A fully abstract relational model of Syntactic Control of Interference

Guy McCusker

Algol-like Languages

- In *The Essence of Algol* Reynolds pointed out that Algol 60 was a powerful and elegant combination of higher-order functional programming with imperative programming.
- A large body of semantic research in imperative programming has been developed, focussing on so-called *Algol-like languages*.

Idealized Algol

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

- the language of while-loops:

$$\begin{array}{l} y := 1; \\ \text{while } x > 0 \text{ do} \\ \quad y := y \times x; \\ \quad x := x - 1 \end{array}$$

- the simply-typed λ-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:

$$(\lambda x : \mathtt{var}.\lambda y : \mathtt{var}.M)zz$$

  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

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

In our version, all base types are allowed to have side effects, so we include terms such as

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

in the type `nat`.

Interference control

The typing rule for application imposes Reynolds's constraint:

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

Here Γ and Δ must be disjoint contexts. In logical terms, this is an *affine* type system.

Semantics of IA and SCI

There has been considerable research into the semantics of these languages:

- classical Milne-Strachey models with global store
- functor category models [Reynolds, Oles, O'Hearn, Tennent]
- refined models incorporating parametricity and other logical relations [O'Hearn, Tennent, Sieber]
- operationally-based reasoning techniques [Pitts]
- "event-based" models [Reddy, Abramsky, McCusker]

Event-based models

- Reddy's *object spaces* and the *game semantics* of Idealized Algol are what we term *event-based* models.
- A computation is interpreted as a sequence of observable events of some kind.
- A program is interpreted as a set of these sequences: a "trace set" or "strategy".

How good are the models?

- Reddy's model applies to SCI, and the games model to full IA.
- The game semantics of full IA has been shown to be *fully abstract*. That is, programs are operationally equivalent if and only if they have the same denotation.
- In this talk, we show that Reddy's model of SCI is also fully abstract.
- We also give an alternative presentation of Reddy's model.

A model based on monoids

- We will give an event-based model built from monoids.
- A type will be interpreted as a *monoid of observations*, determining what may be observed of a program.
- The monoid operation corresponds to sequential composition of observations.
- A program will be interpreted as a relation from input observations to output observations.
- In practice, all the monoids will be list monoids!

The category

We construct a category $\mathbf{MonRel}$ as follows.

- Objects are monoids
- A morphism $A \to B$ is a relation between A and B such that

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

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

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

  The last constraint expresses the idea that compound observations can only be produced by producing their sub-observations.

An alternative characterization

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

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

  as morphisms from A to B.
- We can also say that $\mathbf{Rel}$ is $(\mathbf{Set}_{\mathcal{P}})^{\mathrm{op}}\ldots$
- $\mathcal{P}$ works on $\mathbf{Mon}$, and it turns out that

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

Categorical structure of Mon

This characterization lets us translate categorical structure from $\mathbf{Mon}$ to $\mathbf{MonRel}$. $\mathbf{Mon}$ has:

- products, given by pairing
- coproducts, given by a slightly unwieldy construction
- a "powerset" monad which is commutative with respect to $\times$
- a forgetful functor to $\mathbf{Set}$ with a left-adjoint given by the list functor.

Categorical structure of MonRel

This allows us to establish certain facts about $\mathbf{MonRel}$:

- products in $\mathbf{Mon}$ give a monoidal structure on $\mathbf{Mon}_{\mathcal{P}}$, which in turn gives one on $\mathbf{MonRel}$.
- coproducts in $\mathbf{Mon}$ give coproducts in $\mathbf{Mon}_{\mathcal{P}}$ and hence products in $\mathbf{MonRel}$.
- the freeness of the list functor allows us to construct certain exponentials.

Exponentials

Let C be any set and A and B be monoids.

$$\begin{aligned}
& (\mathbf{Mon}_{\mathcal{P}})^{\mathrm{op}}(A \otimes B, C^*) \\
\cong\ & \mathbf{Mon}(C^*, \mathcal{P}(A \times B)) \\
\cong\ & \mathbf{Set}(C, \mathcal{P}(UA \times UB)) \\
\cong\ & \mathbf{Rel}(C, UA \times UB) \\
\cong\ & \mathbf{Set}(UA \times C, \mathcal{P}UB) \\
\cong\ & (\mathbf{Mon}_{\mathcal{P}})^{\mathrm{op}}(B, (UA \times C)^*)
\end{aligned}$$

Hence $(UA \times C)^*$ gives an exponential $A \multimap C^*$.

Semantics of types

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

  $$[\![\mathtt{nat}]\!] = \omega^*$$

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

  $$[\![\mathtt{comm}]\!] = 1^*$$

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

Semantics of types

- At type $\mathtt{var}$, there are two kind of event: reading from a variable, and writing to a variable, so

$$[\![\mathtt{var}]\!] = \{\mathsf{read}(n), \mathsf{write}(n) \mid n \in \omega\}^*$$

- The exponential in $\mathbf{MonRel}$ can now be used to interpret the function type of SCI. All types are interpreted as list monoids.

Some examples

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

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

Some examples

After λ-abstraction, the program

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

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

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

Remarks

The semantic element

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

illustrates some important facts about our 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 $n = m$... yet.

Semantics of new

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

$$((s_1, *)(s_2, *) \cdots (s_k, *), *)$$

  where each s_i is a sequence of $\mathsf{read}(n)$ and $\mathsf{write}(n)$ events.
- Call such a sequence a *cell-trace* if reads and writes 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

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

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

Another example

The program

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

is modelled as a relation with elements like:

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

Abstracting on z gives a relation with elements like:

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

Example, continued

Relational composition with the semantics of new forces $n = n''$, yielding

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

which is the semantics of

$$\mathsf{new}\ z\ \mathsf{in}\ z := !x; x := !y; y := !z$$

Operational notions

- SCI can be given an operational semantics in the standard fashion, yielding a relation of the form

$$s, M \Downarrow s', M'$$

  which says that term M in state s evaluates to term M' in new state s'.
- A notion of *program equivalence* can be defined: $M \cong M'$ iff for all contexts $C[-]$ such that $C[M]$ and $C[M']$ are closed terms of type `comm`,

$$C[M] \Downarrow \iff C[M'] \Downarrow .$$

Correctness

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

$$[\![M]\!] = [\![M']\!] \Rightarrow M \cong M'.$$

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

$$M \cong M' \Rightarrow [\![M]\!] = [\![M']\!].$$

- The proof hinges on a *definability* result: we show that the part of the model in the image of $[\![-]\!]$ 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

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

  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

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

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

  such that if $(s, a') \in [\![\mathsf{produce}(a)]\!]$ and s is a cell-trace with a certain initial and final value, then $a = a'$.
- The variable y is typically used to count the total number of accesses of the term, so that a need not be a constant sequence.

Base type tests

Consider the element $1 \cdot 2 \cdot 3 \in [\![\mathtt{nat}]\!]$.
The test for this element is:

$$\begin{array}{l} \text{if } x = 1 \text{ then} \\ \quad \text{if } x = 2 \text{ then} \\ \quad\quad \text{if } x = 3 \text{ then skip} \\ \text{else diverge.} \end{array}$$

Base type producers

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

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

Higher type definability

- To illustrate the proof at higher types, consider $A \multimap \mathtt{comm}$. A typical element of this type is a sequence of pairs $(a, *)$ where $a \in A$.
- We show how to test for and produce the singleton sequences of this kind.

Testing higher types

A test for $(a, *)$ must be a term

$$x : A \multimap \mathtt{comm} \vdash M : \mathtt{comm}.$$

It works by

- calling x once: this demands the single $*$ event of $(a, *)$
- supplying $\mathsf{produce}(a)$ as argument to x: this means that the function x can receive a as input
- checking that the appropriate state transitions for $\mathsf{produce}(a)$ take place: this ensures that x has received all of a, and no more.

Testing higher types

Here's the code:

$$\begin{array}{l}\text{new } y := 0 \text{ in} \\ \quad x(\text{produce}(a)); \\ \quad \text{if } !y = 3 \text{ then skip else diverge.}\end{array}$$

(assuming 0 and 3 are the appropriate initial and final values for y).

Producing higher types

To produce $(a, *)$ is merely to test for a, so we can define

$$\text{produce}(a, *) = \lambda x.\text{test}(a).$$

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

Theorem: The model of SCI in $\mathbf{MonRel}$ is fully abstract.

Conclusion

- We have reconstructed Reddy's model of SCI using a simple category of monoids and relations.
- By proving a definability result, we have shown that Reddy's model is fully abstract.
- Thus, Reddy's model was the first ever fully abstract model of a higher-order imperative programming language, and nobody knew it until now!
- Ongoing work aims to extend this result to handle more sophisticated interference-controlled languages, using the more detailed models of game semantics.

References