From Algol to Polymorphic Linear Lambda-calculus

Peter W. O'Hearn
Queen Mary & Westfield College
and
John C. Reynolds
Carnegie Mellon University

In a linearly-typed functional language one can define functions that consume their arguments in the process of computing their results. This is reminiscent of state transformations in imperative languages, where execution of an assignment statement alters the contents of the store. We explore this connection by translating two variations on Algol 60 into a purely functional language with polymorphic linear types. On one hand the translations lead to a semantic analysis of Algol-like programs, in terms of a model of the linear language. On the other hand they demonstrate that a linearly-typed functional language can be at least as expressive as Algol.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: Parametric polymorphism, logical relations, local state, linear logic

Contents

Name: P.W. O'Hearn
Address: Department of Computer Science, Queen Mary and Westfield College, London E1 4NS, UK. e-mail: ohearn@dcs.qmw.ac.uk
Name: J.C. Reynolds
Address: School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213-3891, USA. e-mail: john.reynolds@cs.cmu.edu

1. INTRODUCTION

Traditional denotational semantics models imperative languages using state-to-state functions [Scott and Strachey 1971; Tennent 1991]. This approach successfully accounts for the fact that storage variables take on different values at different times during computation, but it does not cope nearly as well with the idea that a state change destructively alters the contents of the store.

To see the difficulty, suppose we use a function $p : \mathit{States} \to \mathit{States} \times \mathit{Values}$ to model the behaviour of an expression with side effects. Because a state is treated as a value like any other we are free, in the semantics, to manipulate any such function in a decidedly non-imperative manner. For instance, we can define a function

$$snap : (States \rightarrow States \times Values) \rightarrow (States \rightarrow States \times Values)$$

as follows:

$$snap(p)s = [s, v], \quad \text{where } ps = [s', v].$$

(Here we are ignoring issues of non-termination.) This *snap* operator is a perfectly reasonable mathematical function, but if we try to read it in an imperative fashion it contradicts the intuitive understanding of a state transformation as altering the store. Instead, it displays a "snapback" effect; $snap(p)$ proceeds by executing p, producing a new state s' and value v, and then snapping the state back to its initial value s. The use of p here does not destroy the initial state s.

In this paper we present an approach that better captures the imperative character of state transformations. The approach is based on a combination of *linear typing* and *parametric polymorphism*, and is given, formally, via syntactic translations from two variations on Algol 60 into a linear polymorphic λ-calculus. The translations are based on the idea that a program is linearly polymorphic in the type of the state; this allows for a subtle interplay between the copyability of specific values put into the store, and the inability of a program to copy the entire store. We analyze the translations using a model of the target language.

Although our analysis mainly focuses on the resulting semantics of the source languages, the translations can be regarded as well as indicating the imperative nature of the target language. That is, although the linear polymorphic calculus is a purely functional language, the translations can be regarded as providing an imperative reading of a range of types in the functional target.

Before continuing we would like to stress that the "problem" with the traditional semantics should be understood in its historical context. Indeed, Strachey on a number of occasions emphasized the fundamentally different way that the state and environment are used. For example:

> "The state transformation produced by obeying a command is essentially irreversible and it is, by the nature of the computers we use, impossible to have more than one version of [the state] available at any one time."
> C. Strachey [1972]

And Scott identified the non-copyability of state as crucial:

> "We will be tempted to copy ρ [the environment], but we will never generally feel free to ask for a copy of the *whole* computer store – there is just no room for that." D.S. Scott [1972]

But, while Scott and Strachey's prose vividly distinguished the state from the environment, in 1972 the theoretical techniques were not yet in place to allow for a precise description of the imperative, or irreversible, nature of state change, as expressed informally by them.

In 1975, one of the authors (Reynolds) attempted to use the polymorphic λ-calculus [Girard 1972; Reynolds 1974] to describe Algol, discovering much of the translation we will exhibit in Sections 4.1, 5.1, and 5.2. At the time, this seemed to be a quixotic effort to define a well-understood language in terms of a less understood one. (The author overlooked the fact that the translation avoided impredicativity.)

The intervening years, however, have seen the development of a relational semantics of polymorphism [Reynolds 1983]; possible world semantics of imperative languages [Reynolds 1981b; Oles 1982]; a connection between polymorphism and local state [O'Hearn and Tennent 1995]; and linear logic [Girard 1987]. Drawing upon all of these developments, we are now able to give a refined translation of Algol-like languages into the *linear* polymorphic λ-calculus that, when coupled with a relational semantics for the latter language, gives a more abstract description of Algol that earlier formulations of its semantics.

The translations are essentially a recasting of the the functor-category semantics developed by Reynolds [1981b] and Oles [1982] in the early eighties, using a linear polymorphic λ-calculus in place of a functor category. Their store shapes are replaced by type variables, natural transformations are replaced by polymorphic functions, and state-to-state functions are replaced by linear functions. This use of polymorphism is as in the parametric-functor semantics of O'Hearn and Tennent [1995], but refined by the use of linearity.

In the remainder of this Introduction we give an extended, informal, description of the main elements underlying our approach.

1.1 Linear Typing and State Transformations

The central idea, on which linear logic hinges, is that of a *linear function*. The guiding intuition is that a linear function "uses" its argument exactly once; as a result, it cannot freely copy or ignore its argument, because doing so would violate the use criterion. One often speaks also of a linear function as "consuming" its argument in the process of producing its result. The connection between use and consumption is that, after a linear function has used its argument once, the argument is no longer available, because to use it again would constitute two uses. The problem with snapback is that it uses the initial state twice, once when producing an intermediate result and again when producing a final answer. Thus, it is not linear in its state argument.

Linear logic is based on Girard's identification of the structural rules of logic as a source of discarding and copying data [Girard 1987]:

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \text{Weakening} \qquad \frac{\Gamma, B, B \vdash A}{\Gamma, B \vdash A}\ \text{Contraction.}$$

Weakening introduces a dummy assumption: In computational terms it may be understood as transforming a computation depending on Γ into a computation depending on Γ and B, but which ignores B. In Girard's resource description of logical rules, the ignoring of the B component involves the discarding of a datum of type B. Similarly, Contraction involves copying: From a computation depending on two B-typed values a computation depending on only one can be obtained, if we have the ability to duplicate that value and supply the two copies to the original computation.

Linear logic is a refinement of traditional logic which arises by restricting the use of Weakening and Contraction. When the logic is used as a type system for a programming language, this control over structural rules translates into restrictions on the occurrences of identifiers within terms [Abramsky 1993; Wadler 1991; Benton et al. 1993]. These restrictions result in a type $A \multimap B$ of linear functions, which cannot discard or duplicate their arguments through free use of Weakening or Contraction.

To connect these ideas back to imperative computation, let us try to write *snap* in a programming notation:

$$\mathit{snap}(p) \quad = \quad \lambda s.\,\mathbf{let}\,[s', v]\,\mathbf{be}\,p\,s\,\mathbf{in}\,[s, v].$$

This term uses both Contraction and Weakening: Contraction corresponds to the two non-binding occurrences of the initial state s in the body of the λ-expression, and Weakening to the absence of s' in $[s, v]$. As a result, if we were to use $\mathit{States} \multimap \mathit{States} \otimes \mathit{Values}$ as the type of side-effecting expressions (where $\mathit{States} \otimes \mathit{Values}$ is a type of "eager pairs") then $\mathit{snap}(p)$ would fail to typecheck; snapback is excluded by linear typing.

There is thus a tantalizing analogy between linear functions and imperative state transformations. So it is natural to ask whether, or the extent to which, linear logic can give rise to an improved semantic treatment of state.

As a first test of the analogy, we might translate a basic imperative language, such as the language of **while** programs, into a linear functional language. It is clear that one could express typical constructs, such as sequencing, assignment and iteration, in terms of linear functions.

This is all well and good, but it only connects up imperative and linear functional programming on a basic level, for an imperative language without procedures. And such a language does *not* in fact provide a satisfactory test. For, basic sequential imperative languages, without procedures, already possess a satisfactory foundation, with simple semantic models based on partial functions on states and logics based on Hoare triples or weakest preconditions. It is difficult to see how this understanding could be improved by phrasing the semantics in terms of linear types.

How can this be? We began by describing problems in traditional semantics based on state-to-state functions, and the language of **while** programs uses precisely that kind of semantics. Consider again the snapback example: *snap* is a function from state transformations to state transformations; in imperative terms it is a procedure that expects an expression thunk as an argument. It takes an *arbitrary* state transformation, runs it, and then restores the state to its initial value. The whole discussion of snapback and irreversibility hinged on having procedures, which are missing from the language of **while** programs.

We can go further still if we use local variables: We can then write programs whose observable behaviour is sensitive to whether or not snapback is present in the semantics:

$$\mathit{snaptester} \;=\; \lambda p.\,\mathbf{new}\;x.\,x := 0;\; p(x := x + 1);\; \mathbf{if}\;x > 0\;\mathbf{then\;diverge}.$$

The termination/nontermination behaviour of *snaptester* is equivalent to that of $\lambda p.\,p(\mathbf{diverge})$. The reason is that if p executes its argument at all then the value of x on termination of $p(x := x + 1)$ will be greater than 0, since there is no way for p to alter the value of x other than by using its argument. Snapback contradicts this informal reasoning, since $\mathit{snaptester}(\mathit{snap})$ converges while $\mathit{snaptester}(\mathbf{diverge})$

diverges.

Thus, it makes sense to consider imperative languages that have procedures and local state, in addition to assignment. In this paper we consider two such languages, based on Idealized Algol [Reynolds 1981b].

1.2 Polymorphism, Data Abstraction and Store Shapes

It is evident how to model state transformations with linear functions, but now we must consider how to model procedures and local state. We might attempt to do so directly in a simply-typed linear language, using $\multimap$ to model state transformations and a conventional function type $\rightarrow$ to model procedures, but there is a further problem: It is not obvious how we might account for the interaction of procedures and local state (as given, for example, in *snaptester*).

To expand on this last last point, consider how a "counter class" can be programmed in an Algol-like language using procedures and local state [Reynolds 1978]:

$$\mathit{newcounter} \;=\; \lambda p.\,\mathbf{new}\; x.\, x := 0;\; p(x := x + 1, x).$$

This code works by declaring a local variable x, and then passing the ability to increment and read x to the procedure p. (The second argument x of p is implicitly dereferenced from a variable to an expression, so that it cannot be assigned to by p.). Because the procedure p can never access the local variable x we are assured, for example, that the value of x can never be decremented. This illustrates how a form of data abstraction results from the combination of procedures and local state; it is hard to see how this phenomenon could be modelled in a *simply typed* version of linear λ-calculus.

This discussion has been leading toward our choice of target language. We can account for data abstraction and local state using *polymorphic* types [Reynolds 1974; O'Hearn and Tennent 1995], so our target language will be a linearly-typed, polymorphic λ-calculus.

We can now sketch the main ideas behind the translations. The starting point is to allow for multiple state types instead of only one. In terms of the polymorphic target language we regard type variables as ranging over various "store shapes" or state types, so that in a type $\alpha \multimap \alpha$ of state transformations the type variable α can be instantiated to a variety of different representations of the state. The basic idea is that programs working with different store shapes act on separate parts of the store.

To see how this works recall the counter class above. An argument p to *newcounter* is a procedure that accepts a command and an expression as arguments, and produces a command as a result. We assign p the polymorphic type

$$\forall\beta.(\beta \multimap \beta)\&(\beta \multimap \beta \otimes \mathbf{nat}) \rightarrow (\alpha \otimes \beta \multimap \alpha \otimes \beta).$$

The idea is that the state in use when p is called is partitioned into the α-typed part, which p may access directly, and the β-typed part, about which p knows nothing. The argument type $\beta \multimap \beta$ corresponds to a command for changing this unknown state, and $\beta \multimap \beta \otimes \mathbf{nat}$ to a natural number-valued expression (possibly with side effects).

The type constructors & and $\rightarrow$ here are for conventional product and function types; they are not subject to linearity restrictions. The mixing of linear and non-linear type constructors in the type of p implies that it is only the state that must be used linearly; the two arguments, of types $\beta \multimap \beta$ and $\beta \multimap \beta \otimes \mathbf{nat}$, may be used zero, one, or many times, as is common in imperative languages.

Now, if we apply the counter class to such a p, i.e., *newcounter*(p), we obtain a function of type $\alpha \multimap \alpha$:

$$\begin{aligned}&\underline{\lambda} s : \alpha.\, \mathbf{let}\ [s', n']\ \mathbf{be}\ p[\mathbf{nat}]\langle \underline{\lambda} n.\, n+1, \underline{\lambda} n.\, [n,n]\rangle [s,0]\\ &\quad\ \ \mathbf{in}\ s'.\end{aligned}$$

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. Intuitively, the independence of non-local procedures from local state corresponds to the parametricity of a polymorphic function whose type argument ranges over possible pieces of local state [Reynolds 1983; O'Hearn and Tennent 1995].

This example also illustrates how the move from simple to polymorphic types has an additional effect, beyond enabling a treatment of data abstraction. To see this, consider that we have used Contraction and Weakening of $\mathbf{nat}$-typed identifiers: Contraction is used for dereferencing, in $\lambda n.\, [n,n]$, and Weakening of n' is used to model deallocation of the local variable on block exit. (We also sometimes need Weakening to model updates.) These uses of Contraction and Weakening do not contradict the intuitive connection between linearity and state change, because the polymorphic uses of $\mathbf{nat}$ by p (obtained by instantiating β) will still all be linear. This point deserves careful consideration, and we will return to it several times, but the general idea is that polymorphic instantiation mediates between the linear way that state is manipulated, and the use of non-linear values to make up specific states.

Local-variable declarations are a special mechanism for ensuring absence of interference through shared variables. We can also use polymorphic typing to treat non-interference more generally. For example, consider the type

$$\forall \beta \, \forall \gamma.\, (\beta \multimap \beta) \& (\gamma \multimap \gamma \otimes \mathbf{nat}) \to (\alpha \otimes \beta \otimes \gamma \multimap \alpha \otimes \beta \otimes \gamma).$$

In imperative terms, a procedure of this type accepts two arguments, one a command and the other a side-effecting expression. If q is such a procedure then in an application $q[A][B]\langle c, e\rangle$ it is never possible to use c to change the state in a way that affects a future use of e. This is because, in q, using c produces a β-typed value, while e expects a γ-typed value, and these types do not match up. So the use of different type variables for the arguments means that the output state of one cannot be used as the input state of the other. Again in imperative terms, we take this to mean that the two arguments c and e don't interfere.

We now proceed to present the translations, and the semantics. Our two source languages are Idealized Algol [Reynolds 1981b] (without jumps or coercions, and with side effects in expressions) and syntactic control of interference [Reynolds

1978] (without passivity). The target language is based on the $\multimap$, $\otimes$, $\&$, $\rightarrow$ (or "!") fragment of intuitionistic linear logic [Girard 1987; Barber and Plotkin 1997], extended with a fixed-point operator and a predicative form of polymorphism. The semantic model of the target language is based on strict continuous functions and binary relational parametricity [Reynolds 1983]. We analyze the model by looking at sample equivalences, and by characterizing the structure of first-order types in terms of domain equations for resumptions.

In Section 8 we will describe the connection to functor-category semantics in some detail, but for the most part we will work directly with the polymorphic language and its model.

2. TWO VARIATIONS ON ALGOL

Our imperative languages are based on the analysis of Algol 60 given in [Reynolds 1978; Reynolds 1981b]. 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 [Reynolds 1978] (also, [O'Hearn et al. 1999]).

Both languages use the following grammar of types:

$$\begin{array}{rcll} \varphi & ::= & \mathbf{exp} \mid \mathbf{acc} \mid \mathbf{comm} & \text{primitive types} \\ \theta & ::= & \varphi \mid \theta \times \theta' \mid \theta \rightarrow \theta' & \text{types} \end{array}$$

The primitive type **exp** is the type of natural-number expressions, **acc** is the type of acceptors, and **comm** is the type of commands. Commands change the state of the store but do not produce values, and an acceptor changes the state when it is supplied with an integer. The type **var** of storage variables is an abbreviation for $\mathbf{acc} \times \mathbf{exp}$. The factors of **var** give the basic capabilities of updating and accessing a storage variable.

2.1 Idealized Algol

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

$$\frac{}{\Gamma, x : \theta \vdash x : \theta} \qquad \frac{\Gamma \vdash M : \theta}{\widetilde{\Gamma} \vdash M : \theta} \text{ where } \widetilde{\Gamma} \text{ is a permutation of } \Gamma$$

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

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

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

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

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

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

$$\frac{\Gamma \vdash M : \mathbf{exp} \to \mathbf{comm}}{\Gamma \vdash \mathbf{byvalue}\, M : \mathbf{acc}} \qquad \frac{\Gamma \vdash M : \mathbf{acc} \quad \Gamma \vdash N : \mathbf{exp}}{\Gamma \vdash M := N : \mathbf{comm}}$$

Idealized Algol contains the functional constructs of PCF [Plotkin 1977]. Of the imperative constructs, $\mathbf{new}(\lambda x.\, C)$ works by binding x to a local storage variable that is initialized to 0, ";" is sequential composition, **skip** is the do-nothing command, and assignment supplies an integer value to an acceptor.

Acceptors were originally introduced as part of a generalized approach to variables [Reynolds 1981b], in which an acceptor was considered simply as a function from data values to commands. On this view acceptors are similar to functions of type $\mathbf{exp} \to \mathbf{comm}$, except that they accept integer values rather than expression-thunks as arguments; they are thus a form of call-by-value procedures. The **byvalue** construct converts a thunk-expecting procedure to an acceptor using a coercion from natural-number values to expressions. (It would be conceivable to provide instead an alternate binding form for call-by-value, as was done in Algol 60 using the keyword **value** with a formal parameter.)

We will often use syntactic sugar in an informal, but hopefully clear, way. For instance, *newcounter* is rendered formally as

$$\begin{array}{l}\lambda p : (\mathbf{comm} \times \mathbf{exp} \to \mathbf{comm}) \to \mathbf{comm}. \\ \quad \mathbf{new}_{\mathbf{comm}}\ (\lambda x : \mathbf{var}.\, (\pi_1 x) := 0; p\langle (\pi_1 x) := (\mathbf{succ}\ \pi_2 x), \pi_2 x\rangle).\end{array}$$

Generally, we omit mention of types in **new** or on λ-bound identifiers, we omit the projections when using a term of type **var**, and we write $\mathbf{new}\, x.\, M$ instead of $\mathbf{new}\,(\lambda x.\, M)$.

An important difference with the original Idealized Algol is that 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 different uses. For example, if

$x : \mathbf{var}$ is a declared variable then $x := x + 1; x$ returns successive natural numbers on successive uses.

We have not attempted to produce an irredundant collection of basic constructs. For instance, the expression $\mathbf{new}_{\mathbf{acc}}\, P$ of type $\mathbf{acc}$ could be eliminated, as it is equivalent to $\mathbf{byvalue}\,(\lambda y : \mathbf{exp}.\, \mathbf{new}_{\mathbf{comm}}\, z.\, Pz := y)$.

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

$$\mathbf{new}_{\mathbf{exp}}\, z.\, z := 1;\; (plus\,(z := z + 1; z)\, 2)$$

diverges since z is incremented each time $\mathbf{pred}$ is evaluated.

Using $\mathbf{new}$ we can program a version of addition that evaluates its arguments once each, left followed by right, and adds the resulting values together:

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

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 no free identifiers in common. (This is another way of saying that the calculus does not have Contraction.) This restriction prevents interference between different identifiers. For instance, y and z are aliases in $((\lambda y \lambda z. \cdots y := a \cdots z := b \cdots)x)x$ if x denotes a storage variable. But a term of this form cannot typecheck in Basic SCI 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 the parallel composition $M \parallel N$ is allowed if the free identifiers of M and N are disjoint. This illustrates the difference with Idealized Algol, where the same construct would (because of interference) lead to indeterminacy.

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

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

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

To illustrate further the difference between SCI and Idealized Algol consider the addition operations $\mathit{leftadd}, \mathit{rightadd} : \mathbf{exp} \to \mathbf{exp} \to \mathbf{exp}$. 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 $\mathit{leftadd}(\mathbf{succ}\ x)(x := \mathbf{succ}\ x; x)$ returns value 12, whereas $\mathit{rightadd}(\mathbf{succ}\ x)(x := \mathbf{succ}\ x; x)$ returns value 13. In contrast, in SCI the arguments to these functions never interfere: The typing rules ensure that in a procedure call $\mathit{leftadd}(e_1)(e_2)$ the procedure $\mathit{leftadd}(e_1)$ and argument e_2 have disjoint sets of free identifiers. As a result, even though we allow side effects in expressions, *leftadd* and *rightadd* are equivalent in SCI.

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

3. A POLYMORPHIC LINEAR λ-CALCULUS

Now we introduce the polymorphic target language. We follow the version of linear type theory formulated by Barber and Plotkin [1997], where two zones are used in a typing context to keep track of *intuitionistic* and *linear* assumptions. The basic idea is that linear assumptions are used once, while intuitionistic assumptions can be used multiple times in a term. (We refer to [Abramsky 1993; Benton et al. 1993; Wadler 1990; Wadler 1991] for further discussions of linear λ-calculi.)

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

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

Type variables are denoted by α (or other Greek letters β, γ). 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.

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.

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 $\to$. For emphasis we use two binding forms, $\underline{\lambda} x : A.\, t$ and $\lambda x : A.\, t$, one for each function type. We will use the same syntax for applying both kinds of function (in effect leaving dereliction implicit in $\to$); no confusion is likely to arise from this.

The system uses typing judgements of the form

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

where the context is broken into an intuitionistic zone Γ and a linear zone Δ.

Identity

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

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

Additives

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

$$\frac{\Gamma; \Delta \vdash t : A \& B}{\Gamma; \Delta \vdash \pi_1\, t : A} \qquad \frac{\Gamma; \Delta \vdash t : A \& B}{\Gamma; \Delta \vdash \pi_2\, t : B}$$

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

Multiplicatives

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

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

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

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

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

Polymorphism

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

NATURAL NUMBERS AND RECURSION

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

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

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

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

In this formulation of linear typing we have introduction and elimination rules for each type constructor. However, we do not have to add explicit rules that permit Weakening and Contraction of "!"-typed identifiers; this is because of the use of two zones, which allows Weakening and Contraction in the intuitionistic zone to be left implicit, as is the case in simply-typed λ-calculus. This allows for a particularly smooth treatment of the intuitionistic function type $\to$, which is attractive for our purposes: We will have need for $\to$, but *not* explicitly for "!."

The characteristic feature of the additive rules is the sharing of typing contexts. For instance, in the introduction rule for &, the linear zone Δ is shared between both premises. On the other hand, the characteristic feature of the multiplicatives is the splitting of typing contexts in the linear zone. For instance, in the introduction rule for $\otimes$, the contexts Δ_1 and Δ_2 must be made up of disjoint collections of identifiers. The absence of Contraction is reflected in this splitting of contexts in the multiplicative rules, and the absence of Weakening is reflected in the rules for identifiers; the linear zone is empty when an identifier from the intuitionistic zone is typed, and of length one when a linear identifier is typed.

One point to notice is the presence of explicit terms for copying and discarding natural numbers. Using these and the rules for I we can define appropriate copying and discarding terms of types $\sigma \multimap \sigma \otimes \sigma$ and $\sigma \multimap I$, for any *closed* Level 1 type σ. 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 of storage variables could be reversed by using local variables to store and restore the values. But we cannot program a general 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 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 takes a judgement

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

in Idealized Algol 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

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

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, $\mathbf{comm}^\star(\alpha \otimes \beta)$ is $\alpha \otimes \beta \multimap \alpha \otimes \beta$. Similarly, a term M gets mapped to a term $M^\star$ with a type variable hole that can be filled with various type variables or Level 1 types: $M^\star\sigma$, $M^\star\sigma'$... and so on. (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 manipulations of environments.)

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.

In the informal translation of *newcounter* in the Introduction we used Weakening and Contraction of **nat**-typed identifiers. With our linear language, however, we have to be more explicit, using **copy** and **discard**. Also, a slight adjustment is needed because the translation of procedure types in Idealized Algol allows α to appear to the left of $\to$, whereas the type for the procedure p in the Introduction used the SCI interpretation (see below) where α does not appear to the left. So p now has type

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

and the translation of *newcounter*(p) is the following function of type $\alpha \multimap \alpha$:

$$\begin{aligned} &\underline{\lambda} s : \alpha.\, \mathbf{let}\ [s', n']\ \mathbf{be}\ p[\mathbf{nat}]\langle (id_\alpha \otimes \underline{\lambda} n.\ \mathbf{succ}\ n), (id_\alpha \otimes \underline{\lambda} n.\, \mathbf{copy}\, n)\rangle[s, 0] \\ &\qquad \mathbf{in}\ (\mathbf{let}\ *\ \mathbf{be}\ (\mathbf{discard}\, n')\ \mathbf{in}\ s'), \end{aligned}$$

where id_α is $\underline{\lambda} s : \alpha.\, s$.

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 polymorphic λ-calculus together with the assumption that **discard** and **copy** give **nat** a comonoid structure. Typically, the basic equations that are valid in models where $\forall$ is interpreted simply as an indexed product are sufficient for reasoning about **new** blocks whose only free identifiers are of primitive type, but parametricity is needed when there are free identifiers of procedural type.

In the Introduction we discussed snapback in the context of a single collection *States* of states. In the polymorphic language snapback would ostensibly be given by a term

$$\Lambda\alpha.\, \underline{\lambda} s : \alpha.\, \mathbf{let}\ [s', n]\ \mathbf{be}\ c\, s\ \mathbf{in}\ [s, n]$$

of type $\forall\alpha.\,(\alpha \multimap \alpha\otimes\mathbf{nat}) \to (\alpha \multimap \alpha\otimes\mathbf{nat})$ which (given the isomorphism $\alpha \cong I\otimes\alpha$) would determine a closed term of type $(\mathbf{exp} \to \mathbf{exp})^{\star} I$. 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

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

Compared to the other translation, each identifier x_i is now associated with a separate state-type α_i; the idea is that each identifier has a separate piece of the store that it acts upon. 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.

For example, the type $\mathbf{comm} \to \mathbf{exp} \to \mathbf{comm}$ translates as

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

which is isomorphic to the type

$$\forall\beta\,\forall\gamma.\,(\beta \multimap \beta) \,\&\, (\gamma \multimap \gamma\otimes\mathbf{nat}) \to (\alpha\otimes\beta\otimes\gamma \multimap \alpha\otimes\beta\otimes\gamma)$$

that we used to illustrate non-interference in the Introduction.

Note that the absence of Contraction in linear logic is *not* being used to account for the absence of Contraction in 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 non-interference, whereas linearity (combined with polymorphism) ensures that there is no snapback.

We give several examples of term translations. First, if x_1 and x_2 are different identifiers of type **comm**, then the translation of $x_1 \parallel x_2$ is

$$\begin{array}{l} x_1 : \alpha_1 \multimap \alpha_1, x_2 : \alpha_2 \multimap \alpha_2; \_ \\ \vdash \underline{\lambda} s : \alpha_1 \otimes \alpha_2.\ \mathbf{let}\ [s_1, s_2]\ \mathbf{be}\ s\ \mathbf{in}\ [x_1 s_1, x_2 s_2] : \alpha_1 \otimes \alpha_2 \multimap \alpha_1 \otimes \alpha_2. \end{array}$$

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

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

$$\begin{array}{l} x_1 : \alpha_1 \multimap \alpha_1, x_2 : \alpha_2 \multimap \alpha_2; \_ \\ \vdash \underline{\lambda} s : \alpha_1 \otimes \alpha_2.\ x_2'(x_1'(s)) : \alpha_1 \otimes \alpha_2 \multimap \alpha_1 \otimes \alpha_2, \end{array}$$

where

$$\begin{array}{lcl} x_1' & = & \lambda s : \alpha_1 \otimes \alpha_2 . \mathbf{let}\ [s_1, s_2]\ \mathbf{be}\ s\ \mathbf{in}\ [x_1 s_1, s_2] \\ x_2' & = & \lambda s : \alpha_1 \otimes \alpha_2 . \mathbf{let}\ [s_1, s_2]\ \mathbf{be}\ s\ \mathbf{in}\ [s_1, y_2 s_2]. \end{array}$$

Although the translation uses $x_2'(x_1'(s))$, which indicates that x_1 is evaluated first, it reduces to **let** $[s_1, s_2]$ **be** s **in** $[x_1 s_1, x_2 s_2]$ using typical reductions of linear λ-calculus. Thus, it is clear that $x_1; x_2$ and $x_1 \parallel x_2$ are equivalent.

Finally, consider our two addition operations *leftadd* and *rightadd*. The translations of *leftadd*$(x_1)(x_2)$ and *rightadd*$(x_1)(x_2)$ are both

$$\begin{array}{l} x_1 : \alpha_1 \multimap \alpha_1 \otimes \mathbf{nat}, x_2 : \alpha_2 \multimap \alpha_2 \otimes \mathbf{nat}; \_ \\ \vdash \underline{\lambda} s : \alpha_1 \otimes \alpha_2.\ \mathbf{let}\ [s_1, s_2]\ \mathbf{be}\ s\ \mathbf{in} \\ \qquad\qquad\qquad \mathbf{let}[[s_1', n], [s_2', m]]\ \mathbf{be}\ [x_1 s_1, x_2 s_2]\ \mathbf{in}\ [[s_1', s_2'], m + n] \\ : \alpha_1 \otimes \alpha_2 \multimap \alpha_1 \otimes \alpha_2 \otimes \mathbf{nat}. \end{array}$$

We regard the two arguments as being evaluated in parallel, possibly altering different portions of the store, before their results are added together. (The translations do not literally result in this term, but in ones that are, by an easy analysis using the semantic model of Section 6, seen to be equivalent to it.)

5. TRANSLATIONS OF TERMS

The detailed translations of terms follow ideas from functor-category semantics [Oles 1982].

5.1 Idealized Algol

We begin by translating assignment. An assignment statement $M := N$ is executed by first evaluating N, obtaining a changed state and value, and then supplying this value and state to M. Thus, assignment is simply a composition of the form

$$\alpha \xrightarrow{N^\star\alpha} \alpha\otimes\mathbf{nat} \xrightarrow{M^\star\alpha} \alpha.$$

More explicitly, we may define

$$(M := N)^\star\alpha \;=\; \underline{\lambda} s:\alpha.\,\mathbf{let}\,[s',m] = (N^\star\alpha\, s)\,\mathbf{in}\;(M^\star\alpha\,[s',m]).$$

To translate $\mathbf{new}_{\mathbf{comm}}$ we follow the pattern of the *newcounter* example in the Introduction:

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

In this equation the expanded state $[s,0]$ and a local variable $v[\alpha]$ (defined below) 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 $P^\star\alpha$ 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 $\mathbf{nat}$.

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

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

The essence of these two operations is the use of **discard** in *assign* and **copy** in *lookup*. Similar assignment 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 assign maps for "storing" elements of any such type. An interesting question is whether such operations make (imperative) operational sense.

The translation of $\mathbf{new}_{\mathbf{exp}}$ is similar:

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

If we had interpreted **comm** as $\alpha\multimap\alpha\otimes I$ then the two **new** constructs could have been treated uniformly, by the same defining equation. The translation of $\mathbf{new}_{\mathbf{acc}}\,P$ is obtained by viewing it as sugar for $\mathbf{byvalue}\,(\lambda y:\mathbf{exp}.\,\mathbf{new}_{\mathbf{comm}}\,z.\,Pz := y)$.

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:

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

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 isomorphisms.

This use of canonical isomorphisms is unpleasant, and is 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 $\mathit{expand}_\theta : \forall\alpha\beta.\, \theta^\star\alpha \multimap \theta^\star(\alpha\otimes\beta)$. Expansions 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. Suppose 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

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

where $(M^\star(\alpha\otimes\beta))[\mathit{expand}_{\theta_i}[\alpha][\beta]x_i/x_i]$ denotes the term obtained by substituting $\mathit{expand}_{\theta_i}[\alpha][\beta]x_i$ 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

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

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 $M^\star_\Gamma\alpha$, indexed by Idealized Algol type assignments Γ.

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

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

The clause for $M; N$ is for the cases when $N : \mathbf{exp}$ or $N : \mathbf{comm}$. The case of $M; N$ when $N : \mathbf{acc}$ is treated by the translation as sugar for the expression $(\mathbf{byvalue}\, \lambda x : \mathbf{exp}.\, M; (N := x))$. Similarly, the clause for **if** is when $N_2, N_3 : \mathbf{comm}$ or $N_2, N_3 : \mathbf{exp}$, and the acceptor case is sugar for the expression $(\mathbf{byvalue}\, \lambda x : \mathbf{exp}.\, \mathbf{if}\, N_1 = 0\, \mathbf{then}\, N_1 := x\, \mathbf{else}\, N_2 := x)$.

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 of [O'Hearn and Tennent 1995] is then a consequence of the definability of these expansions and the

Logical Relations Lemma. The definition goes by induction on θ:

$$\begin{aligned}
\mathit{expand}_{\mathbf{comm}} &= \Lambda\alpha\beta.\, \underline{\lambda}c : \alpha \multimap \alpha.\, \underline{\lambda}[a,b] : \alpha\otimes\beta.\, [c\,a, b] \\
\mathit{expand}_{\mathbf{acc}} &= \Lambda\alpha\beta.\, \underline{\lambda}c : \alpha\otimes\mathbf{nat} \multimap \alpha.\, \underline{\lambda}[[a,b],n] : (\alpha\otimes\beta)\otimes\mathbf{nat}.\, [c[a,n], b] \\
\mathit{expand}_{\mathbf{exp}} &= \Lambda\alpha\beta.\, \underline{\lambda}e : \alpha \multimap \alpha\otimes\mathbf{nat}.\, \underline{\lambda}[a,b] : \alpha\otimes\beta.\, \mathbf{let}\,[a',n] = e\,a \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\;\, [[a',b],n] \\
\mathit{expand}_{\theta\times\theta'} &= \Lambda\alpha\beta.\, (\mathit{expand}_{\theta}[\alpha,\beta])\&(\mathit{expand}_{\theta'}[\alpha,\beta]) \\
\mathit{expand}_{\theta\rightarrow\theta'} &= \Lambda\alpha\beta.\, \underline{\lambda}p.\, \Lambda\gamma.\, i1 \circ p[\beta\otimes\gamma] \circ i2
\end{aligned}$$

Here we have used the shorthand of composition $\circ$ (in functional order) and the functorial action of &. The terms $i1$ and $i2$ are for associativity isomorphisms:

$$i1 : \theta'((\alpha\otimes\beta)\otimes\gamma) \multimap \theta'(\alpha\otimes(\beta\otimes\gamma))$$
$$i2 : \theta((\alpha\otimes\beta)\otimes\gamma) \multimap \theta(\alpha\otimes(\beta\otimes\gamma)).$$

Thus, expansions for function types are defined according to the following diagram:

$$\begin{tikzcd}
\theta^\star(\alpha\otimes(\beta\otimes\gamma)) \arrow[r, "p[\beta\otimes\gamma]"] & \theta'^\star(\alpha\otimes(\beta\otimes\gamma)) \arrow[d, "i1"] \\
\theta^\star((\alpha\otimes\beta)\otimes\gamma) \arrow[u, "i2"] \arrow[r, "\mathit{expand}_{\theta\to\theta'}[\alpha][\beta]p[\gamma]"'] & \theta'^\star((\alpha\otimes\beta)\otimes\gamma)
\end{tikzcd}$$

5.3 Basic SCI

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

First, to translate identifiers, given

$$x_1 : \theta_1, \ldots, x_m : \theta_m \vdash x_i : \theta_i$$

in SCI we must define

$$x_1 : \theta_1^\circ\alpha_1, \ldots, x_m : \theta_m^\circ\alpha_m ; \_ \vdash x_i^\circ(\alpha_1,\ldots,\alpha_m) : \theta_i^\circ(\alpha_1\otimes\cdots\otimes\alpha_m).$$

We define $x_i^\circ(\alpha_1,\ldots,\alpha_m) = e(x_i)$, where $e : \theta_i^\circ\alpha_i \multimap \theta_i^\circ(\alpha_1\otimes\cdots\otimes\alpha_m)$ is obtained by composing an expansion with appropriate symmetry isomorphisms.

For λ-abstraction, given

$$\begin{aligned}
&x_1 : \theta_1^\circ\alpha_1, \ldots, x_m : \theta_m^\circ\alpha_m, y : \theta^\circ\beta ; _ \\
\vdash\;& M^\circ(\alpha_1,\ldots,\alpha_m,\beta) : \theta'^\circ(\alpha_1\otimes\cdots\otimes\alpha_m\otimes\beta)
\end{aligned}$$

we can form the judgement

$$\begin{aligned}
&x_1 : \theta_1^\circ\alpha_1, \ldots, x_m : \theta_m^\circ\alpha_m ; _ \\
\vdash\;& \Lambda\beta.\, \lambda y : \theta^\circ\beta.\, M^\circ(\alpha_1,\ldots,\alpha_m,\beta). : \forall\beta.\, \theta\beta \rightarrow \theta'(\alpha_1\otimes\cdots\otimes\alpha_m\otimes\beta).
\end{aligned}$$

We take this as the interpretation of λ-abstraction in SCI; it is remarkably simple compared to the translation for Idealized Algol.

The SCI 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 we define $(MN)^\circ(\alpha_1, \ldots, \alpha_m)$ to be

$$\big(M^\circ(\alpha_1, \ldots, \alpha_n)[\alpha_{n+1} \otimes \cdots \otimes \alpha_m]\big)\big(N^\circ(\alpha_{n+1}, \ldots, \alpha_m)\big).$$

In this definition the β-component of

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

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

For parallel composition, given

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

we translate as follows:

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

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

The equations for $\mathbf{new_{comm}}$ and $\mathbf{new_{exp}}$ are as in Idealized Algol, except that the variable $v[\alpha]$ is replaced by $v : (\mathbf{acc} \times \mathbf{exp})^\circ(\mathbf{nat})$, given by $\langle \mathit{assign}, \mathit{lookup} \rangle$, where

$$\mathit{assign} \;=\; \underline{\lambda}[n, m] : \mathbf{nat} \otimes \mathbf{nat}.\, \mathbf{let}\ *\ \mathbf{be}\ (\mathbf{discard}\, n)\ \mathbf{in}\ m$$

$$\mathit{lookup} \;=\; \underline{\lambda} n : \mathbf{nat}.\, \mathbf{copy}\, n.$$

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

Finally, expansions for procedure types are defined as follows:

$$\mathit{expand}_{\theta \to \theta'} \;=\; \Lambda \alpha \beta.\, \underline{\lambda} p.\, \Lambda \gamma.\, i \circ \mathit{expand}_{\theta'}[\alpha \otimes \gamma][\beta] \circ p[\gamma],$$

where i is an isomorphism as indicated in the following diagram

$$\begin{tikzcd}
\theta'^\circ(\alpha \otimes \gamma) \arrow[r, "\mathit{expand}_{\theta'}[\alpha \otimes \gamma][\beta]"] & \theta'^\circ((\alpha \otimes \gamma) \otimes \beta)) \arrow[d, "i"] \\
\theta^\circ(\gamma) \arrow[u, "p[\gamma]"] \arrow[r, "\mathit{expand}_{\theta \to \theta'}[\alpha][\beta] p[\gamma]"'] & \theta'^\circ((\alpha \otimes \beta) \otimes \gamma)
\end{tikzcd}$$

Expansions for the primitive types are the same as in Idealized Algol.

There is another candidate definition for expansions at procedure types, based on the following diagram.

$$\begin{tikzcd}
\theta^\circ(\beta \otimes \gamma) \arrow[r, "{p[\beta \otimes \gamma]}"] & \theta'^\circ(\alpha \otimes (\beta \otimes \gamma)) \arrow[d, "i"] \\
\theta^\circ(\gamma) \arrow[u, "{expand_\theta[\gamma][\beta]}"] \arrow[r, "{expand_{\theta \to \theta'}[\alpha][\beta]p[\gamma]}"'] & \theta'^\circ((\alpha \otimes \beta) \otimes \gamma)
\end{tikzcd}$$

The equivalence of these two definitions depends on a naturality property for p, which we will be able to verify later (Theorem 11) using a semantic model of the linear calculus.

6. THE STRICT PARAMETRICITY MODEL

Having defined the translations we now want to analyze them in more detail. The approach we take is to look at a simple semantic model of the target language, and push it as far as possible. This will allow us (in Section 7) to very quickly go beyond previous functor models [Oles 1982; O'Hearn and Tennent 1995]. Another possibility for analysis would have been to directly relate contextual equivalence relations for the source and target languages: A syntactic study along these lines would be interesting, but is outside the scope of the present paper. Additionally, the semantic model contains useful information beyond contextual equivalence, which can be viewed as providing relational reasoning principles, and which leads to representation results.

The model is based on strict continuous functions and relational parametricity; we call it simply the strict parametricity model. It 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 many of these examples.

6.1 Semantics of Types

In the strict parametricity model, types are interpreted as cpo's, i.e., directed-complete partial orders possessing a least element [Plotkin 1983; Abramsky and Jung 1994]. Level 1 types will always denote countable, flat cpo's, where there is a discretely-ordered set of elements arranged just above a least element $\bot$.

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 directly by their cpo counterparts:

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

On the right-hand side I is the two-point cpo, $(\cdot)_\bot$ 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 [Reynolds 1983]. Suppose $\mathcal{R}$ (a relation environment) is a function mapping each type variable into a pointed (i.e., $\langle\bot,\bot\rangle$-containing) 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. (Completeness means that the relation is closed under lubs of directed sets in $[\![A]\!]_{\mathcal{D}} \leftrightarrow [\![A]\!]_{\mathcal{D}'}$. This condition is needed for recursion to be compatible with the Logical Relations Lemma, below.)

$$\begin{array}{rcl}
[\![\alpha]\!]_{\mathcal{R}} & = & \mathcal{R}\alpha \\
[\![I]\!]_{\mathcal{R}} & = & \{\langle a,a\rangle \mid a \in I\} \\
[\![\mathbf{nat}]\!]_{\mathcal{R}} & = & \{\langle n,n\rangle \mid n \in N_\bot\} \\
[\![\sigma\otimes\sigma']\!]_{\mathcal{R}} & = & \{\langle [d,d'],[e,e']\rangle \mid \langle d,e\rangle \in [\![\sigma]\!]_{\mathcal{R}} \ \wedge\ \langle d',e'\rangle \in [\![\sigma']\!]_{\mathcal{R}}\} \\
[\![A \,\&\, B]\!]_{\mathcal{R}} & = & \{\langle\langle d,d'\rangle,\langle e,e'\rangle\rangle \mid \langle d,e\rangle \in [\![A]\!]_{\mathcal{R}} \ \wedge\ \langle d',e'\rangle \in [\![B]\!]_{\mathcal{R}}\} \\
[\![A \to B]\!]_{\mathcal{R}} & = & \{\langle f,f'\rangle \mid \forall\langle d,d'\rangle \in [\![A]\!]_{\mathcal{R}}.\, \langle fd, f'd'\rangle \in [\![B]\!]_{\mathcal{R}}\} \\
[\![A \multimap B]\!]_{\mathcal{R}} & = & \{\langle f,f'\rangle \mid \forall\langle d,d'\rangle \in [\![A]\!]_{\mathcal{R}}.\, \langle fd, f'd'\rangle \in [\![B]\!]_{\mathcal{R}}\} \\
[\![!A]\!]_{\mathcal{R}} & = & \{\langle d,e\rangle \mid \langle d,e\rangle \in [\![A]\!]_{\mathcal{R}} \ \vee\ d = e = \bot\}.
\end{array}$$

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 $\otimes$, $[x,y]$ is the least element $\langle\bot,\bot\rangle$ if x or y is $\bot$ and is the pair $\langle x,y\rangle$ otherwise. We often refer to the relational action of type constructors directly, for example by writing $R\otimes S$ instead of $[\![\alpha\otimes\beta]\!](\alpha\mapsto R, \beta\mapsto S)$.

The relational action of $\otimes$ deserves comment. A basic justification for the definition is the following two properties that it satisfies:

—If $\langle f,g\rangle \in R \multimap S$ and $\langle f',g'\rangle \in R' \multimap S'$ then $\langle f\otimes f', g\otimes g'\rangle \in R\otimes R' \multimap S\otimes S'$,
—$R\otimes S \multimap T \ \cong\ R \multimap (S \multimap T)$.

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 (needed for the Logical Relations Lemma below). The second is a relationship we expect between $\otimes$ and $\multimap$.

Next, note that we do not have the property that

$\langle a,b\rangle\big(R\otimes S\big)\langle c,d\rangle$ implies aRc and bSd.

For instance if $\bot Rc$ and bSd then $\langle\bot,\bot\rangle\big(R\otimes S\big)\langle c,d\rangle$ even when we do not have $\bot Sd$. The property fails because of the use of $[\cdot,\cdot]$ in the definition of $R\otimes S$.

Finally, a particular subtlety is that the formula for $R\otimes S$ does not always produce a complete relation, for complete relations R and S on arbitrary cpo's. For example, consider relations $R : N_\bot \leftrightarrow I$ and $S : N_\bot \leftrightarrow Vnat$ where $R = \{\langle\bot,\bot\rangle, \langle\bot,*\rangle\}$ and $S = \{\langle\bot,0\rangle, \langle 1,2\rangle, \langle 2,3\rangle, ...\}$, and $Vnat$ is the vertical natural numbers. Then $R\otimes S$ contains a chain of all tuples $\langle\langle\bot,\bot\rangle,\langle *,n\rangle\rangle$, for $n > 1$, but it does not contain the lub $\langle\langle\bot,\bot\rangle,\langle *,\infty\rangle\rangle$. Nevertheless, $R\otimes S$ is strict and trivially complete whenever

R and S are strict relations on flat (or finite height) cpo's. In the semantics of our polymorphic language $\otimes$ is only ever applied to flat cpo's, so the definition of the relational action is suitable for this language.

Returning to the definition of the model, we can now define the cpo and relational semantics for $\forall$. For this, let $Level1$ stand for the collection of countable, flat cpo's. A polymorphic function will be an element of an indexed product, indexed by cpo's in $Level1$, subject to a relational parametricity condition:

$$\begin{array}{rcl} [\![\forall\alpha.A]\!]_{\mathcal{D}} & = & \{p \in \prod_{d\in Level1} [\![A]\!]_{\mathcal{D}|\alpha\mapsto d} \mid \\ & & \quad \forall d, d' \in Level1\ ,\ \forall r : d \leftrightarrow d'.\ \langle p[d], p[d']\rangle \in [\![A]\!]_{\mathcal{I}_{\mathcal{D}}|\alpha\mapsto r} \\ & & \}, \\ & & \text{ordered pointwise} \end{array}$$

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

Here $\mathcal{I}_{\mathcal{D}}$ maps each α to the identity relation $\Delta_{\mathcal{D}\alpha}$ on $\mathcal{D}\alpha$. The notation $r : d \leftrightarrow d'$ indicates that r is a relation between (flat) cpo's d and d' that relates their least elements. 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).

With these definitions one may verify that each $[\![A]\!]_{\mathcal{D}}$ is a cpo and each $[\![A]\!]_{\mathcal{R}}$ is a complete relation on cpo's. This ensures that the least fixed-point operator exists in the model.

By "parametric model" we mean a model satisfying 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 [Reynolds 1983], and the definition of $\forall$ is arranged precisely to ensure the identity property (which is included here 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

$$x_1 : A_1, \ldots, x_n : A_n; y_1 : B_1, \ldots, y_m : B_m \vdash t : C$$

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

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

We omit the detailed definition, which is standard, but we do describe the uniformity property of this family. For this, we extend relational actions from types to typing contexts using the actions for $\otimes$ and $!$:

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

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 $[\![t]\!]_{\mathcal{D}}$ are proven simultaneously by induction on t.

7. WORKING WITH THE MODEL

Our aim in this section is to illustrate how strict, relational parametricity can be used to reason about types and terms in the source languages. All of the examples go beyond those that can be treated properly in either the basic functor category model of Oles [1982] or the parametric functor models of O'Hearn and Tennent [1995], and Sieber [1996]. (The ability to relate $\perp$ to non-$\perp$ states is responsible for the improvement over [O'Hearn and Tennent 1995].)

From now on we will often mix notation for polymorphic types and cpo's. For instance, we write

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

instead of the more cumbersome

$$[\![\forall\beta.\,(\alpha \otimes \beta \multimap \alpha \otimes \beta) \rightarrow (\alpha \otimes \beta \multimap \alpha \otimes \beta)]\!]_{(\alpha \mapsto S)}.$$

Similarly, we write $M^{\star}S$ and $M^{\circ}S$ to indicate functions, and $\theta^{\star}S$ and $\theta^{\circ}S$ to indicate cpo's, obtained by composing the translations with the semantics of the linear langauge.

7.1 Snapback

Example 1. We consider a snapback operator of type

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

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

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

for countable flat cpo's S and $s \in S$. We show that *snap* is not parametric.

Consider any $p \in [\![\forall\alpha.\,(\alpha \multimap \alpha \otimes \mathbf{nat}) \rightarrow (\alpha \multimap \alpha \otimes \mathbf{nat})]\!]$. We show the following property (where $2 = \{0, 1\}$):

If	$e : 2_{\perp} \multimap 2_{\perp} \otimes N_{\perp}$ is such that $e(s) = \langle 1, 1\rangle$ when $s \neq \perp$,
and	$p[2_{\perp}]e0 = \langle 0, 1\rangle$
then	$p[2_{\perp}]\perp 0 = \langle 0, 1\rangle$.

This property says that if you use e at all, the final state can no longer be 0: Using e causes an irreversible state change.

To prove the property, suppose the two assumptions hold for p and e, and 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 $\langle e,\perp\rangle \in [R \multimap R \otimes \Delta_{N_\perp}]$ and by parametricity $\langle p[2_\perp]e0, p[2_\perp]\perp 0\rangle \in R \otimes \Delta_{N_\perp}$. Since $\langle 0,1\rangle$ is only $R \otimes \Delta_{N_\perp}$-related to itself the property follows.

In contrast, the property is not satisfied by *snap* because $\mathit{snap}[2_\perp]\perp 0 = \perp$, even though we will have $\mathit{snap}[2_\perp]e0 = \langle 0,1\rangle$ for any e satisfying the first assumption in the property. 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 countable, flat cpo's $D \in \mathit{Level}1$, is not in $[\![\forall\alpha.\, \alpha \multimap I]\!]$.

The failure of the naturality of this Weakening map is reminiscent 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 *snaptester* equivalence from the Introduction:

$$\textbf{new}\ x.\, x := 0;\ p(x := x+1);\ \textbf{if}\ (x > 0)\ \textbf{then diverge} \quad\equiv\quad p(\textbf{diverge}).$$

To show the equivalence we argue that the translations

$$(2)\qquad \begin{array}{l} \underline{\lambda} s:\alpha.\, \textbf{let}\ [s',n']\ \textbf{be}\ p[\textbf{nat}](id_\alpha \otimes \underline{\lambda} n.\, n+1)[s,0] \\ \quad\ \textbf{in let}\ [s'',n'']\ \textbf{be}\ (\textbf{if}\, n' > 0\, \textbf{then}\, \Omega\, s'\, \textbf{else}\, s') \\ \quad\quad \textbf{in}\ (\textbf{let}\ *\ \textbf{be}\, (\textbf{discard}\, n'')\ \textbf{in}\ s'') : \alpha \multimap \alpha \end{array}$$

and

$$(3)\quad \begin{array}{l} \underline{\lambda} s:\alpha.\ \textbf{let}\ [s',n]\ \textbf{be}\ p[I](\Omega_{(\alpha\otimes I)\multimap(\alpha\otimes I)})\,[s,*]\ \textbf{in} \\ \quad\quad \textbf{let}\ *\ \textbf{be}\ (\textbf{discard}\, n)\ \textbf{in}\ s' \end{array}$$

are equivalent, where

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

Let S 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 $c : S\otimes N_\perp \multimap S\otimes N_\perp$ be $id_S \otimes \mathit{succ}$, where *succ* is the strict extension of successor. Then

$$\langle c,\perp\rangle \in \Delta_S \otimes R \multimap \Delta_S \otimes R.$$

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

$$\langle p[N_\perp]c[s,0]\,,\, p[I]\perp[s,*]\rangle \in \Delta_S \otimes R.$$

Next, for any $s \in S$ suppose that $p[N_\perp]c[s,0] = [s',n]$ and $p[I]\perp[s,*] = [s'',x]$. Using $\langle [s',n],[s'',x]\rangle \in \Delta_S \otimes R$ we can argue that (2)=(3) by cases as follows:

—If $[s',n] = \perp$ then $[s'',x] = \perp$ by the definition of R and $\otimes$, and so (2)s=(3)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 (2)s=(3)s, since both will, when beginning in state s, return s'.

  —If $n > 0$ then we know that (2) applied to s returns $\perp$. But since $n > 0$ we know, from the definition of R and parametricity, that $[s'',x] = \perp$, and so both (2) and (3) 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

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

for $p : \mathbf{comm} \times \mathbf{exp} \to \mathbf{comm}$, 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 [O'Hearn and Tennent 1995]) indicates that the strict-function model provides an extension of the reasoning methods presented there.

Example 3. Since the cpo model of PCF contains functions such as "parallel or," [Plotkin 1977] one might expect similar functions to arise in the model here. But, in the presence of side effects, if we are to evaluate two arguments in parallel then indeterminacy typically results, unless we use snapback. For example, a "parallel or" function of type $(\mathbf{exp} \to \mathbf{exp} \to \mathbf{exp})^\circ I$ in SCI would be determined by a function of type

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

with the following defining property:

$$por[S_1][S_2]\langle e_1, e_2\rangle\,[s_1, s_2] \;=\; \begin{cases} \langle s_1, s_2, 0\rangle & \text{if } \exists s_1'.\, e_1(s_1) = \langle s_1', 0\rangle \\ & \quad\ \vee\ \exists s_2'.\, e_2(s_2) = \langle s_2', 0\rangle \\ \perp & \text{otherwise} \end{cases}$$

This function is not parametric, because $por[I][-]\langle \perp, -\rangle$ would essentially be the unary snapback which, as we saw in Example 1, is not parametric. Note here that the condition $\exists s_1'.\, e_1(s_1) = \langle s_1', 0\rangle$ is testing whether running e_1 results in termination with result 0, and possibly some state change; it, and the other disjunct, are false in the case of non-termination and also in the case of producing a result other than 0.

A more subtle form of parallelism and snapback, which is *not* correctly treated by the model, will be considered at the end of this section.

7.2 Sample Type Analyses

Example 4. The polymorphic type $(\mathbf{comm} \to \mathbf{comm})^\star I$ corresponds to closed terms of type $\mathbf{comm} \to \mathbf{comm}$ in Idealized Algol. We can calculate its structure as follows:

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

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

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

where $c^0 = \mathbf{skip}$ and $c^{i+1} = c; c^i$. This representation of $\mathbf{comm} \to \mathbf{comm}$ should be compared to that of O'Hearn and Tennent [1995], 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 $N_\perp$. (It is in fact the usual parametricity argument for Church numerals [Plotkin 1980; Reynolds 1983], extended to take appropriate care of the presence of $\perp$.) Let $p \in [\![\forall\beta.\, (\beta \multimap \beta) \to (\beta \multimap \beta)]\!]$ and consider any flat cpo 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, $(p[N_\perp]\, succ\, 0)\, R\, (p[D]\, c\, d)$. Next, if $(p[N_\perp]\, succ\, 0) = \perp$ then the definition of R ensures that $(p[D]\, c\, d) = \perp$, and so $p = \perp$. And 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. We may conclude that $[\![\forall\beta.\, (\beta \multimap \beta) \to (\beta \multimap \beta)]\!]$ is isomorphic 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 [O'Hearn and Tennent 1995], relations that relate $\perp$ to non-$\perp$ elements were not considered, which is why this argument could not go through.

Example 5. The SCI interpretation $(\mathbf{comm} \to \mathbf{comm})^\circ I$ turns out to be the same as $(\mathbf{comm} \to \mathbf{comm})^\star I$. However, when we replace I by a non-trivial flat cpo S a difference appears. We calculate

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

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 command argument 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 independent of β. This independence is reflected in certain equivalences between terms, such as between procedures $\lambda c.\, c; x := 1$ and $\lambda c.\, x := 1; c$ of type $\mathbf{comm} \to \mathbf{comm}$.

The same type in Idealized Algol translates as follows:

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

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 in terms of resumptions. This will be presented as a special case of a representation result in Section 10.1.

Example 6. Next we consider a type for functions with two arguments:

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

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 $i = 1, 2$ and i, ℓ is the cons of i onto ℓ. The parametricity argument showing the last isomorphism is essentially the same as in Example 4. With this representation we can see which argument an element of this type evaluates first: i, ℓ evaluates the i'th argument.

Example 7. Two-argument curried functions in SCI are less rigidly sequential, in that it is not necessary for one argument to be evaluated first:

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

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

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

In contrast to Example 6, the evaluations of the two command arguments can proceed completely independently, in parallel.

7.3 A Limitation of Binary, Strict Parametricity

Independence from evaluation order in SCI gives rise to examples that require stronger principles than binary relational parametricity to treat correctly.

Example 8. We define a function *rotate* in the denotation of the type

$$\begin{aligned} \forall\alpha_1\alpha_2\alpha_3.\,(\alpha_1 \multimap \alpha_1 \otimes \mathbf{nat}) \& (\alpha_2 \multimap \alpha_2 \otimes \mathbf{nat}) \& (\alpha_3 \multimap \alpha_3 \otimes \mathbf{nat}) \\ \longrightarrow \quad (\alpha_1 \otimes \alpha_2 \otimes \alpha_3 \multimap \alpha_1 \otimes \alpha_2 \otimes \alpha_3 \otimes \mathbf{nat}) \end{aligned}$$

This type is isomorphic to $(\mathbf{exp} \to \mathbf{exp} \to \mathbf{exp} \to \mathbf{exp})^{\circ}I$. The definition is

$$
\begin{array}{l}
\mathit{rotate}[S_1, S_2, S_3]\langle c_1, c_2, c_3\rangle[s_1, s_2, s_3] \\
= \quad \begin{cases}
[s_1', s_2', s_3, 0] & \text{if } c_1 s_1 = [s_1', 0] \neq \bot \ \wedge \ c_2 s_2 = [s_2', 0] \neq \bot \\
[s_1, s_2', s_3', 1] & \text{if } c_2 s_2 = [s_2', 1] \neq \bot \ \wedge \ c_3 s_3 = [s_3', 1] \neq \bot \\
[s_1', s_2, s_3', 2] & \text{if } c_1 s_1 = [s_1', 2] \neq \bot \ \wedge \ c_3 s_3 = [s_3', 2] \neq \bot \\
\bot & \text{otherwise}
\end{cases}
\end{array}
$$

Informally, to evaluate *rotate* we begin by evaluating its three arguments in parallel. Of the three possible rightmost conditions above only one can apply. If it is the first, then we return the altered states s_1' and s_2', while leaving s_3 unchanged. Similarly we leave s_1 unchanged in the second case and s_2 in the third. Intuitively, this involves a form of snapback in each argument. Nevertheless, *rotate* does exist in the model. (We leave the verification of parametricity conditions as an exercise.)

This example shows a limitation of the strict-function model of the polymorphic calculus, and of the corresponding model of Basic SCI. In particular, because of the example we would not expect a result to the effect that the syntactic type $(\mathbf{exp} \to \mathbf{exp} \to \mathbf{exp} \to \mathbf{exp})^{\circ}I$ is, in the model, characterized by SCI-definable elements: It seems reasonably clear that *rotate* is not definable in the linear language (though a rigorous proof would be very involved).

Although this points to a limitation in the strict parametricity model, it does not indicate a problem in our syntactic translation. We believe that the elements of the cpo $(\mathbf{exp} \to \mathbf{exp} \to \mathbf{exp} \to \mathbf{exp})^{\circ}I$ that are definable by source and target language terms are the same. Furthermore, it appears that these definable elements (and their lubs) form a cpo that is isomorphic to the cpo $N^* \times N^* \times N^* \to_s N_\bot$ of Kahn-Plotkin sequential functions, where N^* is a stream cpo.

The *rotate* function is a variation on Berry's [1978] example of a stable function that is not sequential. It came to our attention in the context of SCI because of Reddy's coherence space model, where a version of it is also present. This indicates that neither model fully accounts for sequentiality, or for the absence of snapback. Reddy has suggested that one might hope that the approach to sequentiality using games [Abramsky et al. ; Hyland and Ong 1994] might help to resolve this issue. Another possibility would be to use a stronger form of parametricity based, perhaps, on Kripke relations, which have also proven useful in approaching sequentiality [O'Hearn and Riecke 1995; Riecke and Sandholm 1997].

8. RELATION TO FUNCTOR MODELS

While we have said that our translations are based on functor-category models, the precise relationship to them may not be obvious. In particular, the translation for Idealized Algol relies on special properties of the functor-category model defined by Oles [1982, 1997], and does not use standard structure found in all functor categories. Furthermore, the translation for Basic SCI is unusual, in that each identifier is associated with a separate store shape. In this section we describe the ingredients allowing connections between the polymorphic and functor-category forms of semantics to be drawn.

For Idealized Algol we define a logical relation between our model and the functor-category model of Oles (allowing for side effects in expressions). The logical relation establishes that our model agrees with Oles's on the meanings of closed terms of primitive type. A corollary of this is a computational adequacy result, connecting the model to an operational semantics for Idealized Algol. Interestingly, parametricity is not needed for this connection to be made, and it goes through for a model in which $\forall$ is simply an indexed product. For Basic SCI we relate the style of interpretation where each identifier is associated with a separate store shape to a notion of multi-map between functors.

It would be helpful to have some familiarity with functor-category models here. In the next section we return to an analysis of the translations in their own right.

8.1 Idealized Algol

We recall the category Σ of store shapes [Oles 1982; Oles 1987].

—The objects are countable sets.

—The morphisms from W to X are pairs of set-theoretic functions $\varphi : X \to W, \rho : W \times X \to X$ such that
(1) $\forall x \in X.\, \rho\langle\varphi(x), x\rangle \;=\; x$;
(2) $\forall x \in X.\, \forall w \in W.\, \varphi(\rho\langle w, x\rangle) \;=\; w$;
(3) $\forall x \in X.\, \forall w, w' \in W.\, \rho\langle w, \rho\langle w', x\rangle\rangle \;=\; \rho\langle w, x\rangle$.

The propotypical example is the "expansion" morphism $(\varphi, \rho) \colon W \to W \times Y$ where $W \times Y$ is the set-product, φ is the first projection and $\rho\langle w', \langle w, y\rangle\rangle = \langle w', y\rangle$. The composite of morphisms $(\varphi, \rho) \colon W \to X$ and $(\varphi', \rho') \colon X \to Y$ is $(\varphi'', \rho'') \colon W \to Y$ such that $\varphi'' = \varphi'; \varphi$ and $\rho''\langle w, y\rangle = \rho'\langle\rho\langle w, \varphi'(y)\rangle, y\rangle$. The identity morphism on W is (φ, ρ) such that $\varphi(w) = w$ and $\rho(w, w') = w$. (φ, ρ) is an isomorphism iff φ is a bijection.

Let **Predom** denote the category of predomains (possibly bottomless cpo's) and continuous functions. The functor category $\mathbf{Predom}^{\Sigma}$ is Cartesian closed. In fact, $\mathbf{Predom}^{C}$ is Cartesian closed, for any small category C: The internal hom can be given in a standard way using the definition

$$(F \Rightarrow G)X \;=\; \mathbf{Predom}^{C}[C[X, -] \times F, G]. \qquad \text{(Standard Definition)}$$

(Note that this formula is covariant in X, which gives the action on morphisms.) However, in the case that $C = \Sigma$ a factorization of morphisms shown by Oles allows the internal hom to be calculated in a special way.

LEMMA 3. **(Expansion Factorization Lemma)** *Any morphism $(\varphi, \rho) : X \to Y$ in Σ factors as an expansion followed by an isomorphism $X \xrightarrow{e} X \times Z \xrightarrow{i} Y$.*

LEMMA 4. **(Exponent Representation Lemma)** *The internal hom $\Rightarrow$ in $\mathbf{Predom}^{\Sigma}$ is such that*

$$(F \Rightarrow G)X \;\cong\; \mathbf{Predom}^{\Sigma}[F(X \times -), G(X \times -)], \textit{ ordered pointwise}$$

*for any functors $F, G : \Sigma \to$ **Predom** and any countable set X. (We are using $X \times -$ to denote the evident endofunctor on Σ; but note that $\times$ is not the categorical product in Σ.)*

Proof: The proof of the Factorization Lemma is in [Oles 1982]; we sketch why the Representation Lemma follows from it. Any $p \in (F \Rightarrow G)X$ accepts a store shape Y and a pair $((\varphi,\rho) : X \to Y, a \in FY)$ as arguments, and produces a result in GY. Using Oles's lemma, we can factor (φ,ρ) into a composite $X \xrightarrow{e} X \times Z \xrightarrow{i} Y$. By functoriality $F(i^{-1})$ and Gi are both isomorphisms, where i^{-1} is the isomorphism inverse to i. By naturality for the isomorphisms, $p[Y]((\varphi,\rho),a)$ must be equal to $Gi(p[X \times Z](e,(Fi^{-1})a)$. Thus, we can see that p is completely determined by its action on expansions, and this leads in a straightforward fashion to the representation result. ■

Notice that the formula for the internal hom in the Representation Lemma is formally similar to the translation of function types in Idealized Algol. With this it is possible to establish a connection with the translation $(\cdot)^\star$, which we now outline.

First, given the Representation Lemma, we might as well *define* the internal hom as

$$(F \Rightarrow G)X \;=\; \mathbf{Predom}^{\Sigma}[F(X \times -), G(X \times -)]. \qquad \text{(Alternate Definition)}$$

If $e : X \to X \times Y$ is an expansion then the required functorial action of $F \Rightarrow G$ is given (suppressing associativity isomorphisms) by

$$((F \Rightarrow G)e\, p)[W] \;=\; p[Y \times W] : F(X \times Y \times W) \to G(X \times Y \times W).$$

If $i : X \to Y$ is an isomorphism then

$$((F \Rightarrow G)i\, p)[W] \;=\; F(i \times W)^{-1}; p[W]; G(i \times W),$$

and the functorial action for arbitrary Σ-morphisms is then obtained from the Expansion Factorization Lemma.

In order to interpret recursion we need to work in the full subcategory whose objects are functors F where [Oles 1982]: (i) FX has a least element, for each store shape X, and (ii) $F(\varphi,\rho)$ preserves least elements, for each Σ-morphism (φ,ρ). All of the objects we mention will in fact lie in this subcategory.

We can now define a functor $\mathcal{O}[\![\theta]\!]$ for each Idealized Algol type θ. (The product $F \times G$ of functors is defined pointwise as usual.)

$$\begin{aligned}
\mathcal{O}[\![\mathbf{comm}]\!]W \;&=\; W_\perp \multimap W_\perp\\
\mathcal{O}[\![\mathbf{comm}]\!](\varphi,\rho)cs \;&=\; \begin{cases} \perp & \text{if } s = \perp \text{ or } c(\varphi s) = \perp\\ \rho\langle c(\varphi(s)), s\rangle & \text{otherwise}\end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathcal{O}[\![\mathbf{acc}]\!]W \;&=\; W_\perp \otimes N_\perp \multimap W_\perp\\
\mathcal{O}[\![\mathbf{acc}]\!](\varphi,\rho)c\langle s,n\rangle \;&=\; \begin{cases} \perp & \text{if } s = \perp \text{ or } c[(\varphi s), n] = \perp\\ \rho\langle c[\varphi(s), n], s\rangle & \text{otherwise}\end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathcal{O}[\![\mathbf{exp}]\!]W \;&=\; W_\perp \multimap W_\perp \otimes N_\perp\\
\mathcal{O}[\![\mathbf{exp}]\!](\varphi,\rho)es \;&=\; \begin{cases} \perp & \text{if } s = \perp \text{ or } e(\varphi s) = \perp\\ \rho\langle \rho(s,s'), n\rangle & \text{when } \langle s', n\rangle = e(\varphi s)\end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{O}[\![\theta \to \theta']\!] = \mathcal{O}[\![\theta]\!] \Rightarrow \mathcal{O}[\![\theta']\!]\\
&\mathcal{O}[\![\theta \times \theta']\!] = \mathcal{O}[\![\theta]\!] \times \mathcal{O}[\![\theta']\!]
\end{aligned}$$

The morphism part of $\mathcal{O}[\![\mathbf{comm}]\!]$ appears, at first sight, to be nonlinear in the store, where s is copied. Apart from the fact that the two occurrences of s are in different arms of a pairing for a categorical product $\langle \cdot, \cdot \rangle$, this is not a problem because when (φ, ρ) is an expansion the "otherwise" clause of the definition is equivalent to $\rho\langle cx, y\rangle$, and this is what we use in the translation to linear λ-calculus.

Now we set up a logical relation between $(\cdot)^\star$ and $\mathcal{O}[\![\cdot]\!]$, the former understood in terms of the strict parametricity model. For each type θ and countable set X, we define a relation

$$R_\theta^X \subseteq \theta^\star X_\perp \times \mathcal{O}[\![\theta]\!]X.$$

For $\theta = \mathbf{comm}$, $\mathbf{acc}$ or $\mathbf{exp}$, R_θ^W is an identity relation, and product types use the obvious pointwise definition. For procedures,

$$\langle p, q\rangle \in R^X_{\theta \to \theta'} \iff \forall W.\, \langle a, b\rangle \in R_\theta^{X \times W} \Rightarrow \langle p[W_\perp]a, q[W]b\rangle \in R_{\theta'}^{X\times W}.$$

We define R_Γ^X by viewing a type assignment Γ as a product as usual.

Lemma 5. *Each R_θ^X is a pointed, complete relation.*

Lemma 6. **(Expansion Relatedness Lemma)** *If $\langle a, b\rangle \in R_\theta^X$ then*

$$\langle (expand_\theta[X_\perp][Y_\perp]a), (\mathcal{O}[\![\theta]\!](e)b)\rangle \in R_\theta^{X\times Y},$$

where $e : X \to X \times Y$ is the expansion in Σ.

The proof of the Expansion Relatedness Lemma is straightforward, as the definition of $expand_\theta$ merely copies the definition of $\mathcal{O}[\![\theta]\!]e$ in the functor category.

Each judgement $\Gamma \vdash M : \theta$ in Idealized Algol is interpreted as a natural transformation

$$\mathcal{O}[\![M]\!] : \mathcal{O}[\![\Gamma]\!] \stackrel{\cdot}{\to} \mathcal{O}[\![\theta]\!].$$

The interpretations are essentially as in the translations $(\cdot)^\star$, except that we use environment manipulations in place of substitution. Representing environments $u \in \mathcal{O}[\![\Gamma]\!]X$ as functions from identifiers to appropriately-typed values, representative equations are:

$$\begin{aligned}
\mathcal{O}[\![M; M']\!]Wus &= \mathcal{O}[\![M']\!]Wu(\mathcal{O}[\![M]\!]Wus)\\
\mathcal{O}[\![\lambda x : \theta.\, M]\!]Wu[X]d &= \mathcal{O}[\![M]\!](W \times X)\,((\mathcal{O}[\![\Gamma]\!]fu) \mid x \mapsto d)\\
\mathcal{O}[\![M(M')]\!]Wu &= h(\mathcal{O}[\![M]\!]Wu[I](i\mathcal{O}[\![M']\!]Wu))\\
\mathcal{O}[\![\mathbf{new_{comm}}\, M]\!]Wuw &= \begin{cases} w' & \text{if } \mathcal{O}[\![M]\!]Wu[N]v[w, 0] = [w', n]\\ \perp & \text{if } \mathcal{O}[\![M]\!]Wu[N]v[w, 0] = \perp. \end{cases}
\end{aligned}$$

In these definitions, i and h are suitable isomorphisms, $f : W \to W \times X$ is the expansion morphism, N is the set of natural numbers. The standard local variable v is $\langle a, e\rangle$, where $e : (X \times N)_\perp \multimap ((X \times N) \times N)_\perp$ and $a : ((X \times N) \times N)_\perp \multimap (X \times N)_\perp$ are the unique strict functions such that

$e\langle x, n\rangle = \langle\langle x, n\rangle, n\rangle$, and
$a\langle\langle x, n\rangle, m\rangle = \langle x, m\rangle$.

(Notice that a and e are obtained by lifting ρ and φ from the expansion map $(\varphi, \rho) : X \to X \times N$ in the category Σ.)

LEMMA 7. *Suppose $\Gamma \vdash M : \theta$ in Idealized Algol, and that $\langle u, u' \rangle \in R^X_\Gamma$. Then $\langle (M^\star X_\perp\, u), (\mathcal{O}[\![M]\!]X\, u') \rangle \in R^X_\theta$.*

Proof: By induction on M. We indicate the key cases of **new** and λ-abstraction.

For $\mathbf{new}_\varphi(M)$, by induction we know that $\langle p, q \rangle \in R^X_{\mathbf{var} \to \varphi}$, where p is the denotation of M in the strict parametricity model and q in the Oles model. $\mathbf{new}_\varphi(M)$ is defined in the models by supplying local variables v and v' to p and q, and deallocating on termination. Clearly, by the definition of R for procedure types, this case will hold if we can show that the two standard variables are in relation $R^{X \times N}_{\mathbf{var}}$, and this is easy to check from their definitions above and in Section 5.

For the case $\Gamma \vdash \lambda x : \theta.\, M : \theta \to \theta'$, suppose $\langle d, d' \rangle \in R^{X \times Y}_\theta$. We need to show that $\langle m, m' \rangle \in R^{X \times Y}_{\theta'}$, where

$$\begin{aligned} m &= ((M^\star(\alpha \otimes \beta))[\mathit{expand}_{\theta_i}[\alpha][\beta]x_i/x_i])(\alpha \mapsto X_\perp, \beta \mapsto Y_\perp)(u \mid x \mapsto d), \\ m' &= \mathcal{O}[\![M]\!](X \times Y)(\mathcal{O}[\![\Gamma]\!]eu' | x \mapsto d'). \end{aligned}$$

Here, $e : X \to X \times Y$ is the expansion, m is the translation of a λ-abstraction, applied in the strict parametricity model to an appropriate world $Y_\perp$ and argument d', and m' is obtained from the meaning of λ-abstraction in the functor category. Using a substitution lemma for the parametricity model of the linear lambda-calculus we can infer

$$m = (M^\star(X_\perp \otimes Y_\perp)(x_i \mapsto (\mathit{expand}^\star_{\theta_i}[X_\perp][Y_\perp](ux_i)) \,,\, x \mapsto d).$$

Further, $(\mathcal{O}[\![\Gamma]\!](e)u' | x \mapsto d')$ is of the form $(x_i \mapsto \mathcal{O}[\![\theta_i]\!]e(u'x_i), x \mapsto d)$, so we know that the environments $(x_i \mapsto (\mathit{expand}^\star_{\theta_i}[X_\perp][Y_\perp](ux_i)) \,,\, x \mapsto d)$ and $([\![\Gamma]\!](e)u' | x \mapsto d')$ are $R^{X \times Y}_{\Gamma, x:\theta'}$-related, using the Expansion Relatedness Lemma and the assumptions that u and u', and d and d', are related. We can then use the induction hypothesis to conclude that m and m' are related. ■

At primitive types this lemma reduces to the following.

PROPOSITION 8. **(Adequacy with Oles's Model)** *If $\vdash M : \varphi$ in Idealized Algol then $M^\star X_\perp = \mathcal{O}[\![M]\!]X$.*

Furthermore, Lent [1993] has verified that Oles's model satisfies an adequacy correspondence with a suitable operational semantics, to the effect that a closed term of command type converges operationally iff it does not denote $\perp$ (and the proof carries through equally in the presence of expressions with side effects). Thus, adequacy with respect to Oles's model establishes also adequacy with respect to a standard operational semantics. Note that parametricity has not been used thus far in this section, and the adequacy result would go through for the unconstrained model of polymorphism, where $\forall$ is simply indexed product over countable, flat cpo's.

The logical relation, however, contains more information than required for adequacy, which will be put to use in some technical lemmas at the end of the next section.

8.2 Basic SCI

It is not difficult to interpret SCI in $\mathbf{Predom}^\Sigma$, by mimicking $(\cdot)^\circ$, and to verify an adequacy correspondence as was done for Idealized Algol. Rather than do this, we want to relate the somewhat unusual form of interpretation, where each identifier gets a separate piece of the store, to structure found in the functor category; see also [O'Hearn et al. 1999].

First, recall the translation of the SCI function type:

$$(\theta \to \theta')^\circ\alpha \;=\; \forall\beta.\, \theta^\circ\beta \to \theta'^\circ(\alpha \otimes \beta).$$

We can rewrite this in functorial form as follows: If $G, F : \Sigma \longrightarrow \mathbf{Predom}$ are functors, then the functor G^F is defined by the equation

$$(G^F)X \;=\; \mathbf{Predom}^\Sigma[F(-), G(X \times -)], \text{ ordered pointwise.}$$

This formula is covariant in X, which gives the action on morphisms.

What is the sense in which G^F is a function type? One way to explain this would be to find a tensor product of functors satisfying the usual adjunction property

$$\mathbf{Predom}^\Sigma[E \otimes F, G] \;\cong\; \mathbf{Predom}^\Sigma[E, G^F].$$

In fact, G^F is precisely the exponent of functors described by Day [1970], who also gives a recipe for obtaining the required tensor products.

Another, for our purposes more direct, way is to proceed is to look explicitly at maps of multiple arity.

Definition 9. *Suppose $F_1, \ldots, F_n$ and G are functors in $\mathbf{Predom}^\Sigma$. A multi-map*

$$\eta : F_1, \ldots, F_n \longrightarrow G$$

is a family of continuous functions

$$\eta[X_1, \ldots, X_n] : F_1X_1 \times \cdots \times F_nX_n \longrightarrow G(X_1 \times \cdots \times X_n)$$

natural in store shapes $X_1, \ldots, X_n$. (In case $n = 0$ a multi-map is a function from a one-point cpo to $G(1)$, for 1 a one-point set.)

The connection between multi-maps and G^F is immediate from the definitions.

Proposition 10. *Given functors $F_1, \ldots, F_n$, F and G, multi-maps*

$$\eta : F_1, \ldots, F_n, F \longrightarrow G$$

are in bijective correspondence with multi-maps

$$\eta : F_1, \ldots, F_n \longrightarrow G^F.$$

Proposition 10 could be restated in more abstract terms by saying that functors in $\mathbf{Predom}^\Sigma$, together with multi-maps, comprise a *closed multi-category* [Lambek 1989]. We regard this as a kind of categorical justification for, or commentary on, the form of the translation of SCI.

At this point the reader may be wondering why we arranged the $(\cdot)^\circ$ translation in this multi-map form: It would be more standard to follow the structure of a monoidal closed category, where we would interpret a term using a map out of a tensor product. There are two reasons. First, explicitly associating a different piece of local state with each identifier is very appealing, as it directly models the idea that distinct identifiers don't interfere. Second, it is not clear to us how to describe Day's tensor product of functors in polymorphic linear λ-calculus. As a result, if we were to add types $\theta \otimes \theta'$ of noninterfering pairs to SCI, it would not be evident how to interpret them in the linear calculus.

9. ON NATURALITY AND PARAMETRICITY

We have seen a close relationship between our translations and semantic equations in a functor-category model. The main difference is that functor models require meanings to be natural with respect to expansions, where in the strict parametricity model we use binary relational parametricity. In this section we explore circumstances under which the former is implied by the latter. We begin by considering SCI, where the situation is simpler.

9.1 Basic SCI

In SCI, parametricity implies naturality at all types.

THEOREM 11. **(Naturality Theorem for SCI)** *Any $p \in (\theta_1 \to \theta_2)^\circ S$ is natural with respect to expansions: for all flat cpo's S_1 and S_2,*

$$\begin{array}{ccc}
\theta_1^\circ(S_1) & \xrightarrow{\;p[S_1]\;} & \theta_2^\circ(S \otimes S_1) \\
{\scriptstyle expand_{\theta_1}[S_1][S_1']}\downarrow & & \downarrow{\scriptstyle expand_{\theta_2}[S \otimes S_1][S_1']} \\
\theta_1^\circ(S_1 \otimes S_1') & \xrightarrow[\;p[S_1 \otimes S_1']\;]{} & \theta_2^\circ(S \otimes S_1 \otimes S_1')
\end{array}$$

(where we have elided canonical isomorphisms $(S \otimes S_1) \otimes S_2 \leftrightarrow S \otimes (S_1 \otimes S_2)$).

Proof: First, it is easy to see that the result holds for $\theta_1 \to (\theta_2 \times \theta_3)$ iff it holds for $\theta_1 \to \theta_2$ and $\theta_1 \to \theta_3$. By an inductive argument we obtain that it suffices to prove the result for types of the form

$$\theta_1 \to (\theta_2 \to \cdots (\theta_n \to \varphi) \cdots).$$

We prove it for $\varphi = \mathbf{comm}$; the cases of **acc** and **exp** are similar. Also, to keep things simple, we give the proof for the case $n = 2$. This case shows the key role of disjointness between arguments in the proof, and is easily extended to other n.

We are required to show that (a)=(b), where

$$\text{(a)} = p[S_1 \otimes S_1']\big(expand_{\theta_1}[S_1][S_1']a_1\big)[S_2]a_2\,[s, s_1, s_1', s_2]$$

$$\text{(b)} = \big(expand_{\theta_2 \to \mathbf{comm}}[S \otimes S_1][S_1'](p[S_1]a_1)\big)[S_2]a_2\,[s, s_1, s_1', s_2]$$

for all S_2, $[s, s_1, s_1', s_2] \in S \otimes S_1 \otimes S_1' \otimes S_2$, $a_1 \in \theta_1^\circ S_1$ and $a_2 \in \theta_2^\circ S_2$. It is crucial here that a_2 lives in store shape S_2, which is separate from the S_1' component; this allows us to use a relation that fixes S_1' below.

By the definition of *expand* for procedure types, (b) is equal to

$$\text{(c)} = [q, q_1, s_1', q_2], \text{ where } p[S_1]a_1[S_2]a_2[s, s_1, s_2] \;=\; [q, q_1, q_2].$$

Let $R : S_1 \otimes S_1' \leftrightarrow S_1$ be the relation that fixes s_1', i.e. it contains $\langle \bot, \bot \rangle$ and all pairs $\langle [x, s_1'], x \rangle$ where $x \neq \bot$. By the Identity Extension property (a special case of the Isomorphism Functoriality Lemma) we know that

$$\text{(d) } \langle a_1, a_1 \rangle \in \theta_1^\circ \Delta_{S_1} \text{ and } \langle a_2, a_2 \rangle \in \theta_2^\circ \Delta_{S_2}.$$

Further, we claim that

$$\text{(e) } \langle (expand_{\theta_1}[S_1][S_1']a_1), a_1 \rangle \in \theta_1^\circ R.$$

Postponing the proof of (e) for a moment, (d), (e), and parametricity of p imply that (a) and $[q, q_1, q_2]$ are $\Delta_S \otimes R \otimes \Delta_{S_2}$-related, and hence by the definition of R that (a)=(c). Since (b)=(c) we obtain the desired result (a)=(b).

Property (e) follows from parametricity of $\mathit{expand}_{\theta_1} : \forall\beta\forall\gamma.\, \theta_1^\circ\beta \multimap \theta_1^\circ(\beta \otimes \gamma)$. Specifically, we noted $\langle a_1, a_1 \rangle \in \theta_1^\circ \Delta_{S_1}$ in (d), and taking $R' : S_1' \leftrightarrow I$ as the relation containing $\langle \perp, \perp \rangle$ and $\langle s_1', \perp \rangle$, we obtain

(f) $\langle (\mathit{expand}_{\theta_1}[S_1][S_1']a_1), (\mathit{expand}_{\theta_1}[S_1][I]a_1) \rangle \in \theta_1^\circ(\Delta_{S_1} \otimes R')$.

Now, R is equal to the composition of $\Delta_1 \otimes R' : S_1 \otimes S_1' \leftrightarrow S_1 \otimes I$ with the canonical isomorphism $S_1 \otimes I \leftrightarrow S_1$. We can then use Isomorphism Functoriality and parametricity of $\mathit{expand}_{\theta_1}$ once more (with respect to this canonical isomorphism) to obtain (e) from (f). ■

Similarly, we can show that the interpretation

$$x_1 : \theta_1^\circ \alpha_1, \ldots, x_m : \theta_m^\circ \alpha_m \vdash M^\circ(\alpha_1, \ldots, \alpha_n) : \theta^\circ(\alpha_1 \otimes \cdots \otimes \alpha_m).$$

of a judgement satisfies a naturality property corresponding to Definition 9.

9.2 Idealized Algol

The proof of the Naturality Theorem for SCI relies on the simpler interpretation of function types, where the state for procedures and arguments is separate. In Idealized Algol, we have only been able to verify the corresponding result for types of a specific form.

THEOREM 12. **(Naturality Theorem for Idealized Algol)** *If φ is a primitive type then any $p \in (\theta_1 \to \varphi)^\star S$ is natural with respect to expansions:*

$$\begin{array}{ccc}
\theta_1^\star(S \otimes S_1) & \xrightarrow{\quad p[S_1] \quad} & \varphi^\star(S \otimes S_1) \\
{\scriptstyle \mathit{expand}_{\theta_1}[S \otimes S_1][S_1']}\downarrow & & \downarrow{\scriptstyle \mathit{expand}_{\varphi}[S \otimes S_1][S_1']} \\
\theta_1^\star(S \otimes S_1 \otimes S_1') & \xrightarrow[\quad p[S_1 \otimes S_1'] \quad]{} & \varphi^\star(S \otimes S_1 \otimes S_1')
\end{array}$$

(where we have elided canonical isomorphisms $(S \otimes S_1) \otimes S_1' \leftrightarrow S \otimes (S_1 \otimes S_1')$).

The proof can follow the proof of Naturality Theorem in [O'Hearn and Tennent 1995], and is essentially similar to the case of $\theta \to \varphi$ in the proof of Naturality for SCI. The reason that the SCI proof does not generalize can be seen in the case $\theta_1 \to \theta_2 \to \mathbf{comm}$. In Idealized Algol the application $p[S_1 \otimes S_1'](\mathit{expand}[S_1][S_2]a_1)[S_2]a_2$ uses an element $a_2 \in \theta_2^\star(S \otimes S_1 \otimes S_1' \otimes S_2)$ that can change members of S_1', thus preventing the use of a relation that fixes the S_1' component in the proof.

In [O'Hearn and Tennent 1995], Section 9, there was a counterexample to "parametricity implies naturality," which showed that this result failed to hold with φ replaced by arbitrary θ. But the counterexample there used Weakening and Contraction and does not exist in the strict parametricity model. As a result, using the representation in the Resumption Theorem in Section 10.1, we have been able to slightly extend Theorem 12 to cases where φ is replaced by a first-order type $\varphi_1 \times \cdots \times \varphi_n \to \varphi$. The question of whether it holds for all types remains open.

Naturality is important, e.g., for proving the isomorphism $\theta_1 \times \theta_2 \to \theta_3 \;\cong\; \theta_1 \to \theta_2 \to \theta_3$ that is characteristic of Cartesian closed categories. So, even though we have the adequacy result above, we have not verified the indicated CCC isomorphism at higher types in Idealized Algol.

Interesting though the open question may be, it does not pose a fundamental problem for our semantics, as it can be dealt with in at least two ways. First, every Idealized Algol type can be converted to a "canonical form" of a product of types of the form $\theta \to \varphi$, and we could simply use an uncurried presentation of the semantics of types (as in [O'Hearn and Tennent 1995], Section 2). The second option is to add naturality as an explicit additional requirement (as in [O'Hearn and Tennent 1995], Section 7). In either case, we get an interpretation of Idealized Algol using CCC structure, and satisfying all the laws of the typed λ-calculus. Since these two alternatives are thoroughly examined in [O'Hearn and Tennent 1995], the details do not bear repeating here. Instead, we choose to carry working on with the translation $(\cdot)^\star$ as defined (which we have seen satisfies an adequacy property), and simply note that the naturality properties that we can verify are sufficient to support the technical results that we shall prove about low-order types. To this end, we finish this section with some technical lemmas that will be needed later, in Section 10.3.

9.3 Technical Lemmas

We now prove some technical lemmas that will be used to help produce a distinguishing context during the proof of full abstraction in the next section; the reader may wish to move on and refer back to these lemmas as necessary.

The first lemma reduces the domain approximation $p \sqsubseteq q$ between certain polymorphic functions to the ordering $p[N_\perp] \sqsubseteq q[N_\perp]$ of their instantiations to $N_\perp$. This is one of the steps which enables us to use a single local variable, which lives in store shape $N_\perp$, to produce a distinguishing context.

The use of I in the lemma corresponds, semantically, to meanings for closed terms of the indicated type.

Lemma 13. *If $p, q \in \theta^\star I$, where $\theta = (\varphi_1 \times \cdots \times \varphi_n \to \varphi) \to \varphi'$, then $p \sqsubseteq q$ iff $p[N_\perp] \sqsubseteq q[N_\perp]$.*

Proof: The only if part is immediate. For the if part, consider any countable, flat cpo S, and suppose $p[N_\perp] \sqsubseteq q[N_\perp]$. We must show $p[S] \sqsubseteq q[S]$ for arbitrary S. If S is $\{\perp\}$ then the result is trivial. Otherwise, first note that $N_\perp$ and $S \otimes N_\perp$ are isomorphic, since both are countably infinite and flat. Using the Isomorphism Functoriality Lemma and parametricity we infer $p[S \otimes N_\perp] \sqsubseteq q[S \otimes N_\perp]$. We know that p and q are natural with respect to expansions by Theorem 12, and so we can then show that $p[S] \sqsubseteq q[S]$ using naturality with respect to the expansion $S \multimap S \otimes N_\perp$ and the easy fact that expansion maps $\mathit{expand}_{\varphi'}[S_1][S_2]$ reflect order for primitive types φ'. ■

The next lemma refers to the logical relation R between $(\cdot)^\star$ and $\mathcal{O}[\![\cdot]\!]$ defined in Section 8.1.

Lemma 14. *If $\theta = (\varphi_1 \times \cdots \times \varphi_n \to \varphi) \to \varphi'$ then the opposite of relation R^X_θ is single-valued: if $pR^X_\theta m$ and $p' R^X_\theta m$ then $p = p'$.*

The point of this result is that, at first order types, the relationship between the two models can be described by a partial function; an element in the Oles model can correspond to at most one element in the parametricity model (which is more constrained).

Proof: Assume the antecedent. By Theorem 12 we have an injection

$$i : ((\varphi_1 \times \cdots \times \varphi_n) \to \varphi)^\star(X \times W)_\perp \hookrightarrow \mathcal{O}[\![(\varphi_1 \times \cdots \times \varphi_n) \to \varphi]\!](X \times W),$$

and it is clear that if $q \in ((\varphi_1 \times \cdots \times \varphi_n) \to \varphi)^\star(X \times W)_\perp$ then $qR^{X\times W}_{\varphi_1\times\cdots\times\varphi_n\to\varphi}(iq)$. By the logical relation property for p and p' this means that

$$(p[W]q)R^{X\times W}_{\varphi'}(m[W](iq)) \quad \text{and} \quad (p'[W]q)R^{X\times W}_{\varphi'}(m[W](iq)).$$

But the logical relation $R^{X\times W}_{\varphi'}$ at primitive type is an equality relation. Thus, we have shown that $p[W]q = p'[W]q$ for arbitrary W and q, so $p = p'$. ∎

Lemma 15. *If* $\vdash M : \theta$, *where* $\theta = (\varphi_1 \times \cdots \times \varphi_n \to \varphi) \to \varphi'$, *then* M *is natural:*

$$\begin{array}{ccc} I & \xrightarrow{M^\star S} & \theta^\star S \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle expand_\theta[S][S']} \\ I & \xrightarrow[M^\star(S\otimes S')]{} & \theta^\star(S \otimes S') \end{array}$$

Proof: Let $S = X_\perp$ and $S' = X'_\perp$. From Lemma 7 we know that

$$(M^\star X_\perp *)R^X_\theta(\mathcal{O}[\![M]\!]X*) \quad \text{and} \quad (M^\star(X \times X')_\perp *)R^{X\times X'}_\theta(\mathcal{O}[\![M]\!](X \times X')*).$$

By Expansion Relatedness (Lemma 6),

$$(expand_\theta[S][S'](M^\star X_\perp *))R^{X\times X'}_\theta(\mathcal{O}[\![\theta]\!]\, e\, (\mathcal{O}[\![M]\!]X*)),$$

where $e : X \to X \times X'$ is the expansion. Further, by naturality of $\mathcal{O}[\![M]\!]$ we know

$$\mathcal{O}[\![\theta]\!]\, e\, (\mathcal{O}[\![M]\!]X*) \;=\; \mathcal{O}[\![M]\!](X \times X') * .$$

Naturality follows from these facts together with Lemma 14 and the standard isomorphism $(X \times X')_\perp \cong S \otimes S'$. ∎

The order $\sqsubseteq$ on p and q in Lemma 13 is determined pointwise. Likewise, any derivable judgement $\Gamma \vdash M : \theta$ in Idealized Algol determines a family of maps $M^\star(-) : \Gamma^\star(-) \multimap \theta^\star(-)$ in the strict parametricity model, which we also order pointwise.

Lemma 16. *If* $\vdash M, N : \theta$, *where* $\theta = (\varphi_1 \times \cdots \times \varphi_n \to \varphi) \to \varphi'$, *then*

$$M^\star \sqsubseteq N^\star \quad \textit{iff} \quad M^\star I \sqsubseteq N^\star I.$$

Proof: The only if part is immediate from the pointwise order for $\sqsubseteq$. For the if part, if $M^\star I \sqsubseteq N^\star I$ we get $M^\star(I \otimes S) \sqsubseteq N^\star(I \otimes S)$ from naturality (Lemma 15), and then $M^\star S \sqsubseteq N^\star S$ using the Isomorphism Functoriality Lemma (Section 6) with the isomorphism $S \leftrightarrow I \otimes S$. ∎

10. RESUMPTIONS AND IDEALIZED ALGOL

In this section we show that the meanings of first-order types in our model of Idealized Algol may be characterized using domain equations for resumptions. The proof of this result goes by a parametricity argument which characterises the elements of these types exactly. Using this characterization, we derive a full abstraction result for closed terms of second-order type.

We have already seen in Section 7.3 that such representation results do not go through in our model of SCI.

10.1 A Representation Theorem

The domain equation we give will need to deal with a number of different capabilities for the primitive types, and it will help to look at some instances of it before giving the general equation.

To begin, $(\mathbf{comm} \rightarrow \mathbf{comm})^{\star}S$ is an initial solution of the familiar domain equation for deterministic resumptions [Plotkin 1983]:

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

where S is a flat cpo and $\oplus$ is coalesced sum. Using this domain equation we understand a procedure of type $\mathbf{comm} \rightarrow \mathbf{comm}$ as follows. A procedure with a command argument begins by possibly changing the state. It then either terminates or resumes. Resuming causes the argument to be evaluated once, with the state resulting from this evaluation fed through as an argument to the resumed procedure.

For expressions we augment deterministic resumptions with input and output capabilities. The equation for $\mathbf{exp} \rightarrow \mathbf{exp}$ is

$$D \;\cong\; S \multimap \big((S \otimes N_{\perp}) \oplus (S \otimes (N_{\perp} \multimap D))\big).$$

This equation allows for an output value in the component $S \otimes N_{\perp}$ for the final value of execution of the resumption, and an input value in $N_{\perp} \multimap D$ that is obtained from evaluating the input expression once.

For acceptors to the left of $\rightarrow$ consider a procedure $p : \mathbf{acc} \rightarrow \mathbf{comm}$. A procedure call $p(a)$ is a command which may assign a number of values to a during its evaluation. The appropriate domain equation is the one for $\mathbf{comm} \rightarrow \mathbf{comm}$, augmented with natural number values to be supplied to the input acceptor on resumption:

$$D \;\cong\; S \multimap \big(S \oplus (S \otimes D \otimes N_{\perp})\big).$$

For acceptors to the right of $\rightarrow$ consider a procedure $p : \mathbf{acc} \rightarrow \mathbf{acc}$. A procedure call $p(a)$ is an acceptor, so that it accepts a natural number value and then behaves like a command, during evaluation of which a can be used a number of times. Accordingly, $(\mathbf{acc} \rightarrow \mathbf{acc})^{\star}S$ is isomorphic to $N_{\perp} \multimap D$, where D is the domain just given for $\mathbf{acc} \rightarrow \mathbf{comm}$.

This treatment of acceptors can in fact be regarded as being derived from a view of $\mathbf{acc}$ as an infinite product of $\mathbf{comm}$; semantically, $\mathbf{acc}^{\star}S$ is isomorphic to an infinite product $(\mathbf{comm}^{\star}S)^{N}$. For acceptors to the right of $\rightarrow$, the domain $N_{\perp} \multimap D$ is isomorphic to the infinite product D^{N}. To the left of $\rightarrow$ the term $S \otimes D \otimes N_{\perp}$ in the domain equation can be isomorphically rewritten to $\Sigma_{i \in N} S \otimes D$, for Σ the coalesced sum: Below we will use summation in this way to account for explicit product types to the left of $\rightarrow$.

To formulate the general equation, for each primitive type φ we define functors φ^{in}, φ^{arg} and and φ^{res} (for input, argument, and result) on the category of cpo's and strict continuous functions:

$$\begin{array}{lll} \mathbf{exp}^{in} \;=\; (-) & \mathbf{exp}^{arg} \;=\; N_\perp \multimap (-) & \mathbf{exp}^{res} \;=\; (-) \otimes N_\perp \\ \mathbf{acc}^{in} \;=\; N_\perp \multimap (-) & \mathbf{acc}^{arg} \;=\; (-) \otimes N_\perp & \mathbf{acc}^{res} \;=\; (-) \\ \mathbf{comm}^{in} \;=\; (-). & \mathbf{comm}^{arg} \;=\; (-) & \mathbf{comm}^{res} \;=\; (-) \end{array}$$

Now we add the capability to choose among multiple arguments to the domain equations above.

THEOREM 17. **(Resumption Theorem)** *If $\varphi_1, ..., \varphi_n, \varphi$ are primitive Idealized Algol types and S a countable flat cpo, then*

$$(\varphi_1 \times \cdots \times \varphi_n \rightarrow \varphi)^{\star} S$$

is isomorphic to $\varphi^{in}(D)$, where D is an initial solution of the domain equation

$$D \;\cong\; S \multimap \big(\varphi^{res}(S) \oplus (S \,\otimes\, (\Sigma_{i=1}^{n} \varphi_i^{arg}(D)))\big),$$

where $\oplus$ and Σ are coalesced sum.

The summation over $i \in 1, ..., n$ here corresponds to a strong form of sequentiality where a resumption must choose a specific i, which causes the i'th argument to be evaluated before resuming. This is analogous to strategies for picking sequentiality indices in sequential algorithms [Berry and Curien 1982]: In contrast, Kahn-Plotkin sequential functions require mere existence of a sequentiality index, and as a result are less distinguishing than the stronger form of sequentiality found in Idealized Algol. The weaker sequential-function form of sequentiality is closer to functions in SCI, where non-interference leads to less dependence on evaluation order than in Idealized Algol. (So the parallelism in SCI is reminiscent of independence from evaluation order in PCF, and appears to be actually sequential in this weaker sense of sequentiality.)

The proof of the Resumption Theorem is lengthy, and we give it separately in the following subsection. In describing the proof we have attempted to isolate the more conceptual part, the parametricity argument, from the technical development needed to apply the argument to verify the theorem.

10.2 Proof of the Theorem

To keep the notation simple we give the proof for the type $\mathbf{exp} \times \mathbf{exp} \rightarrow \mathbf{exp}$. This requires us to address the most important issues raised by the domain equation, including multiple arguments and possibly non-trivial input and output values. The proof for more or fewer than two arguments involving **comm** or **exp** is a straight notational reworking of what follows, and **acc** requires only minor modifications (recall the remarks about **acc** as a infinite product of **comm** above).

The relevant cpo in the strict parametricity model is

$$\begin{aligned} Z \;=\;& \forall \beta.\, (S \otimes \beta \multimap S \otimes \beta \otimes \mathbf{nat}) \& (S \otimes \beta \multimap S \otimes \beta \otimes \mathbf{nat}) \\ & \rightarrow (S \otimes \beta \multimap S \otimes \beta \otimes \mathbf{nat}). \end{aligned}$$

Let D be an initial solution of the equation $D \cong TD$, where T is the following functor on the category of cpo's and strict continuous functions:

$$T(-) \;=\; S \multimap S \otimes \big(N_\perp \oplus (N_\perp \multimap (-)) \oplus (N_\perp \multimap (-))\big).$$

(The functor $S \multimap \big((S \otimes N_\perp) \oplus (S \otimes \Sigma_{i=1}^{2}(N_\perp \multimap (-)))\big)$ is explicitly given by the statement of the theorem, but it is isomorphic to T, which is obtained by distributing $\oplus$ over $\otimes$.)

Our aim is to show that Z and D are isomorphic. To do this, we proceed in four steps. First, we describe the structure of D. This is straightforward, and follows from standard material on recursive domain equations [Plotkin 1983; Abramsky and Jung 1994]. Second, we give the parametricity argument, which relates uses of polymorphic functions to the structure described in the first step. Steps three and four consist of technical results, which use the information from the first two steps to verify that D and Z are isomorphic.

Step 1: Structure of D.

From the solution of domain equations by the inverse limit construction [Plotkin 1983; Abramsky and Jung 1994], we know that D is a *bounded complete, algebraic cpo*. As a result, any element is the lub of the finite elements beneath it. The first step needed for relating D to Z is to describe this structure explicitly. We do this using step functions. If E is a flat cpo, F is bounded complete and algebraic, $e \in E$ is not $\perp$, and $f \in F$ is finite, then the step function $e \searrow f : E \multimap F$ is such that

$$(e \searrow f)e' \;=\; \begin{cases} f & \text{if } e = e' \\ \perp & \text{otherwise.} \end{cases}$$

The domain D can be described using special finite elements, the atoms. Recall that an atom a in a cpo is an element just above $\perp$, i.e. $a \neq \perp$ and $\forall b.\, b \sqsubseteq a \Longrightarrow b = a \vee b = \perp$. The atoms $a \in D$ can be generated using step functions as follows:

$$\begin{array}{lll} a & ::= & s \searrow [s', e] \qquad\qquad\qquad\qquad (\perp \neq s, s' \in S) \\ e & ::= & n \mid 1 : n \searrow a \mid 2 : n \searrow a \qquad\qquad (n \in N) \end{array}$$

Here, we use n, $1 : n \searrow a$ and $2 : n \searrow a$ to indicate elements in respective components of the coproduct $N_\perp \oplus N_\perp \multimap D \oplus N_\perp \multimap D$. We will follow the usual practice of suppressing isomorphisms between D and TD. For instance, $s \searrow [s', n]$ is, strictly speaking, an element of TD, which we could coerce to D using an isomorphism.

It is not difficult to verify that each element a generated by this grammar is indeed an atom, and also that this exhausts the atoms of D. The role of the atoms in D is summed up by the following lemma.

LEMMA 18. *D is isomorphic to $\{A \subseteq D \mid A$ is a pairwise-consistent set of atoms$\}$, ordered by subset inclusion. (That is, D is a coherence space.)*

Here, two elements are said to be consistent when they have an upper bound. The lemma can be shown using a routine calculation with the inverse limit construction.

Step 2: The Parametricity Argument.

The essential part of the proof is the following use of parametricity. Suppose $p \in Z$, E is a flat cpo, $c_1, c_2 : S \otimes E \multimap S \otimes E \otimes N_\perp$ and $p[E]\langle c_1, c_2\rangle[s_0, e_0] = [s_0', e_0', n'] \neq \perp$. We identify an element of D that tracks the calls of c_1 and c_2.

Let L be the flat cpo whose non-$\perp$ elements are lists of elements drawn from the set coproduct $S' + N + N$, where S' is the set of non-$\perp$ elements of S. We use s, $1 : n$, and $2 : n$, with subscripts or superscripts, to range over components of the coproduct. Define $R : L \leftrightarrow E$ to be the smallest strict relation such that

$$(s_0) R e_0 \qquad\qquad c_i[s, e] = [s', e', n] \neq \perp \text{ and } \ell R e \implies (\ell, s, s', i : n) R e'.$$

The initial and final values of the state on a use of $c_i[s, e]$, together with component i (which is 1 or 2) and produced value n, are recorded in $(\ell, s, s', i : n)$. (We are using a comma for appending an element to a list.) R is single-valued and determines a function $f_R : L \multimap E$, where $f_R \ell = e$ if $\ell R e$, and $f_R \ell = \perp$ if there is no e such that $\ell R e$. For i equal to 1 or 2, define $c_i^* : S \otimes L \multimap S \otimes L \otimes N_\perp$ to be the function where

$$c_i^*[s, \ell] \;=\; \textbf{let } [s', e, n] \textbf{ be } c_i[s, f_R \ell] \textbf{ in } [s', (\ell, s, s', i : n), n].$$

This definition is such that

$$\langle (s_0), e_0 \rangle \in R \qquad \wedge \qquad \langle c_i^*, c_i \rangle \in \Delta_S \otimes R \multimap \Delta_S \otimes R \otimes \Delta_{N_\perp}.$$

Parametricity of p then implies that

$$\langle\, p[L]\langle c_1^*, c_2^*\rangle[s_0, (s_0)]\,,\, p[E]\langle c_1, c_2\rangle[s_0, e_0] \rangle \in \Delta_S \otimes R \otimes \Delta_{N_\perp}.$$

By the definition of R, this means $p[L]\langle c_1^*, c_2^*\rangle[s_0, (s_0)] \neq \perp$. Further, the use of $\Delta_S \otimes R \otimes \Delta_{N_\perp}$ implies that $p[L]\langle c_1^*, c_2^*\rangle[s_0, (s_0)] \in S \otimes L \otimes N_\perp$ is of the form

$$[s_0', (s_0, s_1, s_1', i_1 : n_1, ..., s_k, s_k', i_k : n_k), n']$$

for some, possibly empty, list $s_1, s_1', i_1 : n_1 ..., s_k, s_k', i_k : n_k$ (here s_0' and n' are the final values in $p[E]\langle c_1, c_2\rangle[s_0, e_0] = [s_0', e_0', n']$). From this we can read off the behaviour of $p[E]\langle c_1, c_2\rangle[s_0, e_0]$ as a sequence of calls to c_1 and c_2, together with initial and final states and values produced when each call was made.

This determines an atom

$$s_0 \searrow [s_1, i_1 : (n_1 \searrow (s_1' \searrow \cdots [s_k, i_k : (n_k \searrow (s_k' \searrow [s_0', n']))] \cdots))]$$

in D, which we denote by $a(E, p, c_1, c_2, s_0, e_0)$. The logical relation argument we have just given is a recipe for obtaining $a(E, p, c_1, c_2, s_0, e_0)$ from appropriately typed E, p, c_1, c_2, s_0, e_0, as long as $p[E]\langle c_1, c_2\rangle[s_0, e_0] \neq \perp$.

This completes the most important part of the proof. The remainder of the section consists of technical lemmas, showing how this way of generating atoms in D from elements of Z can be used to establish an isomorphism between D and Z.

Step 3: From D to Z.

We obtain a map from D to Z using the fact that there is an initial T-algebra $TD \xrightarrow{h_D} D$, the structure map h_D of which is an isomorphism [Abramsky and Jung 1994]. Initiality implies that there is a (unique) strict continuous function

$\psi : D \to Z$ making

$$\begin{tikzcd} TD \arrow[r, "T\psi"] \arrow[d, "h_D"'] & TZ \arrow[d, "h_Z"] \\ D \arrow[r, "\psi"'] & Z \end{tikzcd}$$

commute, where the T-algebra structure map $TZ \stackrel{h_Z}{\to} Z$ is as follows:

$$\begin{array}{l} h_Z \;=\; \lambda p \in TZ \\ \qquad \Lambda\beta \;\; \lambda\langle c_1, c_2\rangle : (S \otimes \beta \multimap S \otimes \beta \otimes \mathbf{nat}) \& (S \otimes \beta \multimap S \otimes \beta \otimes \mathbf{nat}) \\ \qquad \underline{\lambda}[s, b] : S \otimes \beta \\ \qquad\qquad \mathbf{let}\ [s', x]\ \mathbf{be}\ p\, s\ \mathbf{in} \\ \qquad\qquad \mathbf{case}\ x\ \mathbf{of} \\ \qquad\qquad\qquad n \;\mapsto\; [s', b, n] \\ \qquad\qquad\qquad 1 : f \;\mapsto\; (\mathbf{let}\ [s'', b', n]\ \mathbf{be}\ c_1[s', b]\ \mathbf{in}\ (f\, n)[\beta]\langle c_1, c_2\rangle[s'', b']) \\ \qquad\qquad\qquad 2 : g \;\mapsto\; (\mathbf{let}\ [s'', b', n]\ \mathbf{be}\ c_2[s', b]\ \mathbf{in}\ (g\, n)[\beta]\langle c_1, c_2\rangle[s'', b']). \end{array}$$

We are using polymorphic λ-calculus notation as a convenient way of defining elements of Z in a way that makes the satisfaction of parametricity conditions obvious. The **case** construct is used to branch on coproduct components, where n, $(1 : f)$, and $(2 : g)$ range over components of $N_\perp \oplus (N_\perp \multimap E) \oplus (N_\perp \multimap E)$. Note that the use of **case** is not a problem for parametricity because we are not branching on anything we are required to be parametric in.

The map ψ can be obtained from general considerations ([Abramsky and Jung 1994], Lemma 5.3.1) as a least fixed-point:

$$\psi \;=\; \mu g.\, (h_D^{-1}; T(g); h_Z).$$

For concreteness, it will be useful to spell out the action of ψ on atoms. Consider an atom

$$a \;=\; s_0 \searrow [s_1, i_1 : (n_1 \searrow (s_1' \searrow \cdots [s_k, i_k : (n_k \searrow s_k' \searrow [s_0', n'])] \cdots))]$$

in D. Then we define the polymorphic function $\psi(a) \in Z$ by induction on the size of a. The base case is for atoms of the form $s_0 \searrow [s_0', n']$, and $\psi(a)$ is then the function

$$\Lambda\beta.\, \lambda\langle c_1, c_2\rangle.\, \underline{\lambda}[s, b].\, \mathbf{if}\, (s = s_0)\, \mathbf{then}\, [s_0', b, n']\, \mathbf{else}\, \perp.$$

For the induction case, suppose that we know the polymorphic function $\psi(a')$ corresponding to the atom

$$a' \;=\; (s_1' \searrow \cdots [s_k, i_k : (n_k \searrow s_k' \searrow [s_0', n'])] \cdots).$$

Then $\psi(a)$ is

$$\begin{array}{l} \Lambda\beta.\, \lambda\langle c_1, c_2\rangle.\, \underline{\lambda}[s, b]. \\ \quad \mathbf{if}\, (s = s_0)\, \mathbf{then} \\ \quad\;\; \big(\mathbf{let}\, [s', b', n] = c_{i_1}[s_1, b]\, \mathbf{in} \\ \qquad \mathbf{if}\, n = n_1\, \mathbf{then}\, \psi(a')[\beta]\langle c_1, c_2\rangle[s', b']\, \mathbf{else} \perp\big) \\ \quad \mathbf{else} \perp. \end{array}$$

It is not difficult to see that this description is consistent with the abstract definition of ψ as a least fixed-point.

From Lemma 18 we know that D is determined by its atoms, and consistency between them. A key point is that ψ respects this consistency structure.

LEMMA 19. *(1) If ψd and $\psi d'$ are consistent in Z then d and d' are consistent in D.*

(2) ψ reflects order: $\psi d \sqsubseteq \psi d' \implies d \sqsubseteq d'$.

Proof: For part (1) of the lemma, by Lemma 18 it suffices to prove the result for atoms only. From the description of atoms in D it is clear that, if d and d' are inconsistent, then we must have a situation in which

$$\begin{array}{rcl} d & = & s_0 \searrow [s_1, i_1 : (n_1 \searrow (s'_1 \searrow \cdots [s_k, i_k : (n_k \searrow s'_k \searrow [q, e])] \cdots))] \\ d' & = & s_0 \searrow [s_1, i_1 : (n_1 \searrow (s'_1 \searrow \cdots [s_k, i_k : (n_k \searrow s'_k \searrow [q', e'])] \cdots))] \end{array}$$

where one of three cases must be true:

(1) $e = m$ and $e' = n$ are in the first component, and $m \neq n$,
(2) $q \neq q'$, or
(3) e and e' are in different components.

For each case it is straightforward to find arguments E, c_1, c_2 and e_0 such that $(\psi\, d)[E]\langle c_1, c_2\rangle[s_0, e_0]$ and $(\psi\, d')[E]\langle c_1, c_2\rangle[s_0, e_0]$ are unequal and non-$\perp$. For instance, in the first case we can choose $E = N_\perp$, $e_0 = 0$, and for y equal to 1 or 2 define c_y to be the least function such that $c_y[s_j, j-1] = [s'_j, j, n_j]$, for $1 < j < k$. A routine calculation with the definition of ψ shows that $(\psi\, d)[E]\langle c_1, c_2\rangle[s_0, e_0]$ and $(\psi\, d')[E]\langle c_1, c_2\rangle[s_0, e_0]$ are inconsistent. The other two cases are similar.

Part (2) of the lemma can also be shown straightforwardly by proving the contrapositive, constructing arguments that show $\psi d \not\sqsubseteq \psi d'$ when $d \not\sqsubseteq d'$. ■

From part (2) of this result it follows that ψ is an an injection that preserves and reflects order, so that D "sits inside" Z. The next result sets the stage for the proof that ψ is bijective; it shows that the behaviour of any polymorphic function $p \in Z$ is completely determined by elements of the form $\psi(a)$ lying below it, for a an atom of D.

LEMMA 20. *(1) $\psi(a(E, p, c_1, c_2, s_0, e_0))[E]c[s_0, e_0] = p[E]\langle c_1, c_2\rangle[s_0, e_0]$*

(2) $\psi(a(E, p, c_1, c_2, s_0, e_0)) \sqsubseteq p$.

Proof: (1) follows straightforwardly from the relational parametricity property for $\psi(a(E, p, c_1, c_2, s_0, e_0))$, with the relation R used to define $a(E, p, c_1, c_2, s_0, e_0)$ above. For (2) we need more work because we must consider cpo's other than E. That is, we must show the following for every flat cpo B:

(a) If $\psi(a(E, p, c_1, c_2, s_0, e_0))[B]\langle h_1, h_2\rangle[q_0, b_0] = [q'_0, b'_0] \neq \perp$ then $p[B]\langle h_1, h_2\rangle[q_0, b_0] = [q'_0, b'_0]$.

Fix B and assume the antecedent of (a). Note that we must have $s_0 = q_0$, or else we would have $\psi(a(E, p, c_1, c_2, s_0, e_0))[B]\langle h_1, h_2\rangle[q_0, b_0] = \perp$.

Let L and $R : L \leftrightarrow E$ be as in the parametricity argument that defined the atom $a(E, p, c_1, c_2, s_0, e_0)$, and define $Q : L \leftrightarrow B$ as the least strict relation such that

$$(s_0) Q b_0 \qquad h_i[q, e] = [q', b', n] \neq \perp \,\wedge\, \ell Q b \Rightarrow (\ell, q, q', i : n) Q b'.$$

We define f_Q and h_i^* similarly to f_R and c_i^*. Again we have

$$\langle (s_0), b_0 \rangle \in Q \quad \wedge \quad \langle h_i^*, h_i \rangle \in \Delta_S \otimes Q \multimap \Delta_S \otimes Q \otimes \Delta_{N_\perp},$$

so, by parametricity (remembering $s_0 = q_0$) we obtain

(b) $\langle\, p[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)]\,,\, p[B]\langle h_1, h_2\rangle[s_0, b_0]\rangle \in \Delta_S \otimes Q \otimes \Delta_{N_\perp}$,

and

(c) $\big\langle\, \psi(a(E, p, c_1, c_2, s_0, e_0))[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)],$
$\psi(a(E, p, c_1, c_2, s_0, e_0))[B]\langle h_1, h_2\rangle[s_0, b_0]\big\rangle \in \Delta_S \otimes Q \otimes \Delta_{N_\perp}$.

Then (b) and (c) imply, since Q is single-valued, that to prove (a) it suffices to show

(d) $\psi(a(E, p, c_1, c_2, s_0, e_0))[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)] \;=\; p[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)]$.

Now, the atom $a(E, p, c_1, c_2, s_0, e_0)$ must be of the form

$$s_0 \searrow [s_1, i_1 : (n_1 \searrow (s_1' \searrow \cdots [s_k, i_k : (n_k \searrow s_k' \searrow [s_0', n'])] \cdots))].$$

Fixing k, for each $0 \le j \le k$ define

$$\ell_j \;=\; (s_0, s_1, s_1', i_1 : n_1 ..., s_j, s_j', i_j : n_j),$$

and, for i equal to 1 or 2, define c_i^{min} to be the function where

$$c_i^{min}[s_j, \ell_{j-1}] \;=\; [s_j', \ell_j, n_j], \text{ for } 1 \le j \le k,$$

and $c_i^{min}[s, \ell] = \perp$ in all other cases. Define $U : L \leftrightarrow L$ to be the relation containing $\langle \perp, \perp \rangle$, all pairs $\langle \ell_j, \ell_j \rangle$ for $0 \le j \le k$, and $\langle \perp, \ell \rangle$ for all ℓ's whose length is greater than $3k + 1$ (which is the length of ℓ_k).

Intuitively, c_i^{min} is like c_i^*, except that it is just defined enough to enable the polymorphic application $p[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)]$ to be non-$\perp$.

More formally, it is easy to see that

$$\langle c_i^{min}, c_i^* \rangle \in \Delta_S \otimes U \multimap \Delta_S \otimes U \otimes \Delta_{N_\perp},$$

and parametricity then implies

$$\langle\, p[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)]\,,\, p[L]\langle c_1^*, c_2^*\rangle[s_0, (s_0)]\rangle \in \Delta_S \otimes U \otimes \Delta_{N_\perp}.$$

We know from the parametricity argument that defined $a(E, p, c_1, c_2, s_0, e_0)$ that

$$p[L]\langle c_1^*, c_2^*\rangle[s_0, (s_0)] \;=\; [s_0', \ell_k, n'].$$

Further, the definition of U implies that

$$\text{if } \langle \ell, \ell_k \rangle \in U \text{ then } \ell \;=\; \ell_k,$$

so we may conclude that

$$p[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)] \;=\; [s_0', \ell_k, n'].$$

In particular, it is not $\perp$.

Next, we claim

(e) $\langle c_i^{min}, h_i^* \rangle \in \Delta_S \otimes U \multimap \Delta_S \otimes U \otimes \Delta_{N_\perp}$.

To see why (e) must hold, consider how c_i^{min} is defined in terms of the atom. If (e) did not hold then we would have

$$\psi(a(E, p, c_1, c_2, s_0, e_0))[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)] = \bot,$$

and that cannot be the case because, by (c) and the definition of Q, it would contradict the assumed antecedent of (a).

From (e) and parametricity of p we obtain

$$\langle\, p[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)]\,,\, p[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)]\rangle \in \Delta_S \otimes U \otimes \Delta_{N_\bot}.$$

By the definition of U and the fact, shown above, that $p[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)] \neq \bot$, this implies

$$\text{(f)}\;\; p[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)] \;=\; p[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)].$$

By a similar (but easier) argument, parametricity of $\psi(a(E, p, c_1, c_2, s_0, e_0))$ with respect to U implies

$$\begin{array}{l}\text{(g)}\;\; \psi(a(E, p, c_1, c_2, s_0, e_0))[L]\langle c_1^{min}, c_2^{min}\rangle[s_0, (s_0)] \\ \;= \;\psi(a(E, p, c_1, c_2, s_0, e_0))[L]\langle h_1^*, h_2^*\rangle[s_0, (s_0)].\end{array}$$

The desired result (d) follows directly from (f) and (g). ∎

Step 4: From Z to D.

Finally, we define the inverse map $\kappa : Z \to D$:

$$\kappa\, p \;=\; \bigsqcup\{a \in D \mid \psi a \sqsubseteq p \text{ and } a \text{ is an atom}\}.$$

Lemma 21. *κ is well-defined.*

Proof: If $\psi d, \psi d' \sqsubseteq p$ then d and d' are consistent by Lemma 19(1). Thus, the set $\{a \in D \mid \psi a \sqsubseteq p$ and a is an atom$\}$ is pairwise consistent, and by Lemma 18 its lub exists. ∎

This shows that κ is a function but we do not, as yet, claim that it is continuous.

Lemma 22. *$\kappa\psi d \;=\; d$.*

Proof:

$$\begin{array}{rll}\kappa\psi d & = \bigsqcup\{a \in D \mid \psi a \sqsubseteq \psi d \text{ and } a \text{ is an atom}\} & \text{(definition)} \\ & = \bigsqcup\{a \in D \mid a \sqsubseteq d \text{ and } a \text{ is an atom}\} & \text{(Lemma 19(2))} \\ & = d & \text{(Lemma 18)}\end{array}$$

∎

Lemma 23. *$\psi\kappa p \;=\; p$.*

Proof: Certainly we have $\psi\kappa p \sqsubseteq p$ by the definition of κ. Conversely, suppose that $p[E]\langle c_1, c_2\rangle[s_0, e_0] = [s_0', e_0']$ and consider $a(E, p, c_1, c_2, s_0, e_0) \in D$. We know from Lemma 20(1) that $\psi(a(E, p, c_1, c_2, s_0, e_0)) \sqsubseteq p$, and since, by definition, $a(E, p, c_1, c_2, s_0, e_0)$ is an atom in D, the inequality $a(E, p, c_1, c_2, s_0, e_0) \sqsubseteq \psi\kappa p$ follows from the definition of κ. This and Lemma 20(2) imply the desired equality $(\psi\kappa p)[E]\langle c_1, c_2\rangle[s_0, e_0] = [s_0', e_0']$, and we are done. ∎

To prove the Resumption Theorem, note that in the specific category of cpo's and strict continuous functions any order-reflecting bijection is an isomorphism. Lemmas 22 and 23 show that ψ is a bijection, so by Lemma 19 we conclude that it is an isomorphism (and we can finally infer that κ is continuous).

This Concludes the Proof of the Resumption Theorem

After all this work it is reasonable to ask if Theorem 17 could be seen as an instance of a more general result about relational parametricity in linear type theory. In suitably parametric models of (intuitionistic) polymorphic λ-calculus types of the form $\forall\alpha.\,(T\alpha \to \alpha) \to \alpha$ denote initial T-algebras, and this paves the way for a characterization of all second-order types in prenex form. Plotkin [1993] in lectures has indicated that the corresponding property in linear polymorphic type theory is that $\forall\alpha.\,(T\alpha \multimap \alpha) \to \alpha$ denotes an initial T-algebra, for covariant functors T on a "linear" category where morphisms correspond to terms of $\multimap$ type. It is not immediately obvious that this implies our result because our translations of first-order Idealized Algol types have quite a different form. Further, in contrast to intuitionistic type theory the mixing of intuitionistic and linear facilities blocks an evident reduction of these and other low order types to the form $\forall\alpha.\,(T\alpha \multimap \alpha) \to \alpha$. Finally, note that the map κ in the proof, which essentially gives (weak) initiality, was defined using details about the structure of solutions of domain equations in the category of cpo's and strict continuous functions.

10.3 A Full Abstraction Result

We know from the Resumption Theorem that the first-order types have algebraic cpo structure. We show that the finite elements are definable in a suitable sense. (This is related, but not identical, to the argument in [O'Hearn and Reddy 1999], Proposition 5.2.)

Let $v_{N_\bot} \in \mathbf{var}^\star N_\bot$ be the evident variable that directly updates and reads from $N_\bot$. It can be obtained from the standard variable $v[I] \in \mathbf{var}^\star(I \otimes N_\bot)$ using the canonical isomorphism $I \otimes N_\bot \leftrightarrow N_\bot$.

LEMMA 24. **(Definability Lemma)** *If $\varphi_1, ..., \varphi_n, \varphi$ are primitive Algol types then every finite element d of $(\varphi_1 \times \cdots \varphi_n \to \varphi)^\star N_\bot$ is definable by an Idealized Algol term*

$$x : \mathbf{var} \vdash M_d : \varphi_1 \times \cdots \varphi_n \to \varphi,$$

in the sense that

$$(M_d^\star N_\bot)\, v_{N_\bot} \;=\; d$$

in the strict parametricity model.

In this result $(M_d^\star N_\bot)$ is a map of type $\mathbf{var}^\star N_\bot \multimap (\varphi_1 \times \cdots \varphi_n \to \varphi)^\star N_\bot$. The standard variable $v_{N_\bot}$ for updating $N_\bot$ is given as the only component in the environment, and is denoted by x in $(M_d^\star N_\bot)\, v_{N_\bot}$.

Proof: Again we concentrate on $\mathbf{exp} \times \mathbf{exp} \to \mathbf{exp}$. It is standard that the finite elements d in the resumption domain can be generated as follows.

$$d \ ::= \ (s_1 \searrow [s'_1, e_1]) \sqcup \cdots \sqcup (s_k \searrow [s'_k, e_k]) \quad \text{(where } s_i, s'_i \in N,\ s_1, ..., s_k \text{ are all distinct)}$$

$$e \ ::= \ n \mid i : (n_1 \searrow d_1) \sqcup \cdots \sqcup (n_m \searrow d_m) \quad \text{(where all } n_j \in N \text{ are distinct, and } i = 1 \text{ or } 2)$$

We define terms N_d, F_e by induction on d and e, where

$$x : \mathbf{var}, c_1 : \mathbf{exp}, c_2 : \mathbf{exp} \vdash N_d : \mathbf{exp}$$
$$x : \mathbf{var}, c_1 : \mathbf{exp}, c_2 : \mathbf{exp} \vdash F_e : \mathbf{exp}.$$

N_d is defined by

$$\begin{aligned} & N_{(s_1 \searrow [s'_1, e_1]) \sqcup \cdots \sqcup (s_k \searrow [s'_k, e_k])} \\ = \ & \mathbf{if}\, x = s_1 \,\mathbf{then}\, x := s'_1 ; F_{e_1} \\ & \quad \vdots \\ & \mathbf{else\ if}\, x = s_k \,\mathbf{then}\, x := s'_k ; F_{e_k} \\ & \mathbf{else}\ \Omega. \end{aligned}$$

F_e is defined according to the component of e:

$$F_n \ = \ n$$

$$\begin{aligned} & F_{i:(n_1 \searrow d_1) \sqcup \cdots \sqcup (n_m \searrow d_m)} \\ = \ & \mathbf{new}_{\mathbf{exp}}\, y.\ y := c_i;\ \mathbf{if}\, y = n_1 \,\mathbf{then}\, N_{d_1} \\ & \qquad \vdots \\ & \qquad \mathbf{else\ if}\, y = n_m \,\mathbf{then}\, N_{d_m} \\ & \qquad \mathbf{else}\ \Omega. \end{aligned}$$

Then we set $M_d = \lambda\langle c_1, c_2\rangle.\, N_d$, and it is straightforward to verify that M_d defines d in the sense of the statement of the Lemma. ■

To formulate the full abstraction result, we will take convergence of closed terms of type **comm** as the observable. (We could also observe integers generated by terms of type **exp**, but this would lead to the same contextual equivalence relation.) To define this precisely, first note that if M is a closed term of type **comm** in Idealized Algol, the family of maps $M^\star(-) : I \multimap \mathbf{comm}^\star(-)$ is completely determined by the component $M^\star I : I \multimap \varphi^\star I$ at I. (This follows from the Logical Relations Lemma, using relations of the form $I \leftrightarrow S$ that fix a state in S.) The resulting function $M^\star I * : I \multimap I$ takes as an argument the state $* \in I$ and produces as a result either $*$ or $\perp$. We take as observable this final value $M^\star I * *$, and we look at approximation in all **comm**-typed contexts.

Theorem 25. **(Full Abstraction to 2nd order)** *Suppose* $\vdash M : \theta$ *and* $\vdash N : \theta$ *in Idealized Algol where* $\theta = (\varphi_1 \times \cdots \varphi_n \to \varphi) \to \varphi'$. *Then*

$$M^\star \sqsubseteq N^\star \quad \Longleftrightarrow \quad \forall C[\cdot].\, C[M]^\star I * * \sqsubseteq C[N]^\star I * *$$

(where $C[\cdot]$ *is understood to be a context such that* $C[M]$ *and* $C[N]$ *are closed terms of type* **comm***).*

Proof: The $\Longrightarrow$ direction is immediate from compositionality. For $\Longleftarrow$, suppose that $[\![M^\star\alpha]\!] \not\sqsubseteq [\![N^\star\alpha]\!]$. We require a distinguishing context where $C[M]^\star \not\sqsubseteq C[N]^\star$.

First suppose that φ' is **comm** or **exp**. By Lemmas 13 and 16, and continuity, there is a finite element d and $n \in N$ such that

$$(M^\star I)[N_\bot]d[*, n] \not\sqsubseteq (N^\star I)[N_\bot]d[*, n].$$

By the Definability Lemma (using also an isomorphism $N_\bot \leftrightarrow I \otimes N_\bot$) there is a term M_d that defines d in an environment where x denotes the standard variable $v[I]$ for local variables.

If φ' is **comm** then $(M^\star I)[N_\bot]d[*, n] = [*, m]$ for some m, and a distinguishing context is

$$C[\cdot] = \mathbf{new}_{\varphi'}\, x.\, x := n; ([\cdot]M_d); \mathbf{if}\, x = m\, \mathbf{then\, skip\, else\, diverge}.$$

If φ' is **exp** then $(M^\star I)[N_\bot]d[*, n] = [*, m, m']$ for some m and m', and a distinguishing context is

$$\begin{aligned} C[\cdot] = \mathbf{new}_{\varphi'}\, x.\, x := n; \mathbf{if}\, ([\cdot]M_d) = m'\, \mathbf{then} \\ \mathbf{if}\, x = m\, \mathbf{then\, skip\, else\, diverge} \\ \mathbf{else\, diverge}. \end{aligned}$$

The proof when φ' is **acc** is similar, if we observe that assignment to an acceptor is needed when using continuity and definability to generate a distinguishing context. ■

We have formulated the full abstraction result for second-order types of a specific form, but it is not difficult (observing remarks on naturality in Section 9.2) to extend the argument to all second-order types. We do not know if the result extends to higher types.

11. RELATED WORK

There are two ways to read the contribution of this paper. One has to do with semantic models of imperative languages, and the other with translating from an imperative language to a (linear) purely functional language. From the first point of view the semantics is being used to analyze the imperative source languages, while from the second the translation may be regarded just as much as telling us something about the functional target language. In discussing related work we consider these points of view in turn.

This paper builds on prior work on functor category semantics [Reynolds 1981a; Oles 1982; Oles 1997; O'Hearn and Tennent 1995; Sieber 1996], the main technical improvement being the elimination of snapback operators. The semantics may in fact be regarded as a refinement of the parametric-functor model of O'Hearn and Tennent [1995], obtained by moving from standard to linear polymorphism. Another difference with these works is our use of a polymorphic target language in place of a functor category (an aspect left implicit in the description of the parametric-functor model). This makes the store shape typing information implicit in a functor category more explicit, and statically checkable; we do not yet fully appreciate the significance of this point.

Pitts [1996] has carried out a study of contextual equivalence in Algol-like languages using operational techniques. His work is a good example of useful interplay between denotational and operational semantics. He proves a "possible worlds" version of the context lemma [Milner 1977], the formulation of which mimicks the structure of a functor category; it characterizes equivalence of functions in terms of applications to arguments at accessible worlds or store shapes. He uses an operational formulation of the relational principles considered here (and in [O'Hearn and Tennent 1995]) to prove his main result. A key point in Pitts's work is that he separates the use of logical relations as reasoning principles from their use in constructing a model. This does not lead to representation results, and it de-emphasizes the connection with linear polymorphic typing, but it does provide a pleasantly simple mathematical expression of the relational principles, and interesting technical results.

A completely different view of imperative computation is given by *implicit-state* models, which interpret imperative programs using histories of events. The basic conception of this approach is similar to ideas in work on processes (e.g. [Milner 1989]), but novel models of Algol-like languages have now been defined using denotational tools. The first of these is due to Reddy [1996], who defined a model for syntactic control of interference using coherence spaces; more recently, Abramsky and McCusker [1997] gave a game model of Idealized Algol.

Reddy's semantics is similar in spirit to domain-theoretic models of functional languages, and is based on a concrete description of domain-theoretic structure associated with SCI types. It accounts especially well for independence between arguments to functions. The model of Abramsky and McCusker is an extension the game semantics of PCF developed by Hyland and Ong [1994], obtained by dropping the "innocence" condition; this results in a clear distinction between functional and imperative behaviours in the model. They show that all finite elements in their model are definable by terms in essentially the same version of Idealized Algol considered here; this leads, after quotienting, to a fully abstract model. Previously, full abstraction had only been obtained up to second-order types [O'Hearn and Reddy 1999], which is where our analysis here (which was carried out around the same time) ends as well.

The difference between our semantics and implicit-state models is striking. For us the primitive concepts are sets of states, and the linearly polymorphic way that states are used. In the implicit models the primitive concepts are events or observations, and interaction of a program with its environment. The conceptual distance between the two approaches is thus very great. It is not obvious how to formulate a precise linkage between them, but to do so would be valuable.

We now turn to related work on linear functional programming. The informal connection between imperative-like state transformations and linear functions was emphasized from the beginning in linear logic. It formed part of the motivation for a number of linear functional languages [Mackie 1994; Lafont 1988; Holmström 1988; Chirimar et al. 1994], where linearity could be used to restrict the number of pointers to functional values and, in some cases, guarantee the safety of destructive array update.

This connection was illustrated particularly clearly by Wadler [1990], by translating an imperative langauge, without procedures, into a linear functional language. We argued in the Introduction that a language without procedures does not itself provide a stringent test for the imperative expressiveness of a linear language, but we do want to emphasize our debt to the work of Wadler, and to other early works on linear logic, for making the connection between imperative-like state transformations and linear functions. To these works we would add the point that moving from simple to polymorphic linear types allows for a treatment of procedures and local state, and as a result it becomes possible to cover a much wider range of imperative programming. We would also add that polymorphism can be used to capture that a non-linear state be *used* linearly.

Prior to the appearance of linear logic, Schmidt [1985] had already studied the relationship between syntactic restrictions on λ-terms and the imperative nature of state transformations. His aim was to detect, in a standard denotational definition, when a parameter was "single threaded." The idea was that this would enable a compiler generator to detect when the parameter could be implemented in a store-like manner, by overwriting. His aim, and form of analysis, was thus different from ours; in particular, he works with simple types, where polymorphism plays a central role here. Some of his basic ideas are reflected in our translations, but one that is not is *passivity* [Reynolds 1978], where multiple copies of a store parameter are allowed in contexts that ensure that they are used in a read-only fashion.

12. DISCUSSION

In the course of the paper we have presented syntactic translations from two Algol-like languages into a polymorphic, linear lambda-calculus, given a semantic model of the linear language, and used it characterize the cpo structure of a number of low-order types. We hope particularly to have convinced the reader that the translations and semantic model provide simple and effective principles that can be utilized in a variety of circumstances. This is highlighted by our work in Section 7, and also by Pitts's work referenced above.

The semantic analysis provided by the strict parametricity model is, however, incomplete in come respects. We were in fact surprised to find that we could push the model as far as we could. To clarify this incompleteness we discuss a number of unanswered questions.

The first question arises from the work in Section 9.2: Does parametricity imply naturality with respect to expansions at all types for the Idealized Algol translation? If the answer is no then the translation of Idealized Algol, as it stands, would not verify the isomorphisms of cartesian closed categories at higher types. We indicated in Section 9.2 that this is not a fundamental problem, as we know a number of ways to overcome it. Furthermore, in Section 8 we verified an adequacy result to the effect that the translation gets convergence at primitive types right, so the translation can be used to *soundly* reason about Idealized Algol programs. But the question is irksome, because one might expect that parametricity *should* imply naturality [Plotkin and Abadi 1993].

Our analysis of equivalence only went as far as second-order types. We have not found a counterexample to full abstraction at higher types in Idealized Algol, but we did find an explicit limitation in the model for SCI in Section 7.3. One could consider a more focused study of contextual equivalence, either by using different models or by a syntactic analysis of the translations. Independently of any specific model we could ask if the translations are fully abstract, i.e., whether they preserve and reflect appropriate notions of contextual equivalence for source and target languages.

It may have seemed odd that we did use the strict function model, since it is actually a model of *relevant* lambda-calculus (it models Contraction). We observed that snapback requires both Contraction and Weakening, so that eliminating one has the effect of banishing snapback operators. But there may be a more comprehensive explanation than this. For primitive types in the translations there is always a unique occurrence of any type variable to the left or right of $\multimap$: Intuitively, if you perform a Contraction, copying a value of one of these types, it must be followed by a Weakening to preserve the uniqueness property (and conversely if a Weakening is performed first). This leads to the question (which we leave imprecisely stated) of whether the translations are the same if we take relevant, linear, or affine lambda-calculus as the target language. When one moves outside the ranges of our translations, beyond Algol-like types, we expect that linearity would play a more crucial role; we would benefit from a more precise understanding of these points.

With this discussion of technical properties, it is well to remember that the original motivation for semantics based on store shapes was much more basic: It was to build a model that made the stack discipline obvious [Reynolds 1981a]. In particular, it is clear from the semantics of types that the shape of the store in the final state obtained by evaluating a command must be the same as the shape in the initial state. While the translation for Idealized Algol follows previous functor-category models closely, in SCI store shapes play a further role, making clear that different identifiers work with different pieces of the state. So, more generally, we may say that semantics based on store shapes aims to communicate a spatial intuition: Programs working with different store shapes act on separate parts of the store, and consequently don't interfere.

Our work here gives an implementation of store shape semantics by translation into a linear polymorphic functional language. We have, for the most part, concentrated on what the resulting semantics says about the source languages, but if we switch focus to the target language then the translations can just as well be regarded as telling us about it. In particular, since we already understand Algol as an imperative language, the translations (and representation results) give us a precise imperative way of reading certain linear types. This does not, however, help us to understand types that lie beyond the ranges of our translations, because there we do not have a prior understanding of an imperative language to fall back on.

A final note on the source languages, and limitations. While we concentrate on call-by-name, the adaptation of our semantic methods to a call-by-value setting does not appear to raise insuperable difficulties. Much more difficult is the inclusion of storable procedures or commands. Idealized Algol and SCI allow only stateless entities, such as integers, to be stored, and we do not know how well our methods might extend to deal with stateful values in the store.

We wonder whether this apparent limitation could be turned into a feature. That is, both Idealized Algol and Basic SCI are higher-order imperative languages that obey a stack discipline for variable declarations. While the stack discipline is made evident by the translations of types, there are other polymorphic types, lying outside the ranges of the translations, that display the same stack-like character. This raises the question of whether the stack-like sublanguage could be demarcated, perhaps leading to an imperative language that is more general and flexible than

Algol, while maintaining efficient storage utilization.

ACKNOWLEDGEMENTS. This work first came to light at a workshop on Syntactic Control of Interference and Linear Logic in Glasgow in August of 1995, where the authors presented independently discovered translations into a linear polymorphic target language. The translations were essentially the same, except that O'Hearn used Idealized Algol as the source language where Reynolds used SCI. We are grateful to the workshop participants, especially the host Phil Wadler, for contributing to a remarkably congenial research atmosphere. Special thanks go to Uday Reddy for discussions on how the strict function model could eliminate snapback, to Gordon Plotkin for emphasizing the suitability of linear typing in the presence of recursion, and to Bob Tennent for numerous discussions, especially in relation to the material in Section 8. Thanks also to Jeff Polakow and Chris Stone for comments on the presentation in the paper, and the referees for helpful suggestions. O'Hearn was supported by NSF grant CCR-92110829 in the initial stages of this work. Reynolds was supported by NSF grant CCR-9409997. Support during extended visits was given to both authors from the Isaac Newton Programme on Semantics of Computation, held in Cambridge in 1995; to O'Hearn from the Electrotechnical Laboratory, Tsukuba, Japan; and to Reynolds from Bell Laboratories, Lucent Technologies, New Jersey.

REFERENCES

ABRAMSKY, S. 1993. Computational interpretations of linear logic. *Theoretical Computer Science 111*, 1–2 (April 12), 3–57.

ABRAMSKY, S., JAGADEESAN, R., AND MALACARIA, P. Full abstraction for PCF. To appear in *Information and Computation.*

ABRAMSKY, S. AND JUNG, A. 1994. Domain theory. In S. ABRAMSKY, D. M. GABBAY, AND T. S. E. MAIBAUM Eds., *Handbook of Logic in Computer Science*, Volume 3, pp. 1–168. Clarendon Press.

ABRAMSKY, S. AND MCCUSKER, G. 1997. Linearity, sharing and state. In P. W. O'HEARN AND R. D. TENNENT Eds., *Algol-like Languages*, Volume 2, pp. 297–330. Boston: Birkhauser. Extended abstract to appear in Proceedings of Linear Logic 1996, *Electronic Notes in Theoretical Computer Science*, volume 3, Elsevier Science, 1996.

BARBER, A. AND PLOTKIN, G. 1997. Dual intuitionistic linear logic. Submitted.

BENTON, N., BIERMAN, G., DE PAIVA, V., AND HYLAND, M. 1993. A term calculus for intuitionistic linear logic. In M. BEZEN AND J. F. GROOTE Eds., *Typed Lambda Calculi and Applications*, Volume 664 of *Lecture Notes in Computer Science* (Utrecht, The Netherlands, March 1993), pp. 75–90. Springer-Verlag, Berlin.

BERRY, G. 1978. Stable models of typed lambda-calculi. *Automata, Languages and Programming, Fifth Colloquium, Udine*, 72–89. Lecture Notes in Computer Science 62.

BERRY, G. AND CURIEN, P.-L. 1982. Sequential algorithms on concrete data structures. *Theoretical Computer Science 20*, 265–321.

BEZEN, M. AND GROOTE, J. F. Eds. 1993. *Typed Lambda Calculi and Applications*, Volume 664 of *Lecture Notes in Computer Science* (Utrecht, The Netherlands, March 1993). Springer-Verlag, Berlin.

BROOKES, S., MAIN, M., MELTON, A., AND MISLOVE, M. Eds. 1995. *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.

CHIRIMAR, J., GUNTER, C. A., AND RIECKE, J. 1994. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming 2*, 6, 194–244.

DAY, B. J. 1970. On closed categories of functors. In S. MAC LANE Ed., *Reports of the Midwest Category Seminar*, Volume 137 of *Lecture Notes in Mathematics*, pp. 1–38. Springer-Verlag, Berlin-New York.

GIRARD, J.-Y. 1972. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Ph. D. thesis, Université Paris VII.

GIRARD, J.-Y. 1987. Linear logic. *Theoretical Computer Science 46*, 1–102.

HOLMSTRÖM, S. 1988. Linear functional programming. *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, Chalmers University.

HYLAND, J. M. E. AND ONG, C.-H. L. 1994. On full abstraction for PCF: I, II and III. Submitted for publication.

LAFONT, Y. 1988. The linear abstract machine. *Theoretical Computer Science 59*, 157–180.

LAMBEK, J. 1989. Multicategories revisited. In J. W. GRAY AND A. SCEDROV Eds., *Categories in Computer Science and Logic*, Volume 92 of *Contemporary Mathematics*, pp. 217–240. American Mathematical Society.

LENT, A. F. 1993. The category of functors from state shapes to bottomless CPOs is adequate for block structure. In *ACM SIGPLAN Workshop on State in Programming Languages* (Copenhagen, Denmark, June 12, 1993), pp. 101–119. Technical report RR-968, Department of Computer Science, Yale University.

MACKIE, I. 1994. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming 4*, 4 (Oct.), 395–433.

MILNER, R. 1977. Fully abstract models of typed λ-calculi. *Theoretical Computer Science 4*, 1–22.

MILNER, R. 1989. *Communication and Concurrency*. Prentice Hall, New York.

O'HEARN, P. W., POWER, A. J., TAKEYAMA, M., AND TENNENT, R. D. 1999. Syntactic control of interference revisited. *Theoretical Computer Science ??*, ???–??? To appear. Preliminary version in [Brookes et al. 1995] and in [O'Hearn and Tennent 1997b].

O'HEARN, P. W. AND REDDY, U. S. 1999. Objects, interference and the Yoneda embedding. *Theoretical Computer Science ??*, ???–??? To appear. Preliminary version in [Brookes et al. 1995].

O'HEARN, P. W. AND RIECKE, J. G. 1995. Kripke logical relations and PCF. *Information and Computation 120(1)*, 107–116.

O'HEARN, P. W. AND TENNENT, R. D. 1995. Parametricity and local variables. *J. ACM 42(3)*, 658–709. Also in [O'Hearn and Tennent 1997b], pages 109–164.

O'HEARN, P. W. AND TENNENT, R. D. Eds. 1997a. *Algol-like Languages*, Volume 1. Birkhauser, Boston.

O'HEARN, P. W. AND TENNENT, R. D. Eds. 1997b. *Algol-like Languages*, Volume 2. Birkhauser, Boston.

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

OLES, F. J. 1987. Lambda calculi with implicit type conversions. Technical Report RC 13245, IBM Research, T. J. Watson Research Center, Yorktown Heights, N.Y.

OLES, F. J. 1997. Functor categories and store shapes. In P. W. O'HEARN AND R. D. TENNENT Eds., *Algol-like Languages*, Volume 2, pp. 3–12. Boston: Birkhauser.

PITTS, A. 1996. Reasoning about local variables with operationally-based logical relations. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science* (New Jersey, USA, 1996), pp. 152–163. IEEE Computer Society Press, Los Alamitos, California. Also in [O'Hearn and Tennent 1997b], pages 165–186.

PLOTKIN, G. 1983. Domains. 1992 TeXed edition of course notes prepared by Yugo Kashiwagi and Hidetaka Kondoh from notes by Tatsuya Hagino.

PLOTKIN, G. AND ABADI, M. 1993. A logic for parametric polymorphism. In M. BEZEN AND J. F. GROOTE Eds., *Typed Lambda Calculi and Applications*, Volume 664 of *Lecture Notes in Computer Science* (Utrecht, The Netherlands, March 1993), pp. 361–375. Springer-Verlag, Berlin.

PLOTKIN, G. D. 1977. LCF considered as a programming language. *Theoretical Computer Science 5*, 223–255.

PLOTKIN, G. D. 1980. Lambda-definability in the full type hierarchy. In J. P. SELDIN AND J. R. HINDLEY Eds., *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pp. 363–373. Academic Press.

PLOTKIN, G. D. 1993. Type theory and recursion (extended abstract). In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science* (Montreal, Canada, 19–23 June 1993), pp. 374. IEEE Computer Society Press.

REDDY, U. S. 1996. Global states considered unnecessary: introduction to object-based semantics. *Lisp and Symbolic Computation*. Special issue on State in Programming Languages. Also in [O'Hearn and Tennent 1997b], pages 227–296.

REYNOLDS, J. C. 1974. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, Volume 19 of *Lecture Notes in Computer Science* (Berlin, 1974), pp. 408–425. Springer-Verlag.

REYNOLDS, J. C. 1978. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (Tucson, Arizona, January 1978), pp. 39–46. ACM, New York. Also in [O'Hearn and Tennent 1997a], pages 273–286.

REYNOLDS, J. C. 1981a. *The Craft of Programming*. Prentice-Hall International, London.

REYNOLDS, J. C. 1981b. The essence of Algol. In J. W. DE BAKKER AND J. C. VAN VLIET Eds., *Algorithmic Languages* (Amsterdam, October 1981), pp. 345–372. North-Holland, Amsterdam. Also in [O'Hearn and Tennent 1997a], pages 67-88.

REYNOLDS, J. C. 1983. Types, abstraction and parametric polymorphism. In R. E. A. MASON Ed., *Information Processing 83*, pp. 513–523. Amsterdam: North Holland.

RIECKE, J. G. AND SANDHOLM, A. 1997. A relational account of call-by-value sequentiality. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science* (Warsaw, Poland, 29 June–2 July 1997), pp. 258–267. IEEE Computer Society Press.

SCHMIDT, D. A. 1985. Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems 7*, 3 (April), 299–310.

SCOTT, D. S. 1972. Mathematical concepts in programming language semantics. In *Proc. 1972 Spring Joint Computer Conference* (1972), pp. 225–34. AFIPS Press, Montvale, N.J.

SCOTT, D. S. AND STRACHEY, C. 1971. Toward a mathematical semantics for computer languages. In J. FOX Ed., *Proceedings of the Symposium on Computers and Automata*, Volume 21 of *Microwave Research Institute Symposia Series* (1971), pp. 19–46. Polytechnic Institute of Brooklyn Press, New York. Also Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, Oxford.

SIEBER, K. 1996. Full abstraction for the second order subset of an ALGOL-like language. *Theoretical Computer Science 168*, 1 (10 Nov.), 155–212.

STRACHEY, C. 1972. The varieties of programming language. In *Proceedings of the International Computing Symposium* (Venice, April 1972), pp. 222–233. Cini Foundation, Venice. Also in [O'Hearn and Tennent 1997a], pages 51–64.

TENNENT, R. D. 1991. *Semantics of Programming Languages*. International Series in Computer Science. Prentice-Hall International.

WADLER, P. 1990. Linear types can change the world! In M. BROY AND C. JONES Eds., *IFIP TC-2 Working Conference on Programming Concepts and Methods* (Sea of Galilee, Israel, April 1990), pp. 347–359. North Holland, Amsterdam.

WADLER, P. 1991. Is there a use for linear logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1991). Proceedings of the 1991 Conference.