Logical Methods in Computer Science
Vol. 6 (1:2) 2010, pp. 1–35
www.lmcs-online.org

Submitted May. 14, 2009
Published Jan. 12, 2010

A GRAPH MODEL FOR IMPERATIVE COMPUTATION

GUY MCCUSKER

Department of Computer Science, University of Bath, Bath BA2 7AY, United Kingdom
e-mail address: G.A.McCusker@bath.ac.uk

Abstract. Scott's graph model is a lambda-algebra based on the observation that continuous endofunctions on the lattice of sets of natural numbers can be represented via their graphs. A graph is a relation mapping finite sets of input values to output values.

We consider a similar model based on relations whose input values are finite sequences rather than sets. This alteration means that we are taking into account the order in which observations are made. This new notion of graph gives rise to a model of affine lambda-calculus that admits an interpretation of imperative constructs including variable assignment, dereferencing and allocation.

Extending this untyped model, we construct a category that provides a model of typed higher-order imperative computation with an affine type system. An appropriate language of this kind is Reynolds's Syntactic Control of Interference. Our model turns out to be fully abstract for this language. At a concrete level, it is the same as Reddy's object spaces model, which was the first "state-free" model of a higher-order imperative programming language and an important precursor of games models. The graph model can therefore be seen as a universal domain for Reddy's model.

1. Introduction

This paper is an investigation into the semantics of imperative programs, using a style of model first proposed by Reddy [19]. Reddy's model was a significant development, because it was the first to model imperative programs without the use of an explicit semantic entity representing the store. Instead, programs are interpreted as "objects" (in Reddy's terminology) which exhibit history-sensitive behaviour. The store is not modelled explicitly; instead one models the behaviour that results from the use of the store.

This new approach turned out to be the key to finding models that are *fully abstract*: that is, models whose equational theory coincides with the operationally defined notion of program equivalence. The first such models for higher-order imperative programming languages to be discovered were based on game semantics [2, 1]. Although these models used several ideas from Reddy's work, it was not known whether Reddy's model was itself fully abstract for the language SCI which it interprets.

In this paper, some of which is a much extended exposition of work first presented in [13], we show that Reddy's model is indeed fully abstract. But more than this, we argue that it arises from a straightforward modification of Scott's well-known $\mathcal{P}\omega$ graph-model of

1998 ACM Subject Classification: F.3.2.
Key words and phrases: Semantics of Programming Languages, Denotational Semantics, Local State.

DOI:10.2168/LMCS-6 (1:2) 2010

the λ-calculus [22]. Just as in Scott's work, we develop a model in which every type-object appears as a retract of a universal object, and it turns out that these retractions are all definable in a slightly extended SCI language. Thus the language has a *universal type*, which leads to a very cheap proof of full abstraction. With some additional effort, we show that the extensions required to establish this universal type are in fact conservative, that is, they do not alter the notion of program equivalence. Therefore the original model is itself fully abstract.

We should remark that the work required to establish conservativity of one of these extensions amounts to a partial definability result which would be enough to prove full abstraction of the original model directly; indeed, that is what was done in [13]. Nevertheless, we believe that the presentation in terms of conservativity is useful, not least because of the ease of establishing full abstraction for the extended language.

1.1. **Related work.** The utility of a universal type for establishing properties of a model is well-known, and was explained in detail by Longley [11]. The central idea of this paper, of modifying Scott's graph model to record slightly different information, has also been used by Longley in [12] to obtain a model of fresh name generation. A similar model construction has been investigated by Hyland et al. [7]. We shall remark further on the connections between these papers and our present work below, although we leave closer investigation for future work.

The denotational semantics of SCI was first treated by O'Hearn [17] using functor categories. Reddy's model [19] was the first to avoid the explicit use of a store-component in the mathematical model, but as mentioned above this model was not known to be fully abstract until a preliminary version of the work being reported here appeared [13]. Joint work of the present author and Wall [23] developed a game semantics for SCI and established a full abstraction result. Laird [9] analysed the fully abstract relational model to show that equivalence of programs in a finitary fragment of SCI is decidable, but observational approximation is not, and went on to construct a fully abstract games model of a version of SCI with control operators, establishing decidability of both equivalence and approximation. The SCI type system itself has been refined and extended in two ways: first by Reynolds, using intersection types [21], and then by O'Hearn et al. [15], using a novel system with two-zone type judgements.

1.2. **Acknowledgments.** The author is very grateful to the many researchers with whom he has discussed this work, including Martin Churchill, Jim Laird, John Longley, Ana Carolin Martins, Peter O'Hearn, John Power and Uday Reddy. The comments of anonymous referees were very useful in the preparation of the final version of the paper. The author also benefitted from the support of two EPSRC research grants during the development and preparation of this paper.

2. Scott's $\mathcal{P}\omega$ model

We begin with a brief review of Scott's $\mathcal{P}\omega$ *graph model* of the λ-calculus, which appeared in the seminal paper *Data Types as Lattices* [22].

Let $\mathcal{P}\omega$ denote the lattice of sets of natural numbers, ordered by inclusion. A continuous function $f : \mathcal{P}\omega \to \mathcal{P}\omega$ is determined by its action on finite sets. Therefore, such an f is determined by the set

$$\mathsf{graph}(f) = \{(S, n) \mid S \subseteq_{\mathsf{fin}} \omega, n \in \omega, n \in f(S)\}.$$

Conversely, let G be a set of pairs (S, n) with $S \subseteq_{\mathsf{fin}} \omega$ and $n \in \omega$. We can define a continuous function $\mathsf{fun}(G) : \mathcal{P}\omega \to \mathcal{P}\omega$ by

$$\mathsf{fun}(G)(S) = \{n \mid \exists S' \subseteq S.(S', n) \in G\}$$

and it is clear that for any continuous f, $\mathsf{fun}(\mathsf{graph}(f)) = f$.

Let $\mathsf{code}(-)$ be any injective encoding

$$\mathsf{code} : \mathcal{P}_{\mathsf{fin}}\omega \times \omega \to \omega.$$

Writing $[\mathcal{P}\omega \to \mathcal{P}\omega]$ for the complete partial order of continuous functions from $\mathcal{P}\omega$ to itself, the mapping

$$f \mapsto \{\mathsf{code}(S, n) \mid (S, n) \in \mathsf{graph}(f)\}$$

is a continuous function $[\mathcal{P}\omega \to \mathcal{P}\omega] \to \mathcal{P}\omega$, and

$$S \mapsto \mathsf{fun}(\{(S', n) \mid \mathsf{code}(S', n) \in S\})$$

is a continuous function $\mathcal{P}\omega \to [\mathcal{P}\omega \to \mathcal{P}\omega]$. These two mappings therefore form a retraction

$$[\mathcal{P}\omega \to \mathcal{P}\omega] \trianglelefteq \mathcal{P}\omega$$

in the category of domains and continuous functions, so that $\mathcal{P}\omega$ is a reflexive object in this category, and thus a model of untyped λ-calculus. For more details on how reflexive objects are used to model λ-calculus, see Barendregt [4].

Scott in fact worked in the other direction: from the $\mathcal{P}\omega$ model he defined a category in which to work, using the *Karoubi envelope* (see for example [10]) of the monoid of endomorphisms of $\mathcal{P}\omega$. One way of presenting this monoid is as follows. Its elements are graphs of continuous functions from $\mathcal{P}\omega$ to itself; explicitly, an element a is a set of pairs (S, n), where $S \subseteq_{\mathsf{fin}} \omega$ and $n \in \omega$, such that

$$(S, n) \in a \wedge S \subseteq S' \implies (S', n) \in a.$$

(It is easy to verify that these are exactly the image of the $\mathsf{graph}(-)$ function.) The monoid operation is the graph representation of function composition, which can be defined by

$$a \cdot b = \{(\bigcup_{i=1}^{k} S_i, n) \mid \exists m_1, \ldots, m_k.(\{m_1, \ldots, m_k\}, n) \in b \wedge (S_i, m_i) \in a, i = 1, \ldots, k\}.$$

The Karoubi envelope of this monoid is the category whose objects are idempotents, i.e. elements a such that $a = a \cdot a$, and maps $f : a \to b$ are elements of the monoid such that $f = a \cdot f \cdot b$. Scott shows that this is a cartesian closed category and notes that it is equivalent to the category of separable continuous lattices and continuous maps. A similar theory yielding a category of cpos was developed by Plotkin [18]. In this paper, we will show that replacing the finite *sets* S in the above construction with finite *sequences* yields a category appropriate for modelling imperative computation.

The monoid in question has as its elements set of pairs (s, n) where s is a finite sequence of natural numbers and n is a natural. Multiplication is defined by

$$a \cdot b = \{(s_1 \cdots s_k, n) \mid \exists m_1, \ldots, m_k.(m_1 \cdots m_k, n) \in b \wedge (s_i, m_i) \in a, i = 1, \ldots, k\}$$

where $s_1 \cdots s_k$ denotes the concatenation of the sequences $s_1, \ldots, s_k$ and we identify singleton sequences with their unique elements.

Let us call this monoid $\mathcal{M}$ and its Karoubi envelope $\mathcal{K}(\mathcal{M})$. Concretely, the connection between $\mathcal{M}$ and Scott's monoid is very straightforward: sequences replace Scott's finite sets, and concatenation replaces union. It seems obvious that the move from Scott's construction to ours is nothing more than replacing one monad, the monad of finite powerset, with another, that of finite sequences, in some formal construction. In fact the situation is not quite so straightforward: in order to set things up in an axiomatic fashion, one appears to require a distributive law of the monad at hand over the powerset monad. While the monad of finite sequences does distribute over $\mathcal{P}$, $\mathcal{P}_{\text{fin}}$ does not. This situation has been studied by Hyland et al. in [7], where models along the lines of Scott's are built axiomatically, using a Kleisli-category construction. Their work only applies to commutative monads, and therefore not to the finite-sequence monad, so is not directly applicable here. Moreover, for our purposes neither the category $\mathcal{K}(\mathcal{M})$ nor the kind of Kleisli construction proposed by Hyland et al. provides the most convenient setting in which to work. Although our model of imperative computation can be seen as living entirely within these categories, we shall propose a somewhat different construction which yields additional structure useful in the analysis of the model.

We note also that Longley has recently shown how a similar category, built from an untyped graph-style model using the monad of finite multisets, as opposed to finite sets or finite sequences, provides a model of fresh name generation [12]. In future work, we plan to investigate the relationships between all these models in greater detail, and explore the constructions at the higher level of generality proposed by Hyland et al.

3. Syntactic Control of Interference

The imperative language we shall model is Reynolds's *Syntactic Control of Interference* (SCI) [20], and this section is devoted to the presentation of its syntax, operational semantics and notion of program equivalence. The language was introduced by Reynolds as an approach to the problem of establishing the non-interference properties of procedures and their arguments required by specification logic. Reddy noticed that it was precisely this interference-free fragment of an Algol-like language which his model could interpret. Later, Reddy and O'Hearn showed that the model could be extended to a full Algol-like language by means of the Yoneda embedding [16], but it was not until the refinement of game semantics was discovered that a fully abstract model for such a language became available.

The SCI language consists of a direct combination of the language of while-loops, local variable allocation and the simply-typed λ-calculus with an affine type discipline. The types of SCI are given by the grammar

$$A ::= \mathtt{nat} \mid \mathtt{comm} \mid \mathtt{var} \mid A \multimap A$$

where the base types are those of natural numbers (`nat`), commands (`comm`) and assignable variables (`var`). The terms of the language are as follows.

$$\begin{array}{rcl} M & ::= & n \mid M + M \mid M - M \mid \ldots \\ & \mid & \texttt{skip} \mid M\,;M \mid M := M \mid !\,M \\ & \mid & \texttt{while}\ M\ \texttt{do}\ M \mid \texttt{ifzero}\ M\ \texttt{then}\ M\ \texttt{else}\ M \\ & \mid & x \mid \lambda x^A.M \mid MM \\ & \mid & \texttt{new}\ x\ \texttt{in}\ M \end{array}$$

where n ranges over the natural numbers, x over a countable set of identifiers, and A over the types of SCI. We adopt the usual conventions with regard to binding of identifiers: $\lambda x^A.M$ binds x in M; terms are identified up to α-equivalence; and $M[N/x]$ denotes the capture-avoiding substitution of N for free occurrences of x in M.

The type system of the language imposes an affine discipline on application: no function is allowed to share free identifiers with its arguments. Typing judgments take the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$$

where the x_i are distinct identifiers, the A_i and A are types, and M is a term. We use Γ and Δ to range over contexts, that is, lists $x_1 : A_1, \ldots, x_n : A_n$ of identifier-type pairs with all identifiers distinct. The well-typed terms are given by the following inductive definition, in which it is assumed that all judgments are well-formed.

λ-calculus:

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \multimap B} \qquad \frac{\Gamma \vdash M : A \multimap B \qquad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

Structural Rules:

$$\frac{\Gamma \vdash M}{\Gamma, x : A \vdash M}\ \text{weakening} \qquad \frac{\Gamma \vdash M}{\widetilde{\Gamma} \vdash M}\ \text{exchange}$$

Arithmetic:

$$\frac{}{\vdash n : \texttt{nat}} \qquad \frac{\Gamma \vdash M : \texttt{nat} \qquad \Gamma \vdash N : \texttt{nat}}{\Gamma \vdash M \odot N : \texttt{nat}}\ \odot \in \{+, -, \ldots\}$$

Sequential composition:

$$\frac{}{\vdash \texttt{skip} : \texttt{comm}} \qquad \frac{\Gamma \vdash M : \texttt{comm} \qquad \Gamma \vdash N : B}{\Gamma \vdash M\,;N : B}\ B \in \{\texttt{comm}, \texttt{nat}, \texttt{var}\}$$

Assignable variables:

$$\frac{\Gamma \vdash M : \texttt{var} \qquad \Gamma \vdash N : \texttt{nat}}{\Gamma \vdash M := N : \texttt{comm}} \qquad \frac{\Gamma \vdash M : \texttt{var}}{\Gamma \vdash !\,M : \texttt{nat}}$$

Control structures:

$$\frac{\Gamma \vdash M : \mathtt{nat} \qquad \Gamma \vdash N : \mathtt{comm}}{\Gamma \vdash \mathtt{while}\ M\ \mathtt{do}\ N : \mathtt{comm}}$$

$$\frac{\Gamma \vdash M : \mathtt{nat} \qquad \Gamma \vdash N_1 : B \qquad \Gamma \vdash N_2 : B}{\Gamma \vdash \mathtt{ifzero}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : B}\ B \in \{\mathtt{comm}, \mathtt{nat}, \mathtt{var}\}$$

Local blocks:

$$\frac{\Gamma, x : \mathtt{var} \vdash M : B}{\Gamma \vdash \mathtt{new}\ x\ \mathtt{in}\ M : B}\ B \in \{\mathtt{comm}, \mathtt{nat}\}$$

In the exchange rule, $\widetilde{\Gamma}$ denotes any permutation of the list Γ. In the rule for application, the assumption that the conclusion is well-formed implies that Γ and Δ contain distinct identifiers. This was key to Reynolds's interference control agenda: in the absence of a contraction rule, the only source of identifier aliasing in the language is through procedure application, so by enforcing the constraint that procedures and their arguments have no identifiers in common, one eliminates all aliasing. It then follows that program phrases with no common identifiers cannot interfere with one another.

Note. Our version of SCI allows side-effects at all base types: see the typing rule for sequential composition. We also include a conditional at all base types. Variable allocation, however, is restricted to blocks of type `comm` and `nat`: terms such as $\mathtt{new}\ x\ \mathtt{in}\ x$ are not permitted, because any sensible operational semantics for such terms would violate the stack discipline for allocation and deallocation of variables.

The operational semantics of the language is given in terms of *stores*, that is, functions from identifiers to natural numbers. A store σ has as its domain a finite set of identifiers, $\mathsf{dom}(\sigma)$. Given a store σ, we write $(\sigma \mid x \mapsto n)$ for the store with domain $\mathsf{dom}(\sigma) \cup \{x\}$ which maps x to n and is identical to σ on other identifiers. Note that this operation may extend the domain of σ.

Operational semantic judgments take the form

$$\Gamma \vdash \sigma, M \Downarrow \sigma', V : A$$

where

- Γ is a context containing only `var`-type identifiers
- σ and σ' are stores whose domain is exactly those identifiers in Γ
- M and V are terms
- A is a type
- $\Gamma \vdash M : A$ and $\Gamma \vdash V : A$
- V is a *value*, that is, a natural number, the constant `skip`, an identifier (which must have type `var`) or a λ-abstraction.

For the sake of brevity we omit the typing information from the inductive definition below, writing judgments of the form $\sigma, M \Downarrow \sigma', V$.

Values and functions:

$$\frac{}{\sigma, V \Downarrow \sigma, V}\ V \text{ a value} \qquad \frac{\sigma, M \Downarrow \sigma', \lambda x^A.M' \quad \sigma', M'[N/x] \Downarrow \sigma'', V}{\sigma, MN \Downarrow \sigma'', V}$$

Operations:

$$\frac{\sigma, M_1 \Downarrow \sigma', n_1 \qquad \sigma', M_2 \Downarrow \sigma'', n_2}{\sigma, M_1 \odot M_2 \Downarrow \sigma'', n}\ n = n_1 \odot n_2, \odot \in \{+, -, \ldots\}$$

Variables:

$$\frac{\sigma, N \Downarrow \sigma', n \qquad \sigma', M \Downarrow \sigma'', x}{\sigma, M := N \Downarrow (\sigma'' \mid x \mapsto n), \mathtt{skip}} \qquad \frac{\sigma, M \Downarrow \sigma', x}{\sigma, !\, M \Downarrow \sigma', \sigma'(x)}$$

Control structures:

$$\frac{\sigma, M \Downarrow \sigma', \mathtt{skip} \qquad \sigma', N \Downarrow \sigma'', V}{\sigma, M\,;N \Downarrow \sigma'', V}$$

$$\frac{\sigma, M \Downarrow \sigma', n}{\sigma, \mathtt{while}\ M\ \mathtt{do}\ N \Downarrow \sigma', \mathtt{skip}}\ n \neq 0$$

$$\frac{\sigma, M \Downarrow \sigma', 0 \qquad \sigma', N \Downarrow \sigma'', \mathtt{skip} \qquad \sigma'', \mathtt{while}\ M\ \mathtt{do}\ N \Downarrow \sigma''', \mathtt{skip}}{\sigma, \mathtt{while}\ M\ \mathtt{do}\ N \Downarrow \sigma''', \mathtt{skip}}$$

$$\frac{\sigma, M \Downarrow \sigma', 0 \qquad \sigma', N_1 \Downarrow \sigma'', V}{\sigma, \mathtt{ifzero}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \Downarrow \sigma'', V}$$

$$\frac{\sigma, M \Downarrow \sigma', n \qquad \sigma', N_2 \Downarrow \sigma'', V}{\sigma, \mathtt{ifzero}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 \Downarrow \sigma'', V}\ n \neq 0$$

Local blocks:

$$\frac{(\sigma \mid x \mapsto 0), M \Downarrow (\sigma' \mid x \mapsto n), V}{\sigma, \mathtt{new}\ x\ \mathtt{in}\ M \Downarrow \sigma', V}$$

Note that in the rule for local blocks, the well-formedness constraints on the conclusion $\sigma, \mathtt{new}\ x\ \mathtt{in}\ M \Downarrow \sigma', V$ mean that the domains of definition of σ and σ' are the same, and do not include x. Therefore the variable x is only available during the execution of the block M.

We remark that, though the operational semantics takes account of the possibility that evaluating a term of function-type could change the store, the fact that all the store-changing term constructs are confined to the base types means that this does not happen: whenever $\sigma, M \Downarrow \sigma', V$ for some M and V of type $A \multimap B$, we have $\sigma = \sigma'$ as a straightforward induction will establish.

We now define a notion of *contextual equivalence* on programs in the usual way: given terms $\Gamma \vdash M, N : A$, we say that M and N are contextually equivalent, and write $M \cong N$, if and only if for every context $C[-]$ such that $\vdash C[M], C[N] : B$ for $B \in \{\mathtt{comm}, \mathtt{nat}\}$, and every value $\vdash V : B$,

$$C[M] \Downarrow V \iff C[N] \Downarrow V.$$

(We omit the unique store over no variables from the operational semantic judgments.)

One can also define a *contextual preorder*: given the same data as above, we write $M \mathrel{\underset{\sim}{\sqsubset}} N$ iff for all contexts $C[-]$ and values V,

$$C[M] \Downarrow V \Longrightarrow C[N] \Downarrow V.$$

4. Reddy's object-spaces model

In this section we give a direct, concrete definition of a semantics for SCI which accords with the model given by Reddy [19]. To begin with we define the model without imposing any structure on it, simply using sets and relations. Later we go on to construct a category in which our modified graph model lives as a monoid of endomorphisms of a particular object, and show that the model of SCI inhabits that category. We shall then exploit the structure of the category to obtain a clean proof of the model's soundness. However, for pedagogical reasons we believe the concrete presentation of the model in this section is worthwhile. In particular, for the fragment of the language without abstraction and application, the model is very simple and intuitively appealing, and its soundness is easy to establish.

4.1. **A model based on events.** The key idea behind Reddy's model is that computations are interpreted not as mappings from initial to final states (i.e. *state transformers*), but using sequences of observable *events*. A program will have as its denotation a set of tuples of such sequences.

A type is interpreted as a set: the set of observable events at that type. We define the semantics of types as follows.

$$\begin{aligned} [\![\mathtt{nat}]\!] &= \mathbb{N}, && \text{the set of natural numbers} \\ [\![\mathtt{comm}]\!] &= \{*\}, && \text{a singleton set} \\ [\![\mathtt{var}]\!] &= \{\mathsf{read}(n), \mathsf{write}(n) \mid n \in \mathbb{N}\} \\ [\![A \multimap B]\!] &= [\![A]\!]^* \times [\![B]\!] \end{aligned}$$

where $[\![A]\!]^*$ denotes the set of finite sequences over $[\![A]\!]$.

The basic event one can observe of a term of type `nat` is the production of a natural number, so $\mathbb{N}$ is the interpretation of `nat`. A closed term of type `comm` can do nothing interesting apart from terminating when executed, so `comm` is interpreted as a singleton set: we will see later that it is the open terms of type `comm` which behave more like state-transformers. At the type `var`, there are two kinds of event: $\mathsf{read}(n)$ events correspond to dereferencing a variable and receiving n as the result, and $\mathsf{write}(n)$ events correspond to assigning n to the variable, and observing termination of this operation.

For the function types, the idea is that a single use of a function $A \multimap B$ will result in a single observable output event from B, but may give rise to a sequence of events in the argument of type A. Compare and contrast with Scott's $\mathcal{P}\omega$ model: there functions are modelled as sets of pairs (S, n) where S is a set of input-observations and n is an output, while here we have sets of pairs (s, n) where the input observations form sequences rather than sets.

The denotation of a term

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

will be a set of tuples

$$(s_1, \ldots, s_n, b)$$

where each $s_i \in [\![A_i]\!]^*$ and $b \in [\![B]\!]$. Again the idea is that such a tuple records the ability of M to produce observable event b while itself observing the sequences s_i of events in (the terms bound to) its free identifiers.

4.1.1. *Remark.* Note that, in this model, the observed behaviour in each variable is recorded separately; that is, there is no record of how interactions with the various variables are interleaved. It is precisely this which means we can only model SCI rather than the full Idealized Algol language. The models based on game semantics refine the present model by breaking each event into two, a start and a finish, and recording the interleaving between actions, thereby overcoming this limitation.

A little notation must be introduced before we give the definition of the semantics. We will abbreviate such tuples $s_1, \ldots, s_n$ as $\vec{s}$, and semantic elements as above will become $(\vec{s}, b)$, or simply b when $n = 0$. We use $\vec{s}\vec{s'}$ to denote the componentwise concatenation of the tuples of sequences $s_1, \ldots, s_n$ and $s'_1, \ldots, s'_n$.

We say that a sequence $s \in [\![\mathtt{var}]\!]^*$ is a *cell-trace* iff every read action in s carries the same value as the most recent write, if any, and zero if there has been no write yet. (A formal definition appears later.)

We now give the definition of the semantics by induction on the typing derivation of terms: for each typing rule, Figure 1 gives an equation which defines the semantics of the term in the rule's conclusion by reference to the semantics of the terms in its hypotheses.

4.2. **Examples.**

- Consider the program swap, defined by

  $$x : \mathtt{var}, y : \mathtt{var}, z : \mathtt{var} \vdash z := !\,x\,;\, x := !\,y\,;\, y := !\,z : \mathtt{comm}.$$

  It is straightforward to compute that $[\![\mathsf{swap}]\!]$ is the set

  $$\{(\mathsf{read}(n)\mathsf{write}(n'), \mathsf{read}(n')\mathsf{write}(n''), \mathsf{write}(n)\mathsf{read}(n''), *) \mid n, n', n'' \in \mathbb{N}\}.$$

  The semantic definitions do not yet enforce variable-like behaviour, so that in particular n and n'' need not be equal.

  However, the semantics of $\mathtt{new}\ z\ \mathtt{in}\ \mathsf{swap}$ selects just those entries in which z behaves like a good variable, so that $n = n''$, and then hides the z-behaviour:

  $$[\![\mathtt{new}\ z\ \mathtt{in}\ \mathsf{swap}]\!] = \{(\mathsf{read}(n)\mathsf{write}(n'), \mathsf{read}(n')\mathsf{write}(n), *) \mid n, n' \in \mathbb{N}\}.$$

  Thus the values in x and y are swapped, and the semantics does not record anything about the use of z or the fact that x was reassigned first.
- The type $\mathtt{comm} \multimap \mathtt{comm}$ has as its elements all pairs of the form

  $$(* \cdot * \cdot * \cdots *, *).$$

  A deterministic program of this type will contain at most one such element in its denotation, corresponding to a "for loop" which executes its argument a fixed, finite number of times. There is also the empty set, corresponding to a program which never terminates regardless of its argument.

$$
\begin{aligned}
[\![x : A \vdash x : A]\!] &= \{(a, a) \mid a \in [\![A]\!]\} \\
[\![\Gamma \vdash \lambda x^A.M : A \multimap B]\!] &= \\
&\{(s_1, \ldots, s_n, (s, b)) \mid (s_1, \ldots, s_n, s, b) \in [\![\Gamma, x : A \vdash M : B]\!]\} \\
[\![\Gamma, \Delta \vdash MN : B]\!] &= \\
&\left\{ (\vec{s}, \vec{t^1} \ldots \vec{t^k}, b) \;\middle|\; \begin{array}{l} \exists a_1, \ldots, a_k.(\vec{s}, (a_1 \ldots a_k, b)) \in [\![\Gamma \vdash M : A \multimap B]\!] \\ \wedge (\vec{t^i}, a_i) \in [\![\Delta \vdash N : A]\!] \text{ for } i = 1, \ldots, k \end{array} \right\} \\
[\![\Gamma, x : A \vdash M : B]\!] &= \{(\vec{s}, \varepsilon, b) \mid (\vec{s}, b) \in [\![\Gamma \vdash M : B]\!]\} \\
[\![\widetilde{\Gamma} \vdash M : A]\!] &= \{(\widetilde{\vec{s}}, a) \mid (\vec{s}, a) \in [\![\Gamma \vdash M : A]\!]\} \\
[\![\vdash n : \mathtt{nat}]\!] &= \{n\} \\
[\![\Gamma \vdash M_1 \odot M_2 : \mathtt{nat}]\!] &= \\
&\{(\vec{s}\vec{s'}, m_1 \odot m_2) \mid (\vec{s}, m_1) \in [\![\Gamma \vdash M_1 : \mathtt{nat}]\!], (\vec{s'}, m_2) \in [\![\Gamma \vdash M_2 : \mathtt{nat}]\!]\} \\
[\![\vdash \mathtt{skip} : \mathtt{comm}]\!] &= \{*\} \\
[\![\Gamma \vdash M ; N : B]\!] &= \\
&\{(\vec{s}\vec{s'}, b) \mid (\vec{s}, *) \in [\![\Gamma \vdash M : \mathtt{comm}]\!], (\vec{s'}, b) \in [\![\Gamma \vdash N : B]\!]\} \\
[\![\Gamma \vdash M := N]\!] &= \\
&\{(\vec{s}\vec{s'}, *) \mid (\vec{s}, n) \in [\![\Gamma \vdash N : \mathtt{nat}]\!], (\vec{s'}, \mathsf{write}(n)) \in [\![\Gamma \vdash M : \mathtt{var}]\!]\} \\
[\![\Gamma \vdash !\, M : \mathtt{nat}]\!] &= \{(\vec{s}, n) \mid (\vec{s}, \mathsf{read}(n)) \in [\![\Gamma \vdash M : \mathtt{var}]\!]\} \\
[\![\Gamma \vdash \mathtt{while}\ M\ \mathtt{do}\ N : \mathtt{comm}]\!] &= \\
&\left\{ (\vec{s^1}\vec{t^1}\vec{s^2}\vec{t^2} \ldots \vec{s^j}\vec{t^j}\vec{s}, *) \;\middle|\; \begin{array}{l} \forall i.(\vec{s^i}, 0) \in [\![\Gamma \vdash M : \mathtt{nat}]\!] \\ \wedge (\vec{t^i}, *) \in [\![\Gamma \vdash N : \mathtt{comm}]\!] \\ \wedge \exists m \neq 0.(\vec{s}, m) \in [\![\Gamma \vdash M : \mathtt{nat}]\!] \end{array} \right\} \\
[\![\Gamma \vdash \mathtt{ifzero}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : B]\!] &= \\
&\{(\vec{s}\vec{t}, b) \mid (\vec{s}, 0) \in [\![\Gamma \vdash M : \mathtt{nat}]\!], (\vec{t}, b) \in [\![\Gamma \vdash N_1 : B]\!]\} \\
&\qquad \cup \\
&\{(\vec{s}\vec{t}, b) \mid \exists m \neq 0.(\vec{s}, m) \in [\![\Gamma \vdash M : \mathtt{nat}]\!], (\vec{t}, b) \in [\![\Gamma \vdash N_2 : B]\!]\} \\
[\![\Gamma \vdash \mathtt{new}\ x\ \mathtt{in}\ M : B]\!] &= \left\{ (\vec{s}, b) \;\middle|\; \begin{array}{l} \exists s.(\vec{s}, s, b) \in [\![\Gamma, x : \mathtt{var} \vdash M : B]\!] \\ \wedge\ s \text{ is a cell trace.} \end{array} \right\}
\end{aligned}
$$

Figure 1: Reddy-style semantics of SCI

4.3. **Soundness for the ground types.** We now prove that our model is sound with respect to the operational semantics for the fragment of the language excluding abstraction, application, and non-base types. We refer to this fragment as bSCI; it is essentially the language of while-programs plus block allocated variables.

First let us introduce a little more notation.

We define a notion of state transition. Given a sequence $s \in [\![\mathtt{var}]\!]^*$, we define the transitions

$$n \xrightarrow{s} n'$$

where n and n' are natural numbers, as follows.

$$\frac{}{n \xrightarrow{[]} n} \qquad \frac{}{n \xrightarrow{[\mathsf{read}(n)]} n}$$

$$\frac{}{n \xrightarrow{[\mathsf{write}(n')]} n'} \qquad \frac{n \xrightarrow{s} n' \quad n' \xrightarrow{s'} n''}{n \xrightarrow{ss'} n''}$$

We write $n \xrightarrow{s}$ to mean that $n \xrightarrow{s} n'$ for some n'. We can now give a precise definition of cell-trace: a sequence $s \in [\![\mathtt{var}]\!]^*$ is a cell-trace if and only if $0 \xrightarrow{s}$. Note also that $n \xrightarrow{s}$ if and only if $\mathsf{write}(n)s$ is a cell-trace.

We extend this to traces involving more than one `var` type as follows. Given a context $x_1 : \mathtt{var}, \ldots, x_n : \mathtt{var}$, an element $s = (s_1, \ldots, s_n) \in [\![\mathtt{var}]\!]^* \times \cdots \times [\![\mathtt{var}]\!]^*$, and stores σ and σ' in variables x_1, ..., x_n, we write

$$\sigma \xrightarrow{s} \sigma'$$

iff

$$\sigma(x_i) \xrightarrow{s_i} \sigma'(x_i)$$

for each i.

Definition Say that a term $\Gamma \vdash M : B$, where B is a base type and Γ contains only `var`-typed variables, is *good* if and only if:

Case $B = \mathtt{comm}$**:** for all stores σ, σ' over Γ,

$$\sigma, M \Downarrow \sigma', \mathtt{skip} \Leftrightarrow \exists(\vec{s}, *) \in [\![M]\!].\sigma \xrightarrow{\vec{s}} \sigma'$$

Case $B = \mathtt{nat}$**:** for all stores σ, σ' over Γ and all $n \in \mathbb{N}$,

$$\sigma, M \Downarrow \sigma', n \Leftrightarrow \exists(\vec{s}, n) \in [\![M]\!].\sigma \xrightarrow{\vec{s}} \sigma'$$

Case $B = \mathtt{var}$**:** $\Gamma \vdash !M : \mathtt{nat}$ is good and for all $n \in \mathbb{N}$, $\Gamma \vdash M := n : \mathtt{comm}$ is good.

Lemma 4.1. *All terms $\Gamma \vdash M : B$ of bSCI, where B is a base type and Γ contains only* `var`*-typed variables, are good in the above sense.*

Proof. We proceed by induction on the structure of the term M. For the constants `skip` and n, the result is trivial. For variables $x : \mathtt{var}$, we must show that both $!x$ and $x := n$ are good.

Unpacking the definitions, we have

$$[\![!\, x]\!] = \{(\vec{\varepsilon}, \mathsf{read}(n), \vec{\varepsilon}, n) \mid n \in \mathbb{N}\}.$$

But $\sigma \xrightarrow{\vec{\varepsilon}, \mathsf{read}(n), \vec{\varepsilon}} \sigma'$ if and only if $\sigma = \sigma'$ and $\sigma(x) = n$, which holds if and only if $\sigma, !\, x \Downarrow \sigma', n$.

For the assignment part, we have

$$[\![x := n]\!] = \{(\vec{\varepsilon}, \mathsf{write}(n), \vec{\varepsilon}, *)\}$$

and $\sigma \xrightarrow{\vec{\varepsilon}, \mathsf{write}(n), \vec{\varepsilon}} \sigma'$ if and only if $\sigma' = (\sigma \mid x \mapsto n)$, which holds if and only if $\sigma, x := n \Downarrow \sigma', \mathtt{skip}$.

For `while` M `do` N, first note that

$$\sigma, \mathtt{while}\ M\ \mathtt{do}\ N \Downarrow \sigma', \mathtt{skip}$$

if and only if there are sequences of stores σ_i and τ_i, for $i = 1, \ldots, n$, such that $\sigma = \sigma_1$, $\sigma' = \tau_n$,

$$\sigma_i, M \Downarrow \tau_i, 0 \qquad \tau_i, N \Downarrow \sigma_{i+1}, \mathtt{skip}$$

for $i = 1, \dots, n-1$ and

$$\sigma_n, M \Downarrow \tau_n, k$$

for some $k \neq 0$. (This can be proved by induction on derivations in the operational semantics of `while`.)

Therefore, applying the inductive hypothesis to M and N, we have that

$$\sigma, \texttt{while}\ M\ \texttt{do}\ N \Downarrow \sigma', \texttt{skip}$$

if and only if there are $\vec{s_1}, \dots, \vec{s_n}$ and $\vec{t_1}, \dots, \vec{t_{n-1}}$ such that

$$(\vec{s_i}, 0) \in [\![M]\!] \qquad (\vec{t_i}, *) \in [\![N]\!]$$

for $i = 1, \dots, n-1$ and

$$(\vec{s_n}, k) \in [\![M]\!]$$

for some $k \neq 0$, and moreover

$$\sigma_i \xrightarrow{\vec{s_i}} \tau_i \qquad \tau_i \xrightarrow{\vec{t_i}} \sigma_{i+1}$$

for $i = 1, \dots, n-1$ and

$$\sigma_n \xrightarrow{\vec{s_i}} \tau_n.$$

But then we have that

$$\sigma_1 \xrightarrow{\vec{s_1}\vec{t_1} \dots \vec{s_{n-1}}\vec{t_{n-1}}\vec{s_n}} \tau_n$$

and

$$(\vec{s_1}\vec{t_1} \dots \vec{s_{n-1}}\vec{t_{n-1}}\vec{s_n}, *) \in [\![\texttt{while}\ M\ \texttt{do}\ N]\!]$$

by definition. Furthermore, all elements of $[\![\texttt{while}\ M\ \texttt{do}\ N]\!]$ with cell-traces in the Γ part are of this form, which establishes the converse.

The case of $\texttt{ifzero}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2$ is similar to this one, and simpler.

Consider the case of $M := N$. By definition of the operational semantics,

$$\sigma, M := N \Downarrow \sigma', \texttt{skip}$$

if and only if there are σ'', σ''', x and n such that

$$\sigma, N \Downarrow \sigma'', n \qquad \sigma'', M \Downarrow \sigma''', x$$

and $\sigma' = (\sigma''' \mid x \mapsto n)$. This is the same as saying

$$\sigma, N \Downarrow \sigma'', n \qquad \sigma'', M := n \Downarrow \sigma', \texttt{skip}. \tag{4.1}$$

By the inductive hypothesis, both N and M are good, and hence by definition of "good" for terms of type `var`, $M := n$ is good, so (4.1) holds if and only if we have

$$(\vec{s}, n) \in [\![N]\!] \qquad (\vec{t}, *) \in [\![M := n]\!]. \tag{4.2}$$

such that

$$\sigma \xrightarrow{\vec{s}} \sigma'' \qquad \sigma'' \xrightarrow{\vec{t}} \sigma'.$$

By definition of the semantics,

$$(\vec{t}, *) \in [\![M := n]\!] \Leftrightarrow (\vec{t}, \mathsf{write}(n)) \in [\![M]\!]$$

so (4.2) holds if and only if

$$(\vec{s}\vec{t}, *) \in [\![M := N]\!].$$

The case of $!\,M$ follows directly from the inductive hypothesis: since M is good, so is $!\,M$.

Finally we consider $\texttt{new}\ x\ \texttt{in}\ M : \texttt{comm}$ (the `nat` case is similar). By definition of the operational semantics,

$$\sigma, \texttt{new}\ x\ \texttt{in}\ M \Downarrow \sigma', \texttt{skip}$$

iff

$$(\sigma \mid x \mapsto 0), M \Downarrow (\sigma' \mid x \mapsto n), \texttt{skip}.$$

By the inductive hypothesis, this is possible if and only if there is some $(\vec{s}, s', *) \in [\![M]\!]$ with

$$\sigma \xrightarrow{\vec{s}} \sigma' \qquad 0 \xrightarrow{s'} n.$$

The second condition above is the definition of s' being a cell-trace, so this holds if and only if $(\vec{s}, *) \in [\![\texttt{new}\ x\ \texttt{in}\ M]\!]$ as required. □

The fact that all terms are good gives us the following soundness result for bSCI.

Corollary 4.2. *For any closed term $\vdash M : B$ of bSCI, where B is* `comm` *or* `nat`*, $M \Downarrow V$ if and only if $[\![M]\!] = [\![V]\!]$.* □

5. A category of monoids and relations

Before going on to establish the soundness of Reddy's model for the whole of SCI, we shall develop a categorical setting for the model, based on monoids and relations. Our monoid $\mathcal{M}$ appears as the monoid of endomorphisms of an object in this category, so the retracts of this object all live in the category $\mathcal{K}(\mathcal{M})$. It happens that all the objects we use to interpret types of SCI are indeed retracts of this object, so the graph construction does indeed yield a category suitable for modelling imperative computation. Nevertheless it is useful to describe the larger category. Not only is its construction straightforward, but also it possesses some structure beyond that of $\mathcal{K}(\mathcal{M})$ which makes the description of Reddy's model more straightforward, and allows the soundness result above to be extended to the whole language using algebraic reasoning.

We believe that there is a more general description of these constructions to be found, perhaps extending the work of [7]; but we leave this for future work.

To build our category, 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.

5.0.1. *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 $U : \mathbf{Mon} \to \mathbf{Set}$.

5.0.2. *Products.* The category **Mon** has 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. The construction given above generalizes to give all small products.

5.0.3. *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.

5.0.4. *Powerset.* The familiar powerset construction on **Set** lifts to **Mon** and retains much of its structure. Given a monoid A, define the monoid $\mathcal{P}A$ 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 make use of the Kleisli category $\mathbf{Mon}_{\mathcal{P}}$. 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 $\mathcal{P}B$. The identity on A is the singleton map which takes each $a \in A$ to $\{a\}$. Morphisms are composed as follows: given maps $f : A \to B$ and $g : B \to C$, the composite $f\,;g : A \to C$ is defined by

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

The fact that powerset is a *commutative monad* on **Mon** means that the product structure on **Mon** lifts to a monoidal structure on $\mathbf{Mon}_{\mathcal{P}}$ 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

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

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 $f : A \to B$ and $g : C \to D$ in $\mathbf{Mon}_{\mathcal{P}}$, we can define $f \otimes g : A \otimes C \to B \otimes D$ as the homomorphism $f \times g\,;\theta_{B,D}$. See for example [8] for more details on this construction.

5.1. **The category.** The category we will use to model SCI is $(\mathbf{Mon}_{\mathcal{P}})^{\mathrm{op}}$. This category can be seen as a category of "monoids and relations" of a certain kind, so we will call it **MonRel**.

We now briefly explore some of the structure that **MonRel** possesses.

5.1.1. *Monoidal structure.* The monoidal structure on $\mathbf{Mon}_{\mathcal{P}}$ 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. An important consequence of this is that projections exist: for any $A_1, \ldots, A_n$ there are canonical maps

$$\pi_i : A_1 \otimes \cdots \otimes A_n \to A_i.$$

5.1.2. *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.

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

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 $A_1 \multimap (A_2 \multimap \ldots (A_n \multimap C^*))$ for any $A_1, \ldots, A_n$. That is to say, the free monoids form an *exponential ideal* in **MonRel**.

Given a map $f : A \otimes B \to C^*$ in **MonRel**, we write $\Lambda(f)$ for the curried map $A \to (B \multimap C^*)$. The counit of the adjunction is written

$$\mathsf{ev} : (B \multimap C^*) \otimes B \to C^*.$$

5.1.3. *Products.* The coproduct in **Mon** is inherited by the Kleisli-category $\mathbf{Mon}_{\mathcal{P}}$, and since **MonRel** is the opposite of this category, **MonRel** has products.

5.1.4. *An alternative characterization.* We can also describe the category **MonRel** concretely, as follows. Objects are monoids, and maps $A \to B$ 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 $a_1, a_2 \in A$ 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 $(\mathbf{Mon}_{\mathcal{P}})^{\mathrm{op}}$. The action of the isomorphism is as follows. Given a map $A \to B$ in $(\mathbf{Mon}_{\mathcal{P}})^{\mathrm{op}}$, that is to say, a homomorphism

$$f : B \longrightarrow \mathcal{P}(A)$$

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

5.1.5. *Recovering the monoid $\mathcal{M}$.* We remark that the monoid of endomorphisms of the object ω^*, the monoid of sequences of natural numbers, is exactly the monoid $\mathcal{M}$ of Section 2. A map $\omega^* \to \omega^*$ consists of a monoid homomorphism $\omega^* \to \mathcal{P}\omega^*$ which is the same as an ordinary function $\omega \to \mathcal{P}\omega^*$. Reversing the arrows and using the characterization of **Rel** as the Kleisli-category for $\mathcal{P}$ on **Set**, this is just a subset of $\omega^* \times \omega$, and it is routine to check that the composition of these sets is as described in Section 2.

It follows that the full subcategory of **MonRel** consisting of objects which are retracts of ω^* can also be seen a subcategory of the Karoubi envelope $\mathcal{K}(\mathcal{M})$, and it will turn out that all the types of SCI are modelled using objects of this subcategory. Just as Scott used the Karoubi envelope of $\mathcal{P}\omega$ as a category for giving semantics, we can use $\mathcal{K}(\mathcal{M})$. However, **MonRel** proves to be a more convenient category, because it possesses additional objects, in particular tensor products such as $\omega^* \otimes \omega^*$, which assist in the description and analysis of our model but do not belong to $\mathcal{K}(\mathcal{M})$.

It is perhaps worth remarking that Reddy's original work struggled to find a satisfying categorical setting for the model, resorting to the use of multicategories in the absence of objects such as $\omega^* \otimes \omega^*$. We believe our new categorical setting paints a more convincing picture.

5.2. **Modelling SCI in MonRel.** We now show how Reddy's model of SCI lives in **MonRel**. Types are interpreted as objects of the category, that is, as monoids. Indeed every type is interpreted as the free monoid over the set which we used for the direct presentation of the semantics given above. Formally we can give an inductive definition of the semantics of types as follows.

$$\begin{aligned} [\![\mathtt{comm}]\!] &= 1^* \\ [\![\mathtt{nat}]\!] &= \mathbb{N}^* \\ [\![\mathtt{var}]\!] &= [\![\mathtt{comm}]\!]^\omega \times [\![\mathtt{nat}]\!] \\ [\![A \multimap B]\!] &= [\![A]\!] \multimap [\![B]\!]. \end{aligned}$$

For the definition of $[\![A \multimap B]\!]$ to make sense it is essential that every $[\![B]\!]$ is a free monoid. This is clear for the base types `comm` and `nat`. Recalling that products in **MonRel** come from coproducts in **Mon**, and that the coproduct of free monoids is again a free monoid, we see that $[\![\mathtt{var}]\!]$ is a free monoid, and therefore by induction every types is interpreted as the free monoid over some alphabet.

Let us write αA for the underlying alphabet of $[\![A]\!]$, and verify that for every type A, αA is the set that was used in the direct presentation of the semantics above.

For `comm` and `nat`, this is clear. To see that the same holds for `var`, recall that products in **MonRel** come from coproducts in **Mon**, which for free monoids are given by disjoint union of alphabets. So

$$\alpha\mathtt{var} = \left(\sum_w 1\right) + \mathbb{N}.$$

The single element of the nth summand of the left component corresponds to $\mathsf{write}(n)$, and the element n of the right component corresponds to $\mathsf{read}(n)$; indeed we will continue to use this notation below. Our reason for giving the semantic definition in the above form will become clear when we come to the semantics of assignment and dereferencing.

Finally, by the definition of exponential,

$$\alpha(A \multimap B) = (\alpha A)^* \times \alpha B$$

which agrees with our previous definition.

For the semantics of terms, we exploit the categorical structure of **MonRel**: the λ-calculus part is interpreted using the monoidal and exponential structure of the category, while the constants are interpreted by defining particular maps in the category, making use of products for those constants which allow their operands to share variables.

A term $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ is interpreted as a map

$$[\![M]\!] : [\![A_1]\!] \otimes \cdots \otimes [\![A_n]\!] \to [\![B]\!].$$

(If Γ is the context $x_1 : A_1, \ldots, x_n : A_n$ we will often abbreviate the object $[\![A_1]\!] \otimes \cdots \otimes [\![A_n]\!]$ as $[\![\Gamma]\!]$). Unpacking definitions, such a map is a homomorphism

$$[\![B]\!] \to \mathcal{P}([\![A_1]\!] \times \cdots \times [\![A_n]\!]).$$

Since all types are interpreted as free monoids, this is the same as an ordinary function

$$\alpha B \to \mathcal{P}((\alpha A_1)^* \times \cdots \times (\alpha A_n)^*)$$

which in turn corresponds to a subset of

$$(\alpha A_1)^* \times \cdots \times (\alpha A_n)^* \times \alpha B.$$

Under this representation, the denotations of terms in **MonRel** have the same form as those in the direct presentation, and we will use the "sets of tuples" when we need to define morphisms explicitly.

A variable is interpreted as the identity map:

$$[\![x : A \vdash x : A]\!] = \mathsf{id} : [\![A]\!] \to [\![A]\!].$$

Weakening is interepreted using projections: if

$$[\![\Gamma \vdash M : B]\!] = f : [\![\Gamma]\!] \to [\![B]\!]$$

then

$$[\![\Gamma, x : A \vdash M : B]\!] = \pi \,;\, f$$

where $\pi : [\![\Gamma]\!] \otimes [\![A]\!] \to [\![\Gamma]\!]$ is a projection map.

Exchange is interpreted using the symmetry isomorphisms: for any permutation on a context taking Γ to $\widetilde{\Gamma}$ there is a corresponding isomorphism $\mathsf{symm} : [\![\widetilde{\Gamma}]\!] \to [\![\Gamma]\!]$, and then

$$[\![\widetilde{\Gamma} \vdash M : A]\!] = \mathsf{symm} \,;\, [\![\Gamma \vdash M : A]\!].$$

Abstraction is interpreted using the currying part of the exponential adjunction: if

$$[\![\Gamma, x : A \vdash M : B]\!] = f : [\![\Gamma]\!] \otimes [\![A]\!] \to [\![B]\!]$$

then

$$[\![\Gamma \vdash \lambda x^A.M : A \multimap B]\!] = \Lambda(f) : [\![\Gamma]\!] \to [\![A]\!] \multimap [\![B]\!].$$

Application is interpreted using ev:

$$[\![MN]\!] = [\![M]\!] \otimes [\![N]\!] \,;\, \mathsf{ev}.$$

It is straightforward to check that these definitions agree with the concrete ones given earlier.

To interpret the basic imperative constructs, we define a collection of maps in the category. For instance, to interpret $\texttt{while}\ M\ \texttt{do}\ N$ we use a map

$$w : [\![\texttt{nat}]\!] \times [\![\texttt{comm}]\!] \to [\![\texttt{comm}]\!]$$

which we will define below, and set

$$[\![\texttt{while}\ M\ \texttt{do}\ N]\!] = \langle [\![M]\!], [\![N]\!] \rangle\ ;\ w.$$

The object $[\![\texttt{nat}]\!] \times [\![\texttt{comm}]\!]$ is the free monoid over the alphabet $\mathbb{N} \cup \{*\}$. We can therefore define w as the set of tuples

$$w = \{(0 * 0 * \cdots 0 * n, *) \mid n \neq 0\}.$$

Maps interpreting $\texttt{ifzero}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2$, $!\,M$ and $M := N$ can be defined similarly and all yield interpretations which agree with the direct one. However, for assignment and dereferencing, the definition of $[\![\texttt{var}]\!]$ as $[\![\texttt{comm}]\!]^\omega \times [\![\texttt{nat}]\!]$ suggests a more abstract definition using projections: there are projections

$$\mathsf{assign}(n) : [\![\texttt{var}]\!] \to [\![\texttt{comm}]\!]$$

for each n, and

$$\mathsf{deref} : [\![\texttt{var}]\!] \to [\![\texttt{nat}]\!]$$

and these are indeed the maps we need. Thus our interpretation of `var` has the kind of "object oriented" flavour advocated by Reynolds: a variable is an object with ω-many write-methods and a read-method, and its semantics is given by the product of these.

Finally the semantics of `new` is given by means of maps of type

$$[\![\texttt{var} \multimap \texttt{comm}]\!] \to [\![\texttt{comm}]\!] \qquad \text{and} \qquad [\![\texttt{var} \multimap \texttt{nat}]\!] \to [\![\texttt{nat}]\!]$$

defined by the sets

$$\{((s, *), *) \mid s \text{ is a cell trace}\}$$

and

$$\{((s, n), n) \mid n \in \mathbb{N}, s \text{ is a cell trace}\}$$

respectively.

5.3. **Soundness of the model of SCI.** We can now show that our model is sound for the whole of SCI, extending the result of Section 4.3.

First a standard lemma which says that substitution is modelled by composition in the category.

Lemma 5.1 (Substitution)**.** *If $\Gamma, x : A \vdash M : B$ and $\Delta \vdash N : A$ are terms of SCI, then so is $\Gamma, \Delta \vdash M[N/x] : B$, and furthermore $[\![M[N/x]]\!] = \mathsf{id}_{[\![\Gamma]\!]} \otimes [\![N]\!]; [\![M]\!]$.* □

With this in place it is standard that β-reduction is soundly modelled, because of the naturality of currying.

Lemma 5.2. *If $\Gamma, x : A \vdash M : B$ and $\Delta \vdash N : A$, then $[\![(\lambda x.M)N]\!] = [\![M[N/x]]\!]$.* □

Both of these Lemmas are proved by a straightforward induction on the structure of terms. They hold for standard reasons, because we are working in a symmetric monoidal category and using exponentials to model function spaces. We can now establish soundness for our model using purely algebraic reasoning: the fact that there is no recursion in the language makes this particularly straightforward. The key is to establish that every ground-type term of the full language has the same behaviour as a term of $bSCI$; a property that is captured by the following definition.

Definition Let $\Gamma \vdash M : A$ be a term of SCI, where Γ contains only `var`-typed variables. We say that M is *bSCI-expressive* iff:

- A is a ground type and there exists a term $\Gamma \vdash M' : A$ of bSCI such that $[\![M]\!] = [\![M']\!]$ and for all stores σ and values $\Gamma \vdash V : A$
$$\sigma, M \Downarrow \sigma', V \iff \sigma, M' \Downarrow \sigma', V$$
or
- $A = A_1 \multimap A_2$ is a function type and for all bSCI-expressive terms $\Delta \vdash N : A_1$, $\Gamma, \Delta \vdash MN : A_2$ is bSCI-expressive.

Note that the first case above implies that all ground-type terms of bSCI with only `var`-typed free variables are automatically bSCI-expressive.

Lemma 5.3. *Let $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ be any term of SCI, and let $\Gamma_i \vdash N_i : A_i$ be bSCI-expressive terms. Then $M[\vec{N_i}/\vec{x_i}]$ is bSCI-expressive.*

Proof. By induction on the structure of M.

Variables: this case is trivial.

Constants: trivial since constant terms are themselves bSCI-terms.

Term formers of bSCI: for terms such as `while` M_1 `do` M_2, we must prove that `while` $M_1[\vec{N}/\vec{x}]$ `do` $M_2[\vec{N}/\vec{x}]$ is bSCI-expressive.

The subterms $M_i[\vec{N}/\vec{x}]$ are bSCI-expressive by inductive hypothesis, and hence there are terms M_1' and M_2' of bSCI such that
$$[\![M_i']\!] = [\![M_i[\vec{N}/\vec{x}]]\!]$$
for $i = 1, 2$, and for all stores σ and values V,
$$\sigma, M_i' \Downarrow \sigma', V \iff \sigma, M_i[\vec{N}/\vec{x}] \Downarrow \sigma', V.$$
By the definition of the operational semantics it follows that
$$\sigma, \texttt{while } M_1' \texttt{ do } M_2' \Downarrow \sigma', V$$
if and only if
$$\sigma, \texttt{while } M_1[\vec{N}/\vec{x}] \texttt{ do } M_2[\vec{N}/\vec{x}] \Downarrow \sigma', V.$$
By the compositionality of the denotational semantics,
$$[\![\texttt{while } M_1' \texttt{ do } M_2']\!] = [\![\texttt{while } M_1[\vec{N}/\vec{x}] \texttt{ do } M_2[\vec{N}/\vec{x}]]\!]$$
and hence `while` $M_1[\vec{N}/\vec{x}]$ `do` $M_2[\vec{N}/\vec{x}]$ is bSCI-expressive, as required.

The cases of other term-formers which are included in $bSCI$, such as `if` and `new`, are similar.

Abstraction: For a term $\lambda x.M$, we must prove that $\lambda x.M[\vec{N}/\vec{x}]$ is bSCI-expressive. Let us write M' for $M[\vec{N}/\vec{x}]$. By the definition of bSCI-expressive, we must show that for all bSCI-expressive terms $P_1, \ldots, P_k$ such that $(\lambda x.M')P_1 \ldots P_k$ is of ground type, $(\lambda x.M')P_1 \ldots P_k$ is bSCI-expressive.

By the inductive hypothesis, $M'[N/x]$ is bSCI-expressive whenever N is. Hence by definition of bSCI-expressivity, $M'[P_1/x]P_2 \ldots P_k$ is bSCI-expressive whenever the P_i are. Therefore there is a term M'' of bSCI such that $[\![M'']\!] = [\![M'[P_1/x]P_2 \ldots P_k]\!]$ and for all stores σ and values V,

$$\sigma, M'' \Downarrow \sigma', V \iff \sigma, M'[P_1/x]P_2 \ldots P_k \Downarrow \sigma', V.$$

But by soundness of β-reduction,

$$[\![(\lambda x.M')P_1 \ldots P_k]\!] = [\![M'[P_1/x]P_2 \ldots P_k]\!] = [\![M'']\!].$$

This is to say that $(\lambda x.M')P_1 \ldots P_k$ is bSCI-expressive whenever the P_i are, so $\lambda x.M'$ is bSCI-expressive.

Application: For a term $M_1 M_2$, we must show that $M_1[\vec{N}/\vec{x}]M_2[\vec{N}/\vec{x}]$ is bSCI-expressive. But by inductive hypothesis,

$$M_i[\vec{N}/\vec{x}]$$

is bSCI-expressive for $i = 1, 2$ and the result follows by definition of bSCI-expressivity at function types.

□

Lemma 5.4. *For any closed term M of type* `nat` *or* `comm`*, $M \Downarrow V$ iff $[\![M]\!] = [\![V]\!]$.*

Proof. By Lemma 5.3, M is bSCI-expressive and hence there is a term M' of bSCI such that $[\![M]\!] = [\![M']\!]$ and $M \Downarrow V$ if and only if $M' \Downarrow V$. By the soundness for bSCI-terms, Corollary 4.2, $M' \Downarrow V$ if and only if $[\![M']\!] = [\![V]\!]$, and the result follows. □

Theorem 5.5 (Equational Soundness)**.** *If $\Gamma \vdash M, N : A$ are terms such that $[\![M]\!] = [\![N]\!]$, then M and N are contextually equivalent.*

Proof. Since the semantics is compositional, for any context $C[-]$, we have $[\![C[M]]\!] = [\![C[N]]\!]$. By Lemma 5.4, $C[M] \Downarrow V$ iff $[\![C[M]]\!] = [\![V]\!]$ iff $[\![C[N]]\!] = [\![V]\!]$ iff $C[N] \Downarrow V$ as required. □

6. Two extensions to the language

In the next section it will be useful to consider a version of SCI extended with two new constructs: erratic choice and a "bad variable" constructor. It will turn out that in a certain sense these extensions add no new expressive power—in technical parlance, they are *conservative* extensions—but they do alter the character of the language at an intuitive level, and allow new programs to be written. More importantly for our purposes, they give rise to the presence of a *universal type* in the language.

6.1. **Erratic choice.** There are several ways to add an erratic choice operation to the language. As long as we are interested only in the "may-converge" version of the $\Downarrow$ predicate, recording what values are possible as the result of a computation without making any guarantee of termination, the simplest form of erratic choice is a random number generator.

We add to the language a constant `random`, with typing rule

$$\frac{}{\Gamma \vdash \texttt{random} : \texttt{nat}}$$

and operational semantics

$$\frac{}{\sigma, \texttt{random} \Downarrow \sigma, n}$$

for any n.

The denotational semantics of `random` in our model is

$$[\![\Gamma \vdash \texttt{random} : \texttt{nat}]\!] = \{(\vec{\varepsilon}, n) \mid n \in \mathbb{N}\}.$$

6.1.1. *Remark.* Note that if we were to treat the must-converge predicate, this unbounded nondeterminism would be very different from finite nondeterminism, and would lead to some technical difficulties in the semantics, cf. [3]. However, for may-convergence, adding `random` to the language is equivalent to adding a mere binary nondeterministic choice operator.

6.2. **Bad variable constructor.** We alluded earlier to the "object-oriented" nature of our denotational semantics of the `var` type: `var` is seen as the product of countably many assignment methods of type `comm` and a dereferencing method of type `nat`. We can import this reading of the `var` type into the syntax of the language by means of a bad-variable constructor `mkvar`, as follows.

The typing rule is

$$\frac{\Gamma \vdash M : \texttt{nat} \multimap \texttt{comm} \quad \Gamma \vdash N : \texttt{nat}}{\Gamma \vdash \texttt{mkvar}\ M\ N : \texttt{var}}$$

For operational semantics, there are three rules:

$$\frac{}{\sigma, \texttt{mkvar}\ M\ N \Downarrow \sigma, \texttt{mkvar}\ M\ N}$$

$$\frac{\sigma, N \Downarrow \sigma', n \quad \sigma', M \Downarrow \sigma'', \texttt{mkvar}\ M_1\ M_2 \quad \sigma'', M_1 n \Downarrow \sigma''', V}{\sigma, M := N \Downarrow \sigma''', V}$$

$$\frac{\sigma, M \Downarrow \sigma', \texttt{mkvar}\ M_1\ M_2 \quad \sigma', M_2 \Downarrow \sigma'', V}{\sigma, !\, M \Downarrow \sigma'', V}$$

The idea is that $\texttt{mkvar}\ M\ N$ is a variable for which the assignment methods are given by the Mn and the dereferencing method is given by N. Thus any genuine variable x is equivalent to

$$\texttt{mkvar}\ (\lambda n. x := n)\ (!\, x)$$

but many other kinds of variable are available, some with very un-variable-like behaviour, such as

$$\texttt{mkvar}\ (\lambda n. \texttt{skip})\ (3)$$

which always gives 3 when dereferenced.

The denotational semantics of `mkvar` is as follows.

$$[\![\texttt{mkvar}\ M\ N]\!] = \{(\vec{s}, \mathsf{write}(n)) \mid (\vec{s}, *) \in [\![Mn]\!]\} \cup \{(\vec{s}, \mathsf{read}(n)) \mid (\vec{s}, n) \in [\![N]\!]\}$$

A somewhat more abstract presentation can be given. First note that the denotations of terms

$$f : \mathtt{nat} \multimap \mathtt{comm} \vdash fn : \mathtt{comm}$$

for each n give us ω-many maps $[\![\mathtt{nat} \multimap \mathtt{comm}]\!] \to [\![\mathtt{comm}]\!]$ and thus a map

$$\mathsf{flatten} : [\![\mathtt{nat} \multimap \mathtt{comm}]\!] \to [\![\mathtt{comm}]\!]^\omega$$

which "flattens" a function into a tuple. Since $[\![\mathtt{var}]\!] = [\![\mathtt{comm}]\!]^\omega \times [\![\mathtt{nat}]\!]$ we can then define

$$[\![\mathtt{mkvar}\ M\ N]\!] = \langle [\![M]\!]; \mathsf{flatten}, [\![N]\!] \rangle.$$

6.2.1. *Remark.* One might argue that the `mkvar` constructor is unnatural from a programmer's point of view. However, the ability to define one's own assignment and dereferencing operators is a useful programming technique which is frequently exploited in languages such as Ruby, for example [5]. This constructor appears in the syntax of most Algol-like languages which have been studied in the theoretical literature, and is available in most models of such languages too. Our result, to follow, which shows that `mkvar` is a conservative extension of SCI is therefore somewhat comforting; moreover this result can be extended to full Idealized Algol, arguing via a game-based model [14].

6.2.2. *Terminology.* We shall refer to the language SCI extended with `mkvar` as $SCI_{\mathtt{mk}}$. The relation of contextual equivalence for this language, defined in the same way as for SCI, will be denoted $\cong_{\mathtt{mk}}$. Note that this relation may distinguish more terms of the pure SCI language than does $\cong$, because contexts may now make use of `mkvar`; in fact we shall see later that this is not the case, so that `mkvar` is a *conservative extension* of the language. Similarly, the language extended with both `mkvar` and `random` will be called $SCI_{\mathtt{mk,ran}}$ and its notion of contextual equivalence will be written $\cong_{\mathtt{mk,ran}}$.

6.3. **Soundness.** We now show that the model of the extended language $SCI_{\mathtt{mk,ran}}$ is sound. The proof is a straightforward extension of the arguments used to establish Lemma 5.4. For the sake of completeness (of the paper, not the model!) we give the formulation here.

Definition A term $x_1 : \mathtt{var}, \ldots, x_n : \mathtt{var} \vdash M : A$ of $SCI_{\mathtt{mk,ran}}$ is *good* iff

- A is `comm` and for all σ, σ',
$$\sigma, M \Downarrow \sigma', \mathtt{skip}$$
if and only if
$$\exists (\vec{s}, *) \in [\![M]\!].\sigma \xrightarrow{\vec{s}} \sigma'.$$
- A is `nat` and for all σ, σ', n,
$$\sigma, M \Downarrow \sigma', n$$
if and only if
$$\exists (\vec{s}, n) \in [\![M]\!].\sigma \xrightarrow{\vec{s}} \sigma'.$$
- A is `var` and for all n, $M := n$ is good and $!\,M$ is good.
- A is $A_1 \multimap A_2$ and for all good $N : A_1$, $MN : A_2$ is good.

Lemma 6.1. *For any term $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ of $SCI_{\mathtt{mk,ran}}$, if $\Gamma_i \vdash M_i : A_i$ are good terms for $i = 1, \ldots, n$, with the Γ_i disjoint, then $\Gamma_1, \ldots, \Gamma_n \vdash M[\vec{M_i}/\vec{x_i}] : B$ is good.*

Proof. By induction on the structure of M. We treat only the cases of `random` and `mkvar`; the arguments for the others are as in the proofs of Lemmas 4.1 and 5.3.

For `random`, the operational semantics says that

$$\sigma, \texttt{random} \Downarrow \sigma, n$$

for any σ and n. But $\sigma \xrightarrow{\vec{\varepsilon}} \sigma$ and

$$(\vec{\varepsilon}, n) \in [\![\texttt{random}]\!]$$

by definition. Conversely, if $\sigma \xrightarrow{\vec{\varepsilon}} \sigma'$ then $\sigma = \sigma'$, so both directions of the required implication hold.

For `mkvar`, we shall show that if M : `nat` $\multimap$ `comm` and N : `nat` are good, then so is `mkvar` M N.

We must show that (`mkvar` M N) := n and !(`mkvar` M N) are good. By the definition of the operational semantics,

$$\sigma, (\texttt{mkvar}\ M\ N) := n \Downarrow \sigma', \texttt{skip}$$

if and only if

$$\sigma, Mn \Downarrow \sigma', \texttt{skip}.$$

Since M and n are good, this happens if and only if

$$\exists (\vec{s}, *) \in [\![Mn]\!].\sigma \xrightarrow{\vec{s}} \sigma'.$$

By definition of the semantics of `mkvar`, this holds iff

$$\exists (\vec{s}, \mathsf{write}(n)) \in [\![\texttt{mkvar}\ M\ N]\!].\sigma \xrightarrow{\vec{s}} \sigma'$$

which in turn holds iff

$$\exists (\vec{s}, *) \in [\![(\texttt{mkvar}\ M\ N) := n]\!].\sigma \xrightarrow{\vec{s}} \sigma'$$

by definition of the semantics of assignment, which completes the argument. The case for dereferencing is proved similarly. □

Corollary 6.2. *For any closed term M of $SCI_{mk,ran}$ having type* `comm`*, $M \Downarrow$`skip` $\Leftrightarrow * \in [\![M]\!]$, and for any closed term M of type* `nat`*, $M \Downarrow n \Leftrightarrow n \in [\![M]\!]$.* □

Note that the statement of this result is a little different from the analogous result for SCI, Corollary 4.2, because of the nondeterminism in the language.

Just as before, this result is enough to allow us to establish the soundness of our model.

Theorem 6.3. *If M and N are terms of $SCI_{mk,ran}$ of the same type and $[\![M]\!] = [\![N]\!]$, then $M \cong_{\texttt{mk,ran}} N$.*

Another simple corollary will prove useful for us later.

Corollary 6.4. *If M and N are closed terms of $SCI_{mk,ran}$ of type* `nat`*, then $M \cong_{\texttt{mk,ran}} N \iff [\![M]\!] = [\![N]\!]$.*

Proof. The right-to-left implication is Theorem 6.3. Left-to-right holds because if M and N are equivalent, then $M \Downarrow n$ if and only if $N \Downarrow n$ for any n, so by Corollary 6.2, $n \in [\![M]\!]$ if and only if $n \in [\![N]\!]$, that is, $[\![M]\!] = [\![N]\!]$. □

7. A universal type and full abstraction

We begin this section with the observation that every type-object $[\![A]\!]$ in **MonRel** is a retract of $[\![\mathtt{nat}]\!]$, confirming our claim that the Karoubi envelope of the monoid $\mathcal{M}$ is an appropriate setting for modelling imperative computation.

This would be little more than an intriguing observation but for the fact that the maps involved in the retractions are *definable* by terms of $SCI_{\mathtt{mk},\mathtt{ran}}$. Thus, not only is $[\![\mathtt{nat}]\!]$ a universal object for the category of type-objects in **MonRel**, but also `nat` is a universal *type* in the language. This gives rise to a very simple proof of the full abstraction of the model of $SCI_{\mathtt{mk},\mathtt{ran}}$. We then show that this result restricts to the smaller language SCI by demonstrating that $SCI_{\mathtt{mk},\mathtt{ran}}$ extends SCI conservatively.

Lemma 7.1. *Let A be any countable set. The monoid A^* is a retract of $[\![\mathtt{nat}]\!] = \omega^*$ in* **MonRel**.

Proof. Let $f : A \to \omega$ be any injective function. We define maps

$$\mathsf{in} : A^* \to \omega^* \qquad \mathsf{out} : \omega^* \to A^*$$

in **MonRel** by the relations

$$\begin{aligned} \mathsf{in} &= \{(a_1 \cdots a_k, f(a_1) \cdots f(a_k)) \mid a_1, \ldots, a_k \in A\} \\ \mathsf{out} &= \{(f(a_1) \cdots f(a_k), a_1 \cdots a_k) \mid a_1, \ldots, a_k \in A\} \end{aligned}$$

It is immediately clear that these are well-defined maps in **MonRel** and that $\mathsf{in}; \mathsf{out} = \mathsf{id}$. □

Since every type object $[\![A]\!]$ is a list-monoid over a countable set, every type-object is a retract of $[\![\mathtt{nat}]\!]$.

We should remark, however, that not every object used to define the semantics of SCI is a retract of $[\![\mathtt{nat}]\!]$. For example one can show that the object $[\![\mathtt{nat}]\!] \otimes [\![\mathtt{nat}]\!]$ does not have this property. The category **MonRel** therefore possesses some advantages over the category $\mathcal{K}(\mathcal{M})$.

We can go further in our description of type-objects as retracts of $[\![\mathtt{nat}]\!]$: the retractions at hand are denotations of terms of $SCI_{\mathtt{mk},\mathtt{ran}}$.

Definition A type A of SCI is a *definable retract* of `nat` iff there are maps $\mathsf{in} : [\![A]\!] \to \omega^*$ and $\mathsf{out} : \omega^* \to [\![A]\!]$ in **MonRel** such that $\mathsf{in}; \mathsf{out} = \mathsf{id}_{[\![A]\!]}$ and furthermore there are terms $x : A \vdash \mathtt{in} : \mathtt{nat}$ and $y : \mathtt{nat} \vdash \mathtt{out} : A$ of $SCI_{\mathtt{mk},\mathtt{ran}}$ such that $[\![\mathtt{in}]\!] = \mathsf{in}$ and $[\![\mathtt{out}]\!] = \mathsf{out}$.

Theorem 7.2. *Every type of SCI is a definable retract of* `nat`.

Proof. By induction on the structure of types. We shall give particular definable retractions for the types `nat`, `comm`, `var` and $\mathtt{nat} \multimap \mathtt{nat}$. The case of a more general function type $A \multimap B$ is then handled inductively, by defining

$$\begin{aligned} x : A \multimap B \vdash \mathtt{in}_{A \multimap B} : \mathtt{nat} &\triangleq \mathtt{in}_{\mathtt{nat} \multimap \mathtt{nat}}(\lambda n : \mathtt{nat}.\mathtt{in}_{\mathtt{B}}(x(\mathtt{out}_{A}(n)))) : \mathtt{nat} \\ y : \mathtt{nat} \vdash \mathtt{out}_{\mathtt{A} \multimap \mathtt{B}} &\triangleq \lambda a : A.\mathtt{out}_{\mathtt{B}}(\mathtt{out}_{\mathtt{nat} \multimap \mathtt{nat}}(y)(\mathtt{in}_{A}(a))) : A \multimap B. \end{aligned}$$

The identity maps clearly make `nat` a definable retract of itself. For the type `comm`, we define

$$\begin{aligned} x : \mathtt{comm} \vdash \mathtt{in}_{\mathtt{comm}} : \mathtt{nat} &\triangleq x; 0 \\ y : \mathtt{nat} \vdash \mathtt{out}_{\mathtt{comm}} : \mathtt{comm} &\triangleq \mathtt{ifzero}\ y\ \mathtt{then}\ \mathtt{skip}\ \mathtt{else}\ \Omega \end{aligned}$$

where Ω is any nonterminating program. It is trivial to verify that these terms have the required property.

For the type $\mathtt{var}$, we make use of nondeterminism. We are going to encode the action of reading a value n from a variable as the number $2n$, and writing n to a variable as $2n+1$ (any effective encoding of a disjoint sum of naturals would do, of course). The $\mathtt{in}$ term randomly assigns to or dereferences from the variable x, and then returns the encoding of what it has done:

$$x : \mathtt{var} \vdash \mathtt{in}_{\mathtt{var}} : \mathtt{nat} \triangleq \mathtt{new}\ r := \mathtt{random}\ \mathtt{in}\ \mathtt{ifzero}\ r\ \ \mathtt{then}\ 2(!x) \\ \mathtt{else}\ (x := r - 1); 2r - 1.$$

The semantics of $\mathtt{in}_{\mathtt{var}}$ therefore consists of all pairs of the forms

$$([\mathsf{read}(n)], 2n) \qquad \text{and} \qquad ([\mathsf{write}(n)], 2n + 1).$$

The $\mathtt{out}$ term makes use of $\mathtt{mkvar}$ to create a variable. Both the reading and writing parts of this variable evaluate the natural number y once. If y is of the form $2n$, then the variable allows n to be read from it; if on the other hand y is $2n + 1$, then the variable allows n to be written to it. No other actions are possible.

$$y : \mathtt{nat} \vdash \mathtt{out}_{\mathtt{var}} : \mathtt{var} \triangleq \mathtt{mkvar}\ \ (\lambda n : \mathtt{nat}.\mathtt{if}\ y = 2n + 1\ \mathtt{then}\ \mathtt{skip}\ \mathtt{else}\ \Omega) \\ (\mathtt{new}\ z := y\ \mathtt{in}\ \mathtt{if}\ \mathsf{even}(!z)\ \mathtt{then}\ !z/2\ \mathtt{else}\ \Omega).$$

The semantics of this term therefore consists of all pairs of the forms

$$([2n], \mathsf{read}(n)) \qquad \text{and} \qquad ([2n + 1], \mathsf{write}(n))$$

thus giving the required retraction.

Finally for $\mathtt{nat} \multimap \mathtt{nat}$, the term $\mathtt{in}$ supplies the function with a randomly generated sequence of inputs, s, observes the output, n, and returns an encoding of the pair (s, n) as a natural number. Compare this with the $\mathsf{code}(-)$ function used to embed $[\mathcal{P}\omega \to \mathcal{P}\omega]$ in $\mathcal{P}\omega$ in Scott's model. To ease the notation we use a liberal dose of syntactic sugar. We assume that an encoding of sequences of natural numbers as naturals exists, and suppress mention of it, so it appears that the variable s in the term below is used to store finite sequences directly. We write ε for the encoding of the empty sequence, $[n]$ for the encoding of the singleton sequence containing the element n, and $\cdot$ for the encoding of concatenation. If n is a number encoding a sequence s, $|n|$ denotes the length of sequence s and n_i denoting the ith element of s. We also use pair notation $\langle s, n\rangle$ for the encoding of this pair as a natural number, and $\mathtt{fst}$ and $\mathtt{snd}$ to compute the projections from such encoded pairs. Finally we allow multiple variables to be allocated and initialized at once, so that $\mathtt{new}\ s := \varepsilon; x := 0\ \mathtt{in}\ M$ means $\mathtt{new}\ s\ \mathtt{in}\ \mathtt{new}\ x\ \mathtt{in}\ s := \varepsilon; x := 0; M$. With these abbreviations at our disposal, $\mathtt{in}_{\mathtt{nat}\multimap\mathtt{nat}}$ is defined as follows.

$$f : \mathtt{nat} \multimap \mathtt{nat} \vdash \mathtt{in}_{\mathtt{nat}\multimap\mathtt{nat}} \triangleq \mathtt{new}\ \ s := \varepsilon; x := 0\ \mathtt{in} \\ x := f(\mathtt{new}\ r := \mathtt{random}\ \mathtt{in}\ (s :=!s \cdot [!r]); !r); \\ \langle !s, !x\rangle.$$

Finally for $\mathtt{out}_{\mathtt{nat}\multimap\mathtt{nat}}$, we take the value $y : \mathtt{nat}$, decode it as a pair (s, n), and return a function which can return n on observation of the input sequence s, but can do nothing

else.

$$y : \mathtt{nat} \vdash \mathtt{out}_{\mathtt{nat}\multimap\mathtt{nat}} \triangleq \lambda z^{\mathtt{nat}}.\mathtt{new}\ \begin{array}{l} y' := y; z' := z; s := \mathtt{fst}(!y'); n := \mathtt{snd}(!y'); x := 0\ \mathtt{in} \\ \mathtt{while}\ \ !x < |!s|\ \mathtt{do} \\ \qquad\qquad \mathtt{if}\ !z'_{!x} = !s_{!x}\ \mathtt{then}\ x :=\!!x + 1\ \mathtt{else}\ \Omega; \\ !n \end{array}$$

□

These definable retractions allow us to prove full abstraction for $SCI_{\mathtt{mk},\mathtt{ran}}$ in a very straightforward fashion.

Theorem 7.3. *The model of $SCI_{\mathtt{mk},\mathtt{ran}}$ in* **MonRel** *is fully abstract. That is, for any closed terms M and N of the same type, $[\![M]\!] = [\![N]\!]$ if and only if $M \cong_{\mathtt{mk},\mathtt{ran}} N$.*

Proof. The left-to-right implication is Theorem 6.3. For the right-to-left, suppose M and N are equivalent terms. Then by definition of equivalence, we also have

$$\mathtt{in}[M/x] \cong_{\mathtt{mk},\mathtt{ran}} \mathtt{in}[N/x].$$

These are closed terms of type `nat`, so by Corollary 6.4, $[\![\mathtt{in}[M/x]]\!] = [\![\mathtt{in}[N/x]]\!]$. By compositionality of the semantics it follows that $[\![\mathtt{out}[\mathtt{in}[M/x]/y]]\!] = [\![\mathtt{out}[\mathtt{in}[N/x]/y]]\!]$. But $[\![\mathtt{out}[\mathtt{in}[M/x]/y]]\!] = [\![M]\!];[\![\mathtt{in}]\!];[\![\mathtt{out}]\!]$ and similarly for N, so we conclude that $[\![M]\!] = [\![N]\!]$ as required. □

8. A model without nondeterminism

We have established full abstraction of our model of $SCI_{\mathtt{mk},\mathtt{ran}}$, which admits both the `mkvar` construct and nondeterminism. Before embarking on our proof that these additional constructs do not change the notion of equivalence in SCI, we first develop a more constrained model in which `random` cannot be interpreted.

Reddy's original object-spaces model did not admit the nondeterministic construct `random`. We use some of Reddy's ideas to construct a variant of the category **MonRel** which contains the same model of $SCI_{\mathtt{mk}}$ but, like Reddy's category, contains no nondeterministic elements. The idea is to introduce a relation of coherence, in the style of Girard's coherence spaces [6].

Definition Given a monoid A, a *coherence relation* $\frown$ on A is a symmetric reflexive binary relation on the underlying set of A such that

prefix closure: if $a_1a_2 \frown a'_1a'_2$ then $a_1 \frown a'_1$
extension: if $aa_1 \frown aa_2$ then $a_1 \frown a_2$.

A useful intution is that elements a and a' are coherent, $a \frown a'$, if they can coexist as possible observations to be made of a single deterministic computation at the same state. So, for instance, distinct natural numbers n and n' will not be coherent in the denotation of `nat`, but $\mathsf{write}(n)$ and $\mathsf{write}(n')$ will be coherent in `var` because a variable may allow any value to be written to it.

Definition The category **MonRelCoh** is defined as follows. Objects are pairs $(A, \frown)$ consisting of a monoid A together with a coherence relation on A, and maps from $(A, \frown_A)$ to $(B, \frown_B)$ are relations R such that R is a map from A to B in **MonRel** and furthermore

- if $a \frown_A a'$, aRb and $a'Rb'$ then $b \frown_B b'$

• if $a \frown_A a'$, aRb and $a'Rb$ then $a = a'$.

Composition is the usual composition of relations.

Lemma 8.1. MonRelCoh *is a category.*

Proof. It is clear that the identity relations are valid maps in **MonRelCoh** so we just need to show that composition preserves the two new constraints on maps. Let $R : A \to B$ and $S : B \to C$ be maps in **MonRelCoh**. Suppose $a \frown_A a'$ and that $aR;Sc$ and $a'R;Sc'$. Then there exist $b, b' \in B$ such that aRb, bSc, $a'Rb'$ and $b'Sc'$. Since $a \frown_A a'$ we have $b \frown_B b'$ and hence $c \frown_C c'$ as required. Now suppose $c = c'$; we shall show that $a = a'$. Since S is a valid map, we have $b = b'$ and then since R is valid, $a = a'$. Hence $R;S$ is a valid map in **MonRelCoh**. □

The following definition is due to Reddy [19].

Definition Given a set A and a symmetric reflexive binary relation $\frown_A$ on A, we define an object of **MonRelCoh** called the *object-space over* A consisting of the free monoid over A with coherence relation defined by:

$$a_1 \dots a_m \frown a'_1 \dots a'_n$$

if and only if

$$\forall i \in \{1, \dots \mathsf{min}(m,n) - 1\}. a_1 \dots a_i = a'_1 \dots a'_i \Rightarrow a_{i+1} \frown_A a'_{i+1}.$$

That is to say, two sequences are coherent if either one is a prefix of the other, or at the first place they differ, the two differing elements are coherent.

Lemma 8.2. *Let $(A, \frown)$ be a set with a coherence relation, and let A^* be the object-space over this structure. Let B be any object of* **MonRelCoh**. *Let R be a relation from UB to A such that if bRa and $b'Ra'$ with $b \frown b'$ then $a \frown a'$ and if $a = a'$ then $b = b'$. Then there is a unique map in* **MonRelCoh** *from B to A^* which extends R; by abuse of notation we also write R for this relation.*

Proof. The unique candidate for this map is the extension of R to a map B to A^* in **MonRel**, exploiting the fact that A^* is the free monoid over A. We just need to show that it is a valid map in **MonRelCoh**.

We first show that if $b \frown b'$ with $bRa_1 \cdots a_n$ and $b'Ra'_1 \cdots a'_{n'}$ then $a_1 \cdots a_n \frown a'_1 \cdots a'_{n'}$. This requires demonstrating that at the first i such that $a_i \neq a'_i$, we have $a_i \frown a'_i$, if such an i exists. We proceed by induction on the minimum of n, n'. In the base case there is nothing to prove, so suppose both n and n' are non-zero.

By the decomposition property, we can find $b_1, \dots, b_n$ such that $b = b_1 \cdots b_n$ and each $b_i Ra_i$, and similarly for b' and the a'_i. By the prefix-closure property in B, $b_1 \frown b'_1$ and hence $a_1 \frown a'_1$. Thus if $a_1 \neq a'_1$, we are done. Otherwise, $a_1 = a'_1$ implies that $b_1 = b'_1$ and then by the extension property of coherence in B, we have $b_2 \cdots b_n \frown b'_2 \cdots b'_{n'}$ and of course $b_2 \cdots b_n Ra_2 \cdots a_n$ and similarly for the b'_i and a'_i. Then the inductive hypothesis gives us the result we require.

We now show that if additionally $a_1 \cdots a_n = a'_1 \cdots a'_{n'}$ then $b = b'$, again by induction on n (which is equal to n'). The base case is guaranteed by the identity reflection property of maps in **MonRel**. In the inductive step, we again decompose b and b' as above, and note that since $a_1 = a'_1$ we have $b_1 = b'_1$. Then we also have $b_2 \cdots b_n Ra_2 \cdots a_n$ and similarly for the b'_i, and conclude by the inductive hypothesis. □

The product, tensor and exponential constructions in **MonRel** all lift to **MonRelCoh**. This can be expressed as follows.

Lemma 8.3. MonRelCoh *is a symmetric monoidal category with products, and the object-spaces form an exponential ideal in* **MonRelCoh**. *Moreover the forgetful functor to* **MonRel** *preserves all this structure on the nose.*

Proof. We just need to define the coherence-relation parts of the various constructions and show that they are well-defined and have the appropriate properties.

For the monoidal structure, coherence is defined pointwise:

$$(a,b) \frown_{A\otimes B} (a',b') \iff a \frown_A a', b \frown_B b'.$$

(To aid legibility in future we will drop the subscripts on the $\frown$ relations where no confusion will arise.)

It is clear that this definition makes $\otimes$ into a bifunctor on **MonRelCoh** and that the associativity, symmetry and unit maps from **MonRel** are well-defined maps in **MonRelCoh** too.

We now consider the exponentials. Let $(A, \frown_A)$ be an object of **MonRelCoh**, and let $(B, \frown_B)$ be a set equipped with a symmetric reflexive binary relation. In **MonRel** the exponential $A \multimap B^*$ is given by the free monoid over $UA \times B$. We shall define a symmetric reflexive binary relation on this set and show that the object-space this defines is the required exponential in **MonRelCoh**.

The coherence relation on $UA \times B$ echoes the definition of map in **MonRelCoh**: $(a,b) \frown (a',b')$ if and only if

- $a \frown_A a' \Rightarrow b \frown_B b'$
- $a \frown_A a' \wedge b = b' \Rightarrow a = a'$.

By Lemma 8.2, maps from an object C into this object space are described by relations from UC to $UA \times B$ which satisfy the appropriate coherence constraints. That is, if $cR(a,b)$ and $c'R(a',b')$ then we have

- $c \frown_C c' \implies (a,b) \frown (a',b')$
- $c \frown_C c' \wedge (a,b) = (a',b') \implies c = c'$.

On the other hand, maps from $C \otimes A$ to B^* are given by relations from $UC \times UA$ to B such that

- $c \frown_C c' \wedge a \frown_A a' \implies b \frown_B b'$
- $c \frown_C c' \wedge a \frown_A a' \wedge b = b' \implies a = a' \wedge c = c'$.

It is straightforward to verify that these are the same constraints, so that we have a natural bijection of homsets:

$$\mathbf{MonRelCoh}(C \otimes A, B^*) \cong \mathbf{MonRelCoh}(C, A \multimap B),$$

as required.

A similar argument shows that products in **MonRel** lift to **MonRelCoh**. For object-spaces, the construction is very straightforward: the product of object-spaces A^* and B^* is the object space over the disjoint union $A + B$, equipped with the coherence relation which relates elements of A if and only if they are related in the object space A^*, and similarly for B, but also relates all elements of A to all elements of B. $\square$

MonRelCoh therefore possesses all the structure we require to model SCI. To lift our model to **MonRelCoh** we just need to give interpretations of the base types and constants. The base types are all interpreted using object spaces, with underlying coherence relations as follows:

- for $\mathtt{nat}$, $n \frown n' \iff n = n'$.
- for $\mathtt{comm}$, $* \frown *$.
- for $\mathtt{var}$, $\mathsf{write}(n) \frown \mathsf{write}(n')$ for all n, n'; $\mathsf{read}(n) \frown \mathsf{read}(n') \iff n = n'$; and $\mathsf{write}(n) \frown \mathsf{read}(n')$ for all n, n'. Note that this makes $\mathtt{var}$ the product object-space of $\mathtt{nat}$ with ω-many copies of $\mathtt{comm}$.

It is easy to check that the constant maps used in the denotations of SCI terms are maps of **MonRelCoh** over the appropriate types. The same applies to $\mathtt{mkvar}$, but not to $\mathtt{random}$: the map $[\![\mathtt{random}]\!]$ clearly violates the coherence constraints since it returns incoherent outputs from coherent (empty) inputs.

Theorem 8.4. *The model of $SCI_{\mathtt{mk}}$ in* **MonRel** *lifts to* **MonRelCoh**. □

Corollary 8.5. *If $\vdash M : A$ is a closed term of $SCI_{\mathtt{mk}}$ and $a, a' \in [\![M]\!]$ then $a \frown a'$. (Here we blur the distinction between maps from the tensor unit into $[\![A]\!]$ and subsets of $[\![A]\!]$.)* □

Thus the model of $SCI_{\mathtt{mk}}$ in **MonRelCoh** captures $SCI_{\mathtt{mk}}$'s deterministic nature: for instance, closed terms of type $\mathtt{nat}$ contain at most one natural number in their denotation.

9. Conservativity results

In this section we show that the extensions of SCI with the $\mathtt{mkvar}$ and $\mathtt{random}$ operators are *conservative*, that is to say, they have no effect on the relation of contextual equivalence for terms of the original SCI language. This means that the new contexts available when the language is extended have no additional discriminating power, and as a result, the full abstraction theorem for $SCI_{\mathtt{mk,ran}}$ also applies to the smaller languages $SCI_{\mathtt{mk}}$ and SCI. As explained in [13], this work shows that Reddy's object-spaces model [19] was the first example of a fully abstract semantics for a higher-order imperative language, though this was not known at the time. Its full abstraction is 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.

Though we present our results in the form of conservativity theorems rather than direct full abstraction proofs, our arguments hinge on partial definability results which would be enough to establish full abstraction of the model for SCI and $SCI_{\mathtt{mk}}$ directly, that is, without appealing to Theorem 7.3, if desired. The proof of conservativity of $\mathtt{mkvar}$ in particular makes heavy use of our definability results, and is essentially the same as the direct proof of full abstraction given in [13]. Nevertheless we believe that presenting the results as conservativity theorems is worthwhile, particularly in light of the relatively cheap proof of full abstraction for $SCI_{\mathtt{mk,ran}}$, and the limited use of definability in the proof of conservativity of $\mathtt{random}$.

9.1. **Definability.** As explained above, our conservativity results are established by means of a partial definability result which demonstrates how certain elements of our model are found as the denotations of terms from SCI and its extensions.

Let us first mention a curious fact. Let $C[-]$ be some context of SCI, so that in particular $C[-]$ does not employ `mkvar`. If

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

then it is also the case that $C[x := 3]\Downarrow$. This inability of `mkvar`-free contexts to distinguish completely between reading and writing into variables is the main obstacle to overcome in our definability proof. The presence of `mkvar` makes quite a difference, since for example a context binding `x` to the term

$$\texttt{mkvar}\ (\lambda y.\texttt{diverge}) \quad (3)$$

will make the first term above converge and the second diverge. This immediately tells us that the addition of `mkvar` is not conservative with respect to the contextual *preorder*. Our work in this section will show that it is conservative with respect to contextual *equivalence*; this came as a surprise.

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

Definition For any SCI type A, we define the *positive and negative read-write orders* $\preceq^+$ and $\preceq^-$ between elements of $[\![A]\!]$ 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 `nat`:
$$n \preceq^+ m \iff n = m \iff n \preceq^- m$$
- At type `var`:
$$\begin{array}{lllll} a & \preceq^+ & a' & \iff & (a = a') \vee \exists n. a = \mathsf{read}(n) \wedge a' = \mathsf{write}(n) \\ a & \preceq^- & a' & \iff & a = a' \end{array}$$
- At type $A \multimap B$:
$$\begin{array}{lllll} (s,b) & \preceq^+ & (s',b') & \iff & s \preceq^- s' \wedge b \preceq^+ b' \\ (s,b) & \preceq^- & (s',b') & \iff & s \preceq^+ s' \wedge b \preceq^- b' \end{array}$$

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

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

Lemma 9.1. *Let A be any type of SCI and let $a \in [\![A]\!]$ be any element of the monoid interpreting A. There exists a term*

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

of SCI (not including `mkvar` *or* `random`*) such that $(s, *) \in [\![\mathsf{test}(a)]\!]$ iff $a \preceq^- s$. There also exists a context $\Gamma = x_1 : \texttt{var}, \ldots, x_n : \texttt{var}$, Γ-stores $\mathsf{init}(a)$ and $\mathsf{final}(a)$, and a term*

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

such that for all $a' \in [\![A]\!]$,

$$(\exists s.(s, a') \in [\![\mathsf{produce}(a)]\!] \wedge \mathsf{init}(a) \xrightarrow{s} \mathsf{final}(a)) \iff a \preceq^{+} a'.$$

Proof. We will prove the two parts of this lemma simultaneously by induction on the type A. First note that any $a \in [\![A]\!]$ 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: $\mathsf{test}([]) = \mathtt{skip}$ and $\mathsf{produce}([])$ is any divergent term, with $\mathsf{init}([])$ and $\mathsf{final}([])$ both being the unique store on no variables.

If $a = [a_1, a_2, \ldots, a_n]$, then we can define $\mathsf{test}(a)$ as

$$\mathsf{test}([a_1])\,;\mathsf{test}([a_2])\,;\ldots;\mathsf{test}([a_n]).$$

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

$$\begin{array}{l}
\mathtt{if}\ (!x_1 \neq \sigma(x_1))\ \mathtt{then}\ \mathtt{diverge} \\
\mathtt{else\ if}\ (!x_2 \neq \sigma(x_2))\ \mathtt{then}\ \mathtt{diverge} \\
\ldots \\
\mathtt{else\ if}\ (!x_n \neq \sigma(x_n))\ \mathtt{then}\ \mathtt{diverge} \\
\mathtt{else\ skip}
\end{array}$$

Define $\mathsf{set}(\sigma)$ to be $x_1 := \sigma(x_1)\,;\cdots;\,x_n := \sigma(x_n)$.

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

$$\begin{array}{ll}
\Gamma, x : \mathtt{var} \vdash \lambda \vec{y_i}^{\vec{A_i}}. & x :=\!!x + 1; \\
& \mathtt{if}\ (!x = 1)\ \mathtt{then}\ \mathsf{produce}(a_1) y_1 \ldots y_k \\
& \mathtt{else\ if}\ (!x = 2)\ \mathtt{then}\ \mathsf{check}(\mathsf{final}(a_1)); \\
& \qquad\qquad\qquad\qquad \mathsf{set}(\mathsf{init}(a_2)); \\
& \qquad\qquad\qquad\qquad \mathsf{produce}(a_2) y_1 \ldots y_k \\
& \ldots \\
& \mathtt{else\ if}\ (!x = n)\ \mathtt{then}\ \mathsf{check}(\mathsf{final}(a_{n-1})); \\
& \qquad\qquad\qquad\qquad \mathsf{set}(\mathsf{init}(a_n)); \\
& \qquad\qquad\qquad\qquad \mathsf{produce}(a_n) y_1 \ldots y_k \\
& \mathtt{else\ diverge}
\end{array}$$

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

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

For the type $\mathtt{comm}$, we define

$$\begin{array}{rcl}
\mathsf{test}(*) & = & x : \mathtt{comm} \vdash x : \mathtt{comm} \\
\mathsf{produce}(*) & = & y : \mathtt{var} \vdash y :=\!!y + 1 : \mathtt{comm} \\
\mathsf{init}(*) & = & (y \mapsto 0) \\
\mathsf{final}(*) & = & (y \mapsto 1)
\end{array}$$

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

The type $\mathtt{nat}$ is handled similarly:

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

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

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

For $\mathsf{produce}(\mathsf{write}(n))$, 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:

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

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

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

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

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

The non-interference between function and argument allows us to define these terms very simply: for $\mathsf{test}(s, b)$ 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 $\mathsf{init}(s)$ and $\mathsf{final}(s)$ states. For $\mathsf{produce}(s, b)$ 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. □

9.2. **Conservativity of `random`.**

Lemma 9.2 (`random` is conservative)**.** *Let $\Gamma \vdash M, N : A$ be terms of $SCI_{\mathtt{mk}}$ such that $M \cong_{\mathtt{mk}} N$. Then $M \cong_{\mathtt{mk,ran}} N$.*

Proof. It suffices to consider *closed* terms, because in all the language fragments we consider, open terms M and N are equivalent if and only if their closures $\lambda\vec{x}.M$ and $\lambda\vec{x}.N$ are equivalent.

So, let $\vdash M, N : A$, suppose $M \cong_{\mathtt{mk}} N$ and let $C[-]$ be a context, possibly employing `random`, such that $C[M] \Downarrow \mathtt{skip}$. We shall show that $C[N] \Downarrow \mathtt{skip}$ by induction on the number of occurrences of `random` in $C[-]$.

The base case, where $C[-]$ does not employ `random` at all, is trivial: $C[-]$ is a $SCI_{\mathtt{mk}}$ context, so since $M \cong_{\mathtt{mk}} N$, we have $C[N] \Downarrow \mathtt{skip}$.

For the inductive step, let $C'[-]$ be the context obtained from $C[-]$ by replacing one occurrence of `random` with a fresh variable r of type `nat`. Then for any term P, $C[P] \Downarrow \mathtt{skip}$ if and only if $(\lambda r.C'[P])(\mathtt{random}) \Downarrow \mathtt{skip}$.

Since $(\lambda r.C'[M])(\mathtt{random}) \Downarrow \mathtt{skip}$, Corollary 6.2 implies that

$$(\varepsilon, *) \in [\![(\lambda r.C'[M])(\mathtt{random})]\!].$$

By definition of $[\![\mathtt{random}]\!]$ and the semantics of application, there must exist a sequence s of natural numbers such that $(s, *) \in [\![\lambda r.C'[M]]\!]$.

By Lemma 9.1, there is a term

$$x : \mathtt{nat} \to \mathtt{comm} \vdash \mathsf{test} : \mathtt{comm}$$

not involving `random`, such that $(t, *) \in [\![\mathsf{test}]\!]$ iff $t = (s, *)$.

We therefore have $(\varepsilon, *) \in [\![(\lambda x.\mathsf{test})(\lambda r.C'[M])]\!]$ and hence by Corollary 6.2, $(\lambda x.\mathsf{test})(\lambda r.C'[M]) \Downarrow \mathtt{skip}$. But $(\lambda x.\mathsf{test})(\lambda r.C'[-])$ is a context involving the same number of occurrences of `random` as does $C'[-]$, so by inductive hypothesis we also have $(\lambda x.\mathsf{test})(\lambda r.C'[N]) \Downarrow \mathtt{skip}$. Therefore $(\varepsilon, *) \in [\![(\lambda x.\mathsf{test})(\lambda r.C'[N])]\!]$, which is only possible if $(s, *) \in [\![\lambda r.C'[N]]\!]$. But then

$$(\varepsilon, *) \in [\![(\lambda r.C'[N])(\mathtt{random})]\!]$$

and hence by Corollary 6.2 again, $(\lambda r.C'[N])(\mathtt{random}) \Downarrow \mathtt{skip}$. Finally we can conclude that $C[N] \Downarrow \mathtt{skip}$ as required. □

Corollary 9.3. *The model of $SCI_{\mathtt{mk}}$ in* **MonRel** *is fully abstract.* □

9.3. **Conservativity of `mkvar`.**

Lemma 9.4. *Let A^* be an object-space interpreting a type of SCI in* **MonRelCoh** *and let $a, a' \in A^*$.*

- *If $a \preceq^- a'$ and $a \frown a'$ then $a = a'$.*
- *If $a \preceq^+ a'$ then $a \frown a'$.*

Proof. By induction on type. We consider only the cases of singleton sequences; the general cases follow easily.

For `comm` and `nat`, both $\preceq^-$ and $\preceq^+$ are the identity relations, so the results hold trivially. For `var`, $\preceq^-$ is again the identity relation completing this case. For $\preceq^+$, the result follows from the fact that $\mathsf{read}(n) \frown \mathsf{write}(n)$.

For the inductive step, consider elements (s,b) and (s',b') of $A \multimap B$. If $(s,b) \preceq^- (s',b')$ then $s \preceq^+ s'$ and $b \preceq^- b'$. By the inductive hypothesis on type A, $s \frown s'$ so if $(s,b) \frown (s',b')$ then we also have $b \frown b'$. The inductive hypothesis on B then gives us $b = b'$ and hence $s = s'$ as required. If $(s,b) \preceq^+ (s',b')$ then $s \preceq^- s'$ and $b \preceq^+ b'$. Then if $s \frown s'$, the inductive hypothesis gives us $s = s'$. Induction also tells us that $b \frown b'$, and hence $(s,b) \frown (s',b')$ as required. □

Lemma 9.5 (`mkvar` is conservative)**.** *Let $\Gamma \vdash M, N : A$ be terms of SCI such that $M \cong N$. Then $M \cong_{\mathtt{mk}} N$.*

Proof. As in Lemma 9.2 we consider only closed terms. Suppose $\vdash M, N : A$ with $M \cong N$ and let $(\varepsilon, a) \in [\![M]\!]$ be any element of the denotation of M. By Lemma 9.1 there is a term $x : A \vdash \mathsf{test}(a) : \mathtt{comm}$ such that $(a', *) \in [\![\mathsf{test}(a)]\!]$ if and only if $a \preceq^- a'$. We therefore have $(\varepsilon, *) \in [\![(\lambda x.\mathsf{test}(a))M]\!]$, and hence $(\lambda x.\mathsf{test}(a))M \Downarrow \mathtt{skip}$ by Corollary 4.2. By hypothesis we have $(\lambda x.\mathsf{test}(a))N \Downarrow \mathtt{skip}$, so that $(\varepsilon, *) \in [\![(\lambda x.\mathsf{test}(a))N]\!]$. Therefore there is some a' such that $a \preceq^- a'$ and $(\varepsilon, a') \in [\![N]\!]$. Symmetrically we can find a'' such that $a' \preceq^- a''$ and $(\varepsilon, a'') \in [\![M]\!]$.

By Corollary 8.5, $a \frown a''$ and then by Lemma 9.4, $a = a''$ and hence $a = a'$. It follows that $[\![M]\!] = [\![N]\!]$ and hence $M \cong_{\mathtt{mk}} N$ by Theorem 6.3. □

Corollary 9.6. *The model of SCI in* **MonRel** *is fully abstract.* □

We remark that Reddy was not aware that his model was fully abstract; indeed it was believed not to be.

10. Conclusions

We have shown that a simple amendment of Scott's $\mathcal{P}\omega$ graph-model gives rise to a model of imperative computation, in the event-based style of Reddy's object-spaces model and later models based on game semantics. Moreover we have shown that this model contains a universal type, thus yielding a very cheap proof of full abstraction for the language $SCI_{\mathtt{mk},\mathtt{ran}}$. With some additional work we have established full abstraction for the original SCI language via conservativity results; this was not known prior to our work.

We believe that the general approach of constructing models in this way is of interest and has the potential to give rise to a range of interesting concrete models and some useful insights at a more abstract level. We intend to develop an axiomatic presentation of our constructions, expanding on the work of Hyland et al. [7]. At present it is not clear whether the more refined game-based models can be presented in this style; this remains a topic for further investigation.

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 P. W. O'Hearn and R. D. Tennent, editors, *Algol-like Languages*, pages 297–329 of volume 2. Birkhaüser, 1997.

[3] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
[5] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc., January 2008.
[6] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
[7] M. Hyland, M. Nagayama, J. Power, and G. Rosolini. A category-theoretic formulation of engeler-style models of the untyped λ-calculus. In *Proc. MCFSIT 2004, Electronic Notes in Theoretical Computer Science volume 161*, pages 43–57, 2006.
[8] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.
[9] J. Laird. Decidability in syntactic control of interference. *Theoretical Computer Science*, 394:64–83, 2008.
[10] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.
[11] J. Longley. Universal types and what they are good for. In *Domain theory, logic and computation: Proceedings of the 2nd International Symposium on Domain Theory*, number 3 in Semantic Structures in Computation, pages 25–63. Kluwer, 2003.
[12] J. Longley. Interpreting localized computational effects using operators of higher type, extended abstract. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008, Athens, Proceedings*, number 5028 in Lecture Notes in Computer Science. Springer Verlag, 2008.
[13] G. McCusker. A fully abstract relational model of syntactic control of interference. In *Proceedings, Computer Science Logic (CSL) 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 247–261. Springer-Verlag, 2002.
[14] G. McCusker. On the semantics of the bad variable constructor in Algol-like languages. In S. Brookes and P. Panangaden, editors, *Proceedings, Nineteenth Conference on the Mathematical Foundations of Programming Semantics, Montreal 2003*, Electronic Notes in Theoretical Computer Science. Elsevier, 2003.
[15] 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.
[16] P. W. O'Hearn and U. Reddy. Objects, interference and the Yoneda embedding. In M. Main and S. Brookes, editors, *Mathematical Foundations of Programming Semantics: Proceedings of 11th International Conference*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B.V., 1995.
[17] P. W. O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3(4):435–465, 1993.
[18] G. Plotkin. T^ω as a universal domain. *J. Computer and System Sciences*, 17:209–236, 1978.
[19] U. S. Reddy. Global state considered unnecessary: Object-based semantics for interference-free imperative programs. *Lisp and Symbolic Computation*, 9(1), 1996.
[20] J. C. Reynolds. Syntactic control of interference. In *Conf. Record 5th ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.
[21] J. C. Reynolds. Syntactic control of inference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium, ICALP 89, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer, 1989.
[22] D. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.
[23] M. Wall. *Games for Syntactic Control of Interference*. PhD thesis, University of Sussex, 2005.