Type Reconstruction for SCI *

Howard Huang Uday Reddy
Department of Computer Science
The University of Illinois
{hhuang,reddy}@cs.uiuc.edu

January 23, 1996

Abstract

We present a type reconstruction algorithm for SCIR [10], a type system for a language with syntactic control of interference. SCIR guarantees that terms of passive type do not cause any side effects, and that distinct identifiers do not interfere. A reconstruction algorithm for this type system must deal with different *kinds* (passive and general) and different *uses* of identifiers (passive and active). In particular, there may not be a unique choice of kinds for type variables. Our work extends SCIR typings with *kind constraints*. We show that principal type schemes exist for this extended system and outline an algorithm for computing them.

1 Introduction

Researchers interested in functional programming languages have recently turned their attention to extending these languages with suitably-controlled state manipulation facilities [20, 9, 1, 12, 11]. This effort has brought into focus the pioneering work of Reynolds [16, 17, 18] in the late 70's and early 80's, devoted to the analysis and refinement of imperative programming languages using "functional programming principles" (lambda calculus and its equivalences). In retrospect, Reynolds's work may be seen to have addressed two significant issues. In designing a higher-order programming language with imperative features, (i) how does one retain the functional reasoning principles (such as the equivalences of lambda calculus), and (ii) how does one retain the imperative reasoning principles (such as Hoare logic)? Part of the answer to (i) was already contained in the design of Algol 60 with its call-by-name parameter mechanism, and Reynolds's work brings this to the forefront [17]. This part of Reynolds's analysis is adopted *ipso facto* in the recent work in functional programming. The other part of the answer to (i) as well his answer to (ii) are contained in Reynolds's "Syntactic Control of Interference" or SCI [16]. Unfortunately, these ideas of Reynolds have had little impact on the afore-mentioned work in functional languages, though they are clearly applicable. We explain the two aspects of SCI in turn.

How does one retain functional reasoning principles? In defining a function procedure that returns, say an integer, one often wants to use an algorithm

*This work was funded by the National Science Foundation under grant number NSF-CCR-93-03043.

that creates local variables and manipulates them internally.[1] To preserve the standard reasoning principles of integers, the procedure should only manipulate local variables without making any changes to the global variables. But, most programming languages have no checks to ensure this. Consequently, terms of type integer may well involve global state changes (often called "side effects"), thwarting standard reasoning about integers. Reynolds's proposal in SCI contains a comprehensive treatment of this issue via a classification of types into *active* and *passive* types. Computations of passive types (like integer) are guaranteed not to have any global side effects so that standard reasoning is applicable.

How does one retain imperative reasoning principles? The usual reasoning principles of imperative programs tend to assume that distinct identifiers do not *interfere*, i.e., using one identifier in a computation does not affect the meaning of the other identifier. Aliasing is a common example of interference where two identifiers happen to denote the same (mutable) variable. Aliasing is explicitly ruled out in the usual formulations of Hoare logic [5, 13]. However, other kinds of interference arise in languages with (higher-order) procedures. For instance, if a procedure p modifies a global variable x, calling p interferes with x. Again, Reynolds gives a comprehensive set of rules to guarantee that distinct identifiers do not interfere.

Unfortunately, Reynolds notes that there is a problem with his rules in that a legal term can be reduced to an illegal term via the standard reduction rules of lambda calculus. ("Subject reduction" fails.) Some of the type systems proposed for functional languages with imperative features [20, 1] have this problem as well. The problem remained unresolved for over a decade until, recently, O'Hearn *et al.* [10] proposed a revised type system in "Syntactic Control of Interference Revisited" (SCIR). Their proposal uses modern proof-theoretic techniques inherited from linear logic and logic of unity [2, 3], and possesses the subject reduction property. We should note that Reynolds himself presented a solution to this problem [19], but it goes much beyond the original proposal by involving conjunctive types. We do not pursue the new Reynolds system in this paper as SCIR is a simpler system applicable to a wide range of programming languages and type systems.

Modern functional programming languages, pioneered by Milner's work on ML [8], possess *type reconstruction* systems, where the programmer is allowed to omit type declarations and the compiler fills in this information in the most general fashion. The adoption of SCI type regimen to these languages necessitates a type reconstruction system. In this paper, we present a type reconstruction algorithm for SCIR.

The SCIR type system is unorthodox in that it involves two separate zones ("passive" and "active" zones) in the typing judgements. This corresponds to the fact that the free identifiers of a term are classified into two classes. The movement of the identifiers between zones depends on the types of the subterms where the identifiers occur, which in turn depends upon certain zones being empty. Thus, it is by no means obvious that type reconstruction is possible for SCIR.

Moreover, the type reconstruction algorithm has to account for the fact

[1] Throughout this paper, we use the term "variable" for mutable variables. Variables in the sense of lambda calculus are called "identifiers."

that there are two *kinds* of types (general types and passive types) in the type system. Correspondingly, type schemes must keep track of the kinds of type variables. The choice between the kinds of type variables is not always unique. For instance, the term $\lambda f.\, \lambda x.\, fxx$ has the "principal" type scheme $(\alpha \to \alpha \to \beta) \to \alpha \to \beta$ but is legal only if either α or β is passive. To get around this difficulty, we associate *kind constraints* with type schemes, which are boolean constraints specifying possible kinds of type variables. For example, the type of $\lambda f.\, \lambda x.\, fxx$ may be expressed as:

$$\textbf{passive}\,\alpha \vee \textbf{passive}\,\beta \Rightarrow (\alpha \to \alpha \to \beta) \to \alpha \to \beta$$

With this adaptation, we show that principal type schemes exist for SCIR and give an algorithm to compute them.

1.1 Related Work

While a large body of work exists in type reconstruction algorithms, we do not know of any systems where kind constraints are involved.

Some work similar to ours is that on type reconstruction for linear logic-based type systems [22, 7] and the related system of single-threaded lambda calculus [4]. Constraints like ours occur in these systems, though they are not concerned with kinds. A further difference is that there is no type rule similar to *Passification* in these systems, but this rule is central in SCIR.

Our research also resembles the work on effect systems and their reconstruction algorithms [6, 21], but it is hard to make a detailed comparison because those systems are designed for call-by-value languages whereas SCI is designed for lambda calculus-based call-by-name languages.

2 Issues in SCI

As mentioned in Introduction, SCI is concerned with two issues:

1. ensuring that passive-typed computations have no side effects, and
2. ensuring that distinct identifiers do not interfere.

In this section, we motivate these issues and discuss the challenges they pose for designing a formal type system.

As mentioned in Introduction, we would like to permit integer-typed expressions which allocate and manipulate local state. Consider an operator of the form **do**[**int**] $x.\, C$ where x is a bound identifier of type **var**[**int**] and C is a command term. Its semantics is to allocate a local variable bound to x, execute C in its context and, finally, return the content of x as the value of the expression. Such an expression form arises, for example, in the analysis of "function procedures" of Algol 60. Similar expression forms have been proposed for functional languages: the *Obs-elim* rule of Imperative Lambda Calculus [20], the **pure** operator of λ_{var} [9] and the `runST` operator of Glasgow Haskell [11]. To ensure that such an expression is well-behaved, one must verify that the embedded command C has no global effects other than to the variable x. A naive design is to classify types into *passive types* (like **int**) and *active types* (like **comm**

and **var**[**int**]) and then insist that all free identifiers of C other than x are of passive types. Rules similar to this are used in Imperative Lambda Calculus and the type system for λ_{var} [20, 1]. Unfortunately, this naive design runs into the problem of subject reduction.

Consider the terms:

$$\begin{array}{lll} (M_1) & \lambda x{:}\,\mathbf{int}.\,\mathbf{do}[\mathbf{int}]\ r.\ r := x & : \quad \mathbf{int} \to \mathbf{int} \\ (N_1) & \pi_1\ \langle !v,\ v := !v + 1\rangle & : \quad \mathbf{int} \end{array}$$

where π_1 is the first projection and ! is a dereferencing operator for variables. Both terms are legal. (The body of **do** has no free identifiers of active types other than r.) Hence, the application $(M_1 N_1)$ should be a legal term. However, beta-reduction of $(M_1 N_1)$ yields:

$$(P_1) \qquad \mathbf{do}[\mathbf{int}]\ r.\ r := \pi_1\langle !v,\ v := !v + 1\rangle$$

where the body of **do** contains the free identifier v of active type **var**[**int**]. This term is illegal with the naive rule for **do** and subject reduction fails.

Intuitively, the term P_1 may be considered legal because under a call-by-name semantics the assignment $v := !v+1$ will never be executed, and no global state changes take place during the evaluation of P_1.

To avoid the subject reduction problem, Reynolds classifies *occurrences* of free identifiers as active or passive. A free occurrence of x in a term is considered *passive* if it is in a passive-typed subterm; otherwise, the occurrence is *active*. The term $\mathbf{do}[\mathbf{int}]\,x.C$ is legal if all free identifier occurrences in C other than x are passive occurrences. Since all occurrences of v in P_1 are in the subterm $\pi_1\langle !v,\ v := !v + 1\rangle$ which is of passive type **int**, P_1 is legal.

To ensure that distinct identifiers do not interfere, SCI requires that in any application term (MN), the function term M and the argument term N do not interfere. This means, in essence, that the respective computations of M and N can proceed concurrently and the results are still determinate. One can make this fact explicit by adding a non-interfering parallel composition operator

$$\_ \parallel \_ : \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}$$

so that $C_1 \parallel C_2$ is a command that runs C_1 and C_2 in parallel.

How should we decide if M and N are non-interfering? Since we have already decided to focus on occurrences of free identifiers, we might insist that all common free identifiers of M and N should have only passive occurrences in these terms. If x is used actively in M, then x may not appear in N (even passively). Yet, this design runs into the subject reduction problem. Consider the terms:

$$\begin{array}{ll} (M_2) & \lambda c{:}\,\mathbf{comm}.\ r := \pi_1\langle !v,\ v := 0 \parallel c\rangle \\ (N_2) & w := \pi_1\langle !v,\ v := 1\rangle \end{array}$$

The only common free identifier is v and all its occurrences are passive. Hence, the application $(M_2 N_2)$ is legal. However, beta-reduction yields:

$$(P_2) \qquad r := \pi_1\ \langle !v,\ v := 0 \parallel w := \pi_1\langle !v,\ v := 1\rangle\rangle$$

Here, the second component $v := 0 \parallel w := \pi_1\langle !v,\ v := 1\rangle$ is not legal in Reynolds's system because v has an active occurrence in the subterm $v := 0$.

Again, intuitively, this term should be legal because the assignment $v := 0$ and the inner assignment $v := 1$ will never be executed.

This problem has proved sufficiently intricate that it remained unsolved for over a decade. The solution due to O'Hearn *et al.* [10] is obtained by using an explicit contraction rule (together with a clear separation of passive and active free identifiers). The term

$$(P_2') \qquad r := \pi_1 \langle !v,\ v := 0 \parallel w := \pi_1 \langle !v',\ v' := 1 \rangle \rangle$$

is clearly legal. Since all occurrences of v and v' in P_2' are are passive occurrences, one can use a Contraction step to rename v' to v and obtain the term P_2.

This example shows how a legal term may have seemingly illegal subterms. The legality of the subterm $v := 0 \parallel w := \pi_1 \langle !v,\ v := 1 \rangle$ cannot be decided by looking at the subterm alone. One must appeal to the fact that the subterm occurs in an overall context of a passively-typed term.

The type system is sufficiently intricate that its soundness is by no means obvious. One must demonstrate soundness by exhibiting an adequate semantic model that has a *reflective subcategory* of passive types. Several semantic models with this property are now available [14, 15, 10]. So, the soundness is not in question.

3 The SCIR Type System

The type terms of SCIR have the following context-free syntax:

data types	δ	::=	$\mathbf{int} \mid \mathbf{bool}$
types	θ	::=	$\delta \mid \mathbf{var}[\delta] \mid \mathbf{comm} \mid \theta \times \theta \mid \theta \otimes \theta \mid \theta \to \theta \mid \theta \to_p \theta$

Data types are types for values storable in variables. A data type δ used as a type denotes expressions producing δ-typed values (sometimes written as $\mathbf{exp}[\delta]$). The type of commands is represented by **comm**. There are two kinds of product types: the components of a cross pair ($\times$) may interfere with each other, but the components of a tensor pair ($\otimes$) may not. The type $\mathbf{var}[\delta]$ can be thought of as an abbreviation for $(\delta \to \mathbf{comm}) \times \delta$. Passive functions, which do not assign to any global variables, are given a special type constructor $\to_p$.

Passive types ϕ form a subset of the set of types:

$$\phi ::= \delta \mid \phi \otimes \phi \mid \phi \times \phi \mid \theta \to \phi \mid \theta \to_p \theta$$

A term with a passive type cannot modify any global variables, so it cannot affect the outcome of any other term.

Typing judgements are of the form $\Pi \mid \Gamma \vdash M : \theta$ where M is a term, and θ its type. The type context is partitioned into the *passive zone* Π and the *active zone* Γ, which contain the passive free identifiers and the active free identifiers of M respectively. The passive free identifiers can only be used passively, i.e., in passive-typed subterms. Note that the types assigned to identifiers by Π do not have to be passive.

The typing rules are shown in Figure 1. Identifiers can only be moved from Γ to Π when M has a passive type (Passification). There are no restrictions

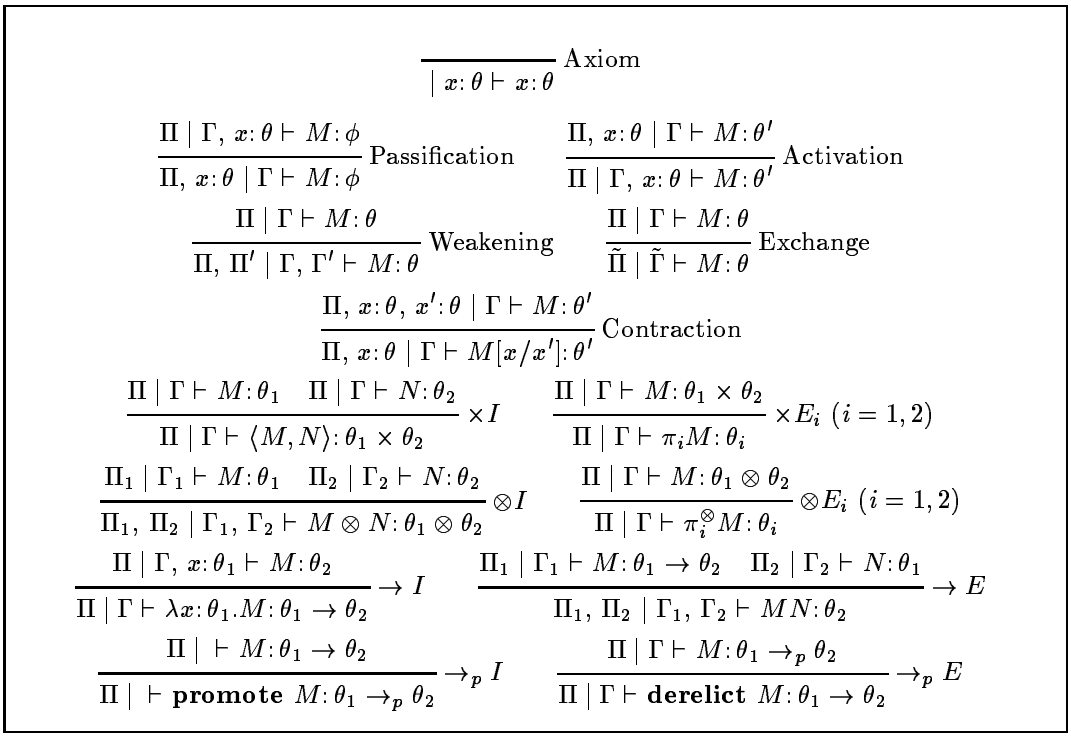

Figure 1: SCIR Typing Rules

on the Activation rule, so identifiers can move from Π to Γ at any time. The $\times I$ rule allows unrestricted identifier sharing, but $\otimes I$ and $\to E$ both require disjoint type contexts in the hypotheses. Sharing is only achieved through Contraction, which allows passive free identifiers to be used multiple times in a term. Finally, note that Γ must be empty in $\to_p I$.

We can use these basic types to define several constants useful in "real" programming languages such as Idealized Algol. For example,

$:=_\delta$	:	$\mathbf{var}[\delta] \times \delta \to \mathbf{comm}$	(assignment)
$!_\delta$	:	$\mathbf{var}[\delta] \to \delta$	(dereference)
$\mathbf{new}_\delta$	:	$(\mathbf{var}[\delta] \to \mathbf{comm}) \to \mathbf{comm}$	(local variable creation)
;	:	$\mathbf{comm} \times \mathbf{comm} \to \mathbf{comm}$	(sequential composition)
$\|\|$	:	$\mathbf{comm} \otimes \mathbf{comm} \to \mathbf{comm}$	(parallel composition)
$\mathbf{do}_\delta$	:	$(\mathbf{var}[\delta] \to_p \mathbf{comm}) \to \delta$	(block expression)
$\mathbf{rec}_\theta$	:	$(\theta \to_p \theta) \to \theta$	(recursion)

We will use $\mathbf{new}[\delta]\ x.\,M$, $\mathbf{do}[\delta]\ x.\,M$ and $\mathbf{rec}[\theta]\ x.\,M$ as syntactic sugar for $\mathbf{new}_\delta\ \lambda x{:}\mathbf{var}[\delta].M$, $\mathbf{do}_\delta\ (\mathbf{promote}\ \lambda x{:}\mathbf{var}[\delta].M)$ and $\mathbf{rec}_\theta\ (\mathbf{promote}\ \lambda x{:}\theta.M)$ respectively.

$$
\begin{array}{rcll}
C(\Gamma; x) & = & \text{let} & \theta = \text{lookup}(x, \Gamma) \\
 & & \text{in} & \text{passify}(\theta, \{x\}, \emptyset, \emptyset) \\
 & & & \\
C(\Gamma; \lambda x{:}\theta'.M) & = & \text{let} & (\theta, A, P, E) = C(\Gamma \cup \{x : \theta'\}; M) \\
 & & \text{in} & \text{if } x \in E \text{ then } \texttt{error} \\
 & & & \text{else } (\theta' \to \theta, A - \{x\}, P - \{x\}, E) \\
 & & & \\
C(\Gamma; MN) & = & \text{let} & (\theta' \to \theta, A_1, P_1, E_1) = C(\Gamma; M) \\
 & & & (\theta', A_2, P_2, E_2) = C(\Gamma; N) \\
 & & & E = (A_1 \cap A_2) \cup (A_1 \cap P_2) \cup (P_1 \cap A_2) \cup (E_1 \cup E_2) \\
 & & \text{in} & \text{passify}(\theta, (A_1 \cup A_2) - E, (P_1 \cup P_2) - E, E) \\
 & & & \\
C(\Gamma; \langle M, N \rangle) & = & \text{let} & (\theta_1, A_1, P_1, E_1) = C(\Gamma; M) \\
 & & & (\theta_2, A_2, P_2, E_2) = C(\Gamma; N) \\
 & & & P = (P_1 \cup P_2) - (P_1 \cap A_2) - (P_2 \cap A_1) \\
 & & \text{in} & (\theta_1 \times \theta_2, A_1 \cup A_2, P, E_1 \cup E_2) \\
 & & & \\
C(\Gamma; \pi_i M) & = & \text{let} & (\theta_1 \times \theta_2, A, P, E) = C(\Gamma; M) \\
 & & \text{in} & \text{passify}(\theta_i, A, P, E) \\
 & & & \\
C(\Gamma; \textbf{promote } M) & = & \text{let} & (\theta_1 \to \theta_2, \emptyset, P, \emptyset) = C(\Gamma; M) \\
 & & \text{in} & (\theta_1 \to_p \theta_2, \emptyset, P, \emptyset) \\
 & & & \\
C(\Gamma; \textbf{derelict } M) & = & \text{let} & (\theta_1 \to_p \theta_2, A, P, E) = C(\Gamma; M) \\
 & & \text{in} & (\theta_1 \to \theta_2, A, P, E) \\
 & & & \\
\text{passify}(\theta, A, P, E) & = & \multicolumn{2}{l}{\begin{cases} (\theta, \emptyset, A \cup P \cup E, \emptyset) & \text{if } \theta \text{ is passive} \\ (\theta, A, P, E) & \text{otherwise} \end{cases}}
\end{array}
$$

Figure 2: Type Checking Algorithm for SCIR

4 Type Checking

The main issue in type checking for SCIR is the noncompositionality of the type system: a term M may be well-typed in some context Γ, even if its subterms seemingly are not. For example, given c: **comm**, the term $\pi_1\langle 3, c \parallel c\rangle$ is typable, but the subterm $c \parallel c$ is not. Our approach is to keep track of the free identifiers in a term M that are used actively, but must be passified (and contracted) because of sharing. If M occurs in the context of a larger passive term, then all its free identifiers have passive occurrences. Otherwise, the sharing is illegal.

An outline of an SCIR type checking algorithm C is shown in Figure 2. Given a set of type assumptions Γ of the form $\{x_1{:}\theta_1, x_2{:}\theta_2, \ldots, x_n{:}\theta_n\}$ and a term M, $C(\Gamma, M)$ returns either `error` or a quadruple (θ, A, P, E). The type of M is θ, and A, P and E form a partition of the free identifiers of M such that:

- A contains the non-shared, actively-used identifiers,
- P is the set of identifiers (which may be shared) that are used passively,

and

- E contains shared identifiers that are used actively.

A term M is not typable in a context Γ if the type checker returns `error` or if it returns a quadruple where E is not empty. If E is empty, then all sharing is legal and M is typable.

The algorithm description uses pattern matching to bind variables as well as to express restrictions on values and types. In a recursive call $(\theta, A, P, E) = C(\Gamma; M)$, each component of the left side must be unifiable with the corresponding component of the right side. The auxiliary function *passify* moves all free identifiers to P whenever a term has a passive type.

As an example, the results of type checking the two terms mentioned above are shown here:

$$\begin{array}{lcl} C(\{c\colon \mathbf{comm}\}; c \parallel c) & = & (\mathbf{comm}; \emptyset; \emptyset; \{c\}) \\ C(\{c\colon \mathbf{comm}\}; \pi_1\langle \mathbf{3}, c \parallel c\rangle) & = & (\mathbf{int}; \emptyset; \{c\}; \emptyset) \end{array}$$

The running time of the algorithm depends on the way sets are represented. Using ordinary lists, for example, results in a running time of $O(n^3)$, where n is the length of the term.

5 Type Reconstruction

Extend the language of terms with untyped lambda terms of the form $\lambda x.\, M$. Given a term M in the extended language, type reconstruction is the problem of finding a well-typed term M' in the original language such that M is obtained by erasing some or all type declarations in M'. As is well-known, such a term M' is not unique. One must use type variables to cover all possibilities.

A *type scheme* μ is a type term which can possibly contain type variables α. Recall that SCIR distinguishes the subset of passive types from the set of all types. How do we tell whether or not a type variable α represents a passive type? We define two kinds (second-order types), `Passive` and `Type`, corresponding to the two classes. We write $\alpha :: \kappa$ to indicate that type variable α has kind κ. Given a kind assignment K for the set of type variables occurring in μ, we can determine if μ is passive or not. The SCIR type system can be extended to type schemes using kind assignments.

An SCIR type reconstruction algorithm is responsible for deducing both missing type information as well as a kind assignment for the type variables. Unfortunately, a typing judgement may be valid under several kind assignments. For example, the term $\lambda f.\, \lambda x.\, fxx$ can be typed under two different kind assignments, as shown in Figure 3. If $\alpha ::$ `Passive`, then x and x' can be passified immediately after they are introduced. On the other hand, if $\beta ::$ `Passive`, then x and x' can be passified after $\rightarrow E$. In both cases, the two identifiers can be contracted, so fxx is typable.

To get around this difficulty, we use *kind constraints*, boolean constraints which represent the class of *all* kind assignments under which a term is typable. Let p be a unary predicate with the semantics that $p(\mu)$ means μ is passive. When μ is a ground type, $p(\mu)$ can be simplified to true if μ is a passive type,

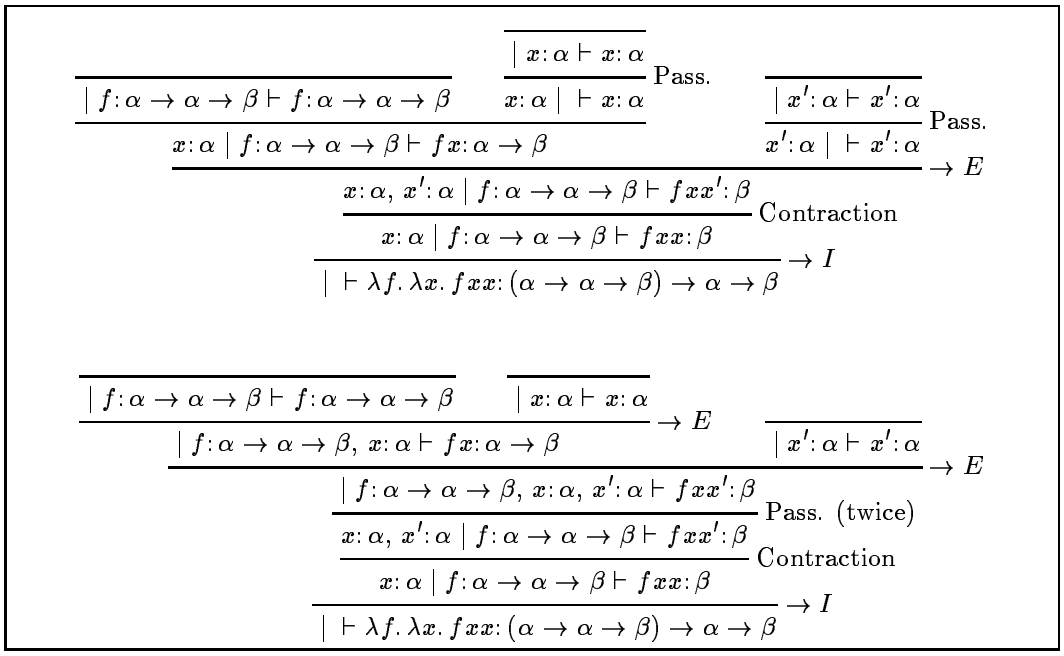

Figure 3: Derivations for $\lambda f.\, \lambda x.\, fxx$ when $\alpha :: \texttt{Passive}$ (top) and when $\beta :: \texttt{Passive}$ (bottom).

and false otherwise. For type variables α, $p(\alpha)$ is true under a kind assignment K if $(\alpha :: \texttt{Passive}) \in K$. Simplifications are shown below:

$$\begin{array}{rcl} p(\delta) & \equiv & \text{true} \\ p(\mathbf{var}[\delta]) & \equiv & \text{false} \\ p(\mathbf{comm}) & \equiv & \text{false} \\ p(\mu_1 \times \mu_2) & \equiv & p(\mu_1) \wedge p(\mu_2) \\ p(\mu_1 \otimes \mu_2) & \equiv & p(\mu_1) \wedge p(\mu_2) \\ p(\mu_1 \to \mu_2) & \equiv & p(\mu_2) \\ p(\mu_1 \to_p \mu_2) & \equiv & \text{true} \end{array} \tag{1}$$

Kind constraints are described by the following grammar:

$$C ::= \text{true} \mid \text{false} \mid p(\mu) \mid C \vee C' \mid C \wedge C'$$

A constraint C is *satisfiable* if there exists some kind assignment K under which C can be simplified to true.

To support type reconstruction, we define a modified type system SCIR_K which maintains kind constraints for type variables. A judgement is of the form $A \vdash M\colon \mu\ [G]$ where the assumption list A is of the form

$$x_1\colon \nu_1\ [P_1; C_1],\ \ldots,\ x_n\colon \nu_n\ [P_n; C_n]$$

and P_i, C_i and G stand for kind constraints. Sometimes we also use the vector notation $\vec{x}\colon \vec{\nu}\ [\vec{P}; \vec{C}]$ to represent an assumption list. Each identifier x_i in the

$$\frac{}{x\colon \mu\ [p(\mu);\mathrm{true}] \vdash x\colon \mu\ [\mathrm{true}]}\ \text{Axiom}$$

$$\frac{A,\ x\colon \mu_1\ [P;C] \vdash M\colon \mu_2\ [G]}{A \vdash \lambda x.M\colon \mu_1 \to \mu_2\ [G \wedge C]} \to I \qquad \frac{A \vdash M\colon \mu_2\ [G]}{A \vdash \lambda x.M\colon \mu_1 \to \mu_2\ [G]} \to I' \quad (x \text{ not in } A)$$

$$\frac{\vec{x}\colon \vec{\nu}\ [\vec{P};\vec{C}],\ \vec{y}\colon \vec{\nu}_1\ [\vec{P}_1;\vec{C}_1] \vdash M\colon \mu_1 \to \mu_2\ [G_1] \quad \vec{x}\colon \vec{\nu}\ [\vec{P}';\vec{C}'],\ \vec{z}\colon \vec{\nu}_2\ [\vec{P}_2;\vec{C}_2] \vdash N\colon \mu_1\ [G_2]}{\begin{array}{l}\vec{x}\colon \vec{\nu}\ [(\vec{P} \wedge \vec{P}') \vee p(\mu_2);(\vec{P} \wedge \vec{P}') \vee p(\mu_2)],\ \vec{y}\colon \vec{\nu}_1\ [\vec{P}_1 \vee p(\mu_2);\vec{C}_1 \vee p(\mu_2)],\\ \quad \vec{z}\colon \vec{\nu}_2\ [\vec{P}_2 \vee p(\mu_2);\vec{C}_2 \vee p(\mu_2)] \vdash MN\colon \mu_2\ [G_1 \wedge G_2]\end{array}} \to E$$

$$\frac{\vec{x}\colon \vec{\nu}\ [\vec{P};\vec{C}],\ A_1 \vdash M\colon \mu_1\ [G_1] \quad \vec{x}\colon \vec{\nu}\ [\vec{P}';\vec{C}'],\ A_2 \vdash N\colon \mu_2\ [G_2]}{\vec{x}\colon \vec{\nu}\ [\vec{P} \wedge \vec{P}';\vec{P} \wedge \vec{P}'],\ A_1,\ A_2 \vdash M \otimes N\colon \mu_1 \otimes \mu_2\ [G_1 \wedge G_2]} \otimes I$$

$$\frac{\vec{x}\colon \vec{\nu}\ [\vec{P};\vec{C}] \vdash M\colon \mu_1 \otimes \mu_2\ [G]}{\vec{x}\colon \vec{\nu}\ [\vec{P} \vee p(\mu_i);\vec{C} \vee p(\mu_i)] \vdash \pi_i^{\otimes} M\colon \mu_i\ [G]} \otimes E_i\ (i = 1,2)$$

$$\frac{\vec{x}\colon \vec{\nu}\ [\vec{P};\vec{C}],\ A_1 \vdash M\colon \mu_1\ [G_1] \quad \vec{x}\colon \vec{\nu}\ [\vec{P}';\vec{C}'],\ A_2 \vdash N\colon \mu_2\ [G_2]}{\vec{x}\colon \vec{\nu}\ [\vec{P} \wedge \vec{P}';\vec{C} \wedge \vec{C}'],\ A_1,\ A_2 \vdash \langle M,N\rangle\colon \mu_1 \times \mu_2\ [G_1 \wedge G_2]} \times I$$

$$\frac{\vec{x}\colon \vec{\nu}\ [\vec{P};\vec{C}] \vdash M\colon \mu_1 \times \mu_2\ [G]}{\vec{x}\colon \vec{\nu}\ [\vec{P} \vee p(\mu_i);\vec{C} \vee p(\mu_i)] \vdash \pi_i M\colon \mu_i\ [G]} \times E_i\ (i = 1,2)$$

$$\frac{\vec{x}\colon \vec{\nu}\ [\vec{P};\vec{C}] \vdash M\colon \mu_1 \to \mu_2\ [G]}{\vec{x}\colon \vec{\nu}\ [\overrightarrow{\mathrm{true}};\overrightarrow{\mathrm{true}}] \vdash \mathbf{promote}\ M\colon \mu_1 \to_p \mu_2\ [G \wedge (\bigwedge \vec{P})]} \to_p I \qquad \frac{A \vdash M\colon \mu_1 \to_p \mu_2\ [G]}{A \vdash \mathbf{derelict}\ M\colon \mu_1 \to \mu_2\ [G]} \to_p E$$

Figure 4: Type rules for SCIR_K.

list is associated with 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.) A judgement $A \vdash M\colon \mu\ [G]$ can be read as "The term M has type μ in 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." From this reading, it is clear that the P-constraint is what is needed for an identifier to be passified, whereas C- and G-constraints arise from identifiers that have already been passified. The difference between C- and G-constraints is that they typically refer to occurrences of free and bound identifiers respectively.

The type rules of SCIR_K are shown in Figure 4. There is some notation that should be explained. For $\otimes I$ and $\times I$, the environments A_1 and A_2 have disjoint domains. Similarly, $\vec{y}$ and $\vec{z}$ are disjoint in the $\to E$ rule. Identifiers common to both hypotheses are represented by $\vec{x}$, although the type and constraint information may differ in each hypothesis.

Boolean operations on constraint vectors can be defined straightforwardly.

If $\vec{P} = P_1, \ldots, P_n$ and $\vec{P'} = P'_1, \ldots, P'_n$,

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

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

Note the following facts about the SCIR_K system:

1. The rules are syntax-directed. Given a term M, there is at most one derivation for M. (Although there are two rules $\rightarrow I$ and $\rightarrow I'$ for deriving $\lambda x.M$, only one is applicable depending on whether or not x occurs free in M.)

2. There are no structural rules. All the identifiers of an assumption list A occur in the term M in a derivable judgement.

3. For every identifier x in an assumption list of a derivable judgement, the passification constraint logically implies the contraction constraint.

4. For every derivable judgement $A \vdash M\colon \mu\ [G]$, the constraint $p(\mu)$ logically implies every passification and contraction constraint in A.

The first two facts mean that we can devise a type reconstruction algorithm for SCIR_K in the standard fashion (using unification in the Hindley-Milner style).

5.1 Explanation

The *passification constraint* of an identifier specifies the condition necessary for that identifier to be passified. The constraint will simplify to true if and only if all occurrences of the identifier appear in passively-typed subterms. This constraint may become weaker as the derivation progresses—in particular, through applications of the $\times E_i$, $\otimes E_i$ and $\rightarrow E$ rules. Intuitively, this corresponds to situations where the identifier appears in the context of a larger, passively-typed term.

The *contraction constraint* of an identifier is similar to its passification constraint, but represents conditions arising from occurrences that have already been contracted. Hence, the contraction constraints must necessarily hold for the term to be well-typed. Like passification constraints, contraction constraints may become weaker as the derivation progresses.

The *global constraint* records the conditions that cannot be weakened further in the rest of the derivation. If the global constraint of a term M is false, then there are identifiers in M that must be passified but cannot be, regardless of M's context. Neither M nor any term containing M is typable.

The following derivations illustrate the key aspects of SCIR_K. First, the

term $\lambda c\colon \mathbf{comm}.\ \pi_1\langle 3, c \parallel c\rangle$ can be typed as shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{c\colon \mathbf{comm}\ [\text{false}; \text{true}] \vdash c\colon \mathbf{comm}\ [\text{true}]}\ \text{Axiom}}{c\colon \mathbf{comm}\ [\text{false}; \text{false}] \vdash c \parallel c\colon \mathbf{comm}\ [\text{true}]}\ \parallel (\otimes I)}{c\colon \mathbf{comm}\ [\text{false}; \text{false}] \vdash \langle 3,\ c \parallel c\rangle\colon \mathbf{int} \times \mathbf{comm}\ [\text{true}]}\ \times I}{c\colon \mathbf{comm}\ [\text{true}; \text{true}] \vdash \pi_1\langle 3,\ c \parallel c\rangle\colon \mathbf{int}\ [\text{true}]}\ \times E_1}{\vdash \lambda c\colon \mathbf{comm}.\ \pi_1\langle 3, c \parallel c\rangle\colon \mathbf{comm} \to \mathbf{int}\ [\text{true}]} \to I$$

(Recall that $p(\mathbf{comm})$ is false, while $p(\mathbf{int})$ is true.) The subterm $c \parallel c$ *appears* to be illegal, but it may safely occur within any passively-typed term, in which case the contraction constraint on c is weakened and becomes satisfiable.

The next example shows that the term $\lambda c\colon \mathbf{comm}.\ (c \parallel c)$ is illegal independent of its context.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{c\colon \mathbf{comm}\ [\text{false}; \text{true}] \vdash c\colon \mathbf{comm}\ [\text{true}]}\ \text{Axiom}}{c\colon \mathbf{comm}\ [\text{false}; \text{false}] \vdash c \parallel c\colon \mathbf{comm}\ [\text{true}]}\ \parallel (\otimes I)}{\vdash \lambda c\colon \mathbf{comm}.\ (c \parallel c)\colon \mathbf{comm} \to \mathbf{comm}\ [\text{false}]} \to I$$

Once the identifier c becomes lambda-bound, it is no longer free and cannot be passified. The global constraint will be false in any continuation of this derivation.

As an intricate example, consider $\lambda f.\ \lambda g.\ \lambda x.\ f(\pi_1 x \otimes \pi_1 x); g(\pi_2 x)$. Define $\mu = \alpha \otimes \alpha \to \mathbf{comm}$ and $\nu = \beta \to \mathbf{comm}$. The two immediate subterms can be typed as follows:

$$f\colon \mu\ [\text{false}; \text{true}],\ x\colon \alpha \times \beta\ [p(\alpha); p(\alpha)] \vdash f(\pi_1 x \otimes \pi_1 x)\colon \mathbf{comm}\ [\text{true}]$$

$$g\colon \nu\ [\text{false}; \text{true}],\ x\colon \alpha \times \beta\ [p(\beta); \text{true}] \vdash g(\pi_2 x)\colon \mathbf{comm}\ [\text{true}]$$

Applying $\times I$ to these two judgements only affects the constraints of x.

$$f\colon \mu\ [\text{false}; \text{true}], g\colon \nu\ [\text{false}; \text{true}], x\colon \alpha \times \beta\ [p(\alpha) \wedge p(\beta); p(\alpha)]$$
$$\vdash f(\pi_1 x \otimes \pi_1 x); g(\pi_2 x)\colon \mathbf{comm}\ [\text{true}]$$

The passification and contraction constraints of x are different because only the first two occurrences of x are contracted. The third occurrence does not have to be contracted due to the $\times I$ rule. From this judgement we can derive the principal typing:

$$\vdash \lambda f.\ \lambda g.\ \lambda x.\ f(\pi_1 x \otimes \pi_1 x); g(\pi_2 x)\colon \mu \to \nu \to (\alpha \times \beta) \to \mathbf{comm}\ [p(\alpha)]$$

Finally we outline a derivation for the term $\lambda f.\ \lambda x.\ f x x$ mentioned at the beginning of this section. The first application of $\to E$ is straightforward and results in the judgement

$$f\colon \alpha \to \alpha \to \beta\ [p(\beta); \text{true}],\ x\colon \alpha\ [p(\alpha) \vee p(\beta); \text{true}] \vdash f x\colon \alpha \to \beta\ [\text{true}]$$

Applying fx to x and continuing the derivation:

$$\dfrac{\dfrac{f\colon \alpha \to \alpha \to \beta\ [p(\beta); \text{true}],\ x\colon \alpha\ [p(\alpha) \vee p(\beta); p(\alpha) \vee p(\beta)] \vdash fx\colon \beta\ [\text{true}]}{f\colon \alpha \to \alpha \to \beta\ [p(\beta); \text{true}] \vdash \lambda x.\, fx\colon \alpha \to \beta\ [p(\alpha) \vee p(\beta)]} \to I}{\vdash \lambda f.\, \lambda x.\, fx\colon (\alpha \to \alpha \to \beta) \to \alpha \to \beta\ [p(\alpha) \vee p(\beta)]} \to I$$

The term is typable in any kind assignment which maps α or β (or both) to the kind `Passive`.

5.2 Soundness

A *type substitution* σ maps type variables to types. By $\sigma(A \vdash M\colon \mu\ [G])$ we mean the type judgement obtained by applying σ to all type terms in the judgement. We call the result an *instance* of the original judgement. An SCIR judgement $\Pi \mid \Gamma \vdash M'\colon \theta$ is said to be an *instance* of $A \vdash M\colon \mu\ [G]$ if there is a type substitution σ such that

1. $M = \text{erase}(M')$,
2. $\sigma(A \vdash M\colon \mu\ [G])$ is the same as $\Pi \mid \Gamma \vdash M'\colon \theta$ except for the constraints and type declarations in M', and
3. in $\sigma(A \vdash M\colon \mu\ [G])$, the global constraint, all the C-constraints, and the P-constraints of all identifiers in $\text{dom}(\Pi)$ simplify to true.

An *extension* of a judgement $A \vdash M\colon \mu\ [G]$ has the form $\vec{x}\colon \vec{\nu}\ [\overrightarrow{\text{true}}; \overrightarrow{\text{true}}]$, $A \vdash M : \mu\ [G]$ (obtained by adding identifiers with trivial constraints). An SCIR judgement $\Pi \mid \Gamma \vdash M'\colon \theta$ is said to be *covered* by an SCIR_K judgement $A \vdash M\colon \mu\ [G]$ if it is an instance of an extension of the latter.

Theorem 1 (Soundness) *If the judgement $A \vdash M\colon \mu\ [G]$ is derivable in $SCIR_K$ and covers an SCIR judgement $\Pi \mid \Gamma \vdash M'\colon \theta$, the latter is derivable in SCIR.*

5.3 Completeness

Lemma 2 *If $A \vdash M\colon \mu\ [G]$ is derivable, then every instance $\sigma(A \vdash M\colon \mu\ [G])$ is derivable.*

Lemma 3 *If $x\colon \nu\ [P; C]$, $x'\colon \nu\ [P'; C']$, $A \vdash M\colon \mu\ [G]$ is derivable, then $x\colon \nu\ [P \wedge P'; C'']$, $A \vdash M[x/x']\colon \mu\ [G]$ is derivable, for some C-constraint C''.*

Proof: By induction on the derivation of the given judgement. If the last derivation step has two hypotheses, where x appears in one and x' appears in the other, we can substitute x for x' and derive the conclusion.

Theorem 4 (Completeness) *If $\Pi \mid \Gamma \vdash M'\colon \theta$ is derivable in SCIR, then there exists a derivable judgement $A \vdash M\colon \mu\ [G]$ of $SCIR_K$ that covers $\Pi \mid \Gamma \vdash M'\colon \theta$.*

Proof: By induction on the derivation of $\Pi \mid \Gamma \vdash M' : \theta$. Consider the last derivation step. Some of the key cases are outlined below.

- *Passification.* By induction, $\Pi \mid x : \theta', \Gamma \vdash M' : \phi$ is covered by some $x : \nu\ [P; C],\ A \vdash M : \mu\ [G]$ using a substitution σ. Since $\sigma(\mu)$ is passive, $\sigma(P)$ holds by fact 4. Thus, the same SCIR_K judgement also covers the conclusion $\Pi, x : \theta' \mid \Gamma \vdash M' : \phi$.
- *Contraction.* This follows from Lemma 3.
- $\otimes I$. By induction, there exist

  $$A_1 \vdash M : \mu_1\ [G_1] \qquad \text{and} \qquad A_2 \vdash N : \mu_2\ [G_2]$$

  which cover

  $$\Pi_1 \mid \Gamma_1 \vdash M' : \theta_1 \qquad \text{and} \qquad \Pi_2 \mid \Gamma_2 \vdash N' : \theta_2$$

  So we can derive

  $$A_1,\ A_2 \vdash M \otimes N : \mu_1 \otimes \mu_2\ [G_1 \wedge G_2]$$

  If σ_1 and σ_2 are the substitutions by which the SCIR_K hypotheses cover the SCIR hypotheses, then $\sigma_1 \oplus \sigma_2$ covers the conclusion.
- $\times I$. By induction, the hypotheses of $\times I$ are covered by

  $$\vec{x} : \vec{\nu}\ [\vec{P}; \vec{C}],\ A_1 \vdash M : \mu_1\ [G_1]$$

  and

  $$\vec{x} : \vec{\nu'}\ [\vec{P'}; \vec{C'}],\ A_2 \vdash N : \mu_2\ [G_2]$$

  Since the assumption lists in these judgements both cover $\Pi \cup \Gamma$, there exists a most general type substitution σ_0 such that $\sigma_0(\vec{\nu}) = \sigma_0(\vec{\nu'})$. By Lemma 2,

  $$\sigma_0(\vec{x} : \vec{\nu}\ [\vec{P}; \vec{C}],\ A_1 \vdash M : \mu_1\ [G_1])$$

  and

  $$\sigma_0(\vec{x} : \vec{\nu'}\ [\vec{P'}; \vec{C'}],\ A_2 \vdash N : \mu_2\ [G_2])$$

  are also derivable. Then we can apply the SCIR_K $\times I$ rule.

5.4 Reconstruction Algorithm

It is relatively straightforward to translate the type rules into a reconstruction algorithm K, where

$$K(M) = (A, \mu, G)$$

if and only if

$$A \vdash M : \mu\ [G]$$

is derivable in SCIR_K. Whenever two subterms containing a common identifier x are combined (using $\rightarrow E$, $\otimes I$, or $\times I$), K must find a most general unifier for the types of x in the subterms. If no such unifier exists, then the term cannot be typed. The kind constraints can be simplified using the simplification rules (1) as well as the laws of boolean algebra. If the G-constraint simplifies to false, then the term cannot be typed.

The reconstruction algorithm takes exponential time in the worst case, as in the case of Hindley-Milner type inference.

5.5 Comparison with linear type reconstruction

In [22], Wadler gives a type reconstruction algorithm for a linear type system based on "standard types." Our reconstruction algorithm is somewhat reminiscent of this algorithm and we have indeed derived significant inspiration from Wadler's work. On the other hand our algorithm differs in more ways than it resembles Wadler's. In the first place, SCIR is considerably more sophisticated than the linear type system. In addition to the promotion ($\rightarrow_p I$) and dereliction ($\rightarrow_p E$) rules "on the right," which correspond to $!I$ and $!E$ in the linear type system, SCIR also has promotion and dereliction rules "on the left" (Passification and Activation rules). The main challenge of our reconstruction algorithm is in handling the left-rules (which are necessarily implicit in the syntax), while Wadler's algorithm is concerned with making the right-rules implicit. This probably accounts for the differences in the constraint maintenance in the two algorithms. While we need to associate constraints with individual free identifiers, Wadler's algorithm requires a single global constraint per judgement.

On the other hand, it would also be desirable to make the right-promotion and dereliction rules implicit in the SCIR syntax. Were we to do so, one would expect that some features of the Wadler's algorithm would resurface in the context of SCIR.

6 Conclusion

We have presented a type reconstruction algorithm for Syntactic Control of Interference. The algorithm is derived in a logical fashion via an inference system SCIR_K that has unique derivations for terms. This system is shown sound and complete with respect to the original type system.

We have implemented a prototype of the type reconstruction algorithm in Prolog (using naive boolean simplifications). As a measure of the complexity of types inferred, we considered the index

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

and found it to be in the range 0–1 for typical library functions. Thus the complexity of inferred types is within practical limits.

A topic for future research is to incorporate let-based polymorphism in the style of ML. This would involve incorporating kind constraints in the syntax of type schemes as in $\forall\beta.\, p(\alpha) \vee p(\beta) \Rightarrow \mu[\alpha, \beta]$. The intriguing feature of such type schemes is that the kind constraint sometimes refers to free type variables (α) as well as bound type variables (β). More work is needed to understand the implications of this feature.

Further work also remains to be done in making various coercions implicit. Programming convenience demands that the **derelict** operator and the dereferencing operator should be omitted and the same projection operators should be usable for both tensor products and cross products. A more ambitious goal would be to make the promotion operator implicit. All such implicit syntactic features would increase the number of possible typings for terms and, very likely, the complexity of the type scheme syntax as well.

References

[1] 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.

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

[3] J.-Y. Girard. On the unity of logic. *Annals of Pure and Appl. Logic*, 59:201–217, 1993.

[4] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343. IEEE Computer Society Press, June 1990.

[5] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symp. Semantics of Algorithmic Languages*, volume 188 of *Lect. Notes Math.*, pages 102–116. Springer-Verlag, 1971.

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

[7] I. Mackie. Lilac: A functional programming language basedon linear logic. *J. Functional Program.*, 4(4):395–433, Oct 1994.

[8] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

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

[10] 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.

[11] S. L. Peyton Jones and J. Launchbury. State in Haskell. *J. Lisp and Symbolic Comput.*, 1996. (to appear).

[12] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Twentieth Ann. ACM Symp. on Princ. of Program. Lang.* ACM, 1993.

[13] G. J. Popek *et. al.* Notes on the design of EUCLID. *SIGPLAN Notices*, 12(3):11–18, 1977.

[14] U. S. Reddy. Passivity and independence. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 342–352. IEEE Computer Society Press, July 1994.

[15] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *J. Lisp and Symbolic Computation*, 1996. (to appear.).

[16] J. C. Reynolds. Syntactic control of interference. In *ACM Symp. on Princ. of Program. Lang.*, pages 39–46. ACM, 1978.

[17] 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.

[18] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982.

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

[20] 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.

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

[22] 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).