable way to delete all elements of a list `data`. Unfortunately, `reclaim` modifies `next(`i`)` to be the head of the old free list, so the second node that is reclaimed actually belongs to the free list, not `data`! To ensure correctness, the arguments to a function should not interfere with the function itself, but it is not always obvious when this condition holds. In this case, the execution of `reclaim` adversely affects `map` via the shared array `next`.

$$
\begin{array}{l}
\text{map} \equiv \lambda p\colon \mathsf{int} \to \mathsf{comm}.\ \lambda i\colon \mathsf{int}.\\
\quad \mathsf{if}\ (i \neq 0)\ \mathsf{then}\\
\qquad p\ i;\\
\qquad \text{map}\ p\ \text{next}(i)\\
\\
\text{reclaim} \equiv \lambda i\colon \mathsf{int}.\\
\quad \text{next}(i) := \text{free};\\
\quad \text{free} := i
\end{array}
$$

Figure 1: Example of procedures which interfere

Language designers have tried to address interference using a variety of approaches. In 1978, Reynolds proposed Syntactic Control of Interference, or **SCI** [17], which syntactically restricts a language to prevent undesired interference and make it clear where interference occurs. O'Hearn *et al.* later developed **SCIR** [13], a type system and semantics for **SCI**. But **SCI** also needs practical type inference algorithms, which have long been available for functional languages with state (but in the absence of interference control), such as ML and Haskell.

Huang and Reddy first studied the problem of type inference for **SCIR** [7]. It can be shown that **SCIR** does not have principal typings; there may exist many possible type derivations for a term because of the interaction between the Passification and Contraction rules. Huang and Reddy were able to infer principal typings in their $\mathbf{SCIR}_K$ system by extending **SCIR** judgements with *kind constraints* that specify the conditions under which a term is well-typed.

Our work improves upon their system in two important ways. First, the promote and derelict operators are explicit in the term syntax of $\mathbf{SCIR}_K$, but they were implicit in Reynolds's original proposal. Intuitively, these operations do not affect any change of meaning; i.e., the implicit language is coherent [15]. In this paper, we devise type inference algorithms that allow many of these coercion operators to be left implicit.

Second, there is no provision for polymorphism in $\mathbf{SCIR}_K$. Traditional type inference techniques for let terms, such as using type quantification [3] or substitution [12], do not seem applicable to **SCIR**. We suggest a different approach which supports polymorphism in conjunction with interference control.

Passive types of **SCI** form what are called *reflective type classes* in [16]. Various other instances of reflective type classes that are useful in imperative programming are mentioned there. Applicative types in imperative lambda calculus [20] and typed λ_{var} [2] also form reflective type classes. The constraint-based techniques we present for passive types are applicable to all such reflective type classes.

The next section of this paper reviews the principles of **SCI** and presents the **SCIR** type system. Section 3 introduces our extended type system with implicit coercions and let-based polymorphism. In Section 4, we outline an inference algorithm based on this type system. Concluding remarks are given in Section 5.

Proofs are omitted from this paper due to space limitations. The full paper will be available as [23].

1.1 Related Work

Constraint-based type inference systems have been designed for many languages, including ones that support subtyping [6, 11], objects [5], overloading [8, 22], and linear types [21]. The inference issues in [21] are especially close to ours, and we received considerable inspiration from this work. One novel aspect of our system which does not arise in these other works is the need to associate constraints with each free identifier of a term, instead of requiring just a single constraint per typing judgement.

Other lines of research also address issues of interference. *Effect systems* [10] use types to specify, but not control, the side effects that may occur in program phrases.

Research on extending functional languages with state includes [9, 20, 2]. 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 for **ILCR**, an imperative lambda calculus [24]. In particular, **ILCR** 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 [18]. There are two kinds of types. *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:

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

Types δ (sometimes written $\mathsf{exp}[\delta]$) are the types of state-dependent expressions which yield δ-typed values. $\mathsf{var}[\delta]$ is the type of storage variables that hold δ-typed values, and comm is the type of commands that modify the state. Type constructors $\times$ and $\to$ are as in typed lambda calculus.

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

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

The functional side of the language is λ-calculus with pairs, conditionals and the recursive Y-combinator. The term $\mathsf{do}[\delta]\lambda x.\, M$ allows computations which manipulate only local state to be embedded in a δ-typed term: the command M is executed with x bound to a new variable of type $\mathsf{var}[\delta]$. 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," or changes to non-local variables other than x.

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

2.1 SCIR

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

Terms without common free identifiers can be said to be noninterfering only if *distinct* identifiers do not interfere. In our example language, the only way to bind identifiers is via new, do and function applications. new and do always create fresh storage locations, which cannot be aliased. For applications MN, **SCI** requires that M and N do not interfere. If they did interfere, then the lambda-bound identifier of the function M could be an alias to other free identifiers of M. For example, in the term

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

the function and the argument interfere (they both have the free identifier y), and x and y are aliases within the function body.

This basic definition of interference is too restrictive. It disallows terms such as $(\mathsf{plus}\ \mathsf{get}(x))(\mathsf{get}(x))$ because x occurs free in both the function and argument. But intuitively, since x is never assigned to, the subterms cannot interfere with each other.

To allow such terms, 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] In our language, the types δ are passive, and a product type $\theta_1 \times \theta_2$ is passive iff θ_1 and θ_2 are both passive types. A function type $\theta_1 \to \theta_2$ is passive iff θ_2 is passive. The argument type is inconsequential, because as long as the body has a passive type, it cannot "use" any arguments of non-passive type.

A free identifier x of a term M is a *passive free identifier* if it always occurs in some passively-typed subterm of M, *regardless* of whether or not x has a passive type. For example, in the addition term above, the type of x, viz. $\mathsf{var}[\mathsf{int}]$, is not passive. But x is a passive free identifier, because both occurrences of x are in subterms of passive type (namely $\mathsf{get}(x)$, which has type int).

We can now relax the original definition of interference, and say that two terms interfere if their shared identifiers are not *passive* free identifiers. Therefore, the addition term above is considered well-typed.

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. For instance, the function $\lambda c.\, c; c$ is considered free from side effects because it can only change state through its argument c. 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 "!" where the type $!\theta$ represents values of type θ that do not change global variables. The type of $\lambda c.\, c; c$ then, would be $!(\mathsf{comm} \to \mathsf{comm})$.[2]

[1] This intuition is formalized by the semantic models of [15, 13].

[2] Our types $!(\theta \to \theta')$ correspond to the passive function types $\theta \to_p \theta'$ in Reynolds's presentation [17]. Adding a generic "!" constructor is a useful notational convenience.

We extend the type syntax of our language with "!" and define the passive types ϕ:

$$\begin{array}{lcl} \theta & ::= & \delta \mid \mathsf{var}[\delta] \mid \mathsf{comm} \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \mid !\theta \\ \phi & ::= & \delta \mid \phi_1 \times \phi_2 \mid \theta \to \phi \mid !\theta \end{array}$$

We formalize the above intuitions in **SCIR**, whose type rules are shown in Figure 2. This system is similar to the one presented in [13], but extended with "!". Typing judgements are of the form $\Pi \mid \Gamma \vdash M\!:\theta$, where M is a term and θ is its type. The typing context is partitioned into the *passive zone* Π, which contains passive free identifiers, and the *active zone* Γ, which may contain any free identifier.

The Passification rule says that all free identifiers in a passively-typed term are passive. Activation lets us ignore the fact that x is a passive free identifier, which is necessary for abstracting x with $\to I$. Weakening permits free identifiers to be added to the context. Identifier sharing is achieved via Contraction, which allows sharing of two *passive* free identifiers.

The rule $\times I$ allows unrestricted identifier sharing between the components of the pair. For $\to E$ the zones $\Pi_1, \Pi_2, \Gamma_1, \Gamma_2$ must be disjoint to prevent interference between distinct identifiers, but passive free identifiers can occur in both the operator and the operand, as a consequence of Contraction. Any term with no active free identifiers may be explicitly promoted to a passive type using Promotion. The Dereliction rule is needed for applying the elimination rules to a promoted term.

It is worth noting that in **SCIR**, the type $A \times B \to C$ is *not* isomorphic to $A \to B \to C$. As a result, the currying transformation is not valid in **SCIR**: if N and L interfere, then the application $(M\langle N, L\rangle)$ may be typable even when $(\mathrm{curry}(M)NL)$ is not.[3]

Many Algol constructs can be defined as constants in this type system:

$$\begin{array}{lcl} :=_\delta & : & \mathsf{var}[\delta] \times \delta \to \mathsf{comm} \\ \mathsf{get}_\delta & : & \mathsf{var}[\delta] \to \delta \\ \mathsf{if}_\theta & : & \mathsf{bool} \times \theta \times \theta \to \theta \\ \mathsf{rec}_\theta & : & !(\theta \to \theta) \to \theta \\ \mathsf{do}[\delta] & : & !(\mathsf{var}[\delta] \to \mathsf{comm}) \to \delta \\ \mathsf{new}[\delta] & : & (\mathsf{var}[\delta] \to \mathsf{comm}) \to \mathsf{comm} \\ ; & : & \mathsf{comm} \times \mathsf{comm} \to \mathsf{comm} \\ \mathsf{skip} & : & \mathsf{comm} \end{array}$$

To illustrate **SCIR**, we explain how to derive a typing for the `map` procedure from Figure 1. It is

[3] The type system presented in [13] includes a *tensor product* constructor $\otimes$, such that $A \otimes B \to C$ is isomorphic to $A \to B \to C$. We do not discuss the tensor product here due to space constraints, but see [7].

$$\frac{}{\mid x{:}\,\theta \vdash x{:}\,\theta}\ \text{Axiom}$$

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

$$\frac{\Pi \mid \Gamma \vdash M{:}\,\theta}{\Pi,\, \Pi' \mid \Gamma,\, \Gamma' \vdash M{:}\,\theta}\ \text{Weakening} \qquad \frac{\Pi,\, x{:}\,\theta,\, x'{:}\,\theta \mid \Gamma \vdash M{:}\,\theta'}{\Pi,\, x{:}\,\theta \mid \Gamma \vdash M[x/x']{:}\,\theta'}\ \text{Contraction}$$

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

$$\frac{\Pi \mid \Gamma,\, x{:}\,\theta_1 \vdash M{:}\,\theta_2}{\Pi \mid \Gamma \vdash \lambda x{:}\,\theta_1.M{:}\,\theta_1 \to \theta_2}\ {\to}I \qquad \frac{\Pi_1 \mid \Gamma_1 \vdash M{:}\,\theta_1 \to \theta_2 \quad \Pi_2 \mid \Gamma_2 \vdash N{:}\,\theta_1}{\Pi_1,\, \Pi_2 \mid \Gamma_1,\, \Gamma_2 \vdash MN{:}\,\theta_2}\ {\to}E$$

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

Figure 2: Type rules for **SCIR**

easy to translate `map` into **SCIR** syntax, treating the array `next` as a function of type $\mathsf{int} \to \mathsf{var[int]}$:

$$\begin{aligned} \text{map} \equiv\quad & \mathsf{rec}(\mathsf{promote}\ (\lambda \text{map}.\ \lambda p.\ \lambda i.\\ & \quad \mathsf{if}\ (i = 0)\ \mathsf{then\ skip}\\ & \quad \mathsf{else}\ p\ i;\ \text{map}\ p\ \mathsf{get}(\text{next}(i)))) \end{aligned}$$

Even though the type of `next` is not passive, `next` is a passive free identifier:

$$\frac{i{:}\,\mathsf{int} \mid \text{next}{:}\,\mathsf{int} \to \mathsf{var[int]} \vdash \mathsf{get}(\text{next}(i)){:}\,\mathsf{int}}{i{:}\,\mathsf{int},\ \text{next}{:}\,\mathsf{int} \to \mathsf{var[int]} \mid\ \vdash \mathsf{get}(\text{next}(i)){:}\,\mathsf{int}}\ \text{Pass.}$$

The rest of the derivation is straightforward. Promotion can be applied because `next` is the only free identifier of the function, and it is passive.

3 Type Inference

In this section, we define the problem of type inference for **SCIR** with implicit coercions and let-based polymorphism. The typing rules presented here form the specification of the inference algorithm of Section 4. In Section 3.1, we present a sound and complete inference system where only $\mathsf{derelict}$ is implicit. We allow $\mathsf{promote}$ to be made implicit in Section 3.2 and then discuss polymorphism in Section 3.3.

3.1 Implicit Dereliction

The input to the type inference algorithm will be a preterm in which the type declarations of lambda-bound identifiers and $\mathsf{derelict}$ operations are omitted. The algorithm succeeds if there is some way to fill in the missing information to obtain a well-typed **SCIR** term. It will produce a *principal typing* which represents all possible ways that the information can be filled in.

The context-free grammar for preterms is as follows:

$$e \quad ::= \quad c \mid x \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{promote}\ e$$

For clarity, we omit product types from our discussion, since they are not immediately relevant to the issue of implicit promotion and dereliction. Adding products is straightforward, as shown in [7].

We must extend the type syntax presented in the last section with type variables. In addition, we restrict our attention to *standard types*, where the "!" constructor may be applied at most once to any type term. There is no loss of expressiveness, since for any passive type ϕ, we can show that $!\phi \ \cong\ \phi$ [15, 13]. Then, since $!\theta$ is a passive type, $!\theta \ \cong\ !!\theta$. The syntax of standard types is:[4]

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

The types ρ range over types without an outermost "!" constructor, and type variables α range over ρ types. 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 $!^0\rho$

[4] Many types expressible in this syntax are not practically useful, such as $!^1\mathsf{var}[\delta]$. In practice, we use a more restricted syntax that is described in Section 4.1. These restrictions do not affect the type inference issues in any essential manner.

$$
\begin{array}{lcl}
p(!^n\rho) & = & (n = 1 \vee p(\rho)) \\
p(\delta) & = & \text{true} \\
p(\mathsf{comm}) & = & \text{false} \\
p(\mathsf{var}[\delta]) & = & \text{false} \\
p(\theta_1 \to \theta_2) & = & p(\theta_2)
\end{array}
$$

Figure 3: The definition of p

denotes ρ. Annotation variables i stand for the 0 and 1 annotations of "!".

The subset of passive types ϕ is defined via

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

We introduce a predicate $p(\theta)$ that is true iff θ is passive. The definition of p (other than for type variables) is given in Figure 3. To determine the passivity of types which contain type and annotation variables, we require *kind assignment* that maps type variables to the kinds "passive" and "general," and maps annotation variables to 0 or 1. Then $p(\alpha)$ holds under a kind assignment K iff K maps α to the kind "passive," and $p(!^i\rho)$ holds iff K maps i to 1 or $p(\rho)$ holds.

However, an **SCIR** term can typecheck under *different* kind assignments. Consider the preterm $\mathsf{promote}\ (f\,x)$, where the standard types for f and x should be of the form $!^i(!^j\alpha \to !^k\beta)$ and $!^l\alpha$. For the term to typecheck, both free identifiers f and x must be used passively, so *either* the types of f and x are both passive (so $!^i(!^j\alpha \to !^k\beta)$ and $!^l\alpha$ are passive), *or* $(f\,x)$ has a passive type (i.e., $!^k\beta$ is passive).

Our system 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:

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

A constraint is *satisfiable* if there is some kind assignment under which it can be simplified to true. Two constraints C_1 and C_2 are equal 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, the kind constraint under which $\mathsf{promote}\ (f\,x)$ typechecks in **SCIR** is $(p(!^i(!^j\alpha \to !^k\beta)) \wedge p(!^l\alpha)) \vee p(!^k\beta)$, which can be simplified to $(i = 1 \wedge p(!^l\alpha)) \vee p(!^k\beta)$.

Note that if we typecheck a term from the bottom up, we cannot tell that this constraint is necessary until the $\mathsf{promote}$ operator is discovered. This suggests that we must associate kind constraints with free identifiers of a term as well as the term itself.

To present the type inference issues abstractly, we devise another system of rules. The judgements in our new system, called *inference judgements*, are of the form

$$\{x_i\!:\theta_i\ [P_i; C_i]\}_i \vdash e\!:\theta\ [G]$$

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.

The constraint C_i indicates the condition under which all occurrences of the free identifier x_i can be contracted. (Recall that in a well-typed term $(M\,N)$, all free identifiers common to M and N should be passive, so that Contraction can be applied in **SCIR**.) 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 $\mathcal{A} \vdash e\!:\theta\ [G]$ can be read as "The term e has type θ under the typing context $\mathcal{A}$, as long as the G-constraint and all C-constraints in $\mathcal{A}$ hold. Further, all free identifiers x whose P-constraints in $\mathcal{A}$ hold are passive free identifiers." Formally, the semantics of inference judgements is defined as follows:

Definition 1

1. A judgement $\Pi \mid \Gamma \vdash e\!:\theta$ is *implicitly derivable* if there is a derivable **SCIR** term $\Pi \mid \Gamma \vdash M\!:\theta$ such that e is obtained by erasing type declarations and $\mathsf{derelict}$ from M.

2. A judgement $\Pi \mid \Gamma \vdash e\!:\theta'_0$ is an *instance* of an inference judgement $\{x_i\!:\theta_i\ [P_i; C_i]\}_i \vdash e\!:\theta'\ [G]$ iff there is a substitution σ for the type variables in the latter such that

   (a) the judgement $\sigma(\{x_i\!:\theta_i\}_i \cup \{z_j\!:\theta_j\}_j \vdash e\!:\theta')$ is the same as $\Pi, \Gamma \vdash e\!:\theta'_0$ for some $\{z_j\!:\theta_j\}_j$ (which represent weakening),

   (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 (instances with no type variables) are implicitly derivable in **SCIR**.

A type system for deriving valid inference judgements is shown in Figure 4. We use the following notations for typing contexts when $\mathcal{A} = \{x_i\colon \theta_i\ [P_i; C_i]\}_i$ and $\mathcal{A}' = \{x_i\colon \theta_i\ [P_i'; C_i']\}_i$:

$$\begin{array}{rcl} \mathcal{A} \wedge \mathcal{A}' & = & \{x_i\colon \theta_i\ [P_i \wedge P_i'; P_i \wedge P_i']\}_i \\ \mathcal{A} \vee p(\theta) & = & \{x_i\colon \theta_i\ [P_i \vee p(\theta); C_i \vee p(\theta)]\}_i \\ \mathcal{A}[\text{true}] & = & \{x_i\colon \theta_i\ [\text{true}; \text{true}]\}_i \\ \mathcal{A}|_{\text{TA}} & = & \{x_i\colon \theta_i\}_i \\ P(\mathcal{A}) & = & \bigwedge_i P_i \end{array}$$

The Axiom rule says that the term x has type $!^m\rho$ in a context which assigns type $!^n\rho$ to the identifier x, provided $m \leq n$ (which represents the possibility of implicitly derelicting the term). Also, the P-constraint of x indicates that x is a passive free identifier if $!^n\rho$ is a passive type. The C-constraint is trivially true because x only appears once in the term and does not need to be contracted.

In an **SCIR** derivation for an abstraction term, any necessary contractions for x must have occurred before the binding of x in $\rightarrow I$. Thus, when x becomes bound in the $\rightarrow I$ rule of the implicit system, its C-constraint cannot be further weakened and must be added to the G-constraint. The second rule $\rightarrow I'$ handles the case where x does not occur in e.

We need not consider the possibility of dereliction in the rules $\rightarrow I$ and $\rightarrow I'$, because of the following property:

Lemma 2 In any **SCIR** derivation, dereliction steps must occur after an Axiom, Promotion, $\times E_i$ or $\rightarrow E$ step, followed by zero or more structural steps (Weakening, Contraction, Passification, or Activation).

The most complex rule is $\rightarrow E$. $\mathcal{A}$ and $\mathcal{A}'$ should contain the free identifiers that are common to e_1 and e_2, and they should assign the same types to those identifiers. The P and C-constraints of the identifiers may differ, however. $\mathcal{B}_1$ and $\mathcal{B}_2$ contain identifiers which appear only in e_1 or e_2, respectively.

If the function e_1 has a passive call type, then $(e_1\, e_2)$ also has a passive type (before considering implicit dereliction), and the free identifiers are all passive. Thus, all P and C-constraints in the typing context are weakened with the disjunct $p(!^n\rho)$. Further, all free identifiers shared between e_1 and e_2 *must* be passive so they can be contracted. Hence the C-constraint of a shared identifier should be the same as its P-constraint. Finally, the term may be implicitly derelicted, so its type is $!^m\rho$, with $m \leq n$.

The Promotion rule ensures that all free identifiers of e are passive by adding all P-constraints in the context to the G-constraint. Then, since $G \wedge P(\mathcal{A})$ must hold for $\mathsf{promote}\ e$ to be well-typed, the P-constraints and C-constraints of all free identifiers must be true. There are no constraints on the annotation variable m due to the possibility of an implicit dereliction.

Examples of derivable and valid inference judgements are:

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

$$\begin{array}{l} f\colon !^i(!^j\alpha \rightarrow !^k\beta)\ [\text{true}; \text{true}], \\ x\colon !^l\alpha\ [\text{true}; \text{true}] \\ \quad \vdash \mathsf{promote}\ (f\ x)\colon !^n\beta \\ \qquad [((i = 1 \wedge p(!^l\alpha)) \vee p(!^k\beta)) \wedge (j \leq l)] \end{array}$$

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

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

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

Theorem 4 (Completeness)
Every judgement implicitly derivable in **SCIR** is an instance of a derivable inference judgement.

3.2 Implicit Promotion

In this section, we consider the issues of implicit promotion. It is possible to make the promote operator implicit simply by combining the Promotion rule of Figure 4 with each of the other four rules, much as we merged Dereliction with the other **SCIR** rules. However, we believe that promote should not be left completely implicit in the language for two reasons.

First, promote serves as a valuable program annotation. It documents the important fact that its argument is free from side effects, and can thus aid programmers in reasoning about interference. (On the other hand, the derelict operator does not provide any useful information about its argument.)

Secondly, implicit promotion may lead to excessively large constraints in typings. As can be seen in Figure 4, the global constraint of a term $\mathsf{promote}\ e$ must "ensure" that all free identifiers of e are passive. If implicit promotion is possible at every step of a derivation, then the size of the global constraint may grow very quickly.

A more practical solution is to allow promote to be left implicit only in conjunction with the constants

$$\frac{}{x{:}\,!^n\rho\ [p(!^n\rho);\mathrm{true}] \vdash x{:}\,!^m\rho\ [m \le n]}\ \text{Axiom}$$

$$\frac{\mathcal{A},\, x{:}\,\theta_1\ [P;C] \vdash e{:}\,\theta_2\ [G]}{\mathcal{A} \vdash \lambda x.e{:}\,!^0(\theta_1 \to \theta_2)\ [G \wedge C]} \to I \qquad \frac{\mathcal{A} \vdash e{:}\,\theta_2\ [G]}{\mathcal{A} \vdash \lambda x.e{:}\,!^0(\theta_1 \to \theta_2)\ [G]} \to I' \quad (x \text{ not in } \mathcal{A})$$

$$\frac{\mathcal{A},\, \mathcal{B}_1 \vdash e_1{:}\,!^0(\theta' \to !^n\rho)\ [G_1] \quad \mathcal{A}',\, \mathcal{B}_2 \vdash e_2{:}\,\theta'\ [G_2]}{(\mathcal{A} \wedge \mathcal{A}') \vee p(!^n\rho),\, \mathcal{B}_1 \vee p(!^n\rho),\, \mathcal{B}_2 \vee p(!^n\rho) \vdash e_1\, e_2{:}\,!^m\rho\ [G_1 \wedge G_2 \wedge m \le n]} \to E$$
$$(\mathcal{A}|_{\mathrm{TA}} = \mathcal{A}'|_{\mathrm{TA}}, \mathrm{dom}(\mathcal{A}) \cap \mathrm{dom}(\mathcal{B}_1) \cap \mathrm{dom}(\mathcal{B}_2) = \emptyset)$$

$$\frac{\mathcal{A} \vdash e{:}\,!^0\rho\ [G]}{\mathcal{A}[\mathrm{true}] \vdash \mathsf{promote}\ e{:}\,!^m\rho\ [G \wedge P(\mathcal{A})]}\ \text{Promotion}$$

Figure 4: System of rules for type inference

$$\frac{\mathcal{A} \vdash e{:}\,!^n(!^k\rho \to !^k\rho)\ [G]}{\mathcal{A}[\mathrm{true}] \vdash \mathsf{rec}\ e{:}\,!^m\rho\ [G \wedge m \le k \wedge (n = 1 \vee P(\mathcal{A}))]}\ \mathsf{rec}$$

$$\frac{\mathcal{A} \vdash e{:}\,!^n(!^0\mathsf{var}[\delta] \to !^0\mathsf{comm})\ [G]}{\mathcal{A}[\mathrm{true}] \vdash \mathsf{do}[\delta]\ e{:}\,!^0\delta\ [G \wedge (n = 1 \vee P(\mathcal{A}))]}\ \mathsf{do}[\delta]$$

Figure 5: Additional type rules for implicit promotion

rec and do. These are two of the most commonly used operators in programs, but since their arguments *must* have promoted function types, explicit promotion is redundant. Permitting one to write $\mathsf{do}[\delta]\, e$ instead of $\mathsf{do}[\delta](\mathsf{promote}\ e)$ and $\mathsf{rec}\ e$ instead of $\mathsf{rec}(\mathsf{promote}\ e)$ eliminates many promote operators in typical programs.

We extend the type system of Figure 4 with the two additional rules shown in Figure 5. These rules combine Promotion with rules for applying rec and $\mathsf{do}[\delta]$. The conjunct $(n = 1 \vee P(\mathcal{A}))$ of the G-constraint is due to implicit promotion: if e does not already have a promoted function type (i.e., if $n = 0$), then e is promoted implicitly by constraining its free identifiers to all be passive.

By changing the definition of *implicitly derivable* from Definition 1 to account for the erasure of promote from the terms $\mathsf{do}[\delta](\mathsf{promote}\ M)$ and $\mathsf{rec}(\mathsf{promote}\ M)$, the soundness and completeness theorems in Section 3.1 can be carried over to this extended system.

Theorem 5 (Soundness and Completeness) All derivable inference judgements are valid and every judgement implicitly derivable in **SCIR** is an instance of a derivable inference judgement.

An inference judgement for a term e has instances only if the G and C-constraints are satisfiable. Otherwise, there does not exist any **SCIR** term corresponding to e, and e should be considered untypable. The satisfiability of kind constraints, and hence typability, can be decided in polynomial time for our system:

Theorem 6 (Satisfiability) Consider constraints generated by the type system with implicit dereliction and implicit promotions for $\mathsf{do}[\delta]$ and rec. There exists an algorithm which checks the satisfiability of the constraint, in time polynomial in the size of the constraint.

As an example of the new rec rule, consider the last step in the derivation for the term $\mathsf{rec}(f)$:

$$\frac{\begin{array}{l} f{:}\,!^n(!^l\alpha \to !^l\alpha)\ [n = 1 \vee p(!^l\alpha);\mathrm{true}] \\ \quad \vdash f{:}\,!^m(!^l\alpha \to !^l\alpha)\ [m \le n] \end{array}}{\begin{array}{l} f{:}\,!^n(!^l\alpha \to !^l\alpha)\ [\mathrm{true};\mathrm{true}] \\ \quad \vdash \mathsf{rec}(f){:}\,!^k\alpha\ [m \le n \wedge k \le l \wedge \\ \qquad (m = 1 \vee (n = 1 \vee p(!^l\alpha)))] \end{array}}$$

3.3 Polymorphism

In this section, we discuss the addition of Hindley-Milner style polymorphism to **SCIR**. Languages which support such polymorphism, like ML, provide a let construct of the form $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. Type inference for let is usually handled with either quantified types or substitution. Unfortunately, neither approach is directly applicable to **SCIR**.

In Damas and Milner's inference algorith [3], e_2 is typechecked under a context that assigns a quantified type $\forall\alpha.\,\theta$ to x. But in **SCIR**, the type variable to be quantified over might occur in the constraints. This implies that constraints should be incorporated into type schemes, as in $\forall\alpha.\, p(\alpha) \vee p(\beta) \Rightarrow \theta$. The situation

$$\frac{\mathcal{A} \vdash (\lambda x'.\, e_2')\,\langle e_1,\, \ldots,\, e_1\rangle\colon \theta\ [G]}{\mathcal{A} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\colon \theta\ [G]}\ \mathsf{let}_1\ (x \text{ is free in } e_2)$$

(e_2' is e with each occurrence i of x replaced by $\pi_i x$)

$$\frac{\mathcal{A} \vdash (\lambda x.\, e_2)\ e_1\colon \theta\ [G]}{\mathcal{A} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\colon \theta\ [G]}\ \mathsf{let}_2\ (x \text{ not free in } e_2)$$

Figure 6: Additional type rules for let-based polymorphism

is complicated by the fact that the constraint may refer to both bound (α) and free (β) type variables. More study is needed to understand how this method can be applied to **SCIR**.

The substitution-based approach to type inference [12] treats the let term as $e_2[e_1/x]$ instead. But this would not be sound in **SCIR**. From the desugared form of the let term, $(\lambda x.\, e_2)e_1$, it is clear that $\lambda x.\, e_2$ should not interfere with e_1. However, this condition cannot be enforced by typechecking $e_2[e_1/x]$, so $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ cannot be considered equivalent to $e_2[e_1/x]$.

We suggest a third approach as a tentative solution. The main idea is to desugar a let term into an application to ensure non-interference between e_1 and e_2. But the traditional desugared form $(\lambda x.\, e_2)e_1$ does not permit polymorphic uses of x within e_2. Instead, replace all n occurrences of x in e_2 by terms $\pi_1 x', \pi_2 x', \ldots, \pi_n x'$, and call the resulting term e_2'. Then, typechecking $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is equivalent to typechecking

$$(\lambda x'.\, e_2')\langle e_1, e_1, \ldots, e_1\rangle$$

Each distinct "copy" of e_1, and hence each term $\pi_i x'$, can be assigned a different typing, effectively simulating polymorphism. Furthermore, by using the cross product type to form n-tuples[5] the n "copies" of e_1 are not considered to interfere with one another.

More precisely, we add the type rules shown in Figure 6 to the rules of Figures 4 and 5. The rule let_2 is needed to typecheck e_1 and to control interference between e_1 and e_2 when x is not free in e_2.

For example, $\mathsf{let\ id} = \lambda x.\, x\ \mathsf{in}\ \langle \mathrm{id}\ 3,\ \mathrm{id}\ (y{:=}3)\rangle$ can be typechecked by deriving the following judgement:[6]

$$\begin{array}{l} y\colon !^i\mathsf{var}[\mathsf{int}]\ [i = 1; \mathrm{true}] \\ \quad \vdash (\lambda x'.\, \langle(\pi_1\, x')\ 3,\ (\pi_2\, x')\ (y{:=}3)\rangle)\langle \lambda x.\, x,\ \lambda x.\, x\rangle \\ \qquad : (!^j\mathsf{int}) \times \mathsf{comm}\ [\mathrm{true}] \end{array}$$

[5]As mentioned in Section 3.1, the inference system can readily be extended with product types.

[6]We will write ρ instead $!^0\rho$ to simplify the presentation.

4 Inference Algorithm

The type rules described in Section 3 essentially form a "logic program" for type inference. They are syntax-directed and are closed under type substitution. Much of the type inference algorithm is derived directly from these rules. However, naively implementing the rule let_1 would be very inefficient, because of the need to perform term substitution on e_2 and to typecheck e_1 multiple times in the tuple term. Instead, we adopt the type inference method of [12, 11.3.3].

Our type inference algorithm T takes two arguments: a preterm e and an *environment* mapping identifiers to typings. The algorithm returns a pair consisting of a typing judgement for e and another environment.

To typecheck $\mathsf{let}\ x = e_1 \mathsf{in}\ e_2$, we first typecheck e_1 just once, and then typecheck e_2 in an environment that associates x with the typing of e_1. So, the environment argument of T stores the principal typing of e_1, and each occurrence of x in e_2 will receive an instantiation of that typing, with all type and annotation variables fresh. This eliminates the need to repeatedly typecheck e_1, as well as the need to substitute x with $\pi_i x$ in e_2.

To simulate the effect of let_1, though, we still need to infer a typing and constraints for $\langle e_1, \ldots, e_1\rangle$, without actually checking e_1. The environment returned by T is used for this purpose. If $T(e, E) = \langle \mathcal{A} \vdash e\colon \theta\ [G],\ I\rangle$ for some environment E and preterm e, then I is an environment that associates the typing of m-tuple $\langle e_1, \ldots, e_1\rangle$ with the let-bound identifier x, where m is the number of occurrences of x in e.

An environment mapping x to a fresh copy of e_1's typing is returned (as the second argument) from an invocation $T(x, E)$. When two terms which share a let-bound identifier x are combined, as for the $\to E$ rule, the instantiations of x are also combined, according to the cross product rules of [7].

Thus, when the let body e_2 is typechecked in $T(e_2, E) = \langle \mathcal{A} \vdash e_2\colon \theta\ [G],\ I\rangle$, the environment I will associate x with a typing that represents the typing of the n-tuple. Then, application of the $\to E$ rule yields the typing for let.

The inference algorithm T is sound and complete with respect to the type rules of Section 3: if T succeeds for a preterm e, then the typing returned by T is derivable in the inference system, and is a principal typing of e. Further, if a preterm e is typable in the inference system, then the algorithm succeeds and returns a principal typing for e. The reader is referred to [23] for further details, including proofs of correctness.

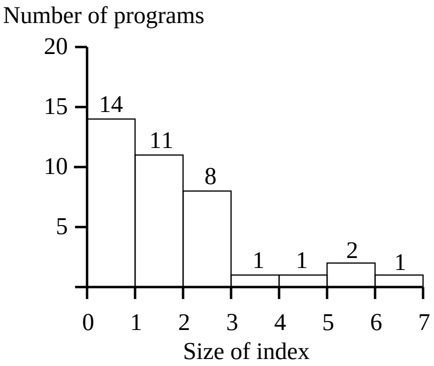

Table 1: Distribution of program indices

4.1 Practical Considerations

Implicit promotion and dereliction may cause the types in our system to become quite complicated. In this section we discuss preliminary experience with our prototype of a **SCIR** typechecker, and suggest some techniques to reduce the size of the constraints.

We implemented a typechecker for **SCIR** based on the inference algorithm described in this section, and tested it with the 38 programs from [1, Appendix B]. As a rough measure of the complexity of the types inferred, we examined the index

$$\frac{\text{number of atomic formulas in constraints}}{\text{length of type}}$$

This index was in the range 0–7 for all examples, with most of the indices falling below 3. So, we feel the complexity of types inferred by our system is reasonable. Table 1 shows a breakdown of the number of programs which fall within each index range.

In general, terms with polymorphic types have larger constraints, while constraints are significantly shorter when some ground types can be inferred. Longer terms also tend to yield longer constraints, due to more possibilities of dereliction as well as more opportunities for passification of free identifiers. We also found that our implicit promotions do not usually add to the size of the constraints, since most Algol-style programs do not access free identifiers from within recursive procedures or blocks.

We introduce some simplifications which can help to further reduce the size and complexity of types and constraints. The simplifications are suggested by the the following isomorphisms [15, 13]:

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

Thus, types of the form $!\phi$ and $!(\theta_1 \times \theta_2)$ are redundant and can be eliminated (by setting their annotation variables to 0). Also, types of the form $!\mathsf{var}[\delta]$ and $!\mathsf{comm}$ serve no purpose because there are no values of such types, other than undefined values. We can eliminate these types as well.

The only useful ! constructors are those applied to $\rightarrow$ types and type variables. We use this fact to formulate the following abbreviated notation:

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

With these simplifications and notations, we can show the following typings:

$$\begin{array}{l} f\colon \mathsf{int} \times \mathsf{comm} \rightarrow^i \mathsf{int} \times \mathsf{comm}\ [\mathrm{true}; \mathrm{true}], \\ y\colon \mathsf{var}[\mathsf{int}]\ [\mathrm{true}; \mathrm{true}] \\ \quad \vdash \mathsf{let}\ \mathrm{twice} = \lambda f.\, \lambda x.\, f\,(f\,x) \\ \qquad \mathsf{in}\ \pi_1(\mathrm{twice}\, f\, \langle 3,\, y := 1\rangle) \quad : \mathsf{int}\ [i = 1] \end{array}$$

$$\begin{array}{l} \mathrm{next}\colon \mathsf{int} \rightarrow \mathsf{var}[\mathsf{int}]\ [\mathrm{false}; \mathrm{true}], \\ \mathrm{free}\colon \mathsf{var}[\mathsf{int}]\ [\mathrm{false}; \mathrm{true}] \\ \quad \vdash \texttt{reclaim}\colon \mathsf{int} \rightarrow \mathsf{comm}\ [\mathrm{true}] \end{array}$$

$$\begin{array}{l} \mathrm{next}\colon \mathsf{int} \rightarrow \mathsf{var}[\mathsf{int}]\ [\mathrm{false}; \mathrm{false}], \\ \mathrm{free}\colon \mathsf{var}[\mathsf{int}]\ [\mathrm{false}; \mathrm{true}] \\ \quad \vdash (\texttt{map reclaim})\colon \mathsf{int} \rightarrow \mathsf{comm}\ [\mathrm{true}] \end{array}$$

The first term illustrates cross products and polymorphic terms. The G-constraint $i = 1$ represents the only possible way that the type of f can be passive, and the only way that f can be a passive free identifier.

The next term is the `reclaim` function of Figure 1. The free identifiers `next` and `free` cannot be passified. The last example is the application (`map reclaim`) from the Introduction. The C-constraint of `next` is false, so `next` is not a passive free identifier, and the term is not typable.

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 left implicit, and introduced a technique for adding let-based polymorphism to the language. The complexity of typings produced by our system remains within practical limits.

Possible areas for future work include investigating how type quantification may be introduced into the system. It would also be interesting to study extensions of **SCI** to include references, as in **ILC** [19, 20].

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

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, London, 1988.

[2] 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.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *ACM Symp. on Princ. of Program. Lang.*, pages 207–212, 1982.

[4] M. Dawson and P. Taylor, editors. *Hypatia Electronic Library.* Queen Mary and Westfield College, URL //hypatia.dcs.qmw.ac.uk.

[5] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Object Oriented Prog. Syst., Lang. and Applications*, pages 169–184. ACM, 1995.

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

[7] 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.

[8] M. P. Jones. *Qualified types: Theory and Practice.* Cambridge University Press, Cambridge, England, 1994.

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

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

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

[12] J. C. Mitchell. *Foundations of Programming Languages.* MIT Press, 1997.

[13] 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 Semanatics: Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theor. Comput. Sci.* Elsevier, 1995. (Reprinted as Chapter 18 of [14]).

[14] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes).* Birkhäuser, Boston, 1997.

[15] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation*, 9:7–76, 1996. (Reprinted as Chapter 19 of [14]).

[16] U. S. Reddy. Objects and classes in Algol-like languages. In *Fifth Intern. Workshop on Foundations of Object-oriented Languages.* ACM, Jan 1998.

[17] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978. (Reprinted as Chapter 10 of [14]).

[18] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981. (Reprinted as Chapter 3 of [14]).

[19] 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.

[20] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In *Algol-like Languages* [14], chapter 9, pages 235–272.

[21] 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).

[22] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *ACM Symp. on Princ. of Program. Lang.* ACM, January 1989.

[23] H. Yang and H. Huang. Type reconstruction for syntactic control of interference, part 2. Technical report, University of Illinois. To appear.

[24] H. Yang and U. S. Reddy. Imperative lambda calculus revisited. Electronic manuscript, [4], Aug 1997.