1 Relational semantics: A warm-up exercise

In this section, we develop a model of interference-free Algol in the relational framework. That is, we interpret

- types as sets, and
- programs as relations

Such a model is admittedly not abstract enough. The directional information of functions is not represented in relations. However, it is conceptually helpful to suppress the directionality information for a “warm-up exercise” because the modelling of state involves intricate directionality issues. It will be seen that all the problematic equivalences that have appeared in the literature can already be validated in the relational model. Moreover, the relational framework is quite expressive: the category of sets and relations forms a $*$-autonomous category and, as shown in [Bar91], can be extended to a model of linear logic (which includes typed lambda calculus).

The framework of sets and relations is precisely the semantic framework of logic programming. Thus, our relational model gives a very good “operational feel” as logic programs. However, it should be mentioned that this is idealized logic programming without commitment to sequential control. The effects that we require can be achieved in concurrent logic programming languages such as Concurrent Prolog, Parlog or Janus [Gre87, SKL90, Sha83]. In fact, our technique for modelling states is very close to the traditional techniques used in these languages. On the other hand, these effects cannot be achieved in sequential Prolog.

A relation is defined between two sets as follows:

\[ R : A \leftrightarrow B \]
\[ R(x, y) \leftrightarrow \psi \]

where $\psi$ is a formula built from relation applications, equality, conjunction, disjunction and existential quantification. For readability, we often suppress existential quantifiers and write definitions as

\[ R(x, y) \leftarrow \psi \]

Such a “clause” means $R(x, y) \leftrightarrow \exists \bar{z}. \psi$ where $\bar{z} = V(\psi) \setminus \{x, y\}$. We use the logic programming conventions of writing conjunction as “;” and disjunction as “;”. Types include
• primitive types such as \textit{int} and \textit{bool},
• the singleton type \textit{unit}, whose element is written as “()”, and
• finite products and disjoint unions.

(The categorically inclined may note that product and \textit{unit} give a monoidal structure to the category and that disjoint union is the categorical product as well as the coproduct). We often write disjoint unions with \textit{constructor} symbols for injections, \textit{e.g.},

\[\text{type result} = \text{fail} \mid \text{succ(int)}\]

which is syntactic sugar for \textit{unit + int}. We also allow polymorphic type constructions with recursive definitions, \textit{e.g.},

\[\text{type list(α) = nil} \mid α :: \text{list(α)}\]

which denotes the least set \(L\) such that \(L = \text{unit + } α \times L\). All these constructions have well-defined semantics. See, for example, [LR91]. Recursion in relation definitions is interpreted by the least fixed point in \(A \leftrightarrow B\) (viewed as \(P(A \times B)\)).

In the following, we deliberately blur the distinction between syntax and semantics and use synonymously terms like “value” and “term” and “relation” and “predicate”. The point is that typed logic programs of the kind we deal with have standard semantics. If there is any confusion, the reader may eventually refer to the coherent semantics (Section 3) for a purely semantic account.

1.1 Modelling functions

First, we define a type constructor \textit{closure} such that \textit{closure(A)} denotes copyable closures for values of type \(A\). We take an abstract point of view and model closures as trees of \textit{messages} sent to some representation of the closure itself:

\[\text{type closure(α) = discard} \mid \text{use(α)} \mid \text{closure(α) @ closure(α)}\]

The \textit{discard} message sent to a closure discards it, the message \textit{use(x)} uses the closure once (with its contents obtained via \(x\)), and the message \textit{ms1@ms2} makes two copies of the closure sending the subtrees \(ms1\) and \(ms2\) to the copies respectively.

Secondly, we define a predicate scheme

\[\text{mkclosure}[^\ell : \tilde{A}; \psi; u : C]\]

where \(\tilde{t} : \tilde{A}\) is a sequence of terms such that each type \(A\) in \(\tilde{A}\) is a \textit{closure} type, \(\psi\) is a formula and \(u : C\) is a term. All of these use some type context \(Γ\). Further, the free variables of the parameters are such that \(V(\psi) \subseteq V(\tilde{t}) \cup V(u)\). The intuition behind these parameters is that, given inputs \(\tilde{t} : \tilde{A}\), a \(C\)-typed value can be produced by executing the goal formula \(\psi\) and then returning \(u\). Given such a computation, \textit{mkclosure} expresses a computation that produces a \textit{closure(C)}-typed result from the same set of inputs. (Notice, however, that the above explanation in terms of inputs and outputs is only to support one’s intuition. Logic programs themselves define relations which have no notion of inputs and outputs). \textit{mkclosure}[^\ell : \tilde{A}; \psi; u : C] is a predicate \(P\) defined recursively as follows:

\begin{align*}
P : A_1 \times \ldots \times A_k & \leftrightarrow \text{closure}(C) \\
P(\tilde{z}, \text{discard}) & \leftarrow \tilde{z} = \text{discard} \\
P(\tilde{z}, \text{use(val)}) & \leftarrow \tilde{z} = \tilde{t}, \text{val} = u, \psi \\
P(\tilde{z}, ms1 @ ms2) & \leftarrow \tilde{z} = \tilde{x} @ \tilde{y}, P(\tilde{x}, ms1), P(\tilde{y}, ms2)
\end{align*}
\[ mkinc : \text{unit} \leftrightarrow \text{closure}(\text{int} \times \text{int}) \]

\[ \text{mkinc}((\text{discard})) \leftarrow \]
\[ \text{mkinc}([f]) \leftarrow f = (x, y), \text{add}(x, 1, y) \]
\[ \text{mkinc}(\text{ms1} \circ \text{ms2}) \leftarrow \text{mkinc}(\text{ms1}), \text{mkinc}(\text{ms2}) \]

\[ mkcomp : \text{unit} \leftrightarrow \text{closure}((\text{closure}(\alpha \times \beta) \times (\text{closure}(\beta \times \gamma) \times (\alpha \times \gamma)))) \]

\[ \text{mkcomp}((\text{discard})) \leftarrow \]
\[ \text{mkcomp}([F]) \leftarrow F = (f, (g, (x, z))), g = [(x, y)], f = [(y, z)] \]
\[ \text{mkcomp}(\text{ms1} \circ \text{ms2}) \leftarrow \text{mkcomp}(\text{ms1}), \text{mkcomp}(\text{ms2}) \]

---

Figure 1: Examples of closures

Here, \( \bar{z} = \text{discard} \), \( \bar{z} = \bar{t} \) and \( \bar{z} = \bar{x} \circ \bar{y} \) mean their obvious component-wise expansions.

**Notation:** We often abbreviate \( \text{use}(t) \) to \([t]\).

The closure type, together with product types, is adequate to define function values. For example, the functions

\[ \text{inc} = \lambda x. x + 1 \]
\[ \text{comp} = \lambda f. \lambda g. \lambda x. f(gx) \]

are expressed by the closures:

\[ \text{mkinc} \equiv \text{mkclosure}[[; \text{add}(x, 1, y); (x, y)]] \]
\[ \text{mkcomp} \equiv \text{mkclosure}[[; \exists y. g = \text{use}(x, y), f = \text{use}(y, z); (f, (g, (x, z)))]]] \]

These schemes are shown in their expanded form in Figure 1. Generally, a function of type \( A \rightarrow B \) is modelled by a closure of type \( \text{closure}(\text{closure}(A) \times B) \). However, when it is clear that the function is used exactly once, we can suppress the outermost closure construction.

An expression of the form \( \text{comp inc inc} \) can now be simulated by using the closure \( \text{comp} \) once and the closure \( \text{inc} \) twice and performing the necessary function applications. This is done in the formula (viewed as a relation between () and \( h \)):

\[ \exists \text{comp, inc, F, f, g. mkcomp(comp), mkinc(inc), comp = [F], inc = [f]@[g], F = (f, (g, h))] \]

This is equivalent to

\[ \exists y_1, x_2. \text{add}(x_1, 1, y_1), \text{add}(x_2, 1, y_2), x_2 = y_1, h = (x_1, y_2) \]

Thus, \( h \) is the relation such that \( h(x, y) \iff y = x + 2 \). Adding an atom such as \( h = (1, y) \) to the above formula (and viewing it as a relation between () and \( y \)) simplifies it to \( y = 3 \).

We can, in fact, define another closure that represents the twice composition of \( \text{inc} \):

\[ \text{mkinc}^2 \equiv \text{mkclosure}[[\text{comp, inc}; \exists f. \text{inc} = \text{use}(f) @ \text{use}(g), \text{comp} = \text{use}(f, (g, h)); h]] \]

This is shown in expanded form in Figure 2. Each time a use message is sent to this closure, it, in turn, sends a use message to \( \text{comp} \) and two use messages to \( \text{inc} \).

Why does this kind of translation work? A function, in the set-theoretic sense, is nothing but a set of pairs (the function graph). In practice, one can do with multisets of pairs instead. The closure type gives a way to form such multisets. This means the values of a closure type satisfy the following equivalences of a **commutative monoid**:
\( mkinc2 : closure(\text{comptype}) \times closure(int \times int) \leftrightarrow closure(int \times int) \)

where \( \text{comptype} = closure(closure(int \times int) \times (closure(int \times int) \times (int \times int))) \)

\[
\begin{align*}
\text{mkinc2}(\text{comp}, \text{inc}, \text{discard}) & \leftarrow \\
\text{comp} = \text{discard}, \text{inc} = \text{discard} \\
\text{mkinc2}(\text{comp}, \text{inc}, [h]) & \leftarrow \\
\text{inc} = \lfloor f \rfloor @ \lfloor g \rfloor, \text{comp} = \lfloor (f, (g, h)) \rfloor \\
\text{mkinc2}(\text{comp}, \text{inc}, \text{ms1} @ \text{ms2}) & \leftarrow \\
\text{comp} = c1 @ c2, \text{inc} = i1 @ i2, \text{mkinc2}(c1, i1, \text{ms1}), \text{mkinc2}(c2, i2, \text{ms2})
\end{align*}
\]

Figure 2: Closure for \( inc2 \) with free variables

\[
\begin{align*}
\text{ms} @ \text{discard} & \equiv \text{ms} \\
\text{discard} @ \text{ms} & \equiv \text{ms} \\
\text{ms1} @ (\text{ms2} @ \text{ms3}) & \equiv (\text{ms1} @ \text{ms2}) @ \text{ms3} \\
\text{ms1} @ \text{ms2} & \equiv \text{ms2} @ \text{ms1}
\end{align*}
\]

It is easy to verify that the \( \text{mkclosure} \) scheme preserves these equivalences.

Lemma 1 If \( P : A_1 \times \ldots \times A_k \leftrightarrow closure(C) \) is an instance of the \( \text{mkclosure} \) scheme, then for all \( \bar{x}_1, \bar{x}_2 : A \) and for all \( m_1, m_2 : closure(C) \), \( \bar{x}_1 \equiv \bar{x}_2 \) and \( m_1 \equiv m_2 \) implies \( P(\bar{x}_1, m_1) \Leftrightarrow P(\bar{x}_2, m_2) \).

The equivalences of commutative monoid, however, do not entirely cut down closures to functions. For instance, \( [(1, 1)] @ [(1, 2)] \) is a valid closure, but it cannot arise in the translation of any lambda term. (A function cannot map 1 to both 1 and 2). Capturing this feature of functions requires a notion of \textit{directionality} and \textit{coherence}. We will address this issue in Section 3.

1.2 Modelling state

The type \( \text{state}(\delta) \) denotes states containing values of type \( \delta \). Again, we take an abstract view and model states by the messages accepted by them. Since states cannot be copied, the messages to a state form a \textit{stream} rather than a tree (as in the case of closures). So, the definition of state types is as follows:

\[
\text{type state}(\delta) = \text{done} \mid \text{read}(\delta \times \text{state}(\delta)) \mid \text{write}(\delta \times \text{state}(\delta))
\]

The type variable \( \delta \) ranges over primitive types such as \( \text{int} \), \( \text{bool} \) etc. We limit the use of the \text{state} type constructor to such primitive types. This buys us some simplicity. Moreover, since Algol variables can only hold primitive typed values, it is adequate for modelling Algol.

States are created using the following predicate:

\[
\begin{align*}
\text{mkstate} : \text{unit} & \leftrightarrow \text{state}(\delta) \\
\text{mkstate}(\text{ms}) & \leftarrow \text{mkstate'}(\text{INIT}_\delta, \text{ms}) \\
\text{mkstate'}(\text{val}, \text{done}) & \leftarrow \\
\text{mkstate'}(\text{val}, \text{read}(x, \text{ms})) & \leftarrow x = \text{val}, \text{mkstate'}(\text{val}, \text{ms}) \\
\text{mkstate'}(\text{val}, \text{write}(y, \text{ms})) & \leftarrow \text{mkstate'}(\text{y}, \text{ms})
\end{align*}
\]

4
We assume that every primitive type \( \delta \) has a special constant \( \text{INIT}_\delta \) whose value is undetermined. Simple imperative programs can be modelled using this primitive. For example, the effect of the Algol block

\[
\text{new } i. \ (i := 0; i := i + 1)
\]

is achieved by the formula (viewed as a relation \( \text{unit} \leftrightarrow \text{unit} \))

\[
\exists x, y. \ \text{mkstate}(\text{write}(0, \text{read}(x, \text{write}(y, \text{done}))), \text{add}(x, 1, y))
\]

More generally, we need a way to thread a state through multiple commands. This is made possible by the append operation on state streams:

\[
\text{append} : \text{state}(\delta) \times \text{state}(\delta) \leftrightarrow \text{state}(\delta)
\]

\[
\text{append}(\text{done}, y, z) \leftarrow z = y
\]

\[
\text{append}(\text{read}(x, x), y, z) \leftarrow z = \text{read}(x, z'), \text{append}(x, y, z')
\]

\[
\text{append}(\text{write}(x, x), y, z) \leftarrow z = \text{write}(x, z'), \text{append}(x, y, z')
\]

The body of the above Algol block can now be modelled as follows:

\[
\text{mkstate}(i), \ \text{append}(i_1, i_2, i), \ i_1 = \text{write}(0, \text{done}), \ i_2 = \text{read}(x, \text{write}(y, \text{done})), \ \text{add}(x, 1, y)
\]

The streams \( i_1 \) and \( i_2 \) constitute the messages sent to the state variable by the two commands respectively. The sequencing between the two commands is represented by the appending of the two state streams.

### 1.3 State-dependent objects

To be able to define functions that takes states as inputs, we need a general mechanism to form state-dependent objects. We again take an abstract view and models such objects by the streams of messages they support:

\[
\text{type object}(\alpha) = \text{done} \mid \text{use}(\alpha) \mid \text{object}(\alpha) \wedge \text{object}(\alpha)
\]

(In [Red93], the “\( \wedge \)” operator was written as “\( \wedge\wedge \)”. The “\( \wedge \)” symbol of the earlier paper is not used here). As for closures, we often abbreviate \( \text{use}(t) \) to \( [t] \). On the surface, messages of objects look similar to those of closures. However, their interpretation is subtly different. The \( \text{done} \) message sent to an object releases it for future messages, and the message \( \text{ms}_1 \wedge \text{ms}_2 \) causes the message stream \( \text{ms}_1 \) to be sent to the object first and, when it is exhausted, causes \( \text{ms}_2 \) to be sent to the object. Note that \( \text{ms}_1 \wedge \text{ms}_2 \) does not involve any copying. Since objects, in general, have internal states, they cannot be copied.

Objects satisfy the equations of a monoid (without commutativity):

\[
\text{ms} \wedge \text{done} \equiv \text{ms}
\]

\[
\text{done} \wedge \text{ms} \equiv \text{ms}
\]

\[
\text{ms}_1 \wedge (\text{ms}_2 \wedge \text{ms}_3) \equiv (\text{ms}_1 \wedge \text{ms}_2) \wedge \text{ms}_3
\]

The absence of commutativity is an essential feature of the history-sensitivity of objects.

There are two basic mechanisms to construct objects. First, states can be coerced to objects. We call such objects variable objects (where “variable” is used in the sense of Algol). A variable objects accepts two forms of use messages: a put message for modifying its value and a get message for reading the value.
\( \text{mkvar : unit} \leftrightarrow \text{object}(\text{var}(\delta)) \)

\( \text{mkvar}(\text{ms}) \leftarrow \text{mkvar'}(\text{INIT}_\delta, \text{ms}) \)

\( \text{mkvar'}(\text{v}, \text{done}) \leftarrow \)
\( \text{mkvar'}(\text{v}, [\text{get}(\text{val})]) \leftarrow \text{val} = \text{v} \)
\( \text{mkvar'}(\text{v}, [\text{put}(\text{val})]) \leftarrow \)
\( \text{mkvar'}(\text{v}, [\text{get}(\text{val})]^\text{ms}) \leftarrow \text{val} = \text{v}, \text{mkvar'}(\text{v}, \text{ms}) \)
\( \text{mkvar'}(\text{v}, [\text{put}(\text{val})]^\text{ms}) \leftarrow \text{mkvar'}(\text{val}, \text{ms}) \)
\( \text{mkvar'}(\text{v}, (\text{ms1} \^ \text{ms2})^\text{ms}) \leftarrow \text{mkvar'}(\text{v}, \text{ms1} \^ (\text{ms2} \^ \text{ms})) \)

Figure 3: Creation of a variable object

\text{type \text{var}(\delta) = \text{put}(\delta) \mid \text{get}(\delta) }

\text{varobject : state(\delta) \leftrightarrow \text{object}(\text{var}(\delta)) }

\text{varobject}(\text{st}, \text{done}) \leftarrow \text{st} = \text{done} \)
\( \text{varobject}(\text{st}, [\text{put}(\text{val})]) \leftarrow \text{st} = \text{write}(\text{val}, \text{done}) \)
\( \text{varobject}(\text{st}, [\text{get}(\text{val})]) \leftarrow \text{st} = \text{read}(\text{val}, \text{done}) \)
\( \text{varobject}(\text{st}, \text{ms1} \^ \text{ms2}) \leftarrow \text{append}(\text{st1}, \text{st2}, \text{st}), \text{varobject}(\text{st1}, \text{ms1}), \text{varobject}(\text{st2}, \text{ms2}) \)

Note that all the operations of the variable object are carried out on the underlying state. Since states can be treated as variable objects, we eliminate states from our presentation and use variable objects in their place. For this purpose, we define a predicate for creating a variable object:

\( \text{mkvar : unit} \leftrightarrow \text{object}(\text{var}(\delta)) \)
\( \text{mkvar}(\text{ms}) \leftarrow \text{mkstate}(\text{st}), \text{varobject}(\text{st}, \text{ms}) \)

A direct definition of this predicate is given in Figure 3. The Algol command \texttt{new i. (i := 0; i := i + 1)} can now be expressed using variable objects as

\( \text{mkvar}(i), i = i1 \^ i2, i1 = [\text{put}(0)], i2 = [\text{get}(x)] \^ [\text{put}(y)], \text{add}(x, 1, y) \)

The second mechanism for creating objects is the predicate scheme

\text{\texttt{mkobject}[\bar{t} : \bar{A}; \phi; u : C]}

which is similar to the \text{\texttt{mkclosure}} scheme except that the types in \( \bar{A} \) can be either closure types or object types. \text{\texttt{mkobject}}[\bar{t} : \bar{A}; \phi; u : C] is a predicate \( P \) defined recursively as follows:

\( P : A_1 \times \ldots \times A_k \leftrightarrow \text{object}(C) \)
\( P(\bar{z}, \text{done}) \leftarrow \bar{z} = \text{done/discard} \)
\( P(\bar{z}, \text{use}(\text{val})) \leftarrow \bar{z} = \bar{t}, \text{val} = u, \phi \)
\( P(\bar{z}, \text{ms1} \^ \text{ms2}) \leftarrow \{ z_i = x_i \cdot y_i/\bar{x_i} @ \bar{y_i} \}_i, P(\bar{x}, \text{ms1}), P(\bar{y}, \text{ms2}) \)

The choice between \texttt{done} and \texttt{discard} in the first clause and that between \( x_i \cdot y_i \) and \( x_i @ y_i \) in the third clause are made based on the type \( A_i \) (which is either an object type or a closure type).

For an example, consider a counter object that returns an integer each time it is used and, simultaneously, increments its internal state. To make such an object, we can use the following instance of \texttt{mkobject}:
\begin{align*}
\text{counter} \, \text{object} & : \text{object}(\text{var} \, \text{int}) \leftrightarrow \text{object} \, \text{int} \\
counter \, \text{object}(v, \text{done}) & \leftarrow v = \text{done} \\
counter \, \text{object}(v, [x]) & \leftarrow v = [\text{get}(x)] \sim [\text{put}(x')], \text{add}(x, 1, x') \\
counter \, \text{object}(v, ms1 \sim ms2) & \leftarrow v = v1 \sim v2, \text{counter} \, \text{object}(v1, ms1), \text{counter} \, \text{object}(v2, ms2)
\end{align*}

Figure 4: A counter object

\begin{align*}
\text{project}1 & : \text{object}(A|B) \leftrightarrow \text{object}(A) \\
\text{project}2 & : \text{object}(A|B) \leftrightarrow \text{object}(B) \\
\text{project}1 & (\text{obj}, \text{done}) \leftarrow \text{obj} = \text{done} \\
\text{project}1 & (\text{obj}, [x]) \leftarrow \text{obj} = [\text{inl}(x)] \\
\text{project}1 & (\text{obj}, m1 \sim m2) \leftarrow \text{obj} = \text{obj}1 \sim \text{obj}2, \text{project}1(\text{obj}1, m1), \text{project}1(\text{obj}2, m2) \\
\text{project}2 & (\text{obj}, \text{done}) \leftarrow \text{obj} = \text{done} \\
\text{project}2 & (\text{obj}, [x]) \leftarrow \text{obj} = [\text{inr}(x)] \\
\text{project}2 & (\text{obj}, m1 \sim m2) \leftarrow \text{obj} = \text{obj}1 \sim \text{obj}2, \text{project}2(\text{obj}1, m1), \text{project}2(\text{obj}2, m2)
\end{align*}

Figure 5: Projection of objects

\begin{align*}
\text{counter} \, \text{object} & \equiv \text{mkobject}[v; v = [\text{get}(x)] \sim [\text{put}(x')], \text{add}(x, 1, x'); x]
\end{align*}

Its expanded form is shown in Figure 4. To create a counter, we use the predicate

\begin{align*}
\text{mkcounter}(\text{counter}) & \leftarrow \text{mkvar}([\text{put}(0)] \sim v), \text{counter} \, \text{object}(v, \text{counter})
\end{align*}

If we use the object by setting \text{counter} = [x_1] \sim [x_2] \sim \ldots the variables \( x_1, x_2, \ldots \) get bound to successive integers \( 0, 1, \ldots \). The reader may wish to trace through the definitions and verify that this is the case.

The most appealing aspect of the model we are building is that it allows hierarchical abstractions of state-dependent values. The constructor \text{mkvar} internally creates and uses a state. Likewise, the constructor \text{mkcounter} internally creates and uses a variable. The internal representations are entirely hidden from the users of these abstractions. We will see, in Section 2, that this feature is crucial to establishing many equivalences that have proved problematic in the previous models of state.

An important special case of the \text{mkobject} scheme is projection of object components. An object of type \text{object}(A|B) can be coerced to \text{object}(A) by the following instance:

\begin{align*}
\text{project}1 & : \text{object}(A|B) \leftrightarrow \text{object}(A) \\
\text{project}1 & \equiv \text{mkobject}[\text{use}(\text{inl}(x)); \text{true}; x]
\end{align*}

The expanded form of \text{project}1 as well as \text{project}2 are shown in Figure 5. These projections essentially “forward” messages sent to the projected object to the original object. Projection of the first component of a variable object gives a “write only” version of the variable.

\begin{align*}
\text{writeonly} & : \text{object}(\text{var}(\delta)) \leftrightarrow \text{object}(\delta) \\
\text{writeonly} & \equiv \text{mkobject}[\text{use}([\text{put}(x)]; \text{true}; x]
\end{align*}
Such write-only versions model the $l$-values or acceptor components. The $r$-values can be similarly projected but, since the multiple values read from the same state are equal, we can coerce the $r$-values to closures:

$$\text{readonly} : \text{object}(\text{var}(\delta)) \leftrightarrow \text{closure}(\delta)$$

$$\text{readonly}(v, \text{discard}) \leftrightarrow v = \text{done}$$

$$\text{readonly}(v, [x]) \leftrightarrow v = [\text{get}(x)]$$

$$\text{readonly}(v, m1 \circ m2) \leftrightarrow \text{merge}(v1, v2, v), \text{readonly}(v1, m1), \text{readonly}(v2, m2)$$

$$\text{merge} : \text{object}(\text{var}(\delta)) \times \text{object}(\text{var}(\delta)) \leftrightarrow \text{object}(\text{var}(\delta))$$

$$\text{merge}(ms1, ms2, v) \leftrightarrow ms1 = \text{done}, v = ms2;\mspace{1em} v = [\text{get}(x)]; ms1 = [\text{get}(x)], v = ms2;\mspace{1em} ms2 = \text{done}, v = ms1;\mspace{1em} ms2 = [\text{get}(y)], v = ms1$$

We show that this construction is well-behaved by a series of elementary results. Define a variable stream to be any $v \in \text{object}(\text{var}(\delta))$ such that $\text{mkvar}(m_1 \wedge v \wedge m_2)$ for some $m_1, m_2 \in \text{object}(\text{var}(\delta))$.

**Theorem 2** If $\text{mkvar}(m_1 \wedge v \wedge m_2)$ then

1. $v = [\text{put}(x)] \wedge [\text{get}(y)]$ iff $\text{mkvar}(m_1 \wedge [\text{put}(x)] \wedge m_2)$ and $y = x$.

2. $v = [\text{get}(x)] \wedge [\text{get}(y)]$ iff $\text{mkvar}(m_1 \wedge [\text{get}(x)] \wedge m_2)$ and $y = x$.

As a corollary of the second property, we have that the variable streams $[\text{get}(x)] \wedge [\text{get}(y)]$ and $[\text{get}(y)] \wedge [\text{get}(x)]$ are observationally equivalent. This gives the result:

**Theorem 3** The coercion $\text{readonly}$ is functional, i.e., whenever $\text{readonly}(v, m1)$ and $\text{readonly}(v, m2)$, $m1 \equiv m2$ is a commutative monoid equivalence.

**Theorem 4** For all variable streams $v_1, v_2 : \text{object}(\text{var}(\delta))$ and $m_1, m_2 : \text{closure}(\delta)$, $v_1 \equiv v_2$ (as monoid equivalence) and $m_1 \equiv m_2$ (as commutative monoid equivalence) implies $\text{readonly}(v_1, m_1) \Leftrightarrow \text{readonly}(v_2, m_2)$.

### 1.4 Modelling interference-free Algol

The types of Algol with syntactic control of interference are modelled as follows. Each type has two interpretations. The output interpretation, denoted $\theta^\circ$, applies whenever $\theta$ is used as the type of an output (or result) of a phrase. The input interpretation, denoted $\theta^*$, applies when $\theta$ is used as the type of an input of a phrase. Note that a similar situation arises in modelling typed lambda calculus (and intuitionistic logic) in intuitionistic logic [Gir87, Sec. 7].

The output interpretation of types is

$$\begin{align*}
(\delta \text{ exp})^\circ &= \delta \\
(\theta_1 \rightarrow_R \theta_2)^\circ &= \theta_1^\circ \times \theta_2^\circ
\end{align*}$$

$$\begin{align*}
(\delta \text{ acc})^\circ &= \delta \\
(\theta \rightarrow \alpha)^\circ &= \theta^* \times \alpha^\circ \\
(\delta \text{ var})^\circ &= \text{var}(\delta)
\end{align*}$$

The input interpretation is

$$\begin{align*}
\phi^* &= \text{closure}(\phi^\circ) \\
\alpha^* &= \text{object}(\alpha^\circ)
\end{align*}$$
The reason for the distinction between input and output interpretations may be understood operationally as follows: a computation sends messages to its inputs in order to obtain their values. On the other hand, there is no message sending involved in constructing the output. Some examples of the type interpretations follow:

\[
\begin{align*}
(\delta \text{var} \rightarrow \text{comm})^0 &= \text{object}(\text{var}(\delta)) \times \text{unit} \\
(\text{comm} \rightarrow \text{comm})^o &= \text{object}(\text{unit}) \times \text{unit} \\
(\delta \text{var} \rightarrow p \delta \text{var})^o &= \text{object}(\text{var}(\delta)) \times \text{var}(\delta) \\
(\delta \text{var} \rightarrow \delta \text{var})^o &= \text{closure}(\text{object}(\text{var}(\delta)) \times \text{var}(\delta)) \\
(\delta \text{var} \rightarrow \delta \text{var})^* &= \text{object}(\text{object}(\text{var}(\delta)) \times \text{var}(\delta))
\end{align*}
\]

An Algol phrase \(x_1 : \theta_1, \ldots, x_n : \theta_n \vdash p : \theta\) is interpreted as a relation \(R \subseteq \theta_1^* \times \cdots \times \theta_n^* \leftrightarrow \theta^o\). We describe the relation compactly using the suggestive notation

\[
t_1, \ldots, t_n \vdash \{\psi\} u
\]

where \(\Gamma \vdash t_1 : \theta_1^*, \ldots, \Gamma \vdash t_n : \theta_n^*\) and \(\Gamma \vdash u : \theta^o\) are terms and \(\Gamma \vdash \psi\) Formula is a formula. This is meant to denote the relation \(R\) such that

\[
R(x_1, \ldots, x_n, z) \iff \exists y. x_1 = t_1 \land \ldots \land x_n = t_n \land z = u \land \psi
\]

where \(y = \text{dom} \Gamma\).

We define the interpretations of Algol phrases by induction on their type derivations. If

\[
\begin{array}{c}
\Theta_1 \vdash p_1 : \theta_1 \\
\vdots \\
\Theta_i \vdash p_k : \theta_k
\end{array}
\]

is an Algol type inference rule, we use an interpretation rule of the form

\[
t_1 \vdash \{\psi_1\} u_1, \quad t_k \vdash \{\psi_k\} u_k \\
\vdash t' \vdash \{\psi'\} u'
\]

This means the obvious: if the interpretation of \(\Theta_i \vdash p_i : \theta_i\) is the relation denoted by \(t_i \vdash \{\psi_i\} u_i\), then the interpretation of \(\Theta' \vdash p' : \theta'\) is the relation denoted by \(t' \vdash \{\psi'\} u'\).

Here are the interpretations for each Algol type rule. For the identity group:

\[
\begin{array}{c}
\begin{array}{c}
\text{Id} \quad \vdash i : \theta \\
\text{Cut} \quad \frac{\Theta_1 \vdash p : \phi \quad \Theta_2, i : \phi \vdash q : \theta}{\Theta_1, \Theta_2 \vdash q[p/i] : \theta}
\end{array}
\end{array}
\]

For the structural rules:

\[
\begin{array}{c}
\begin{array}{c}
\text{Exchange} \quad \frac{\Theta, i_1 : \theta_1, i_2 : \theta_2, \Theta' \vdash p : \theta}{\Theta, i_2, i_1 : \theta_2, \Theta' \vdash p : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Weak} \quad \frac{\Theta \vdash p : \theta}{\Theta, i : \phi \vdash p : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Weak} \quad \frac{\Theta, i_1 : \phi, i_2 : \phi \vdash p : \theta}{\Theta, i : \phi \vdash p[i/i_1, i/i_2] : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Contr} \quad \frac{\Theta, i : \phi \vdash \{\psi\} u}{\Theta, i : \phi \vdash p[i/i_1, i/i_2] : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Exchange} \quad \frac{\Theta, i_1 : \theta_1, i_2 : \theta_2, \Theta' \vdash p : \theta}{\Theta, i_2, i_1 : \theta_2, \Theta' \vdash p : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Weak} \quad \frac{\Theta \vdash p : \theta}{\Theta, i : \phi \vdash p : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Weak} \quad \frac{\Theta, i_1 : \phi, i_2 : \phi \vdash p : \theta}{\Theta, i : \phi \vdash p[i/i_1, i/i_2] : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Contr} \quad \frac{\Theta, i : \phi \vdash \{\psi\} u}{\Theta, i : \phi \vdash p[i/i_1, i/i_2] : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Exchange} \quad \frac{\Theta, i_1 : \theta_1, i_2 : \theta_2, \Theta' \vdash p : \theta}{\Theta, i_2, i_1 : \theta_2, \Theta' \vdash p : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Weak} \quad \frac{\Theta \vdash p : \theta}{\Theta, i : \phi \vdash p : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Weak} \quad \frac{\Theta, i_1 : \phi, i_2 : \phi \vdash p : \theta}{\Theta, i : \phi \vdash p[i/i_1, i/i_2] : \theta}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Contr} \quad \frac{\Theta, i : \phi \vdash \{\psi\} u}{\Theta, i : \phi \vdash p[i/i_1, i/i_2] : \theta}
\end{array}
\end{array}
\]
For the variable rules:

\[ \Theta, i : \delta \text{ exp} \triangleright p : \theta \]
\[ \Theta, i : \delta \text{ var} \triangleright p : \theta \]
\[ \Theta, i : \delta \text{ acc} \triangleright p : \theta \]
\[ \Theta, i : \delta \text{ var} \triangleright c : \text{ comm} \]
\[ \Theta \triangleright \text{ new} \delta \text{ var } i, c : \text{ comm} \]

For commands:

\[ \triangleright \text{skip} : \text{ comm} \]
\[ \Theta \triangleright a : \delta \text{ acc} \quad \Theta \triangleright e : \delta \text{ exp} \]
\[ \Theta \triangleright a := e : \text{ comm} \]
\[ \Theta \triangleright c_1 : \text{ comm} \quad \Theta \triangleright c_2 : \text{ comm} \]
\[ \Theta \triangleright (c_1 \parallel c_2) : \text{ comm} \]
\[ \Theta_1 \triangleright c_1 : \text{ comm} \quad \Theta_2 \triangleright c_2 : \text{ comm} \]
\[ \Theta_1, \Theta_2 \triangleright (c_1 \parallel c_2) : \text{ comm} \]

For terms \( t_1, t_2 \) of type \( \text{ closure}(A) \), \( \text{ dup}[t_1; t_2] \) means the term \( t_1 @ t_2 \). For terms of type \( \text{ object}(A) \), \( \text{ dup}[t_1; t_2] \) means \( t_1 \circ t_2 \). We extend this notation pair-wise for sequences of terms:

\[ \text{ dup}[t_{11}, \ldots , t_{1k}; t_{21}, \ldots , t_{2k}] \equiv \text{ dup}[t_{11}; t_{21}], \ldots , \text{ dup}[t_{1k}; t_{2k}] \]

For higher-order phrases, the interpretation is:

\[ \Theta, i : \theta \triangleright p : \theta' \rightarrow \mathcal{R} \]
\[ \Theta_1 \triangleright p : \theta \quad \Theta_2, i : \theta \triangleright q : \theta' \rightarrow \mathcal{L} \]

Finally, the generic conditional construction

\[ \Theta \triangleright p : \text{ bool exp} \quad \Theta \triangleright q_1 : \theta \quad \Theta \triangleright q_2 : \theta \]

is interpreted by

\[ t' \triangleright \{ \psi' \} u \quad t_1 \triangleright \{ \psi_1 \} v_1 \quad t_2 \triangleright \{ \psi_2 \} v_2 \]
\[ \text{ dup}[t'; x] \triangleright \{ \psi', (u = tt, x = t_1, y = v_1; v = ff, x = t_2, y = v_2) \} y \]
2 Examples

Example 1 The command

\[ \textbf{new } x. \ x := 1 : \textbf{comm} \]

is equivalent to \textbf{skip} because \( x \) is never read before it is discarded. Its stream semantics is

\[ \triangleright \{ \text{mkvar([put(1)])} \} () \]

It is easy to see that the goal is equivalent to true.

\[ \text{mkvar([put(1)])} \]
\[ = \text{mkvar'(undef, [put(1)])} \]
\[ = \text{true} \]

So, the semantics is equivalent to () which is also the semantics of \textbf{skip}. \( \Box \)

Example 2 A slight variation of the above example appears in [MS88]:

\[ c : \textbf{comm} \triangleright \textbf{new } x. \ c : \textbf{comm} \]

This command should be equivalent to \( c \) because \( c \) does not have access to the local variable \( x \). The stream semantics of the command is

\[ [c] \triangleright \{ \text{mkvar(done), c = ()} \} () \]

Since \( \text{mkvar(done)} \) is equivalent to true, the above is equivalent to

\[ [c] \triangleright \{ c = () \} () \]

This is nothing but the semantics of \( c : \textbf{comm} \triangleright c : \textbf{comm} \). \( \Box \)

Example 3 Consider the block

\[ \textbf{new } x. \ x := 0; \ c; \text{if } x = 0 \text{ then diverge} \]

in the context \( c : \textbf{comm} \). One expects the command to diverge because \( c \) cannot alter the local variable \( x \). The stream semantics of the command is

\[ t \triangleright \{ \text{mkvar([put(0)] } \times x, t = [()], (x = [get(0)] \rightarrow \text{diverge}) \} () \]

Clearly, \( \text{mkvar([put(0)] } \times x, t = [get(0)]) \) is equivalent to true. So, \( \text{diverge} \) is indeed evaluated and the goal fails to terminate. \( \Box \)

Example 4 The blocks

\[ Q : \textbf{var} \rightarrow \textbf{var} \rightarrow \textbf{comm} \triangleright \textbf{new } x. \ \textbf{new } y. \ x := 0; \ y := 0; \ Q \ x \ y \]

and

\[ Q : \textbf{var} \rightarrow \textbf{var} \rightarrow \textbf{comm} \triangleright \textbf{new } x. \ \textbf{new } y. \ x := 0; \ y := 0; \ Q \ y \ x \]

are expected to be equivalent. To consider the semantics, let \( Q \) be represented by the object \( q : \text{object(state(int) } \times \text{object(state(int) } \times \text{unit}) \). The semantics of the two blocks is, respectively,
\{\text{mkvar([put(0)]^x), mkvar([put(0)]^y), q = [(x, (y, ()))]} \} ()

and

\{\text{mkvar([put(0)]^x), mkvar([put(0)]^y), q = [(y, (x, ()))]} \} ()

Notice that \(x\) and \(y\) are local variables of the goal formulas. (There is an implicit quantification \(\exists x. \exists y\).) So, simple \(\alpha\) conversion shows their equivalence. \(\square\)

**Example 5** Consider the block

```
new x. new y.
    let incy = (y := y + 1)
in x := 0; y := 0; Q incy; if x = 0 then diverge
```

where \(Q : \text{comm} \rightarrow \text{comm}\) is a free variable. We expect the block to diverge because \(Q\ incy\) has no access to \(x\).

The stream semantics is (assuming \(Q\) is represented by an object \(q : \text{object(object(unit) \times unit)}\)):

\text{mkvar([put(0)]^x), mkvar([put(0)]^y), mkobject([get(v), put(v+1)]; ()](y, incy), q = [(incy, ())], x = [get(0)] \rightarrow diverge}

The combination \(\text{mkvar([put(0)]^x)}\), \(x = [get(0)]\) is equivalent to true. So, the goal fails to terminate. \(\square\)

**Example 6** The following example is attributed to Allen Stoughton in [MS88]:

```
new x.
    let add2 = (x := x + 2)
in x := 0; Q add2; if x mod 2 = 0 then diverge
```

This example fails in support-oriented interpretations [HMT83] and the state-set based possible world semantics [Ole85, Rey81].

The stream semantics of the program is

\text{mkvar([put(0)]^x), x = x1 \^ x2, mkobject([get(v), put(v+2)]; ()](x1, add2), q = [(add2, ())], x2 = [get(w)], w mod 2 = 0 \rightarrow diverge}

Apparantly, \(x1\) can ony be some number of repetitions of the sequence \([get(v), put(v + 2)]\). Hence, by using the definition of \text{mkvar}, the value that is eventually read into \(w\) must be even. So, the stream semantics of the program is divergent.

Note how this example is handled. The procedure \(Q\) is not passed complete acces to the state of \(x\), but, rather, an operation that manipulates the state of \(x\). Thus, the semantics faithfully models what one really intends by the Algol program above. This is in sharp contrast to the approach of [MS88], where \(Q\) is passed the entire state of \(x\), but the possible meanings of \(Q\) are artificially restricted by imposing an “invariant preservation” condition. The artificiality becomes apparant when we consider the next example. \(\square\)

**Example 7** Consider the following block:
**new** x. x := 0; P(x := x + 1)

where \( P : \text{comm} \rightarrow \text{comm} \) is a free variable. Since \( P \) has no way to access \( x \) and \( x \) is discarded after the call of \( P \), the block must be equivalent to \( P(\text{skip}) \). The equivalence fails to hold in the invariant-preservation models of [MS88, OT92] and the noninterference model of [Ten90].

In the stream semantics, the equivalence is straightforward and nothing special needs to be done. The meaning of the block is respectively,

\[
\text{mkvar([-put(0)] - x)}, \\
\text{mkobject([-get(v), put(v+1)]; (); ()](x, add1)}, \\
p = [(\text{add1}, ()]]
\]

Since the state changes carried out by \( \text{add1} \) are never observed, the goal is equivalent to the following simpler goal:

\[
\text{mkobject}[;(); ()](s), p = [(s, ())]
\]

Note that this goal represents the Algol command \( P(\text{skip}) \). □

The example is striking: note that we simply discard \( x \) rather than test for a condition (as in Example 6). So, the only reason the equivalence can fail is if \( P \) is allowed to “snoop inside” its argument and notice what it does. The semantics of [MS88] admits such snooping meanings, and invariant-preservation doesn’t quite do the trick.
3 Coherent semantics

The essential feature missing from the relational semantics is the notion of coherence. A closure is any collection of values whereas it should really be a collection of information tokens for a single “semantic value”. To capture this aspect, we use Girard’s coherent spaces [Gir87, GLT89].

The “web” of a coherent space is a reflexive undirected graph. The vertices are called tokens and the binary relation denoted by the arcs is called coherence. For a coherent space $A$, the set of tokens is denoted by $|A|$ and its coherence relation is denoted $a \sqsubset b \ [\text{mod} A]$ (when $a, b \in |A|$ are coherent). The strict part of the coherence relation is denoted $a \rightsquigarrow b \ [\text{mod} A]$, its complement (incoherence) is denoted $a \leftarrow b \ [\text{mod} A]$ and the reflexive closure of incoherence is denoted $a \dashv b \ [\text{mod} A]$. The coherent space itself is the set of cliques (strongly connected components) of the web, made into a coherent spaces determine each other uniquely. So, we abuse terminology and refer to webs as coherent spaces.

Some simple coherent spaces are the following:

- $\text{int}$, whose tokens are integers and the coherence relation is discrete ($i.e., i \sqsubset j \ [\text{mod int}]$ iff $i = j$),
- $\text{bool}$, whose tokens are the truth values $tt$ and $ff$, and the coherence relation is discrete, and
- $\text{1}$, which has a single token $*$ and the trivial coherence relation.

These spaces are the “directional” versions of the sets $\text{int}$, $\text{bool}$ and $\text{unit}$ used in the relational semantics. Some standard constructions on coherent spaces are:

$$
|A \& B| = |A| + |B| \quad (i, x) \sqsubset (j, y) \iff i = j \Rightarrow x \sqsubset y \\
|A \oplus B| = |A| + |B| \quad (i, x) \sqsubset (j, y) \iff i = j \land x \sqsubset y \\
|A \otimes B| = |A| \times |B| \quad (a, b) \sqsubset (a', b') \iff a \sqsubset a' \land b \sqsubset b' \\
|A \multimap B| = |A| \times |B| \quad (a, b) \multimap (a', b') \iff a \sqsubset a' \Rightarrow b \multimap b' \\
|A^\perp| = |A| \quad a \sqsubset b \ [\text{mod} A^\perp] \iff a \multimap b \ [\text{mod} A]
$$

The spaces $A \& B$ and $A \oplus B$ are two directional versions of the set-theoretic disjoint union construction used in the relational semantics. In the case of $A \& B$, we get a request for the $A$ component or $B$ component and we have to produce the corresponding information. In contrast, for $A \oplus B$, we decide to produce one of the components and produce it. So, the elements of $A \& B$ are pairs (produced lazily) whereas the elements of $A \oplus B$ are injections of elements of $A$ and $B$.

The spaces $A \otimes B$ and $A \multimap B$ are two directional versions of the set-theoretic product construction used in the relational semantics. In the case of $A \otimes B$, we have to produce both the components $A$ and $B$, whereas, for $A \multimap B$, we receive information for $A$ and produce information for $B$. So, the elements of $A \otimes B$ are pairs (produced eagerly) whereas the elements of $A \multimap B$ are functions. $A \otimes B$ is called the tensor product and $A \multimap B$ is called the linear function space.

The coherent space $A^\perp$ is called the dual of $A$. It switches the input-output notion of $A$. To produce an $A^\perp$ is the same as to receive an $A$ and to receive an $A^\perp$ is the same as to produce an $A$. We will have an application of it for modelling acceptors.

We call a coherent space with discrete coherence relation a discrete coherent space. The spaces $\text{int}$, $\text{bool}$ and $\text{1}$ are positive. Further, if $P$ and $Q$ are discrete coherent spaces, $P \oplus Q$ and $P \otimes Q$ are discrete. In other words, their coherence relations are not interesting.
The function space construction \( \to \) gives rise to interesting coherence. If the statement of coherence relation for \( A \to B \) seems surprising, consider the following equivalent formulation:

\[
(a, b) \simeq (a', b') \pmod{A \to B} \iff a \simeq a' \Rightarrow b \simeq b' \text{ and } a \simeq a' \Rightarrow b \simeq b'
\]

The second condition essentially states that a function cannot produce the same output \( b \) for two distinct, but coherent, inputs \( a \) and \( a' \). In other words, if a function maps an element \( X \) to an element \( Y \), each token of \( Y \) comes from a unique token of \( X \). This is used to model the “sequential” nature of the function. (Yet another formulation of the coherence relation is

\[
(a, b) \simeq (a', b') \pmod{A \to B} \iff a \simeq a' \lor b \simeq b'
\]

which is often useful in proofs). For example, the following input output pairs in \(|\text{bool} \& \text{bool} \to \text{bool}|\)
do not form a coherent set:

\[
((1, tt), tt), ((2, tt), tt)
\]

A function which has this behavior would have to demand both the components of its input in parallel and return \( tt \) if either component is \( tt \). On the other hand, the following input-output pairs form a coherent set in \( \text{bool} \otimes \text{bool} \to \text{bool}:\)

\[
((tt, tt), tt), ((tt, ff), ff), ((ff, tt), ff), ((ff, ff), ff)
\]

At the same time, note that inconsistency is prohibited:

\[
((tt, tt), tt), ((tt, tt), ff)
\]
do not form a coherent set.

If \( A \) and \( B \) are coherent spaces, we say that \( f \) is a linear function from \( A \) to \( B \) and write \( f : A \to B \) if \( f \) is an element of the coherent space \( A \to B \). The identity function \( id : A \to A \) is the element \( \{(a, a) : a \in |A| \} \). If \( f : A \to B \) and \( g : B \to C \) then \( g \circ f : A \to C \) is the element

\[
\{(a, c) : \exists b \in |B|. (a, b) \in f \land (b, c) \in g \}
\]

If \( (a, c) \) and \( (a', c') \) are distinct in \( g \circ f \), there exist \( b, b' \in |B| \) such that \( a \simeq a' \lor b \simeq b' \) and \( b \simeq b' \lor c \simeq c' \). So, necessarily, \( a \simeq a' \lor c \simeq c' \) showing \( (a, c) \) and \( (a', c') \) to be coherent. The linear function \( \text{apply} : (A \to B) \otimes A \to B \) is the element

\[
\{(((a, b), a), b) : a \in |A| \land b \in |B| \}
\]

If \( f : A \otimes B \to C \) then \( \text{curry} f : A \to (B \to C) \) is the element

\[
\{(a, (b, c)) : ((a, b), c) \in f \}
\]

These combinators give coherent spaces the structure of a “closed symmetric monoidal category”.

An important function space in our application is \( A \to 1 \). It is given by

\[
|A \to 1| = |A| \times \{*\} \quad (a, *) \simeq (b, *) \pmod{A \to 1} \iff a \simeq b \pmod{A} \lor * \simeq * \pmod{1}
\]

Since \( * \simeq * \) is false, it is convenient to consider a coherent space without “*”. We define it as follows:

\[
|A^\perp| = |A| \quad a \simeq b \pmod{A^\perp} \iff a \simeq b \pmod{A}
\]

\( A^\perp \) is called the dual of \( A \). It is easy to verify that \( h : A^\perp \to (A \to 1) \) given by pairs \( (a, (a, *)) \) is an isomorphism.
Closures The coherent space construction corresponding to the \textit{closure} type constructor is denoted by \(!\). Its tokens are given by \(|!A| = A_{\text{fin}}\), finite subsets of \(|A|\) whose members are pairwise coherent. Its coherence relation is given by

\[ X \circ Y \mod {!A} \iff \forall a \in X. \forall a' \in Y. a \circ a' \mod A \]

An equivalent formulation is \(X \circ Y \iff X \cup Y \in A_{\text{fin}}\). The constructions \textit{discard}, \textit{use}(\(x\)) and \(m_1@m_2\) of the closure types correspond to the empty set, singletons and unions in \(A_{\text{fin}}\) respectively. However, these operations now run \textit{backwards}:

\[
\begin{align*}
    d : !A &\rightarrow 1 & (\emptyset, *) \\
    u : !A &\rightarrow A & (\{a\}, a) \\
    @ : !A &\rightarrow !A \otimes !A & (X_1 \cup X_2, (X_1, X_2))
\end{align*}
\]

This direction is the natural one: information flows \textit{from} the closure to its client. However, the demands for information (messages) are still issued by the clients. So, if one were to model \(!A\) in a programming language based on the coherent space types, one would define

\[
\text{type } !\alpha = 1 \& \alpha \& (!\alpha \otimes !\alpha)
\]

In comparison with the relational type definition given in Sec. 1.1, notice that the nondirectional type construction \("\mid\)\) has been refined into the directional version \("\&\)\. The equivalences of the closure type also hold for \(!A\), but in reverse order, to form what is known as a \textit{cocommutative comonoid}. See [Laf88] for an explicit statement of these equations.

Objects The construction corresponding to the \textit{object} type constructor is denoted \(\uparrow\). Its tokens are given by \(|\uparrow A| = |A|^*\), finite sequences over \(|A|\). Its coherence relation is given by

\[ s \rhd r \mod \uparrow A \iff \exists p, s', r' \in |A|^*. a, b \in |A|, s = pas' \land r = pbbr' \land a \rhd b \mod A \]

In words, distinct sequences \(s\) and \(r\) coherent if they have some common prefix \(p\) and the following tokens \(a\) and \(b\) (in \(s\) and \(r\) respectively) are coherent. This definition captures the notion that a sequence \(s \in |\uparrow A|\) denotes a “history” of an object. Two sequences can coherently form histories of the same object if, at the first point of difference, they exhibit coherent behavior. There may be no relationship between the future behavior of the two histories after this point of difference. The constructions \textit{done}, \textit{use}(\(x\)) and \(m_1 @ m_2\) of the object types correspond to the empty sequence, singleton sequence and concatenations in \(|A|^*\) respectively. Just as for \(!A\), the operations run backwards:

\[
\begin{align*}
    d : \uparrow A &\rightarrow 1 & (\{\}, *) \\
    u : \uparrow A &\rightarrow A & (\{a\}, a) \\
    \wedge : \uparrow A &\rightarrow \uparrow A \uplus \uparrow A & (sr, (s, r))
\end{align*}
\]

Here, \(\uplus\) is a “sequential” tensor product (pronounced “before”) with the obvious definition:

\[ |A \uplus B| = |A| \times |B| \quad (a, b) \sim (a', b') \iff a \sim a' \lor (a = a' \land b \sim b') \]

The operations \(d, u\) and \(\wedge\) satisfy the equations of a \textit{comonoid}.

Variable An Algol variable of type \(\delta\) is modelled by a special kind of object of type \(\uparrow (\delta^\bot \& \delta)\). Note that a token of this type is a sequence of the form

\[(i_1, d_1) \ldots (i_n, d_n)\]

where \(i_j \in \{1, 2\}\) and \(d_j \in \delta\). Two pairs \((i, d), (i', d')\) are coherent \(\mod \delta^\bot \& \delta\) if

\[ i \neq i' \text{ or } i = i' = 1 \land d \sim d' \text{ or } i = i' = 2 \land d \sim d' \]

16
References


