From Algol to Polymorphic Linear Lambda-calculus (Working Draft Version)

Peter W. O'Hearn*
Dept. of Computer Science
Queen Mary & Westfield College
(ohearn@dcs.qmw.ac.uk)

John C. Reynolds
School of Computer Science
Carnegie Mellon University
(john.reynolds@cs.cmu.edu)

July 12, 1996

Abstract

We show how to translate Algol-like programs (with or without syntactical control of interference) into a functional language with polymorphic linear types. The translations are an expression of the functor-category models developed by Reynolds and Oles in the early eighties, using a linear polymorphic lambda-calculus in place of functor categories. On one hand, the translations give rise to semantic models for Algol-like languages, while on the other hand they demonstrate that a linearly typed functional language can be at least as expressive as Algol. Moreover, in such a language one can go beyond conventional imperative languages to construct state transformations that alter the type of the state, or even split or combine states.

1 Introduction

Linear types give rise to a functional programming language where one can define functions that consume their arguments in the process of computing their result. This (informally stated) view of linear functions calls to mind the state transformations that can be effected in imperative languages. In this paper we translate two variations on Algol 60 into (a predicative fragment of) polymorphic linear λ-calculus, extended with a fixed-point operator. The basic idea of the translations is to use different type variables to represent separate portions of the store, and thereby to work with multiple state types at once. Procedures are represented as functions polymorphic in state types and programs that work on different parts of the store communicate with one another using polymorphic instantiation. This results in a view of the information-hiding aspect of local state as parametricity of polymorphic functions. Linearity is used to model the operational intuition that the store is altered in the process of an imperative computation.

*Work partially supported by NSF grant CCR-92110829.

To analyze the translations we consider a model of the polymorphic language based on strict continuous functions and relational parametricity. We show how the model supports reasoning about a number of examples involving local state and irreversibility of state change, and we give a representation theorem that provides an explicit description of the structure of first-order types.

The two translations are an outgrowth of the functor-category semantics developed originally by Reynolds and Oles in the early eighties [11, 8], with the basic ideas expressed here using polymorphic linear λ-calculus in place of functor categories. This results in both a simplification and refinement of the original functor-category model, and a systematization and improvement of the parametric-functor semantics of O'Hearn and Tennent [7].

One way to view this work is as using the linear polymorphic calculus as a semantic metalanguage to analyze the tarnaslated programming languages. An alternative is to view each translation as a mapping between programming languages, one imperative and the other purely functional. The idea that linear type theory is a functional language that can nevertheless faithfully represent a range of imperative features is then buttressed by the semantic analysis of the translations. The striking point is not that *some* translation into a pure language is possible – there are a host of translations that account for behaviour rather crudely – but that the translations preserve subtler aspects of state manipulation related to noninterference, locality, and the irreversible nature of state change.

In this second view the translations perhaps raise more questions than they answer. For instance, the range of types in the linear calculus that support an imperative reading is surely greater than the union of the ranges of the two translations: An intruiging question is what kind of imperative language could express a wider range of the polymorphic types and, indeed, whether the linear calculus might itself be regarded as such a language.

2 Two Variations on Algol

Our imperative languages are based on the analysis of Algol 60 given in [10, 11]. The one, substantial, caveat is that our languages do not account for passivity. Thus, evaluation of a natural-number expression can produce a side effect, and we do not consider a concept of passive type as found in [10] (also, [6]).

Both languages use the following grammar of types.

$$\begin{array}{lll} \varphi & ::= & \mathbf{comm} \mid \mathbf{exp} \qquad \text{primitive types} \\ \theta & ::= & \varphi \mid \theta \times \theta' \mid \theta \longrightarrow \theta' \qquad \text{types} \end{array}$$

The primitive type **exp** is the type of natural-number expressions, and **comm** is the type of commands. Commands change the state but do not produce values. The type **acc** of acceptors is regarded as an abbreviation for ($\mathbf{exp} \rightarrow \mathbf{comm}$), and the type **var** of storage variables is an abbreviation for $\mathbf{acc} \times \mathbf{exp}$. The components of **var** give the basic capabilities of updating and accessing a storage variable.

2.1 Idealized Algol

Idealized Algol extends the functional language PCF by adding four constructs, **new**, **letval**, ";" and **skip**. $\mathbf{new}(\lambda x\,.\,C)$ works by binding x to a local storage variable that is initialized to 0. $\mathbf{letval}\, M\, (\lambda y\,.\,N)$ evaluates M, binds the value obtained to y, and then executes N. Note that the side effect of M is persistent, and not a snapback.

The type rules for Idealized Algol follow. A typing context Γ is a list of assumptions $x : \theta$ pairing identifiers with types, with the proviso that no identifier appears twice.

$$\frac{}{\Gamma, x : \theta \vdash x : \theta} \qquad \frac{\Gamma \vdash t : \theta''}{\widetilde{\Gamma} \vdash t : \theta''} \text{ where } \widetilde{\Gamma} \text{ is a permutation of } \Gamma$$

$$\frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x : \theta\,.\,M : \theta \to \theta'} \qquad \frac{\Gamma \vdash M : \theta \to \theta' \quad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'}$$

$$\frac{\Gamma \vdash M : \theta_1 \times \theta_2}{\Gamma \vdash \pi_i\, M : \theta_i} \text{ where } i \text{ is 1 or 2} \qquad \frac{\Gamma \vdash M : \theta \quad \Gamma \vdash N : \theta'}{\Gamma \vdash \langle M, N \rangle : \theta \times \theta'}$$

$$\frac{}{\Gamma \vdash 0 : \mathbf{exp}} \qquad \frac{\Gamma \vdash N_1 : \mathbf{exp} \quad \Gamma \vdash N_i : \varphi\,,\ i = 2, 3}{\Gamma \vdash \mathbf{if}\ N_1 = 0\ \mathbf{then}\ N_2\ \mathbf{else}\ N_3 : \varphi}$$

$$\frac{\Gamma \vdash M : \mathbf{exp}}{\Gamma \vdash \mathbf{succ}\ M : \mathbf{exp}} \qquad \frac{\Gamma \vdash M : \mathbf{exp}}{\Gamma \vdash \mathbf{pred}\ M : \mathbf{exp}}$$

$$\frac{\Gamma \vdash M : \theta \to \theta}{\Gamma \vdash Y_\theta\, M : \theta} \qquad \frac{\Gamma \vdash M : \mathbf{var} \to \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, M\ : \mathbf{comm}}$$

$$\frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{comm}} \qquad \frac{\Gamma \vdash M : \mathbf{comm} \quad \Gamma \vdash N : \varphi}{\Gamma \vdash M; N : \varphi}$$

$$\frac{\Gamma \vdash M : \mathbf{exp} \quad \Gamma \vdash N : \mathbf{exp} \to \varphi}{\Gamma \vdash \mathbf{letval}_\varphi\, M\, N : \varphi}$$

The **letval** construct gives us a way to force evaluation of an expression argument. We also use it to account for assignment. In Forsythe [13] the assignment $a := e$ of an expression to an acceptor is regarded as sugar for $a(e)$ where $a : \mathbf{exp} \to \mathbf{comm}$. Since, operationally, e is a thunk it is possible to evaluate it multiple times in an assignment. If we desugar $a := e$ instead as $\mathbf{letval}\, e\, a$ then e is evaluated precisely once and the resultant constant-thunk built from e's value is fed to a as an argument; when a is an acceptor created by a **new** declaration, this has the behaviour expected for assignment. More generally, however, for arbitrary $a : \mathbf{exp} \to \varphi$ the procedure call $\mathbf{letval}\, e\, a$ has the behaviour of a call by value evaluation of $a(e)$. (In Algol 60 and Forsythe call by value is achieved by assigning e to a locally-declared variable, and then passing that variable as an argument in e's place.) The **letval** construct is reminiscient of the **case** constructs used in game models of PCF [1, 4].

A sequential composition of the form $M; N$ when $N : \mathbf{exp}$ may result in an "active expression," which may return different natural numbers on successive uses. For example, if $x : \mathbf{var}$ is a declared variable then $x := x + 1; x$ – which desugars as $(\mathbf{letval}(\mathbf{succ}\,\pi_2 x)(\pi_1 x)); \pi_2 x$ – returns successive natural numbers on successive uses.

The following block illustrates the (related) issues of local state and irreversibility of state change that arise in this language.

$$
\begin{aligned}
\mathbf{new}\ \lambda\, x\,.\ & x := 0; \\
& p(x := x + 1); \\
& \mathbf{if}\,(x > 0)\,\mathbf{then\ diverge}
\end{aligned}
$$

The free identifier p has procedural type $\mathbf{comm} \to \mathbf{comm}$. This block has equivalent termination/non-termination behaviour to $p(\mathbf{diverge})$. The idea is that if p uses its argument at all then the state change must be persistent, since it has no access to the local variable x and so no way to restore its value. In contrast, in many semantic models, including those of [11, 8, 7], there exist "snapback" operations that allow the current state of the store to be remembered and restored later. One version of snapback, of type $\mathbf{comm} \longrightarrow \mathbf{comm}$, works by taking a command c and state s as arguments, returning s if cs is terminating and diverging otherwise. Snapback contradicts the intuitive reasoning in this example and runs counter to the basic operational intuition that assignment changes the state instead of making a copy.

It is worth considering how the inclusion of side effects in expressions impacts the coding of arithmetic operations. A typical encoding of addition is

$$
plus \;=\; Y(\lambda\, plus\,.\,\lambda\, x\,.\,\lambda y\,.\,\mathbf{if}\ x = 0\ \mathbf{then}\ y\ \mathbf{else}\ plus\,(\mathbf{pred}\ x)\,(\mathbf{succ}\ y))
$$

In Idealized Algol (with side effects) an evaluation of $plus\, e_1\, e_2$ will evaluate e_1 and e_2 multiple times, perhaps changing the state each time. For example, in the context of a locally-declared variable z, $(plus\,(z := z + 1; z)\,2)$ diverges if z has initial value ≥ 1 since z is incremented each time $\mathbf{pred}$ is used.

Using **letval** we can program a version of addition that evaluates its arguments once each, left followed by right, and adds the resulting values together.

$$
leftadd \;=\; \lambda x\,.\,\lambda y\,.\,\mathbf{letval}\, x\,(\lambda x'\,.\,\mathbf{letval}\, y\,(\lambda\, y'\,.\, plus\, x'\, y'))
$$

We can also define $rightadd = \lambda x\,.\,\lambda y\,.\, leftadd\, y\, x$.

2.2 Basic SCI

Basic SCI (for syntactic control of interference) is similar to Idealized Algol but for a few modifications. First, it uses the affine λ-calculus as its type system, whereas Idealized Algol uses the full simply-typed calculus. The affine calculus is just the usual simply-typed calculus except that the rule for procedure application is restricted so that procedure and argument have disjoint free identifiers. (This is another way of saying that the calculus does not have contraction.) This restricted rule prevents interference between different identifiers. For instance, y and z are aliases in $((\lambda y \lambda z\,.\cdots y \cdots z)x)x$ if x denotes a storage variable; but this term does not typecheck because there is an occurrence of x in a procedure and its argument.

Second, the rule for recursion is restricted to procedures with no free identifiers. This restriction is needed because otherwise a recursive unwinding $\mathbf{Y}(F) \rhd F(\mathbf{Y}(F))$ would violate the disjointness between procedure and argument that is characteristic of Basic SCI.

Third, in Basic SCI we have a determinate form of parallelism, where $M \parallel N$ is allowed if M and N are disjoint. This illustrates the difference with Idealized Algol, where the same construct would (because of aliasing) lead to indeterminacy.

The Forsythe view of an assignment $a := e$ as $a(e)$ does not work well in Basic SCI because this would prohibit interference between a and e. It would not even allow us to write an assignment statement such as $x := x + 1$. But the desugaring of assignment in terms of $\mathbf{letval}\, e\, a$ does allow interference between e and a.

These modifications and additions to Idealized Algol are summed up in the following rules.

$$\frac{\Gamma \vdash M : \theta \rightarrow \theta' \quad \Gamma' \vdash N : \theta}{\Gamma, \Gamma' \vdash MN : \theta'} \qquad \frac{\Gamma \vdash M : \mathbf{comm} \quad \Gamma' \vdash N : \mathbf{comm}}{\Gamma, \Gamma' \vdash M \parallel N : \mathbf{comm}}$$

$$\frac{\vdash M : \theta \rightarrow \theta}{\vdash Y_\theta\, M : \theta}$$

To illustrate further the difference between SCI and Idealized Algol, consider the addition operations $leftadd, rightadd : \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp}$. In Idealized Algol, these operations are inequivalent because of interfering side-effects. For example, in a state where the contents of storage variable x is 5, evaluation of $leftadd(2 * x)(x := x + 1; x)$ returns value 16 whereas $rightadd(2 * x)(x := x + 1; x)$ returns value 18. In contrast, in SCI the arguments to these functions never interfere: The typing rules ensure that in a procedure call $leftadd(e_1)(e_2)$ the procedure $leftadd(e_1)$ and argument e_2 have disjoint sets of free identifiers. As a result (even though we allow for side effects in expressions) $leftadd$ and $rightadd$ are equivalent in SCI.

An interfering version of addition *can* be programmed in SCI using type $\mathbf{exp} \times \mathbf{exp} \rightarrow \mathbf{exp}$ instead of $\mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{exp}$. The point is that SCI types can be used to specify both potential dependence and necessary independence between program parts.

3 Polymorphic Linear λ-calculus

The kind of polymorphism we need for interpreting Algol is predicative in nature, so we work with the following stratification of types.

$$\begin{array}{lcll} \sigma & ::= & \alpha \mid \mathbf{nat} \mid \sigma \otimes \sigma \mid I & \text{Level 1} \\ A & ::= & \sigma \mid \forall \alpha \,.\, A \mid A \multimap A \mid A \rightarrow A \mid A \,\&\, A \mid !A & \text{Level 2} \end{array}$$

Type variables are ranged over by α (or other Greek letters β, γ). This stratification is possible because of the distinction between data types and phrase types (or between storable and denotable values) in Algol. The Level 1 types correspond, intuitively, to *store shapes* in the sense of Reynolds and Oles, whereas

Level 2 types are (after translation) types of phrases in the imperative languages. The essence of the stratification is that the $\forall$ quantifier ranges over only Level 1 types. This is significant because it makes the construction of models much easier than for impredicative calculi.

It would be possible to define $A \to B$ as $!A \multimap B$. But since, for the purpose of the two translations, the only significant uses of "!" would be in this encoding we prefer to work explicitly with both function types $\multimap$ and $\to$. Accordingly, there are two binding forms, $\underline{\lambda} x : A \,.\, t$ and $\lambda x : A \,.\, t$.

The system uses typing judgements of the form

$$\Gamma; \Delta \vdash M : A$$

where the context is broken into an intuitionistic zone Γ and a linear zone Δ. The version of linear type theory we use is taken from Barber [2].

$$\frac{\Gamma;\Delta \vdash t : A}{\widetilde{\Gamma};\widetilde{\Delta} \vdash t : A} \text{ where } \widetilde{\Gamma},\widetilde{\Delta} \text{ are permutations of } \Gamma,\Delta$$

$$\frac{}{\Gamma, x : A; \_ \vdash x : A} \qquad \frac{}{\Gamma; x : A \vdash x : A}$$

$$\frac{\Gamma; \_ \vdash t : A}{\Gamma; \_ \vdash !t : !A} \qquad \frac{\Gamma;\Delta_2 \vdash u : !A \quad \Gamma, x : A; \Delta_1 \vdash t : B}{\Gamma;\Delta_1,\Delta_2 \vdash \mathbf{let}\ !x\ \mathbf{be}\ u\ \mathbf{in}\ t : B}$$

$$\frac{\Gamma, x : A; \Delta \vdash t : B}{\Gamma;\Delta \vdash \lambda x : A.t : A \to B} \qquad \frac{\Gamma;\Delta \vdash t : A \to B \quad \Gamma; \_ \vdash u : A}{\Gamma;\Delta \vdash tu : B}$$

$$\frac{\Gamma;\Delta, x : A \vdash t : B}{\Gamma;\Delta \vdash \underline{\lambda} x : A.t : A \multimap B} \qquad \frac{\Gamma;\Delta_1 \vdash t : A \multimap B \quad \Gamma;\Delta_2 \vdash u : A}{\Gamma;\Delta_1,\Delta_2 \vdash tu : B}$$

$$\frac{\Gamma;\Delta \vdash t : A \& B}{\Gamma;\Delta \vdash \pi_1 t : A} \qquad \frac{\Gamma;\Delta \vdash t : A \quad \Gamma;\Delta \vdash u : B}{\Gamma;\Delta \vdash \langle t,u\rangle : A \& B}$$

$$\frac{\Gamma;\Delta \vdash t : A \& B}{\Gamma;\Delta \vdash \pi_2 t : B} \qquad \frac{\Gamma;\Delta_1 \vdash t : A \quad \Gamma;\Delta_2 \vdash u : B}{\Gamma;\Delta_1,\Delta_2 \vdash [t,u] : A \otimes B}$$

$$\frac{\Gamma;\Delta_2 \vdash u : A \otimes B \quad \Gamma;\Delta_1, x : A, y : B \vdash t : C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathbf{let}\ [x,y]\ \mathbf{be}\ u\ \mathbf{in}\ t : C}$$

$$\frac{}{\Gamma; \_ \vdash * : I} \qquad \frac{\Gamma;\Delta_1 \vdash t : I \quad \Gamma;\Delta_2 \vdash u : A}{\Gamma;\Delta_1,\Delta_2 \vdash \mathbf{let}\ *\ \mathbf{be}\ t\ \mathbf{in}\ u : A}$$

$$\frac{\Gamma;\Delta \vdash t : \forall\alpha.A}{\Gamma;\Delta \vdash t[\sigma] : A[\sigma/\alpha]} \qquad \frac{\Gamma;\Delta \vdash t : A}{\Gamma;\Delta \vdash \Lambda\alpha.t : \forall\alpha.A}\ \alpha \notin \mathbf{fv}(\Gamma,\Delta)$$

$$\frac{}{\Gamma; \_ \vdash 0 : \mathbf{nat}} \qquad \frac{\Gamma; \_ \vdash t : A \to A}{\Gamma; \_ \vdash Y_A\, t : A}$$

$$\frac{\Gamma;\Delta \vdash t : \mathbf{nat}}{\Gamma;\Delta \vdash \mathbf{succ}\ t : \mathbf{nat}} \qquad \frac{\Gamma;\Delta \vdash t : \mathbf{nat}}{\Gamma;\Delta \vdash \mathbf{pred}\ t : \mathbf{nat}}$$

$$\frac{\Gamma;\Delta \vdash t : \mathbf{nat}}{\Gamma;\Delta \vdash \mathbf{copy}\ t : \mathbf{nat} \otimes \mathbf{nat}} \qquad \frac{\Gamma;\Delta \vdash t : \mathbf{nat}}{\Gamma;\Delta \vdash \mathbf{discard}\ t : I}$$

$$\frac{\Gamma;\Delta_1 \vdash u_1 : \mathbf{nat} \quad \Gamma;\Delta_2 \vdash u_2 : \mathbf{nat} \quad \Gamma;\Delta_2 \vdash u_3 : \mathbf{nat}}{\Gamma;\Delta_1,\Delta_2 \vdash \mathbf{if}\ u_1 = 0\ \mathbf{then}\ u_2\ \mathbf{else}\ u_3 : \mathbf{nat}}$$

One point to notice is the explicit terms for copying and discarding natural numbers. Using these and the rules for I, for any *closed* Level 1 type σ we can define appropriate copying and discarding terms of types $\sigma \multimap \sigma \otimes \sigma$ and $\sigma \multimap I$. But we do not have copying or discarding of Level 1 types available generically, as terms of type $\forall\alpha\,.\,\alpha \multimap \alpha \otimes \alpha$ or $\forall\alpha\,.\,\alpha \multimap I$. This distinction between specific and generic copying/discarding is related to the following idea in Idealized Algol: A state change effected by a command on any fixed finite number storage variable could be reversed (using local variables to store and restore the values), but we cannot program a genereral snapback mechanism that reverses state changes for *every* possible shape of the store.

We have used $[s,t]$ as notation for $\otimes$-pairs, reserving $f \otimes g$ for the functorial action of $\otimes$, where

$$f \otimes g \;=\; \lambda x : A \otimes B \,.\, \mathbf{let}\ [y,z]\ \mathbf{be}\ x\ \mathbf{in}\ [fy, gz]$$

when $f : A \multimap A'$ and $g : B \multimap B'$.

We use Ω_A to abbreviate $Y_A(\lambda x : A.x)$. In a fully polymorphic language Y would have type $\forall \alpha \,.\, (\alpha \to \alpha) \to \alpha$, but in the predicative sublanguage this type is not sufficient because α does not range over all types.

4 Translations of Types

In this section we give the translations on the level of types. We consider terms by treating a few examples, leaving the detailed translation to the next section. In presenting the examples we will be somewhat liberal in the use of syntactic sugar and the application of (meaning-preserving) syntactic simplifications in the linear calculus.

4.1 Idealized Algol

The translation for Idealized Algol takes a judgement

$$x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$$

to a judgement

$$x_1 : \theta_1^\star \alpha, \ldots, x_n : \theta_n^\star \alpha; \_ \vdash M^\star \alpha : \theta^\star \alpha$$

in polymorphic linear λ-calculus, where

$$\begin{array}{rcl} \mathbf{comm}^\star \alpha & = & \alpha \multimap \alpha \\ \mathbf{exp}^\star \alpha & = & \alpha \multimap \alpha \otimes \mathbf{nat} \\ (\theta \times \theta')^\star \alpha & = & \theta^\star \alpha \,\&\, \theta'^\star \alpha \\ (\theta \to \theta')^\star \alpha & = & \forall \beta \,.\, \theta^\star(\alpha \otimes \beta) \to \theta'^\star(\alpha \otimes \beta) \end{array}$$

The translation $\theta^\star$ of an Idealized Algol type is a type with a "hole" that can be filled by substituting type variables and other Level 1 types. For instance, $\mathbf{comm}(\alpha \otimes \beta)$ is $\alpha \otimes \beta \multimap \alpha \otimes \beta$. Similarly, a term p gets mapped to a term $p^\star$ with a type variable hole that can be filled with various type variables or Level 1 types: $M^*\sigma$, $M^*\sigma'$... (The translation could be arranged so that each $M^\star$ was a polymorphic function of type

$$\forall \alpha \,.\, \theta_1^\star \alpha \& \cdots \& \theta_n^\star \alpha \to \theta^\star \alpha$$

We prefer, however, to use the term-with-hole representation in order to minimize explicit environment manipulations.)

The only essential uses of linearity involve primitive types and states. In particular, the translations of Algol types always appear in the intuitionistic zone of a typing judgement.

For our first example of a term we consider the block of type **comm**

(1.1) $\quad \mathbf{new}\ (\lambda x : \mathbf{var} \,.\, p(x := x + 1))\ .$

In this block, the free identifier p has type $\mathbf{comm} \rightarrow \mathbf{comm}$ and the bound identifier x denotes a locally-declared variable that is initialized to 0. The translation of the block has type $\alpha \multimap \alpha$, and is defined in the context

$$p : \forall\beta \,.\, (\alpha \otimes \beta \multimap \alpha \otimes \beta) \rightarrow (\alpha \otimes \beta \multimap \alpha \otimes \beta)\ ;\ \_$$

The translation of the block is the term

$$(1.2) \quad \underline{\lambda} s : \alpha \,.\ \mathbf{let}\ [s', n]\ \mathbf{be}\ p[\mathbf{nat}](id_\alpha \otimes (\underline{\lambda} n \,.\, n + 1))\,[s, 0]\ \mathbf{in} \\ (\mathbf{let}\ *\ \mathbf{be}\ (\mathbf{discard}\, n)\ \mathbf{in}\ s')\ .$$

We can see from this how the β-component in the type of p is regarded as ranging over possible pieces of local state. What a local variable declaration does first is extend the state s to $[s, 0]$, i.e. a state with an additional component initialized to 0. In this process of initialization the type of the state changes from α to $\alpha \otimes \mathbf{nat}$, with $\mathbf{nat}$ being the type of the values that can be held by the local variable. Instantiating the β component to $\mathbf{nat}$ allows p to work in this enlarged state: Communication between local state and non-local procedures is achieved through polymorphic instantiation.

To model de-allocation of the local variable the identifier n is discarded in the term (1.2). This illustrates how we need the discarding (and copying) capabilities for specific state types, like $\mathbf{nat}$, even though they are not available polymorphically.

Some equivalences between Idealized Algol terms can be proven using basic laws of polymorphic λ-calculus. For example, the equivalence $(\mathbf{new}\, \lambda\, x \,.\, c) \equiv c$, for an identifier $c : \mathbf{comm}$, follows from basic equivalences of polymorpic λ-calculus together with the assumption that $\mathbf{discard}$ and $\mathbf{copy}$ give $\mathbf{nat}$ a comonoid structure. Typically, the basic equations that are valid in models where $\forall$ is simply indexed product are sufficient for reasoning about $\mathbf{new}$ blocks whose only free identifiers are of ground type, but parametricity is needed when there are free identifiers of procedural type.

Next we consider an example of a snapback operator of type $\mathbf{exp} \rightarrow \mathbf{exp}$. It works by first executing its argument to get a natural number, and then restoring the state to its original value. Snapback would ostensibly be given by a term

$$\Lambda\alpha \,.\, \lambda c : (\alpha \multimap \alpha \otimes \mathbf{nat}) \,.\, \underline{\lambda} s : \alpha \,.\ \mathbf{let}\ [s', n]\ \mathbf{be}\ cs\ \mathbf{in}\ [s, n])$$

of type $\forall\alpha \,.\, (\alpha \multimap \alpha \otimes \mathbf{nat}) \rightarrow (\alpha \multimap \alpha \otimes \mathbf{nat})$ which (given the isomorphism $\alpha \cong \alpha \otimes I$) would determine a closed term of type $(\mathbf{exp} \rightarrow \mathbf{exp})^\star I$. But this term does not have the indicated type because it uses contraction of s and weakening of s', where s and s' are α-typed values. This does not show that no other term produces the behaviour of snapback – for that we will appeal to a semantic model – but it illustrates that it is the general, or polymorphic, snapback that we expect control of structural rules to forbid.

4.2 Basic SCI

We translate

$$x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta$$

in Basic SCI to a judgement

$$x_1 : \theta_1^\circ \alpha_1, \ldots, x_n : \theta_n^\circ \alpha_n ; \_ \vdash M^\circ(\alpha_1, \ldots, \alpha_n) : \theta^\circ(\alpha_1 \otimes \cdots \otimes \alpha_n)$$

in polymorphic linear λ-calculus, where

$$\begin{array}{rcl} \mathbf{comm}^\circ \alpha & = & \alpha \multimap \alpha \\ \mathbf{exp}^\circ \alpha & = & \alpha \multimap \alpha \otimes \mathbf{nat} \\ (\theta \times \theta')^\circ \alpha & = & \theta^\circ \alpha \,\&\, \theta'^\circ \alpha \\ (\theta \to \theta')^\circ \alpha & = & \forall \beta \,.\, \theta^\circ \beta \to \theta'^\circ(\alpha \otimes \beta) \end{array}$$

Comparing to the other translation, each identifier x_i is now associated with a separate piece of state α_i that it acts upon; this makes the noninterference between identifiers in SCI explicit. The other difference is that the procedure type uses $\theta^\circ \beta$ in the argument position instead of $\theta^\circ(\alpha \otimes \beta)$. The result is that the procedure and argument types no longer share type variable α, mirroring the restricted rule for application in SCI which ensures that procedures and their arguments don't interfere.

It is interesting that the substructural aspect of linear logic, particularly the absence of contraction, is *not* modelling the substructural aspect of SCI. Indeed, the translations of SCI types always appear in the intuitionistic zone in the translation of a judgement. It is the use of different type variables that corresponds to the absence of contraction in SCI: Different occurrences of the same SCI type θ, say $x_1 : \theta, x_2 : \theta$, get sent to different types, $x_1 : \theta^\circ \alpha_1, x_2 : \theta^\circ \alpha_2$. Generally, parametric polymorphism is used to model both local state and noninterference, whereas linearity (combined with polymorphism) ensures that there is no snapback.

We give several examples to illustrate the difference with Idealized Algol. If x and y are different identifiers of type **comm** then the translation of $x \parallel y$ is

$$\underline{\lambda} s : \alpha_1 \otimes \alpha_2 \,.\, \mathbf{let}\ [s_1, s_2]\ \mathbf{be}\ s\ \mathbf{in}\ [x s_1, y s_2]$$

Thus, we can see how the disjointness property of SCI is made very explicit: It is obvious that x and y act on disjoint portions of the store, so we can run them in parallel.

Consider now the sequential composition $x; y$ of two command identifiers. This is translated as

$$\underline{\lambda} s : \alpha \,.\, y'(x'(s))$$

where

$$\begin{array}{rcl} x' & = & \lambda s : \alpha \,.\, \mathbf{let}\ [s_1, s_2] = s\ \mathbf{in}\ [x s_1, s_2] \\ y' & = & \lambda s : \alpha \,.\, \mathbf{let}\ [s_1, s_2] = s\ \mathbf{in}\ [s_1, y s_2]. \end{array}$$

Although the translation uses $y'(x'(s))$, which indicates that x is evaluated first, $x'(y'(s))$ reduces, using typical reductions of linear λ-calculus, to **let** $[s_1, s_2]$ **be** s **in** $[xs_1, ys_2]$. Thus, we see that $x; y$ and $x \parallel y$ are equivalent.

Finally, consider our two addition operations *leftadd* and *rightadd*. The translation of $leftadd(x)(y)$ and $rightadd(x)(y)$ are both

$$\begin{array}{l} \underline{\lambda} s : \alpha_1 \otimes \alpha_2 \,.\ \mathbf{let}\ [s_1, s_2]\ \mathbf{be}\ s\ \mathbf{in} \\ \qquad\qquad\qquad \mathbf{let}[[s_1', n], [s_2', m]]\ \mathbf{be}\ [xs_1, ys_2]\ \mathbf{in}\ [[s_1, s_2], m + n] \end{array}$$

where $x : \alpha_1 \multimap \alpha_1 \otimes \mathbf{nat}$ and $y : \alpha_2 \multimap \alpha_2 \otimes \mathbf{nat}$. (The translations do not literally result in this term, but in syntactically distinct terms that are easily, by a simple semantic analysis, seen to be equivalent to this one.)

In Idealized Algol $leftadd(x)(y)$ is translated to

$$\begin{array}{l} \underline{\lambda} s : \alpha\ \mathbf{let}\ [s', m]\ \mathbf{be}\ xs\ \mathbf{in} \\ \qquad \mathbf{let}[s'', n]\ \mathbf{be}\ ys'\ \mathbf{in}\ [s'', m + n] \end{array}$$

where $x, y : \alpha \multimap \alpha \otimes \mathbf{nat}$, and it is clear that the number obtained by evaluating y can depend on the evaluation of x.

4.3 Acceptors

In our illustrative languages the type **acc** of acceptors is an abbreviation for the procedure type **exp** $\rightarrow$ **comm**. An alternative, taken in the original Idealized Algol [11], is to treat acceptors as an additional primitive type, interpreted by

$$\mathbf{acc}^{\star}\alpha \;=\; \alpha \otimes \mathbf{nat} \multimap \alpha \;\; (\cong\; \mathbf{nat} \multimap \mathbf{comm}^{\star}\alpha).$$

In this simpler view an acceptor is passed a value instead of a thunk.

In Algol 60, a formal parameter in a procedure declaration could be specified as **value**. This, in effect, defined a separate type of call by value procedures, distinct from the the standard Algol call by name procedures. There was, however, an important restriction on value parameters specified in the Algol 60 report: "A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values. [5]" We may consider acceptors as call by value procedures in the original Algol 60 sense of call by value. Observing the just-quoted restriction on arguments, it would be possible to consider a call by value procedure type with the following interpretation

$$(\tau \rightarrow_v \theta)^{\star}\alpha \;=\; \tau \multimap \theta^{\star}\alpha$$

with τ ranging over data types such as **nat** and **bool**. A more challenging question is whether this kind of restricted call by value mechanism can be relaxed using the framework of linear lambda-calculus.

5 Translations of Terms

The translations of terms follow ideas from functor category semantics. The reader may wish to skim this section and refer back when appropriate, as most of what we say in the following sections has more to do with types than the details of the term translations.

5.1 Idealized Algol

The translation of **new** follows the pattern of the example in the last section.

$$(\mathbf{new}\,P)^\star\alpha \;=\; \underline{\lambda} s:\alpha\,.\,\mathbf{let}\;[s',n]\;\mathbf{be}\,(P^\star\alpha)[\mathbf{nat}]\,(v[\alpha])\,[s,0]\;\mathbf{in}$$
$$\mathbf{let}\;*\;\mathbf{be}\;(\mathbf{discard}\,n)\;\mathbf{in}\;s'$$

In this equation the expanded state $[s,0]$ and a local variable $v[\alpha]$ are passed as arguments to the procedure P, and the final value n of the local variable is discarded on termination of P. The translation $P^\star$ of P has type $\forall\beta\,.\,\mathbf{var}^\star(\alpha\otimes\beta)\to\mathbf{comm}^\star(\alpha\otimes\beta)$ and polymorphic instantiation is used to set the β-component to **nat**.

The local variable v is given by the term $\Lambda\alpha\,.\,\langle assign[\alpha], lookup[\alpha]\rangle$ of type $\forall\alpha\,.\,((\mathbf{exp}\to\mathbf{comm})\times\mathbf{exp})^\star(\alpha\otimes\mathbf{nat})$, where

$$\begin{aligned}
assign = \Lambda\alpha\,.\,\Lambda\beta\;\lambda x:\mathbf{exp}^\star((\alpha\otimes\mathbf{nat})\otimes\beta)\,.&\\
\underline{\lambda} s:(\alpha\otimes\mathbf{nat})\otimes\beta\,.&\\
\mathbf{let}\;[[[a,n],b],m]\;\mathbf{be}\,xs\;\mathbf{in}&\\
\mathbf{let}\;*\;\mathbf{be}\;(\mathbf{discard}\,n)\;\mathbf{in}\;[[a,m],b]&\\
lookup = \Lambda\alpha\,.\,\underline{\lambda}\,s:\alpha\otimes\mathbf{nat}\,.&\\
\mathbf{let}\;[a,n]\;\mathbf{be}\;s\;\mathbf{in}&\\
\mathbf{let}\;[n',n'']\;\mathbf{be}\;(\mathbf{copy}\,n)\;\mathbf{in}\;[[a,n'],n'']&
\end{aligned}$$

These give the capability for updating and accessing the **nat** component of the state. Simply, $assign[\alpha][\beta]$ updates the **nat**-component of a state of type $(\alpha\otimes\mathbf{nat})\otimes\beta$ and $lookup[\alpha]$ reports the value of the **nat**-component of a state of type $\alpha\otimes\mathbf{nat}$. The types of these operations have more information than seems necessary, e.g. the α and β components in *update*. The *update* operation would be simpler if we included acceptors as an additional primitive type. The essence of the two operations, however, is the use of **discard** in *update* and **copy** in *lookup*. Similar update and lookup operations are therefore available if we replace **nat** by any type that has appropriate versions of copy and discard maps.

Since all types of the form $!A$ have copy and discard maps, if we were to consider an impredicative polymorphic calculus we could write lookup and update maps for "storing" elements of any $!A$ type. An interesting question is whether such operations make (imperative) operational sense.

To translate an application $(M\,N)^\star$, where $M:\theta\to\theta'$ and $N:\theta$, we must use translations of M and N with the following types.

$$\begin{aligned}
&M^\star\alpha:\forall\beta\,.\,\theta^\star(\alpha\otimes\beta)\longrightarrow\theta'^\star(\alpha\otimes\beta)\\
&N^\star\alpha:\theta^\star\alpha
\end{aligned}$$

In order to apply $M^\star$ to $N^\star$ we apply $M^\star\alpha$ to the unit I of $\otimes$, and then use a canonical isomorphism $\alpha\otimes I\cong\alpha$ to make the types of the procedure and argument match up. That is,

$$(MN)^\star\alpha \;=\; h\Big((M^\star\alpha)[I]i(N^\star[\alpha])\Big)$$

where $h : \theta'^{\star}(\alpha \otimes I) \multimap \theta^{\star}\alpha$ and $i : \theta^{\star}\alpha \multimap \theta'^{\star}(\alpha \otimes I)$ are terms coding canonical isomorphisms.

We will not give the explicit definitions of h and i, but simply say that they are defined in a standard way by induction on types. If $T(\alpha)$ is a type of the polymorphic calculus with a free type variable α, and $f : \alpha \multimap \beta$ and $g : \beta \multimap \alpha$ are terms ($\beta \notin \mathbf{fv}(T)$), then there is an induced term $T[f,g] : T(\alpha) \multimap T(\beta)$ obtained by applying f for positive occurrences of α and g for negative occurrences. Further, $T[f,g]$ is an isomorphism (say, in the model of the following section) whenever f and g denote inverse iso's.

This use of canonical isomorphisms is unpleasant, and the reader will understand why we prefer often to concentrate on the translation of types instead of terms. These isomorphisms are treated much more smoothly in a semantics based explicitly on a functor category.

The next case we consider is λ-abstraction. We need to define

$$(\lambda x : \theta \,.\, M)^{\star}\alpha : \forall\beta \,.\, \theta^{\star}(\alpha \otimes \beta) \to \theta'^{\star}(\alpha \otimes \beta)$$

in terms of

$$M^{\star}\alpha : \theta'\alpha, \qquad \text{assuming } x : \theta^{\star}\alpha.$$

The type mismatch between $\theta^{\star}\alpha$ and $\theta^{\star}(\alpha \otimes \beta)$ is dealt with now by using "expansion" mappings $expand_{\theta''} : \forall\alpha\beta \,.\, \theta''^{\star}\alpha \multimap \theta''^{\star}(\alpha \otimes \beta)$. Expansion mappings show how a piece of code defined outside the scope of a local-variable declaration can be used within the scope of the declaration. For instance, a command identifier $c : \mathbf{comm}^{*}\alpha$ gets sent to $\underline{\lambda}[s, s'] : \alpha \otimes \beta \,.\, [c(s), s']$. A detailed description of *expand* is postponed until later in this section, but the way expansions are used to treat λ-abstraction can be set out now. If we are given

$$x_1 : \theta_1^{\star} : \alpha, \ldots, x_n : \theta_n^{\star}\alpha, x : \theta^{\star}\alpha \,;\, \_ \vdash M^{\star}\alpha : \theta'\alpha.$$

Then we also have

$$\begin{aligned} x_1 : \theta_1^{\star}(\alpha \otimes \beta), \ldots, x_n : \theta_n^{\star}(\alpha \otimes \beta), x : \theta^{\star}(\alpha \otimes \beta) \,;\, \_ \\ \vdash M^{\star}(\alpha \otimes \beta) : \theta'(\alpha \otimes \beta) \end{aligned}$$

and using a typical substitution lemma for the linear calculus we can infer

$$\begin{aligned} x_1 : \theta_1^{\star} : \alpha, \ldots, x_n : \theta_n^{\star}\alpha, x : \theta^{\star}(\alpha \otimes \beta) \,;\, \_ \\ \vdash (M^{\star}(\alpha \otimes \beta))[expand_{\theta_i}[\alpha][\beta]x_i / x_i] : \theta'(\alpha \otimes \beta) \end{aligned}$$

where $(M^{\star}(\alpha \otimes \beta))[expand_{\theta_i}[\alpha][\beta]x_i / x_i]$ results from substituting $expand[\alpha][\beta]x_i$ for each x_i. Notice that x is not replaced. Given this judgement we can first λ-abstract x and then Λ-abstract β, leading to the definition

$$(\lambda x : \theta \,.\, M)^{\star}\alpha \quad = \quad \Lambda\beta\, \lambda x : \theta^{\star}(\alpha \otimes \beta) \,.\, (M^{\star}\alpha)[expand_{\theta_i}[\alpha][\beta]x_i) / x_i] \,.$$

Note that the use of expansions makes the translation $M^{\star}$ dependent on the types of free identifiers; e.g. $\lambda x : \mathbf{comm} \,.\, y(z)$ gets translated differently, depending on the types of y and z. So to be precise the translation in fact defines terms $M_{\Gamma}^{\star}\alpha$, indexed by Idealized Algol type assignments Γ.

The remaining clauses are straightforward by comparison. A canonical isomorphism $j : \theta(\alpha \otimes I) \multimap \theta(\alpha)$ is used to deal with the function application in the translation of the fixed-point combinator.

$$
\begin{array}{rcl}
x^\star\alpha & = & x \\[1ex]
(\pi_i M)^\star\alpha & = & \pi_i(M^\star\alpha) \\[1ex]
\langle M, N\rangle^\star\alpha & = & \langle M^\star\alpha, N^\star\alpha\rangle \\[1ex]
0^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,[s, 0] \\[1ex]
(\mathbf{if}\, N_1 = 0\, \mathbf{then}\, N_2 & = & \underline{\lambda} s : \alpha\,.\,\mathbf{let}\ [s', y]\ \mathbf{be}\ N_1^\star\, s\ \mathbf{in} \\
\mathbf{else}\, N_3)^\star\alpha & & \qquad \mathbf{if}\, y = 0\, \mathbf{then}\, N_2^\star\alpha\, s'\, \mathbf{else}\, N_3^\star\alpha\, s' \\[1ex]
(\mathbf{succ}\ M)^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,\mathbf{let}\ [x, y]\ \mathbf{be}\ M^\star\alpha\, s\ \mathbf{in}\ [x, (\mathbf{succ}\ y)] \\[1ex]
(\mathbf{pred}\ M)^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,\mathbf{let}\ [x, y]\ \mathbf{be}\ M^\star\alpha\, s\ \mathbf{in}\ [x, (\mathbf{pred}\ y)] \\[1ex]
(Y_\theta\, M)^\star\alpha & = & j(Y_{\theta^\star(\alpha\otimes I)} M^\star\alpha[I]) \\[1ex]
(M; N)^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,N^\star\alpha\,(M^\star\alpha\, s) \\[1ex]
\mathbf{skip}^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,s \\[1ex]
(\mathbf{letval}_{\mathbf{comm}}\, M\, N)^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,\mathbf{let}\, [s', n]\ \mathbf{be}\ M^\star\alpha\, s\ \mathbf{in} \\
& & \qquad\qquad \mathbf{let}\ [s'', x]\ \mathbf{be} \\
& & \qquad\qquad\quad N^\star\alpha[I](\underline{\lambda} s : \alpha\otimes\beta\,.\,[s, n])\,[s', *] \\
& & \qquad\qquad \mathbf{in}\ (\mathbf{let}\ *\ \mathbf{be}\ x\ \mathbf{in}\ s'') \\[1ex]
(\mathbf{letval}_{\mathbf{exp}}\, M\, N)^\star\alpha & = & \underline{\lambda} s : \alpha\,.\,\mathbf{let}\, [s', n]\ \mathbf{be}\ M^\star\alpha\, s\ \mathbf{in} \\
& & \qquad\qquad \mathbf{let}\ [[s'', x], n']\ \mathbf{be} \\
& & \qquad\qquad\quad N^\star\alpha[I](\underline{\lambda} s : \alpha\otimes\beta\,.\,[s, n])\,[s', *] \\
& & \qquad\qquad \mathbf{in}\ (\mathbf{let}\ *\ \mathbf{be}\ x\ \mathbf{in}\ [s'', n'])
\end{array}
$$

The clauses for **letval** are syntactically complex, but the idea is just that first M is evaluated to get an integer value n and a new state s', and then N is evaluated in this new state with the constantly-n thunk passed as an argument. To realize this evaluation of N, we instantiate the β-component of

$$N^\star\alpha : \forall\beta\,.\,\mathbf{exp}^\star(\alpha\otimes\beta) \to \mathbf{comm}(\alpha\otimes\beta)$$

to I. Then, before returning either the resulting state (for $\mathbf{letval}_{\mathbf{comm}}$) or state paired with number (for $\mathbf{letval}_{\mathbf{exp}}$) we dispose of the I-valued component x.

5.2 Expansions

We define functions

$$expand_\theta : \forall\alpha\forall\beta\,.\,\theta^\star\alpha \multimap \theta^\star(\alpha\otimes\beta)$$

where α and β are different type variables. These functions will be given by closed terms. In the present setup, the Expansion Parametricity Lemma from

[7] is then a consequence of the definability of these expansions and the Logical Relations Lemma. The definition goes by induction on θ.

$$
\begin{aligned}
expand_{\mathbf{comm}} &= \Lambda\alpha\beta\,.\,\lambda c : \alpha \multimap \alpha\,.\,c \otimes id_\beta \\
expand_{\mathbf{exp}} &= \Lambda\alpha\beta\,.\lambda e : \alpha \multimap \alpha \otimes \mathbf{nat}\,.\,i1 \circ (e \otimes id_\beta) \\
expand_{\theta\times\theta'} &= \Lambda\alpha\beta\,.(expand_\theta[\alpha,\beta])\&(expand_{\theta'}[\alpha,\beta]) \\
expand_{\theta\longrightarrow\theta'} &= \Lambda\alpha\beta\,.\,\lambda p\,.\,\Lambda\gamma\,.\,i2 \circ p[\gamma] \circ i3
\end{aligned}
$$

Here we have used the shorthand of composition $\circ$ and the functorial action of $\otimes$ and &. The isomorphisms $i1 - i3$ are those obtained from evident unity, symmetry and associativity isomorphisms.

$$i1 : (\alpha \otimes \mathbf{nat}) \otimes \beta \multimap (\alpha \otimes \beta) \otimes \mathbf{nat}$$

$$i2 : \theta'((\alpha \otimes \beta) \otimes \gamma) \multimap \theta'(\alpha \otimes (\beta \otimes \gamma))$$

$$i3 : \theta((\alpha \otimes \beta) \otimes \gamma) \multimap \theta(\alpha \otimes (\beta \otimes \gamma))$$

The arrow type is defined according to the following diagram.

$$
\begin{tikzcd}
\theta(\alpha \otimes (\beta \otimes \gamma)) \arrow[r, "p[\beta \otimes \gamma]"] & \theta'(\alpha \otimes (\beta \otimes \gamma)) \arrow[d, "i2"] \\
\theta((\alpha \otimes \beta) \otimes \gamma) \arrow[u, "i3"] \arrow[r, "expand[\alpha][\beta]p[\gamma]"'] & \theta'((\alpha \otimes \beta) \otimes \gamma)
\end{tikzcd}
$$

5.3 Basic SCI

In the translation for Basic SCI we will gloss over isomorphisms for permuting $\otimes$. The translation is similar in many ways to the one for Idealized Algol: We concentrate on three differences.

The rule for application is of the form

$$\frac{x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta \to \theta' \quad x_{n+1} : \theta_1, \ldots, x_m : \theta_m \vdash N : \theta}{x_1 : \theta_1, \ldots, x_m : \theta_m \vdash MN : \theta'}$$

and the translation is

$$(MN)^\circ(\alpha_1, \ldots, \alpha_n) \;=\; \Big(M^\circ(\alpha_1, \ldots, \alpha_m)[\alpha_{m+1} \otimes \cdots \otimes \alpha_n]\Big)\Big(N^\circ(\alpha_{m+1}, \ldots, \alpha_n)\Big)$$

In this definition the β-component of

$$M^\circ(\alpha_1, \ldots, \alpha_m) : \forall\beta\,.\,\theta^\circ\beta \to \theta'^\circ(\alpha_1 \otimes \cdots \otimes \alpha_n \otimes \beta)$$

is instantiated to the type $\alpha_{m+1} \otimes \cdots \otimes \alpha_n$ of the state for N.

For parallel composition, given

$$\frac{x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \mathbf{comm} \quad x_n + 1 : \theta_1, \ldots, x_m : \theta_n \vdash N : \mathbf{comm}}{x_1 : \theta_1, \ldots, x_m : \theta_m \vdash M \parallel N : \mathbf{comm}}$$

we translate as follows:

$$(M \parallel N)^\circ(\alpha_1,\ldots,\alpha_n) \;=\; \lambda s:\alpha \;\; \begin{array}{l}\mathbf{let}\ [z_1,\ldots,z_m]\ \mathbf{be}\ s\ \mathbf{in}\\ \Big[M^\circ[\alpha_1,\ldots,\alpha_n][z_1,\ldots,z_n]\\ ,N^\circ[\alpha_{n+1},\ldots,\alpha_m][z_{n+1},\ldots,z_m]\Big]\end{array}$$

This interpretation splits the state into disjoint components $z_1,\ldots,z_n$ and $z_{n+1},\ldots,z_m$ that are acted upon independently by M and N.

Finally, the equation for **new** is the same as in Idealized Algol, except that the variable $v[\alpha]$ is replaced by $v : ((\mathbf{exp} \to \mathbf{comm}) \times \mathbf{exp}))^\circ(\mathbf{nat})$ given by $\langle assign, lookup\rangle$, where

$$\begin{array}{ll} assign & = \Lambda\beta\ \lambda x : \mathbf{exp}^\circ(\mathbf{nat} \otimes \beta)\,.\,\underline{\lambda} s : (\mathbf{nat} \otimes \beta)\,.\\ & \quad \mathbf{let}\ [[n,b],m]\ \mathbf{be}\,xs\ \mathbf{in}\\ & \quad\ \ \mathbf{let}\ *\ \mathbf{be}\ (\mathbf{discard}\,n)\ \mathbf{in}\ [m,b]\\ lookup & = \underline{\lambda}\, n : \mathbf{nat}\,.\,\mathbf{copy}\,n \end{array}$$

There is no longer a need for the α component of v because procedures and arguments don't interfere in Basic SCI.

6 Strict Relational Parametricity

In order to analyze our translations we consider a model of the linear polymorphic language based on strict continuous functions and relational parametricity. This model actually supports contraction, though it does not support weakening. Such a simple model is useful for our purpose, it seems, because problematic examples like snapback in imperative languages typically use *both* weakening and contraction. Removing just one of weakening or contraction banishes the specific examples. (But we will return to this point in Section **??**.)

6.1 Semantics of Types

In the strict function model types are interpreted as cpo's, i.e., complete partial orders possessing a least element. More specifically, a type A determines a function $[\![A]\!]$ from type environments to cpo's, where a type environment $\mathcal{D}$ maps type variables to countable, flat cpo's. Most type constructors are interpreted pointwise by thier cpo counterparts.

$$\begin{array}{lcl} [\![\alpha]\!]_{\mathcal{D}} & = & \mathcal{D}\alpha\\ [\![I]\!]_{\mathcal{D}} & = & I\\ [\![\mathbf{nat}]\!]_{\mathcal{D}} & = & N_\perp\\ [\![!A]\!]_{\mathcal{D}} & = & ([\![A]\!]_{\mathcal{D}})_\perp\\ [\![\sigma \otimes \sigma']\!]_{\mathcal{D}} & = & [\![\sigma]\!]_{\mathcal{D}} \otimes [\![\sigma']\!]_{\mathcal{D}}\\ [\![A \,\&\, B]\!]_{\mathcal{D}} & = & [\![A]\!]_{\mathcal{D}} \,\&\, [\![A]\!]_{\mathcal{D}}\\ [\![A \to B]\!]_{\mathcal{D}} & = & [\![A]\!]_{\mathcal{D}} \to [\![B]\!]_{\mathcal{D}}\\ [\![A \multimap B]\!]_{\mathcal{D}} & = & [\![A]\!]_{\mathcal{D}} \multimap [\![B]\!]_{\mathcal{D}} \end{array}$$

On the right-hand side I is the two-point cpo, $(\cdot)_\perp$ is lifting, $\otimes$ is smash product, & is cartesian product, $\multimap$ is strict continuous function space (pointwise ordered), and $\to$ is continuous function space. To define $\forall$ we make use of an auxiliary relational semantics as in [12].

Suppose $\mathcal{R}$ (a relation environment) is a function mapping each type variable into a pointed binary relation between flat cpo's such that

$$\mathcal{R}(\alpha) : \mathcal{D}\alpha \leftrightarrow \mathcal{D}'\alpha, \quad \text{all } \alpha$$

for type environments $\mathcal{D}, \mathcal{D}'$. Then we define a complete relation

$$[\![A]\!]_{\mathcal{R}} : [\![A]\!]_{\mathcal{D}} \leftrightarrow [\![A]\!]_{\mathcal{D}'}$$

as follows.

$$\begin{array}{rcl}
[\![\alpha]\!]_{\mathcal{R}} & = & \mathcal{R}\alpha \\
[\![\mathbf{nat}]\!]_{\mathcal{R}} & = & \{\langle n, n\rangle \mid n \in N_{\perp}\} \\
[\![!A]\!]_{\mathcal{R}} & = & \{\langle x, y\rangle \mid \langle x, y\rangle \in [\![A]\!]_{\mathcal{R}} \ \vee \ x = y = \perp\} \\
[\![\sigma \otimes \sigma']\!]_{\mathcal{R}} & = & \{\langle [a, a'], [b, b']\rangle \mid \langle a, b\rangle \in [\![\sigma]\!]_{\mathcal{R}} \ \wedge \ \langle a', b'\rangle \in [\![\sigma']\!]_{\mathcal{R}}\} \\
[\![A \,\&\, B]\!]_{\mathcal{R}} & = & \{\langle \langle a, a'\rangle, \langle b, b'\rangle\rangle \mid \langle a, b\rangle \in [\![A]\!]_{\mathcal{R}} \ \wedge \ \langle a', b'\rangle \in [\![B]\!]_{\mathcal{R}}\} \\
[\![A \to B]\!]_{\mathcal{R}} & = & \{\langle f, f'\rangle \mid \forall \langle a, a'\rangle \in [\![A]\!]_{\mathcal{R}} \,.\, \langle fa, f'a'\rangle \in [\![B]\!]_{\mathcal{R}}\} \\
[\![A \multimap B]\!]_{\mathcal{R}} & = & \{\langle f, f'\rangle \mid \forall \langle a, a'\rangle \in [\![A]\!]_{\mathcal{R}} \,.\, \langle fa, f'a'\rangle \in [\![B]\!]_{\mathcal{R}}\}
\end{array}$$

The relational actions of $\to$ and $\multimap$ use the same clauses, the difference being that the clause for $\multimap$ assumes that both f and f' are strict. In the definition of $\otimes$, $[d, e]$ is $\perp$ if d or e is $\perp$. We often refer to the relational action of type constructors directly, e.g. writing $R \otimes S$ instead of $[\![\alpha \otimes \beta]\!](\alpha \mapsto R, \beta \mapsto S)$.

We can now define the cpo and relational semantics for $\forall$. For this, let $Level1$ stand for the collection of countable, flat cpo's.

$$\begin{array}{rcl}
[\![\forall\alpha.A]\!]_{\mathcal{D}} & = & \{p \in \prod_{d \in Level1} [\![A]\!](\mathcal{D} \mid \alpha \mapsto d) \\
& & \quad \forall d, d' \in T \ , \ \forall r : d \leftrightarrow d' \,.\, \langle p[d], p[d']\rangle \in [\![A]\!]_{\mathcal{I}_{\mathcal{D}} \mid \alpha \mapsto r} \\
& & \}, \text{ ordered pointwise}
\end{array}$$

$$\langle p, p'\rangle \in [\![\forall\alpha.A]\!]_{\mathcal{R}} \quad \Leftrightarrow \quad \forall d, d' \in Level1 \ , \ \forall r : d \leftrightarrow d' \,.\, \langle p[d], p'[d']\rangle \in [\![A]\!]_{\mathcal{R} \mid \alpha \mapsto r}$$

Here $\mathcal{I}_{\mathcal{D}}$ maps each α to the identity relation on $\mathcal{D}\alpha$.

For the well-definedness of this definition it is crucial that $Level1$ is essentially small – there are only set-many countable, flat cpo's (at least up to isomorphic copies). Because we use predicative polymorphism it is possible to obtain a parametric model without relying on devices such as PERs. With these definitions one may verify that each $[\![T]\!]_{\mathcal{D}}$ is a cpo and each $[\![T]\!]_{\mathcal{R}}$ is a complete relation on cpo's. This ensures that the least fixed-point operator exists in the model.

By "parametric model" we mean what in [12] was referred to as the Identity Extension Lemma:

$$[\![A]\!]_{\mathcal{I}_{\mathcal{D}}} \ = \ \{\langle a, a\rangle \mid a \in [\![A]\!]_{\mathcal{D}}\}$$

This is usually taken as the defining characteristic of relational parametricity, and the definition of $\forall$ is arranged precisely to ensure the identity property (which is included as part of the Isomorphism Lemma).

Lemma 1 (Isomorphism Functoriality Lemma) *For any type A, the relational action of $[\![A]\!]$ is functorial on isomorphisms. That is, (i) if $\mathcal{R}\alpha$ is the graph of an isomorphism between flat cpo's for each α free in A then $[\![A]\!]_{\mathcal{R}}$ is the graph of an isomorphism, and (ii) $[\![A]\!]$ preserves identities and composites of isomorphisms.*

6.2 Semantics of Terms

A typing judgement

$$x_1 : A_1, \ldots, x_n : A_n; B_1, \ldots, B_m \vdash M : C$$

is interpreted by a $\mathcal{D}$-indexed family of strict continuous functions

$$[\![M]\!]_{\mathcal{D}} : ([\![A_1]\!]_{\mathcal{D}})_{\perp} \otimes \cdots \otimes ([\![A_n]\!]_{\mathcal{D}})_{\perp} \otimes [\![B_1]\!]_{\mathcal{D}} \otimes \cdots \otimes [\![B_m]\!]_{\mathcal{D}} \longrightarrow [\![C]\!]_{\mathcal{D}}$$

We omit the detailed definition, which is standard, but we do describe the uniformity property of this family. Relations extend from types to typing contexts. using the action of $\otimes$ and $!$ on relations.

$$\begin{array}{rl} & [\![x_1 : A_1, \ldots, x_n : A_n; y_1 : B_1, \ldots, y_m : B_m]\!]_{\mathcal{R}} \\ = & [\![!A_1]\!]_{\mathcal{R}} \otimes \cdots [\![!A_n]\!]_{\mathcal{R}} \otimes [\![B_1]\!]_{\mathcal{R}} \otimes \cdots \otimes [\![B_m]\!]_{\mathcal{R}} \end{array}$$

Lemma 2 (Logical Relations Lemma) *If*

$\Gamma; \Delta \vdash t : A$

$\mathcal{R}\alpha : \mathcal{D}\alpha \leftrightarrow \mathcal{D}'\alpha$, *for all* α

$\langle \eta, \eta' \rangle \in [\![\Gamma; \Delta]\!]_{\mathcal{R}}$,

then

$\langle\, [\![t]\!]_{\mathcal{D}}\eta\, ,\, [\![t]\!]_{\mathcal{D}'}\eta'\, \rangle \in [\![A]\!]_{\mathcal{R}}$

Since logical relations are used to constrain $\forall$ types, the lemma and the well-definedness of the maps $[\![M]\!]_{\mathcal{D}}$ are proven simultaneously by induction on M.

6.3 On the Relational Action of $\otimes$

The relational action of $\otimes$ is unusual and deserves comment.

First, notice that:

$\langle a, b \rangle R \otimes S \langle c, d \rangle$ does not imply aRc and bSd.

For instance if $\perp Rc$ and bSd then $\langle \perp, \perp \rangle R \otimes S \langle c, d \rangle$ even when $\langle \perp, d \rangle \notin S$.

Second, the formula for $R \otimes S$ does not produce a complete relation, for complete relations R and S on arbitrary cpo's. For example, consider relations $R : N_{\perp} \leftrightarrow I$ and $S : N_{\perp} \leftrightarrow Vnat$ where $R = \{\langle \perp, \perp \rangle, \langle \perp, * \rangle\}$ and $S = \{\langle \perp, 0 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \ldots\}$ and $Vnat$ is the vertical natural numbers. Then $R \otimes S$ contains a chain of all tuples $\langle \langle \perp, \perp \rangle, \langle *, n \rangle \rangle$, for $n > 1$, but it does not contain the lub $\langle \langle \perp, \perp \rangle, \langle *, \infty \rangle \rangle$. Nevertheless, $R \otimes S$ is strict and trivially complete whenever R and S are strict relations on flat (or finight height) cpo's. In the semantics of our predicatively polymorphic language $\otimes$ is only ever applied to flat cpo's, so the definition of the relational action is suitable for this language.

The following two properties are crucial ones satisfied by the definition.

- If $\langle f, g\rangle \in R \multimap S$ and $\langle f', g'\rangle \in R' \multimap S'$ then $\langle f \otimes f', g \otimes g'\rangle \in R \otimes R' \multimap S \otimes S'$.
- $R \otimes S \multimap G \;\cong\; R \multimap (S \multimap G)$.

The first property connects the functorial and relational actions of $\otimes$, and is needed for the interpretation of the elimination rule for $\otimes$ to satisfy relevant parametricity properties. The second is a relationship we expect between $\otimes$ and $\multimap$.

Pitts [9] has emphasized that one can entertain a number of other possible relational actions for $\otimes$, and a number of notions of "relation." One alternative notion of relation is as (complete) subsets of $D \otimes E$. Such relations are not suitable for our purpose because many properties in this paper, including the examples and results in the next three sections, rely on being able to relate $\perp$ and non-$\perp$ elements. The definitions of $\otimes$ that Pitts considers are each appropriate to a different $\multimap$ than our "logical relations" interpretation.

Finally, note that a definition that doesn't work is the componentwise one:

$$R \otimes S \;=\; \{\langle\langle a, b\rangle, \langle a', b'\rangle\rangle \in (A \otimes B) \times (A' \otimes B') \mid aRa' \;\wedge bSb'\}$$

For one thing, this definition fails to satisfy the first property noted above connecting the functorial and relational actions of $\otimes$.

This discussion simply serves to illustrate that care is needed when defining relational parametricity in the presence of $\otimes$.

7 Semantics Examples

In this and what follows we often mix notation for polymorphic types and cpo's. For instance, we write

$$\forall \beta \,.\, (S \otimes \beta \multimap S \otimes \beta) \to (S \otimes \beta \multimap S \otimes \beta)$$

instead of the more cumbersome

$$[\![\forall \beta \,.\, (\alpha \otimes \beta \multimap \alpha \otimes \beta) \to (\alpha \otimes \beta \multimap \alpha \otimes \beta)]\!](\alpha \mapsto S)$$

Example 1. We consider the snapback operator of type

$$\forall \alpha \,.\, (\alpha \multimap \alpha \otimes \mathbf{nat}) \to (\alpha \multimap \alpha \otimes \mathbf{nat}) \;.$$

Were it to exist in the model, this operator would satisfy the following property

$$\mathit{snap}[S_\perp] e\, s \;=\; \begin{cases} \langle s, n\rangle & \text{if } e(s) = \langle s', n\rangle \\ \perp & \text{if } e(s) = \perp \end{cases}$$

for countable sets S and $s \in S$. We show that snap is not parametric.

Consider any $p \in [\![\forall \alpha \,.\, (\alpha \multimap \alpha \otimes \mathbf{nat}) \to (\alpha \multimap \alpha \otimes \mathbf{nat})]\!]$. We show the following property:

If	$c : 2_\perp \multimap 2_\perp \otimes N_\perp$ is such that $c(x) = \langle 1, 1\rangle$ when $x \neq \perp$,
and	$p[2_\perp]c0 = \langle 0, 1\rangle$
then	$p[2_\perp]\perp 0 = \langle 0, 1\rangle$

Here, $2 = \{0, 1\}$. The property says that if you use c at all, the final state can no longer be 0: Using c causes an irreversible state change.

To prove the property, consider the relation $R : 2_\perp \leftrightarrow 2_\perp$ consisting of tuples $\langle 0, 0\rangle$, $\langle \perp, \perp\rangle$, $\langle 1, \perp\rangle$. Then $\langle c, \perp\rangle \in [R \multimap R \otimes \Delta_{N_\perp}]$ and by parametricity $\langle p[2_\perp]c0, p[2_\perp]\perp 0\rangle \in R$. Since $\langle 0, 1\rangle$ is only $R \otimes \Delta_{2_\perp}$-related to itself the property follows. In contrast, the property is not satisfied by *snap* because $p[2_\perp]\perp 0$ can be $\perp$ even when the preconditions on p and c are satisfied. Thus, *snap* does not exist in the model.

The argument for the non-parametricity of snapback is in fact a transcription of the failure of weakening in the category of cpo's and strict functions. Specifically, for any cpo D there is a candidate weakening map $w_D : D \multimap I$ that takes $\perp$ to $\perp$ and all other elements of D to the non-$\perp$ element of I. Though this map exists for all D, it is not natural, in that

$$\begin{tikzcd} D \arrow[r, "w_D"] \arrow[d, "f"'] & I \\ E \arrow[ur, "w_E"'] & \end{tikzcd}$$

does not commute when f takes a non-$\perp$ element to $\perp$. Re-expressing this in relational form, using the graph of such an f, shows that the family w_D of maps, indexed by $D \in Level1$, is not in $[\![\forall\alpha\,.\,\alpha \multimap I]\!]$.

The failure of the naturality of this weakening map is reminiscient of our contention that it is general, or polymorphic, snapback that does not exist in Algol, whereas specific state changes can often be reversed.

Example 2. The reasoning in Example 1 can be used more positively, to verify the equivalence discussed in section 2.

$$\begin{array}{l} \textbf{new}\ \lambda x\,.\ x := 0; \\ \qquad\qquad P(x := x + 1); \\ \qquad\qquad \textbf{if}\,(x > 0)\,\textbf{then diverge} \end{array} \quad \equiv \quad P(\textbf{diverge})$$

To show the equivalence we argue that the translations

$$\begin{array}{ll} (4.1) & \underline{\lambda} s : \alpha\,.\ \textbf{let}\ [s', n]\ \textbf{be}\ p[\textbf{nat}](id_\alpha \otimes \underline{\lambda} n.n + 1)\,[s, 0]\ \textbf{in} \\ & \qquad\qquad \textbf{let}\ *\ \textbf{be}\ (\textbf{discard}\ n)\ \textbf{in}\ (\textbf{if}\ n > 0\ \textbf{then}\ \Omega_{\alpha \multimap \alpha}\, s\ \textbf{else}\ s) \end{array}$$

$$\begin{array}{ll} (4.2) & \underline{\lambda} s : \alpha\,.\ \textbf{let}\ [s', n]\ \textbf{be}\ p[I](\Omega_{(\alpha \otimes I) \multimap (\alpha \otimes I)})\,[s, *]\ \textbf{in} \\ & \qquad\qquad \textbf{let}\ *\ \textbf{be}\ (\textbf{discard}\ n)\ \textbf{in}\ s \end{array}$$

of the two Algol terms are equivalent, where

$$p : \forall\beta\,.\,(\alpha \otimes \beta \multimap \alpha \otimes \beta) \rightarrow (\alpha \otimes \beta \multimap \alpha \otimes \beta)$$

Notice the use of I in (4.2): To cope with a call of p that isn't within a **new** we extend the state with the identity for $\otimes$, which is essentially not to extend the state at all.

Let $S_\perp$ be a countable, flat cpo (the denotation of α) and consider the relation $R : N_\perp \leftrightarrow I$ consisting of the tuples $\langle \perp, \perp \rangle$, $\langle 0, * \rangle$, and $\langle i, \perp \rangle$ for $i > 0$. Let $c : S_\perp \otimes N_\perp \multimap S_\perp \otimes N_\perp$ be $id_{S_\perp} \otimes succ$, where $succ$ is the strict extension of successor. Then

$$\langle c, \perp \rangle \in I_{S_\perp} \otimes R \multimap I_{S_\perp} \otimes R$$

As a result, for arbitrary p we can use the parametricity property to get

$$\langle p[N_\perp]c[s, 0]\,,\, p[I]\perp[s, *] \rangle \in I_{S_\perp} \otimes R\ .$$

Next, for any $s \in S$ suppose that $p[N_\perp]c[s, 0] = [s', n]$ and $p[I]\perp[s, *] = [s'', x]$. Using $\langle [s', n], [s'', x] \rangle \in I_{S_\perp} \otimes R$ we can argue that (4.1)=(4.2) by cases as follows.

- If $[s', n] = \perp$ then $[s'', x] = \perp$ by the definition of R and $\otimes$, and so (4.1)s=(4.2)s
- If $p[N_\perp]c[s, 0] = [s', n] \neq \perp$ then there are two subcases.
  - If $n = 0$ then, by the definition of R, we must have that $[s'', x] = [s', *]$ and obviously (4.1)s=(4.2)s since both will, when beginning in state s, return s'.
  - If $n > 0$ then we know that (4.1) applied to s returns $\perp$. But since $n > 0$ we know, from the definition of R and parametricity, that $x = \perp$, and so both (4.1) and (4.2) return $\perp$ when started in state s.

Many other equivalences can be treated using this form of reasoning. Typically, one finds a relation between pieces of local state and implementations of different objects. For example, to show the equivalence of two "counter classes"

$$\begin{array}{lcl} \mathbf{new}\ \lambda x\ x := 0; & & \mathbf{new}\ \lambda x\ x := 0; \\ \qquad P(x := x + 1, x) & \equiv & \qquad P(x := x + 2, x/2) \end{array}$$

for $P : \mathbf{comm} \times \mathbf{exp} \rightarrow \mathbf{comm}$, we simply use a relation on their local states $R : N_\perp \leftrightarrow N_\perp$ consisting of $\langle \perp, \perp \rangle$ and all pairs $\langle i, 2i \rangle$.

This example, taken from [7], illustrates that the strict-function model provides an extension of the reasoning methods presented there; the additional relations are simply those that relate non-$\perp$ states to $\perp$. This seemingly small extension has a significant impact, especially on the structure of the types.

Example 3. The polymorphic type $(\mathbf{comm} \rightarrow \mathbf{comm})^\star I$ corresponds to closed terms of type $\mathbf{comm} \rightarrow \mathbf{comm}$ in Idealized Algol. We can calculate its structure as follows.

$$\begin{array}{cl} & (\mathbf{comm} \rightarrow \mathbf{comm})^\star I \\ = & \forall \beta\,.\,((I \otimes \beta) \multimap (I \otimes \beta)) \rightarrow ((I \otimes \beta) \multimap (I \otimes \beta)) \\ \cong & \forall \beta\,.\,(\beta \multimap \beta) \rightarrow (\beta \multimap \beta) \\ \cong & N_\perp \end{array}$$

where the last isomorphism uses a parametricity argument in the strict function model. On the level of Idealized Algol we obtain a correspondence

$$\begin{array}{rcl} n & \mapsto & \lambda c : \mathbf{comm} \, . \, c^n \\ \perp & \mapsto & \lambda c : \mathbf{comm} \, . \, \mathbf{diverge} \end{array}$$

where $c^0 = \mathbf{skip}$ and $c^{i+1} = c; c^i$. This representation of $\mathbf{comm} \to \mathbf{comm}$ should be compared to [7, 14], where $(\mathbf{comm} \to \mathbf{comm})I$ is $N_\perp \otimes Vnat^{op}$, with $Vnat^{op}$ the vertical natural numbers flipped upside-down. The $Vnat$ component was concerned exclusively with a form of snapback.

It is instructive to examine the parametricity argument giving the final isomorphism with $N_\perp$. Let $p \in [\![\forall\beta \, . \, (\beta \multimap \beta) \to (\beta \multimap \beta)]\!]$ and consider any D, $d \in D$, and $c : D \multimap D$. Let $R : N_\perp \leftrightarrow D$ be the relation consisting of pairs $\langle \perp, \perp \rangle$ and $\langle m, c^m(d) \rangle$. Then $\langle succ, c \rangle \in R \multimap R$ and, by parametricity, $(p[N_\perp] \, succ \, 0) \, R \, (p[D] \, c \, d)$. If $(p[N_\perp] \, succ \, 0) = \perp$ then the definition of R ensures that $(p[D] \, c \, d) = \perp$, and so $p = \perp$. If $(p[N_\perp] \, succ \, 0) = n \neq \perp$ then the definition of R ensures that $(p[D] \, c \, d) = c^n d$, and so p is the n'th Church numeral. Thus $[\![\forall\beta \, . \, (\beta \multimap \beta) \to (\beta \multimap \beta)]\!]$ is iso to $N_\perp$.

In this argument note that R is strict and single-valued. Note also that n is related to $\perp$ when $c^n d = \perp$. In [7] relations that relate $\perp$ to non-$\perp$ elements were not considered, which is why this argument could not go through.

Example 4 $(\mathbf{comm} \to \mathbf{comm})^\star I$ turns out to be the same as the SCI interpretation $(\mathbf{comm} \to \mathbf{comm})^\circ I$. However, when we replace I by a nontrivial flat cpo S a difference appears. We calculate

$$\begin{array}{rl} & (\mathbf{comm} \to \mathbf{comm})^\circ S \\ = & \forall\beta \, . \, (\beta \multimap \beta) \to ((S \otimes \beta) \multimap (S \otimes \beta)) \\ \cong & S \multimap (S \otimes N_\perp) \end{array}$$

where the last isomorphism is again a straightforward parametricity argument. In the cpo $S \multimap (S \otimes N_\perp)$ the $N_\perp$ part corresponds to use of the β-component, the number of times the input command is executed (so this part is essentially a Church numeral). This cpo illustrates how the computation on the S and β components can be carried out independently: The input state can affect the number of times the command argument is used, but not what happens on each use. Equally, none of the uses of the command argument affects the S-component, which is why there is a single transformation from input state to output that is indepent of β. This independence is reflected in certain equivalences between terms, such as between functions $\lambda c \, . \, c; x := 1$ and $\lambda c \, . \, x := 1; c$ of type $\mathbf{comm} \to \mathbf{comm}$.

The same type in Idealized Algol translates as follows.

$$\begin{array}{rl} & (\mathbf{comm} \to \mathbf{comm})^\star S \\ = & \forall\beta \, . \, (S \otimes \beta \multimap S \otimes \beta) \to ((S \otimes \beta) \multimap (S \otimes \beta)) \end{array}$$

The representation in terms of $S \multimap (S \otimes N_\perp)$ no longer works due to the possibility of interference between procedure and argument in Idealized Algol. For instance, the terms $\lambda c\,.\,c; x := 1$ and $\lambda c\,.\,x := 1; c$ can be now distinguished by being applied to an argument that interferes with x.

An explicit description of the cpo for $(\mathbf{comm} \to \mathbf{comm})^{\star}S$ can be given: It is (an initial solution of) the familiar domain of deterministic resumptions

$$D \;\cong\; S \multimap (S \oplus (S \otimes D))$$

where $\oplus$ is coalesced sum. With this domain equation we understand a procedure $\lambda c\,.\,C$ as a process that can change the state and either immediately terminate, in the S component, or call c and resume. The use of an intermediate state in the right branch of $\oplus$, together with the new input state expected in the resumed D, models the dependence between procedure and argument.

The proof that $(\mathbf{comm} \longrightarrow \mathbf{comm})^{\star}S$ in the strict parametricity model is an initial solution of the domain equation for resumptions is nontrivial, but may be inferred as a special case of a result in the next section.

Example 5. Next we consider a type for functions with two arguments.

$$\begin{array}{rl} & (\mathbf{comm} \times \mathbf{comm} \longrightarrow \mathbf{comm})^{\star}I \\ \cong & \forall\beta\,.\,(\beta \multimap \beta) \times (\beta \multimap \beta) \longrightarrow (\beta \multimap \beta) \\ \cong & (\mathit{list}\{1,2\})_{\perp} \end{array}$$

A list ℓ of 1's and 2's corresponds to $\lambda c : \mathbf{comm} \times \mathbf{comm}\ .\ c^{\ell}$ where $c^{\epsilon} = \mathbf{skip}$ and $c^{i,\ell'} = (\pi_i c); c^{\ell'}$, for $i = 1, 2$. The parametricity argument showing the last isomorphism is essentially the same as in Example 3.

With this representation we can see which argument an element of the type evaluates first: i, ℓ' evaluates its i'th argument first. It follows that certain counter-intuitive functions do not exist, such as a polling operation given by

$$\mathit{poll}[S_{\perp}]\langle c_0, c_1\rangle\, s \;=\; \begin{cases} s & \text{if } c_0 s \neq \perp \text{ and } c_1 s \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

This operation exhibits both snapback and the kind of determinate parallelism found in the continuous function model of PCF.

Example 6. We consider a curried function type to illustrate further the semantic impact of noninterference between procedure and argument in SCI.

$$\begin{array}{rl} & (\mathbf{comm} \to \mathbf{comm} \longrightarrow \mathbf{comm})^{\circ}S \\ \cong & \forall\beta\gamma\,.\,(\beta \multimap \beta) \times (\gamma \multimap \gamma) \longrightarrow (S \otimes \beta \otimes \gamma \multimap S \otimes \beta \otimes \gamma) \\ \cong & S \multimap (S \otimes N_{\perp} \otimes N_{\perp}) \end{array}$$

In this case computation in the β, γ, and S components can all be carried out independently, being again Church numerals in β and γ. If we consider the case where $S = I$ then we obtain a correspondence between $N_{\perp} \otimes N_{\perp}$ and closed SCI terms of type $\mathbf{comm} \to \mathbf{comm} \to \mathbf{comm}$:

$$\begin{array}{lcl} \langle n, m\rangle & \mapsto & \lambda c_1\,.\,\lambda c_2\,.\ c^n \parallel c^m \\ \perp & \mapsto & \lambda c_1\,.\,\lambda c_2\,.\ \mathbf{diverge} \end{array}$$

A representation for $(\mathbf{comm} \to \mathbf{comm} \to \mathbf{comm})^{\star}S$ requires a form of resumption domain once more to account for the possibility of interference. In comparison to Idealized Algol, Examples 4 and 6 show a striking economy in the structure of SCI types.

8 Representations of First-order Types

9 A Full Abstraction Result

10 Objects in Linear λ-calculus

11 Conclusion

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF (extended abstract). pages 1–15.

[2] A. Barber. Dual intuitionistic linear logic. Draft, August 1995.

[3] S. Brookes, M. Main, A. Melton, and M. Mislove, editors. *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*, Tulane University, New Orleans, Louisiana, March 29–April 1 1995. Elsevier Science.

[4] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. 133 pages, ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Luke.Ong/, 1994.

[5] P. Naur (ed.), J. W. Backus, et al. Report on the algorithmic language ALGOL 60. *Comm. ACM*, 3(5):299–314, 1960. Also *Numerische Mathematik* 2:106–136.

[6] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In Brookes et al. [3].

[7] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, May 1995.

[8] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.

[9] A. M. Pitts. Relational properties of recursively defined domains. In *Proceedings, 8th Annual IEEE Symposium on Logic in Computer Science*, pages 86–97, Montreal, Canada, 1993. IEEE Computer Society Press, Los Alamitos, California.

[10] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM, New York.

[11] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland, Amsterdam.

[12] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.

[13] J. C. Reynolds. Preliminary design of the programming language Forsythe. Report CMU–CS–88–159, Computer Science Department, Carnegie Mellon University, June 21, 1988.

[14] K. Sieber. Full abstraction for the second-order subset of an Algol-like language. *Theoret. Comput. Sci.*, 1995. To appear.