

A fully abstract relational model of Syntactic Control of Interference

Guy McCusker

School of Cognitive and Computing Sciences
University of Sussex
Falmer
Brighton BN1 9QH
United Kingdom
guym@cogs.susx.ac.uk

Abstract. Using familiar constructions on the category of monoids, a fully abstract model of Basic SCI is constructed. Basic SCI is a version of Reynolds’s higher-order imperative programming language Idealized Algol, restricted by means of a linear type system so that distinct identifiers are never aliases. The model given here is concretely the same as Reddy’s *object spaces* model, so this work also shows that Reddy’s model is fully abstract, which was not previously known.

Keywords: semantics, Algol-like languages, interference control, full abstraction, object spaces, monoids.

1 Introduction

For over 20 years there has been considerable interest among the semantics community in the study of *Algol-like languages*. Reynolds’s seminal paper [11] pointed out that Algol 60 embodies an elegant and powerful combination of higher-order procedures and imperative programming, and began a strand of research which has generated a great deal of deep and innovative work. Much of this work was recently republished in a two-volume collection [7].

One theme of this research is that of *interference control*, which was also initiated by Reynolds [10]. When reasoning about higher-order programs, one often encounters the need to establish the non-interference of a pair of program phrases: if a (side-effecting) function is guaranteed not to alter variables which are used by its arguments, and vice versa, then more reasoning principles become available. Unfortunately, the common phenomenon of *aliasing* makes it difficult to detect whether two program phrases may interfere with one another: mere disjointness of the sets of variables they contain is not enough. However, Reynolds showed that if one restricts *all* procedure calls so that a procedure and its argument have no variables in common, aliasing is eliminated, and it follows that no procedure call suffers from interference with its argument. In modern terms, this restriction is the imposition of an affine type system on the λ -calculus part of Idealized Algol. The resulting programming language, which O’Hearn terms *Basic SCI*, can be extended in various ways to restore more

programming power [6, 5], but is itself of interest as a minimal alias-free higher-order imperative programming language. (Other approaches to the control of interference and aliasing have also been considered, including *islands* [3] and *regions* [12].)

This paper is a semantic study of the core language, Basic SCI. Using simple constructions in the category of monoids, we build a model of this language. This model turns out to be the same as an existing model due to Reddy [9], which was presented rather differently using coherence spaces. This *object spaces* model was an important precursor of the games-based models of imperative programming languages [2, 1] and the first model of a higher-order imperative language based on traces of observations rather than on state-transformers.

The first, rather minor, contribution of this paper is in showing that Reddy’s model can be reconstructed so simply. We believe that our presentation is more direct and somewhat easier to work with, although it is perhaps less informative since it lacks some of the built-in structure of coherence which Reddy exploits.

The main result of this paper is that our model, and hence Reddy’s, is not merely sound but *fully abstract*: it captures precisely the notion of behavioural equivalence in the language. Reddy’s model was therefore the first example of a fully abstract semantics for a higher-order imperative language, though this was not known at the time; and it remains the only fully abstract model for an interference-controlled language that we are aware of. Its full abstraction is perhaps remarkable since it contains a great many undefinable elements. However, the definable elements do suffice to distinguish any two different elements of the model, and it is this which leads to full abstraction.

It is hoped that this work can be extended to encompass more powerful interference-controlled languages. The addition of *passive types*, whose elements are side-effect free and thus interfere with nothing, is a prime concern. Reddy showed how to extend his model in this direction. In doing so, full abstraction is lost, but of course, the full abstraction of the model of the core language was not known until now. The present work was in fact inspired by an ongoing attempt to model SCI using game-semantic techniques, conducted by the present author in conjunction with Wall [13], as part of a more general semantic study of interference control. We hope that this research will yield a fully abstract model of an extended language; but this remains to be seen.

2 Basic SCI

Basic SCI is the result of imposing an affine type system on Reynolds’s Idealized Algol [11]. The types are given by the grammar

$$A ::= \text{comm} \mid \text{exp} \mid \text{var} \mid A \multimap A.$$

Here `comm` is the type of commands, `exp` is the type of natural-number-valued expressions, and `var` is the type of variables which may store natural numbers.

The terms of the language are as follows.

$$\begin{aligned}
M ::= & x \mid \lambda x^A.M \mid MM \\
& \mid \text{skip} \mid M; M \mid \text{while } M \text{ do } M \\
& \mid M := M \mid !M \\
& \mid \text{succ } M \mid \text{pred } M \mid \text{ifzero } M M M \\
& \mid \text{new } x \text{ in } M
\end{aligned}$$

Here x ranges over an infinite collection of variables, and A ranges over types. We will often omit the type tag on abstractions when it will cause no confusion.

The type system is given by a collection of judgements of the form

$$x_1 : A_1, \dots, x_n : A_n \vdash M : B$$

where the x_i are distinct variables, M is a term, and the A_i and B are types. We use Γ and Δ to range over contexts, that is, lists $x_1 : A_1, \dots, x_n : A_n$ of variable-type pairs with all variables distinct. In the inductive definition which follows, it is assumed that all contexts are well-formed.

$$\begin{array}{c}
\frac{}{\Gamma, x : A \vdash x : A} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \multimap B} \\
\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}
\end{array}$$

Note that in the last rule above, the assumption that Γ, Δ is a well-formed context implies that Γ and Δ have no variables in common.

$$\begin{array}{c}
\frac{}{\Gamma \vdash \text{skip} : \text{comm}} \quad \frac{\Gamma \vdash M : \text{comm} \quad \Gamma \vdash N : A}{\Gamma \vdash M; N : A} \quad A \in \{\text{comm}, \text{exp}, \text{var}\} \\
\frac{\Gamma \vdash M : \text{exp} \quad \Gamma \vdash N : \text{comm}}{\Gamma \vdash \text{while } M \text{ do } N : \text{comm}} \\
\frac{\Gamma \vdash M : \text{var} \quad \Gamma \vdash N : \text{exp}}{\Gamma \vdash M := N : \text{comm}} \quad \frac{\Gamma \vdash M : \text{var}}{\Gamma \vdash !M : \text{exp}} \\
\frac{\Gamma \vdash M : \text{exp}}{\Gamma \vdash \text{succ } M : \text{exp}} \quad \frac{\Gamma \vdash M : \text{exp}}{\Gamma \vdash \text{pred } M : \text{exp}} \\
\frac{\Gamma \vdash M : \text{exp} \quad \Gamma \vdash N : A \quad \Gamma \vdash P : A}{\Gamma \vdash \text{ifzero } M N P : A} \quad A \in \{\text{comm}, \text{exp}, \text{var}\} \\
\frac{\Gamma, x : \text{var} \vdash M : \text{comm}}{\Gamma \vdash \text{new } x \text{ in } M : \text{comm}}
\end{array}$$

The operational semantics is given in terms of *stores* (also known as states). Given a context $\Gamma = x_1 : \mathbf{var}, x_2 : \mathbf{var}, \dots, x_n : \mathbf{var}$, a Γ -store σ is a function from the set $\{x_1, \dots, x_n\}$ to natural numbers. We write $\sigma[x \mapsto n]$ to mean the store which is identical to σ but maps x to n ; this may be used to extend a Γ -store to a Γ, x -store, or merely to update x when x appears in Γ .

We give the operational semantics by means of a type-indexed family of relations. For each base type B , we define a relation of the form

$$\Gamma \vdash \sigma, M \Downarrow_B \sigma', V$$

where $\Gamma \vdash M : B$ and $\Gamma \vdash V : B$ are well-typed terms, Γ contains only \mathbf{var} -typed variables, and σ and σ' are Γ -stores. The term V must be a *value*, that is, either \mathbf{skip} , n or some $x \in \Gamma$.

For each function type $A \multimap B$, we define a relation of the form

$$\Gamma \vdash M \Downarrow_{A \multimap B} V$$

where again Γ contains only \mathbf{var} -typed variables and M and V are well-typed terms of the appropriate type. Again V must be a value, that is, a term of the form $\lambda x.M'$. Note that there is no mention of store in the operational semantics of terms of higher type; this reflects the fact that terms of higher type do not affect and are not affected by the contents of the store until they are applied to arguments.

These relations are defined inductively. We just give a selection of the rules.

$$\frac{}{\Gamma \vdash \sigma, \mathbf{skip} \Downarrow_{\mathbf{comm}} \sigma, \mathbf{skip}}$$

$$\frac{\Gamma \vdash \sigma, M \Downarrow_{\mathbf{comm}} \sigma', \mathbf{skip} \quad \Gamma \vdash \sigma', N \Downarrow_B \sigma'', N'}{\Gamma \vdash \sigma, M; N \Downarrow_B \sigma'', N'}$$

$$\frac{\Gamma \vdash \sigma, N \Downarrow_{\mathbf{exp}} \sigma', n \quad \Gamma \vdash \sigma', M \Downarrow_{\mathbf{var}} \sigma'', x}{\Gamma \vdash \sigma, M := N \Downarrow_{\mathbf{exp}} \sigma''[x \mapsto n], \mathbf{skip}}$$

$$\frac{\Gamma \vdash \sigma, M \Downarrow_{\mathbf{var}} \sigma', x}{\Gamma \vdash \sigma, !M \Downarrow_{\mathbf{exp}} \sigma', n} \sigma'(x) = n$$

$$\frac{\Gamma \vdash \sigma, M \Downarrow_{\mathbf{exp}} \sigma', 0 \quad \Gamma \vdash \sigma', N \Downarrow_{\mathbf{comm}} \sigma'', \mathbf{skip} \quad \Gamma \vdash \sigma'', \mathbf{while} M \mathbf{do} N \Downarrow_{\mathbf{comm}} \sigma''', \mathbf{skip}}{\Gamma \vdash \sigma, \mathbf{while} M \mathbf{do} N \Downarrow_{\mathbf{comm}} \sigma''', \mathbf{skip}}$$

$$\frac{\Gamma \vdash \sigma, M \Downarrow_{\mathbf{exp}} \sigma', n + 1}{\Gamma \vdash \sigma, \mathbf{while} M \mathbf{do} N \Downarrow_{\mathbf{comm}} \sigma', \mathbf{skip}}$$

$$\frac{\Gamma, x : \mathbf{var} \vdash \sigma[x \mapsto 0], M \Downarrow_{\mathbf{comm}} \sigma'[x \mapsto n], \mathbf{skip}}{\Gamma \vdash \sigma, \mathbf{new} x \mathbf{in} M \Downarrow_{\mathbf{comm}} \sigma', \mathbf{skip}}$$

$$\frac{}{\Gamma \vdash \lambda x.M \Downarrow_{A \multimap B} \lambda x.M}$$

$$\frac{\Gamma \vdash M \Downarrow_{A \rightarrow B} \lambda x. M' \quad \Gamma \vdash M'[N/x] \Downarrow_B V}{\Gamma \vdash MN \Downarrow_B V} \text{ } B \text{ a function type}$$

$$\frac{\Gamma \vdash M \Downarrow_{A \rightarrow B} \lambda x. M' \quad \Gamma \vdash \sigma, M'[N/x] \Downarrow_B \sigma', V}{\Gamma \vdash \sigma, MN \Downarrow_B \sigma', V} \text{ } B \text{ a base type}$$

Contextual equivalence We can define the notion of contextual equivalence in the usual way: given terms $\Gamma \vdash M, N : A$, we say that M and N are *contextually* (or *observationally*) *equivalent*, $M \cong N$, iff for all term-contexts $C[-]$ such that $\vdash C[M] : \mathbf{comm}$ and $\vdash C[N] : \mathbf{comm}$, $C[M] \Downarrow \mathbf{skip} \iff C[N] \Downarrow \mathbf{skip}$. (We omit mention of the unique store over no variables). As usual a term-context is a term with one or more occurrences of a “hole” written $-$, and $C[M]$ is the term resulting from replacing each occurrence of $-$ by M . We will often abbreviate the assertion $C[M] \Downarrow \mathbf{skip}$ simply to $C[M] \Downarrow$.

3 A categorical model

In this section we define and explore the structure of the category which our model of Basic SCI will inhabit.

To build our model, we will be making use of the category **Mon** of monoids and homomorphisms, and exploiting the product, coproduct and powerset operations on monoids, and the notion of the free monoid over a set. For the sake of completeness, we review these constructions here.

First some notation. For a monoid A , we use e_A to denote the identity element, and write monoid multiplication as concatenation, or occasionally using the symbol \cdot_A . The underlying set of the monoid A is written as UA .

Free monoids Recall that for any set A , the *free monoid over A* is given by A^* , the monoid of strings over A , also known as the Kleene monoid over A .

The operation taking A to A^* is left-adjoint to the forgetful functor $U : \mathbf{Mon} \rightarrow \mathbf{Set}$.

Products The category **Mon** has finite products. The product of monoids A and B is a monoid with underlying set $UA \times UB$, the Cartesian product of sets. The monoid operation is defined by

$$\langle a, b \rangle \langle a', b' \rangle = \langle a \cdot_A a', b \cdot_B b' \rangle.$$

The identity element is $\langle e_A, e_B \rangle$. Projection and pairing maps in **Mon** are given by the corresponding maps on the underlying sets. The terminal object is the one-element monoid.

Coproducts The category **Mon** also has finite coproducts. These are slightly awkward to define in general, and since we will not be making use of the general construction, we omit it here.

The special case of the coproduct of two free monoids is easy to define. Since the operation of building a free monoid from a set is left adjoint to the forgetful functor U , it preserves colimits and in particular coproducts. For sets A and B , the coproduct monoid $A^* + B^*$ is therefore given by $(A + B)^*$, the monoid of strings over the disjoint union of A and B .

The initial object is the one-element monoid.

Powerset The familiar powerset construction on **Set** lifts to **Mon** and retains much of its structure. Given a monoid A , define the monoid $\wp A$ as follows. Its underlying set is the powerset of UA , that is, the set of subsets of UA . Monoid multiplication is defined by

$$ST = \{x \cdot_A y \mid x \in S, y \in T\}$$

and the identity is the singleton set $\{e_A\}$.

We will exploit the fact that powerset is a *commutative monad* on **Mon**. In particular, we will make use of the Kleisli category \mathbf{Mon}_\wp . This category can be defined concretely as follows. Its objects are monoids, and a map from A to B is a monoid homomorphism from A to $\wp B$. The identity on A is the singleton map which takes each $a \in A$ to $\{a\}$. Morphisms are composed as follows: given maps $f : A \rightarrow B$ and $g : B \rightarrow C$, the composite $f ; g : A \rightarrow C$ is defined by

$$(f ; g)(a) = \{c \mid \exists b \in f(a). c \in g(b)\}.$$

The fact that the powerset monad is commutative means that the product structure on **Mon** lifts to a monoidal structure on \mathbf{Mon}_\wp as follows. We define $A \otimes B$ to be the monoid $A \times B$. For the functorial action, we make use of the *double strength* map

$$\theta_{A,B} : \wp A \times \wp B \longrightarrow \wp(A \times B)$$

defined by

$$\theta_{A,B}(S, T) = \{\langle x, y \rangle \mid x \in S, y \in T\}.$$

This is a homomorphism of monoids. With this in place, given maps $f : A \rightarrow B$ and $g : C \rightarrow D$ in \mathbf{Mon}_\wp , we can define $f \otimes g : A \otimes C \rightarrow B \otimes D$ as the homomorphism $f \times g ; \theta_{B,D}$. See for example [4] for more details on this construction.

The category we will use to model Basic SCI is $(\mathbf{Mon}_\wp)^{\text{op}}$. This category can be seen as a category of “monoids and relations” of a certain kind, so we will call it **MonRel**. We will now briefly explore some of the structure that **MonRel** possesses.

Monoidal structure The monoidal structure on \mathbf{Mon}_\wp described above is directly inherited by **MonRel**. Furthermore, since the unit I of the monoidal structure is given by the one-element monoid, which is also an initial object in **Mon**, I is in fact a terminal object in **MonRel**, so the category has an *affine* structure.

Exponentials Let A and B be any monoids, and C^* be the free monoid over some set C . Consider the following sequence of natural isomorphisms and definitional equalities.

$$\begin{aligned}
& \mathbf{MonRel}(A \otimes B, C^*) \\
&= \mathbf{Mon}(C^*, \wp(A \times B)) \\
&\cong \mathbf{Set}(C, U\wp(A \times B)) \\
&\cong \mathbf{Rel}(C, UA \times UB) \\
&\cong \mathbf{Rel}(UB \times C, UA)
\end{aligned}$$

Similarly we can show that

$$\mathbf{Rel}(UB \times C, UA) \cong \mathbf{MonRel}(A, (UB \times C)^*).$$

The exponential $B \multimap C^*$ is therefore given by $(UB \times C)^*$. It is important to note that the free monoids are closed under this operation, so that we can form $A_1 \multimap (A_2 \multimap \dots (A_n \multimap C^*))$ for any A_1, \dots, A_n . That is to say, the free monoids form an *exponential ideal* in \mathbf{MonRel} .

Products The coproduct in \mathbf{Mon} is inherited by the Kleisli-category \mathbf{Mon}_\wp , and since \mathbf{MonRel} is the opposite of this category, \mathbf{MonRel} has finite products.

An alternative characterization We can also describe the category \mathbf{MonRel} concretely, as follows. Objects are monoids, and maps $A \rightarrow B$ are relations R between the (underlying sets of) A and B , with the following properties:

homomorphism $e_A R e_B$, and if $a_1 R b_1$ and $a_2 R b_2$, then $a_1 a_2 R b_1 b_2$

identity reflection if $a R e_B$ then $a = e_A$

decomposition if $a R b_1 b_2$ then there exist a_1 and $a_2 \in A$ such that $a_i R b_i$ for $i = 1, 2$ and $a = a_1 a_2$.

Identities and composition are as usual for relations. Note that the property of “identity reflection” is merely the nullary case of the property of “decomposition”.

It is routine to show that this definition yields a category isomorphic to $(\mathbf{Mon}_\wp)^{\text{op}}$. The action of the isomorphism is as follows. Given a map $A \rightarrow B$ in $(\mathbf{Mon}_\wp)^{\text{op}}$, that is to say, a homomorphism

$$f : B \longrightarrow \wp(A)$$

we can define a relation R_f between A and B as the set of pairs $\{(a, b) \mid a \in f(b)\}$.

The intuition behind our use of this category is that the objects we employ are monoids of “observations” that one makes of a program phrase. The monoid operation builds compound observations from simple ones. A map in the category tells us what input observations are required in order to produce a given output observation. The decomposition axiom above has something of a flavour of linearity or stability about it: it says that the only way to produce a compound observation is to produce the elements of that observation. This is made more explicit in Reddy’s presentation which is based on coherence spaces.

4 Modelling Basic SCI

The categorical structure of **MonRel** developed above gives us enough to model the affine λ -calculus over base types which are interpreted as free monoids. We now flesh this out to complete the interpretation of Basic SCI by giving appropriate objects to interpret the base types and maps to interpret the constants of the language.

The idea behind our model of Basic SCI is that a program denotes a set of sequences of observable actions. Thus the types of **MonRel** will be interpreted as objects of the form A^* for a set A , that is, as free monoids.

$$\begin{aligned} \llbracket \mathbf{comm} \rrbracket &= 1^* \\ \llbracket \mathbf{exp} \rrbracket &= \mathbb{N}^* \\ \llbracket \mathbf{var} \rrbracket &= (\mathbb{N} + \mathbb{N})^* \end{aligned}$$

Here 1 denotes the one-element set, whose single element we will denote by $*$, \mathbb{N} is the set of natural numbers, and $+$ denotes disjoint union of sets. The two copies of \mathbb{N} used to interpret **var** correspond to the actions of reading a value from a variable and writing a value to a variable, so we will denote the elements of $\mathbb{N} + \mathbb{N}$ as $\mathbf{read}(n)$ and $\mathbf{write}(n)$.

The only observation we can make of a command is that the command terminates, so **comm** is interpreted using sequences over a one-element set. The basic observation one can make of a term of type **exp** is its value, so expressions denote sequences of natural numbers. For variables, one can observe the value stored in a variable, so there is an observation $\mathbf{read}(n)$ for each natural number n , and one can also observe that assigning the number n to a variable terminates, hence the actions $\mathbf{write}(n)$.

The interpretation of types of Basic SCI as objects of **MonRel** is completed by setting $\llbracket A \multimap B \rrbracket = \llbracket A \rrbracket \multimap \llbracket B \rrbracket$. The fact that objects of the form A^* form an exponential ideal in **MonRel** guarantees that the required exponentials exist. Unpacking the definition of exponential, we see that a basic observation that can be made of the function type $A \multimap B$ consists of a pair (s, b) where b is an observation from B and s is a sequence of observations from A . We will use the list-notation $[a_1, \dots, a_n]$ to display such sequences.

Note that the semantics of a term of type $A \multimap B$ does not record the relative order of actions in A and B . For example, the interpretation of the type $\mathbf{comm} \multimap \mathbf{comm} \multimap \mathbf{comm}$ contains elements such as $([*], [*], ([*], *))$, which will belong to the denotation of the term

$$\lambda x^{\mathbf{comm}}. \lambda y^{\mathbf{comm}}. x; x; y$$

but also of

$$\lambda x^{\mathbf{comm}}. \lambda y^{\mathbf{comm}}. x; y; x.$$

This is only correct thanks to the non-interference property of the language: because x and y cannot interfere, the relative order of their execution is irrelevant.

A term $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash M : B$ will be interpreted as a map

$$\llbracket x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash M : B \rrbracket : \llbracket A_1 \rrbracket \otimes \llbracket A_2 \rrbracket \otimes \dots \otimes \llbracket A_n \rrbracket \rightarrow \llbracket B \rrbracket.$$

Using the first definition of **MonRel**, such a map can be seen as a function taking an observation b from B as argument, and returning a set of tuples (s_1, s_2, \dots, s_n) where each s_i is a sequence of observations from A_i . This can be thought of as stipulating the actions that the environment must be prepared to perform in order for the term to produce the action b . Note that the monoid $\llbracket B \rrbracket$ is always the free monoid over some set, so this map is uniquely determined by its action on singleton observations.

In order to define such maps concretely, we will write them as sets of tuples of the form (s_1, \dots, s_n, b) where b is a singleton observation. This notation accords with the concrete presentation of exponentials above.

We now define maps in **MonRel** to interpret the constants of Basic SCI. Recall that the product in **MonRel** of two free monoids A^* and B^* is given by $(A + B)^*$, where $+$ denotes disjoint union. We will use the notation **fst** and **snd** to tag the two components of this disjoint union; when a ternary product is needed, we use **thd** for the third tag.

$$\begin{aligned} \text{skip} : I &\rightarrow \llbracket \text{comm} \rrbracket = \{(e_I, *)\} \\ \text{seq}_A : \llbracket \text{comm} \rrbracket \times A^* &\rightarrow A^* = \{([\text{fst}(*), \text{snd}(a)], a) \mid a \in A\} \\ \text{read} : \llbracket \text{var} \rrbracket &\rightarrow \llbracket \mathbb{N} \rrbracket = \{([\text{read}(n)], n) \mid n \in \mathbb{N}\} \\ \text{write} : \llbracket \text{var} \rrbracket \times \llbracket \text{exp} \rrbracket &\rightarrow \llbracket \text{comm} \rrbracket = \{([\text{snd}(n), \text{fst}(\text{write}(n))], *) \mid n \in \mathbb{N}\} \\ \text{ifz} : \llbracket \text{exp} \rrbracket \times A^* \times A^* &\rightarrow A^* = \{([\text{fst}(0), \text{snd}(a)], a) \mid a \in A\} \\ &\quad \cup \{([\text{fst}(n), \text{thd}(a)], a) \mid n \neq 0, a \in A\} \\ \text{while} : \llbracket \text{exp} \rrbracket \times \llbracket \text{comm} \rrbracket &\rightarrow \llbracket \text{comm} \rrbracket = \{([\text{fst}(0), \text{snd}(*), \text{fst}(0), \text{snd}(*), \dots, \\ &\quad \dots, \text{fst}(0), \text{snd}(*), \text{fst}(n)], *) \mid n \neq 0\} \end{aligned}$$

Similar maps can be defined for the interpretation of the arithmetic constants.

The interpretation of the constructs of the basic imperative language can now be defined in the standard way. For example,

$$\llbracket M; N \rrbracket = \langle \llbracket M \rrbracket, \llbracket N \rrbracket \rangle; \text{seq}$$

For the λ -calculus part of the language, we exploit the affine monoidal structure and the exponentials of the language. Again, these definitions are standard.

For variables:

$$\llbracket \Gamma, x : A \vdash x : A \rrbracket = \text{proj} : \llbracket \Gamma \rrbracket \otimes \llbracket A \rrbracket \rightarrow \llbracket A \rrbracket.$$

This projection map can be defined concretely as the set

$$\{(e_\Gamma, a, a) \mid a \in A\}$$

where e_Γ is the identity of the monoid $\llbracket \Gamma \rrbracket$.

For abstraction:

$$\llbracket \Gamma \vdash \lambda x^A.M : A \multimap B \rrbracket = \Lambda \llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow (\llbracket A \rrbracket \multimap \llbracket B \rrbracket)$$

where Λ denotes the natural isomorphism coming from the exponential structure.

For application:

$$\llbracket \Gamma, \Delta \vdash MN : B \rrbracket = (\llbracket M \rrbracket \otimes \llbracket N \rrbracket); \text{ev} : \llbracket \Gamma \rrbracket \otimes \llbracket \Delta \rrbracket \rightarrow \llbracket B \rrbracket$$

where $\text{ev} : (A \multimap B) \otimes B \rightarrow B$ is the counit of the exponential adjunction.

Finally, we give the semantics of the variable-allocation construct. First note that an element $s \in \llbracket \text{var} \rrbracket$ consists of a sequence of $\text{read}(-)$ and $\text{write}(-)$ actions. We say that s is a *cell-trace* if the values carried by $\text{read}(-)$ actions correspond to the values previously carried by $\text{write}(-)$ actions in the obvious way: s is a cell-trace iff

- whenever $s = [\dots, \text{read}(n), \text{read}(m), \dots]$, $n = m$
- whenever $s = [\dots, \text{write}(n), \text{read}(m), \dots]$, $n = m$.

We can now define a map

$$\text{new} : (\llbracket \text{var} \rrbracket \multimap \llbracket \text{comm} \rrbracket) \rightarrow \llbracket \text{comm} \rrbracket = \{((s, *), *) \mid \text{write}(0) \cdot s \text{ is a cell-trace}\}$$

and then

$$\llbracket \Gamma \vdash \text{new } x \text{ in } M : \text{comm} \rrbracket = \llbracket \lambda x.M \rrbracket; \text{new}.$$

Ignoring all the structure in this semantics and considering the interpretation of a term as a set of tuples, our semantics is identical to that obtained by forgetting all structure in Reddy's semantics. We therefore have:

Lemma 1. *The semantics given above agrees with the object-space semantics of Reddy [9]: writing $\llbracket - \rrbracket_r$ for the Reddy semantics, we have that for any terms M and N of Basic SCI,*

$$\llbracket M \rrbracket_r = \llbracket N \rrbracket_r \iff \llbracket M \rrbracket = \llbracket N \rrbracket.$$

We show the soundness of our semantics by means of a standard sequence of lemmas.

Lemma 2. *For any closed term M of type comm , if $M \Downarrow$ then $\llbracket M \rrbracket = \llbracket \text{skip} \rrbracket$.*

Proof. A straightforward but lengthy induction over the structure of derivations in the operational semantics. Very similar proofs can be found in Reddy's work [9] and in the work on game semantics of Algol-like languages [2].

Lemma 3. *For any closed term $M : \text{comm}$, if $\llbracket M \rrbracket = \llbracket \text{skip} \rrbracket$ then $M \Downarrow$.*

Proof. A Tait-Girard-Plotkin style computability argument is employed [8]. Similar arguments can be found in the works by Reddy and the game-semantics literature cited above.

Theorem 1 (Equational Soundness). *If $\Gamma \vdash M, N : A$ are terms such that $\llbracket M \rrbracket = \llbracket N \rrbracket$, then M and N are contextually equivalent.*

Proof. Since the semantics is compositional, for any context $C[-]$, we have $\llbracket C[M] \rrbracket = \llbracket C[N] \rrbracket$. By Lemmas 2 and 3, $C[M] \Downarrow$ iff $\llbracket C[M] \rrbracket = \llbracket \text{skip} \rrbracket$ iff $\llbracket C[N] \rrbracket = \llbracket \text{skip} \rrbracket$ iff $C[N] \Downarrow$ as required.

5 Full abstraction

In this section we show the converse of our Equational Soundness theorem: Equational Completeness, which states that if two terms are contextually equivalent, then they have the same denotational semantics. In order to do so, we must study the definable elements of our model more closely, and eventually prove a partial definability result. Our proof will involve some programming in Basic SCI, and we will make use of some syntactic sugar to write down programs which we will not explicitly define. It is hoped that this causes no difficulties for the reader.

Let us first mention an interesting fact. If $C[-]$ is some context such that

$$C[\text{if } !x = 3 \text{ then skip else diverge}] \Downarrow,$$

then it is also the case that $C[x := 3] \Downarrow$. This inability of contexts to distinguish completely between reading and writing into variables is the main obstacle to overcome in our definability proof.

The following definition captures the relationship between sequences of observations which is at work in the above example.

Definition 1. *For any SCI type A , we define the positive and negative read-write orders \preceq^+ and \preceq^- between elements of $\llbracket A \rrbracket$ as follows. We give only the definitions for singleton elements; the definitions are extended to sequences by requiring that the elements of the sequences are related pointwise.*

– At type **comm**:

$$* \preceq^+ * \wedge * \preceq^- *$$

– At type **exp**:

$$n \preceq^+ m \iff n = m \iff n \preceq^- m$$

– At type **var**:

$$\begin{aligned} a \preceq^+ a' &\iff (a = a') \vee \exists n. a = \text{read}(n) \wedge a' = \text{write}(n) \\ a \preceq^- a' &\iff a = a'. \end{aligned}$$

– At type $A \multimap B$:

$$\begin{aligned} (s, b) \preceq^+ (s', b') &\iff s \preceq^- s' \wedge b \preceq^+ b' \\ (s, b) \preceq^- (s', b') &\iff s \preceq^+ s' \wedge b \preceq^- b' \end{aligned}$$

In general, $s \preceq^+ t$ iff t can be obtained from s by replacing some occurrences of $\text{read}(n)$ actions in positive occurrences of the type **var** by the corresponding $\text{write}(n)$ actions. The order \preceq^- is the same but operates on negatively occurring actions.

We also need a notion of state transition. Given an element $s \in \llbracket \text{var} \rrbracket$, we define the transitions $n \xrightarrow{s} n'$ where n and n' are natural numbers, as follows.

$$n \xrightarrow{\square} n \quad n \xrightarrow{[\text{read}(n)]} n \quad n \xrightarrow{[\text{write}(n')]} n'$$

$$\frac{n \xrightarrow{s} n' \quad n' \xrightarrow{s'} n''}{n \xrightarrow{ss'} n''}$$

We extend this to traces involving more than one `var` type as follows. Given a context $x_1 : \mathbf{var}, \dots, x_n : \mathbf{var}$, an element $s = (s_1, \dots, s_n) \in \llbracket \mathbf{var} \rrbracket \otimes \dots \otimes \llbracket \mathbf{var} \rrbracket$, and states σ and σ' in variables x_1, \dots, x_n , we write $\sigma \xrightarrow{s} \sigma'$ iff $\sigma(x_i) \xrightarrow{s_i} \sigma'(x_i)$ for each i .

We are now in a position to state our definability result.

Lemma 4. *Let A be any type of Basic SCI and let $a \in \llbracket A \rrbracket$ be any element of the monoid interpreting A . There exists a term*

$$x : A \vdash \mathbf{test}(a) : \mathbf{comm}$$

such that $(s, *) \in \llbracket \mathbf{test}(a) \rrbracket$ iff $a \preceq^- s$. There also exists a context $\Gamma = x_1 : \mathbf{var}, \dots, x_n : \mathbf{var}$, Γ -stores $\mathbf{init}(a)$ and $\mathbf{final}(a)$, and a term

$$\Gamma \vdash \mathbf{produce}(a) : A$$

such that there exists $(s, a') \in \llbracket \mathbf{produce}(a) \rrbracket$ with $\mathbf{init}(a) \xrightarrow{s} \mathbf{final}(a)$ if and only if $a \preceq^+ a'$.

Proof. We will prove the two parts of this lemma simultaneously by induction on the type A . First note that any $a \in \llbracket A \rrbracket$ is a sequence of elements from a certain alphabet. Before beginning the main induction, we show that it suffices to consider the case when a is a singleton sequence. The cases when a is empty are trivial: $\mathbf{test}(\[]) = \mathbf{skip}$ and $\mathbf{produce}(\[])$ is any divergent term. If $a = [a_1, a_2, \dots, a_n]$, then we can define $\mathbf{test}(a)$ as

$$\mathbf{test}([a_1]) ; \mathbf{test}([a_2]) ; \dots ; \mathbf{test}([a_n]).$$

For the `produce` part, suppose that $A = A_1 \multimap A_2 \multimap \dots \multimap A_k \multimap B$ for some base type B , and that the context Γ contains all the variables needed to define the $\mathbf{produce}(a_i)$. For any store σ over variables x_1, \dots, x_n , define $\mathbf{check}(\sigma)$ to be the term

```

if (! $x_1 \neq \sigma(x_1)$ ) then diverge
else if (! $x_2 \neq \sigma(x_2)$ ) then diverge
...
else if (! $x_n \neq \sigma(x_n)$ ) then diverge
else skip

```

Define $\mathbf{set}(\sigma)$ to be $x_1 := \sigma(x_1) ; \dots ; x_n := \sigma(x_n)$.

An appropriate term $\text{produce}(a)$ can then be defined as follows.

$$\begin{aligned} \Gamma, x : \mathbf{var} \vdash \lambda \vec{y}_i^{A_i}. x := !x + 1; \\ \quad \mathbf{if} (!x = 1) \mathbf{then} \text{produce}(a_1)y_1 \dots y_n \\ \quad \mathbf{else if} (!x = 2) \mathbf{then} \text{check}(\mathbf{final}(a_1)); \\ \quad \quad \mathbf{set}(\mathbf{init}(a_2)); \\ \quad \quad \text{produce}(a_2)y_1 \dots y_n \\ \quad \dots \\ \quad \mathbf{else if} (!x = n) \mathbf{then} \text{check}(\mathbf{final}(a_{n-1})); \\ \quad \quad \mathbf{set}(\mathbf{init}(a_n)); \\ \quad \quad \text{produce}(a_n)y_1 \dots y_n \\ \mathbf{else diverge} \end{aligned}$$

The required initial state $\mathbf{init}(a)$ is $\mathbf{init}(a_1)[x \mapsto 0]$, and the final state $\mathbf{final}(a)$ is $\mathbf{final}(a_n)[x \mapsto n]$.

We now define $\text{test}(a)$ and $\text{produce}(a)$ for the case when a is a singleton, by induction on the structure of the type A .

For the type \mathbf{comm} , we define

$$\begin{aligned} \text{test}(\ast) &= x : \mathbf{comm} \vdash x : \mathbf{comm} \\ \text{produce}(\ast) &= y : \mathbf{var} \vdash y := !y + 1 : \mathbf{comm} \\ \mathbf{init}(\ast) &= (y \mapsto 0) \\ \mathbf{final}(\ast) &= (y \mapsto 1) \end{aligned}$$

Note the way the initial and final states check that the command $\text{produce}(\ast)$ is used exactly once.

The type \mathbf{exp} is handled similarly:

$$\begin{aligned} \text{test}(n) &= x : \mathbf{exp} \vdash \mathbf{if} (x = n) \mathbf{then skip else diverge} : \mathbf{comm} \\ \text{produce}(n) &= y : \mathbf{var} \vdash y := !y + 1; n : \mathbf{exp} \\ \mathbf{init}(n) &= (y \mapsto 0) \\ \mathbf{final}(n) &= (y \mapsto 1) \end{aligned}$$

For \mathbf{var} , there are two kinds of action to consider: those for reading and those for writing. For writing we define:

$$\begin{aligned} \text{test}(\mathbf{write}(n)) &= x : \mathbf{var} \vdash x := n : \mathbf{comm} \\ \text{produce}(\mathbf{write}(n)) &= x : \mathbf{var}, y : \mathbf{var} \vdash y := !y + 1; x : \mathbf{var} \\ \mathbf{init}(\mathbf{write}(n)) &= (x \mapsto n + 1, y \mapsto 0) \\ \mathbf{final}(\mathbf{write}(n)) &= (x \mapsto n, y \mapsto 1) \end{aligned}$$

For $\text{produce}(\mathbf{write}(n))$, the variable y checks that exactly one use is made, and the variable x checks that the one use is a write-action assigning n to the variable.

Reading is handled similarly:

$$\text{test}(\mathbf{read}(n)) = x : \mathbf{var} \vdash \mathbf{if} (!x = n) \mathbf{then skip else diverge} : \mathbf{comm}$$

$$\begin{aligned}
\text{produce}(\text{read}(n)) &= x : \text{var}, y : \text{var} \vdash y := !y + 1; x : \text{var} \\
\text{init}(\text{read}(n)) &= (x \mapsto n, y \mapsto 0) \\
\text{final}(\text{read}(n)) &= (x \mapsto n, y \mapsto 1)
\end{aligned}$$

In $\text{init}(\text{read}(n))$, the variable x holds n so that if the expression $\text{produce}(\text{read}(n))$ is used for a read, the value n is returned. The variable x must also hold n finally, so $\text{produce}(\text{read}(n))$ cannot reach the state $\text{final}(\text{read}(n))$ if it is used to write a value other than n . However, it would admit a single $\text{write}(n)$ action. This is the reason for introducing the \preceq relation: if a term of our language can engage in a $\text{read}(n)$ action, then it can also engage in $\text{write}(n)$.

For a function type $A \multimap B$, the action we are dealing with has the form (s, b) where s is a sequence of actions from A and b is an action from B . We can now define

$$\begin{aligned}
\text{test}(s, b) &= x : A \multimap B \vdash \text{new } x_1, \dots, x_n \text{ in} \\
&\quad \text{set}(\text{init}(s)); \\
&\quad (\lambda x^B. \text{test}(b))(x \text{produce}(s)); \\
&\quad \text{check}(\text{final}(s)); \\
\text{produce}(s, b) &= \lambda x^A. \text{test}(s); \text{produce}(b) \\
\text{init}(s, b) &= \text{init}(b) \\
\text{final}(s, b) &= \text{final}(b)
\end{aligned}$$

where x_1, \dots, x_n are the variables used in $\text{produce}(s)$.

The non-interference between function and argument allows us to define these terms very simply: for $\text{test}(s, b)$ we supply the function x with an argument which will produce the sequence s , and check that the output from x is b . We must also check that the function x uses its argument in the appropriate, s -producing way, which is done by means of the $\text{init}(s)$ and $\text{final}(s)$ states. For $\text{produce}(s, b)$ we simply test that the argument x is capable of producing s , and then produce b .

It is straightforward to check that these terms have the required properties.

The following lemma holds because the language we are considering is deterministic.

Lemma 5. *If M is any term of Basic SCI and $(s, t), (s', t') \in \llbracket M \rrbracket$ are such that $(s, t) \preceq^- (s', t')$ then $(s, t) = (s', t')$.*

Theorem 2 (Equational Completeness). *If $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$ are terms of basic SCI such that $M \cong N$ then $\llbracket M \rrbracket = \llbracket N \rrbracket$.*

Proof. Consider any $(s, a) \in \llbracket M \rrbracket$. Supposing that $\Gamma = y_1 : A_1, \dots, y_n : A_n$, we know that $(e_I, (s, a)) \in \llbracket \lambda \vec{y}. M \rrbracket$. The term $\text{test}(s, a)$ from the definability lemma (Lemma 4) therefore has the property that $\llbracket (\lambda x. \text{test}(s, a))(\lambda \vec{y}. M) \rrbracket = \llbracket \text{skip} \rrbracket$ so we know that $(\lambda x. \text{test}(s, a))(\lambda \vec{y}. M) \Downarrow$ by soundness. Since $M \cong N$, we must also have that $(\lambda x. \text{test}(s, a))(\lambda \vec{y}. N) \Downarrow$ and hence $\llbracket (\lambda x. \text{test}(s, a))(\lambda \vec{y}. N) \rrbracket = \llbracket \text{skip} \rrbracket$.

This implies that there is some $(s', a') \in \llbracket N \rrbracket$ such that $((s', a'), *) \in \text{test}(s, a)$. By the defining property of $\text{test}(s, a)$, it is the case that $(s, a) \preceq^- (s', a')$.

Applying a symmetric argument, we can show that there is some $(s'', a'') \in \llbracket M \rrbracket$ such that $(s', a') \preceq^- (s'', a'')$.

Since both (s, a) and (s'', a'') are in $\llbracket M \rrbracket$ and since $(s, a) \preceq^- (s'', a'')$, the previous lemma tells us that $(s, a) = (s'', a'')$ and hence $(s, a) = (s', a')$. Thus $\llbracket M \rrbracket \subseteq \llbracket N \rrbracket$. We can argue symmetrically to show that $\llbracket N \rrbracket \subseteq \llbracket M \rrbracket$ and hence conclude that $\llbracket M \rrbracket = \llbracket N \rrbracket$.

Putting soundness and completeness together yields full abstraction.

Theorem 3 (Full Abstraction). *Terms M and N of Basic SCI are equivalent if and only if $\llbracket M \rrbracket = \llbracket N \rrbracket$.*

References

1. S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 334–344. IEEE Computer Society Press, 1998.
2. S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In O’Hearn and Tennent [7], pages 297–329 of volume 2.
3. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the OOPSLA ’91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, November 1991.
4. B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.
5. P. W. O’Hearn. Resource interpretations, bunched implications and the $\alpha - \lambda$ -calculus. In J.-Y. Girard, editor, *Proceedings, Typed Lambda-Calculi and Applications, L’Aquila, Italy, April 1999*, volume 1581 of *LNCS*, pages 258–279. Springer-Verlag, 1999.
6. P. W. O’Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228(1–2):211–252, 1999. A preliminary version appeared in the proceedings of MFPS XI.
7. P. W. O’Hearn and R. D. Tennent, editors. *Algol-like Languages*. Birkhäuser, 1997.
8. G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
9. U. S. Reddy. Global state considered unnecessary: Object-based semantics for interference-free imperative programs. *Lisp and Symbolic Computation*, 9(1), 1996.
10. J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.
11. J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.
12. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.
13. M. Wall and G. McCusker. A fully abstract game semantics of SCI. Draft, 2002.