



# A fully abstract relational model of Syntactic Control of Interference

Guy McCusker  
University of Sussex

# Idealized Algol

In *The Essence of Algol*, Reynolds proposed an elegant theoretical language, now known as *Idealized Algol*. It combines:

- the language of while-programs:

```
 $y := 1;$   
while  $x > 0$  do  
   $y := y \times x;$   
   $x := x - 1$ 
```

- the simply-typed  $\lambda$ -calculus
- local allocation of storage variables: `new  $x$  in  $C$ .`

# Syntactic Control of Interference

- In *Syntactic Control of Interference*, Reynolds addressed the problem of *aliasing* and its higher-order cousin, *interference*.
- Application can bind distinct identifiers to the same variable:

$$(\lambda x : \text{var} . \lambda y : \text{var} . M) z z$$

This can render programs unclear, hard to reason about, and potentially incorrect.

- The problem is eliminated if we restrict application so that functions do not have any identifiers in common with their arguments.

# The type system

The SCI language is a restricted Idealized Algol. Its types are

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

In our version, all base types contain terms with side effects, for example:

$$(x := 4); 19$$

in the type `nat`.

# Interference control

A *multiplicative* typing rule for application imposes Reynolds's constraint:

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

Here  $\Gamma$  and  $\Delta$  must be disjoint contexts. In logical terms, this is an *affine* type system.

# Interference control

A *multiplicative* typing rule for application imposes Reynolds's constraint:

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B}$$

Here  $\Gamma$  and  $\Delta$  must be disjoint contexts. In logical terms, this is an *affine* type system.

This does not mean variables are used at most once: sequential composition is typed *additively*:

$$\frac{\Gamma \vdash M : \text{comm} \quad \Gamma \vdash N : \text{comm}}{\Gamma \vdash M; N : \text{comm}}$$



## An event-based model

- In a 1996 paper, Reddy gave a model of SCI which he called *object spaces*.
- A computation is interpreted as a sequence of observable events of some kind.
- A program is interpreted as a relation between these sequences. Closed programs are sets of sequences: “trace sets” or “strategies”.
- (Reddy’s model is a precursor of the game-based models of imperative languages.)





## Our contribution

- We give a simple construction of a closed category in which Reddy's model can be seen to live; this was lacking in Reddy's original presentation.
- We show that the model is *fully abstract*: it captures program equivalence precisely.



# Semantics of types

- At type `nat`, observations will be natural numbers. In general, a term of type `nat` can be used many times, so

$$\llbracket \text{nat} \rrbracket = \omega^*$$

the monoid of lists of naturals.

- At type `comm`, the only direct observation is termination of a command, so

$$\llbracket \text{comm} \rrbracket = 1^*$$

the monoid of lists over a one-letter alphabet. We will use  $*$  for the unique atomic observation.

# Semantics of types

- At type `var`, there are two kinds of event: reading from a variable, and writing to a variable, so

$$\llbracket \text{var} \rrbracket = \{ \text{read}(n), \text{write}(n) \mid n \in \omega \}^*$$

- At function types  $A \multimap B$ , events take the form

$$(s, b)$$

where  $s$  is a sequence of  $A$ -events and  $b$  is a single  $B$ -event.

# Semantics of terms

A program

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

is interpreted as a relation over

$$\llbracket A_1 \rrbracket \times \dots \times \llbracket A_n \rrbracket \times \llbracket B \rrbracket.$$

In fact it suffices to consider only singleton sequences in  $B$  (cf. semantics of function types).

# Some examples

A program such as

$$x : \text{var}, y : \text{var} \vdash x := !y; x := !x + 1 : \text{comm}$$

is interpreted as a relation over  $[[\text{var}]] \times [[\text{var}]] \times [[\text{comm}]]$  with elements such as:

$$(\text{write}(n)\text{read}(m)\text{write}(m + 1), \text{read}(n), *)$$

# Some examples

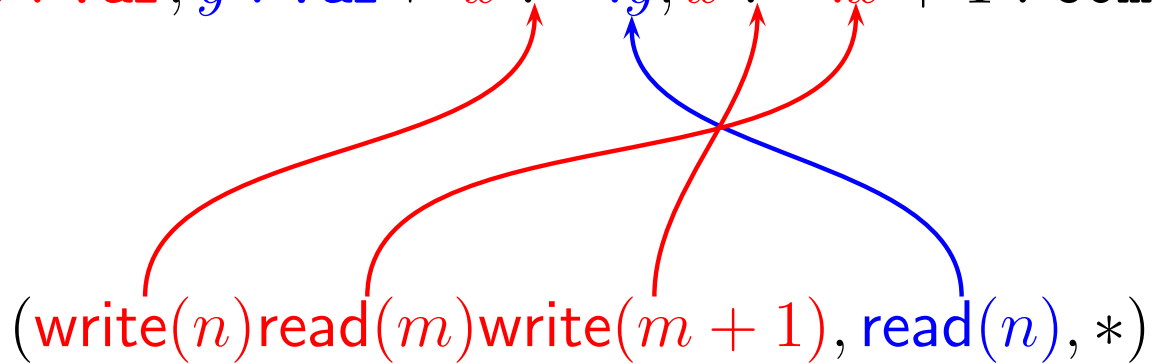
$x : \text{var}, y : \text{var} \vdash x := !y; x := !x + 1 : \text{comm}$

$(\text{write}(n)\text{read}(m)\text{write}(m + 1), \text{read}(n), *)$

# Some examples

$x : \text{var}, y : \text{var} \vdash x := !y; x := !x + 1 : \text{comm}$

$(\text{write}(n)\text{read}(m)\text{write}(m + 1), \text{read}(n), *)$





# Remarks

The semantic element

$$(\text{write}(n)\text{read}(m)\text{write}(m + 1), \text{read}(n), *)$$

illustrates some important facts about the model:

- the order of events in each variable is recorded, but not the relative order of events in distinct variables. This is okay thanks to non-interference.
- nothing in our definition forces  $n = m \dots$  yet.

# Abstraction

After  $\lambda$ -abstraction, the program

$$y : \text{var} \vdash \lambda x. x := !y; x := !x + 1 : \text{var} \multimap \text{comm}$$

is interpreted as a relation over  $\llbracket \text{var} \rrbracket \times (\llbracket \text{var} \rrbracket \times 1)^*$  with elements like

$$(\text{read}(n), (\text{write}(n)\text{read}(m)\text{write}(m + 1), *))$$

## Semantics of new

- The new constructor binds a `var`-type identifier to a storage location, and hides the variable from the outside world.
- It has type  $(\text{var} \multimap \text{comm}) \multimap \text{comm}$ . Elements of the type  $\text{var} \multimap \text{comm}$  look like

$(\text{read}(n)\text{write}(n')\text{read}(n'') \dots, *)$

- Call a sequence of reads and writes a *cell-trace* if the values read and written match up in the expected way.

## Semantics of new

- The semantics of `new` contains all elements of the form

$$((s, *), *)$$

where  $s$  is a cell-trace.

- Relational composition with our example, whose elements look like

$$(\text{read}(n), (\text{write}(n)\text{read}(m)\text{write}(m + 1), *))$$

therefore selects those entries for which  $n = m$ , and then hides all the  $x$ -behaviour.



First contribution:  
A category for all this stuff



# The category

We construct a category `MonRel` as follows.

- Objects are monoids
- A morphism  $A \rightarrow B$  is a relation between  $A$  and  $B$  subject to certain constraints.



## An alternative characterization

- Recall that **Rel** is equivalent to  $\mathbf{Set}_{\mathcal{P}}$ , the Kleisli category of **Set** with respect to the powerset monad.
- This has sets as objects and functions

$$A \rightarrow \mathcal{P}B$$

as morphisms from  $A$  to  $B$ .

- We can also say that **Rel** is  $(\mathbf{Set}_{\mathcal{P}})^{\text{op}}$ ...
- $\mathcal{P}$  works on **Mon**, and it turns out that


$$\mathbf{MonRel} \cong (\mathbf{Mon}_{\mathcal{P}})^{\text{op}}$$

# Categorical structure of MonRel


This characterization lets us translate categorical structure from Mon to MonRel. Mon has:

- products in Mon give monoidal structure on MonRel.
- coproducts in Mon give products in MonRel.
- exponentials  $A \multimap B$  exist in MonRel for all Kleene monoids  $B$ .

This is precisely the categorical structure needed to support our model of SCI.



Second contribution:  
A proof of full abstraction



# Correctness

- Reddy showed that the model is *sound*, so that

$$\llbracket M \rrbracket = \llbracket M' \rrbracket \Rightarrow M \cong M'.$$

- Our main contribution is to show that the converse holds:

$$M \cong M' \Rightarrow \llbracket M \rrbracket = \llbracket M' \rrbracket.$$

- The proof hinges on a *definability* result: we show that the part of the model in the image of  $\llbracket - \rrbracket$  is “big enough”.

# Definability of tests

- We will show that for any monoid  $A$  interpreting a type of SCI and any  $a \in A$ , there is a program

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

whose semantics contains only the pair  $(a, *)$ .

- This implies that there are enough programs in the language to distinguish between any two distinct semantic elements; full abstraction follows.

# Definability of producers

To complete a proof by induction on type, we need a little more.

We simultaneously show that for any  $a \in A$ , there is a term

$$y : \text{var} \vdash \text{produce}(a) : A$$

such that if  $(s, a') \in \llbracket \text{produce}(a) \rrbracket$  and  $s$  is a cell-trace with a certain initial and final value, then  $a = a'$ .

# Base type tests

Consider the element  $1 \cdot 2 \cdot 3 \in \llbracket \text{nat} \rrbracket$ .

The test for this element is:

if  $x = 1$  then  
  if  $x = 2$  then  
    if  $x = 3$  then skip  
  else diverge.

# Base type producers

The producer for  $1 \cdot 2 \cdot 3$  is:

$$y : \text{var} \vdash y := !y + 1; !y : \text{nat}$$

Its only element  $(s, a)$  for which  $s$  is a cell-trace starting with a  $\text{read}(0)$  and ending with a  $\text{read}(3)$  is

$$\left( \begin{array}{l} \text{read}(0)\text{write}(1)\text{read}(1) \\ \text{read}(1)\text{write}(2)\text{read}(2) \\ \text{read}(2)\text{write}(3)\text{read}(3) \end{array} , 1 \cdot 2 \cdot 3 \right)$$

# Base type producers

$$y : \text{var} \vdash y := !y + 1; !y : \text{nat}$$
$$\left( \begin{array}{l} \text{read}(0)\text{write}(1)\text{read}(1) \\ \text{read}(1)\text{write}(2)\text{read}(2) \\ \text{read}(2)\text{write}(3)\text{read}(3) \end{array} , 1 \cdot 2 \cdot 3 \right)$$

# Base type producers

$$y : \text{var} \vdash y := !y + 1; !y : \text{nat}$$
$$\left( \begin{array}{l} \text{read}(0)\text{write}(1)\text{read}(1) \\ \text{read}(1)\text{write}(2)\text{read}(2) \\ \text{read}(2)\text{write}(3)\text{read}(3) \end{array} , 1 \cdot 2 \cdot 3 \right)$$

# Testing higher types

A test for  $(a, b) \in \llbracket A \multimap B \rrbracket$  looks a bit like

$$x : A \multimap B \vdash \text{test}(b)(x(\text{produce}(a))) : \text{comm}$$

The term  $\text{produce}(a)$  supplies  $a$  as input to  $x$ , and we then test that  $b$  is given as output. (Actually, a little more is needed. Don't tell anyone.)

Higher type producers are defined similarly.



## That's all folks

We now have all the tests and producers we need, completing our definability proof.

**Theorem:** The model of SCI in MonRel is fully abstract.

