A Graph Model for Imperative Computation

Guy McCusker
Semantics via Universal Domains

• In *Data Types as Lattices [1976]*, Scott describes a model of untyped \( \lambda \)-calculus based on *continuous endofunctions* on the lattice \( \mathcal{P}_\omega \).

• Using this model as a *universal domain*, a semantic universe for typed computation can be developed.

• Solving recursive domain equations is particularly simple in this setting.

• A similar model due to Plotkin yields a familiar semantic universe of domains and continuous functions.
Trying again

- Scott’s model, and Plotkin’s refinement, are tailored to functional computation.
- It is possible to model imperative programming using continuous functions on domains, but the basic semantic entities are state-free.
- Can we develop a semantic universe of “stateful” entities using techniques similar to those of Scott?
Credit

• The model we end up with is concretely the same as Reddy’s *object spaces* model [1996], although the construction is completely different.

• Lots of the ideas in this talk were inspired by recent work and writings of John Longley.

• This work started life as a kind of “squashed game model”, suggested by Martin Hyland.
Scott’s graph model

- Consider the continuous functions from $\mathcal{P} \omega$ to itself.
- By continuity, any such $f$ is determined by its action on finite sets.
- Therefore, $f$ is determined by the set
  \[ \{(S, n) \mid S \subseteq_{\text{fin}} \omega, n \in f(S)\} \]
  called the graph of $f$.
- One can encode pairs $(S, n)$ as naturals. Any such encoding leads to a retraction
  \[ \mathcal{P} \omega \to \mathcal{P} \omega \sqsubseteq \mathcal{P} \omega \]
Curried functions

• By repeatedly decoding the output, an element of the graph model can be seen as a curried function:

\[
\begin{align*}
n &= \text{code}(S_1, n_1) \\
    &= \text{code}(S_1, \text{code}(S_2, n_2)) \\
    &= \text{code}(S_1, \text{code}(S_2, \text{code}(S_3, n_3))).
\end{align*}
\]

• Thinking along these lines, a term with three free variables is interpreted as a set of tuples

\[(S_1, S_2, S_3, n_3).\]
A silly example

- Using the trivial encoding of natural numbers, the term \( x, y \vdash x + y \) will have in its graph things like
  \[
  (\{3\}, \{5\}, 8).
  \]

- Contraction (variable sharing) boils down to taking union of the input sets, so \( x \vdash x + x \) contains things like
  \[
  (\{3, 5\}, 8)
  \]

- Our choice of encoding means that we admit nondeterminism at the type of natural numbers. This will come up again later.
Higher order functions

• Applying the decoding to the input side gives rise to higher-order functions. For example, the term

\[ f \vdash f(0) \]

will contain entries like

\[ (\{\text{code}([0], 3)\}, 3). \]

• The term \( f, g \vdash f(0) + g(1) \) will contain entries like

\[ (\{\text{code}([0], 3)\}, \{\text{code}([1], 2)\}, 5). \]

• Note that this is the same as \( g(1) + f(0) \): evaluation order is ignored.
Multiplicity

- A term like $f(0) + f(0)$ will have entries like

$$\text{\{\text{code(\{0\}, 3), code(\{0\}, 3)}\}, 6} = \text{\{\text{code(\{0\}, 3)}\}, 6}$$

in its denotation.

- There is nothing in the model to tell us that $f$ has been called twice here: $f(0) + f(0)$ looks the same as $2 \times f(0)$.

- This is exactly right for side-effect-free computation, but can be badly wrong for imperative programs.
Moving to typed computation

- The untyped model gives rise to a model of typed computation via the Karoubi envelope construction.

- We first construct a monoid as follows. The carrier is the collection of sets of graph-elements \((S, n)\) as used above. Think of these as denotations of terms in a single free variable.

- The monoid operation is defined as follows:

\[
\alpha \cdot \beta = \left\{ \left( \bigcup_{i} S_i, n \right) \mid \exists (S_1, n_1) \ldots (S_k, n_k) \in \alpha, \left\{ n_1 \ldots n_k \right\}, n) \in \beta \right\}
\]

This corresponds to substitution.
**Example**

Consider the monoid element corresponding to $f \vdash f(0) + f(1)$, which contains entries like

$$(\{\text{code}\{\{0\}, n\}, \text{code}\{\{1\}, m\}\}, n + m)$$

and the element corresponding to $x \vdash \lambda y. x + y$, with entries like

$$(\{p\}, \text{code}\{\{q\}, p + q\}).$$

Composing gives an element with entries like

$$(\{n, m - 1\}, n + m).$$
The Karoubi Envelope

- The Karoubi envelope is obtained by splitting idempotents in this monoid.
- Objects of the Karoubi envelope are elements $A$ such that $A \cdot A = A$.
- Maps $A \rightarrow B$ are elements $\alpha$ such that

$$A \cdot \alpha \cdot B = \alpha.$$ 

- Since we began with a model of $\lambda$-calculus, this yields a cartesian closed category. See Lambek & Scott for details.
To model imperative computation, we need at least to capture the order and multiplicity of events.

Why not try moving from sets of input-observations to sequences?

Define a monoid as follows. The carrier is the collection of sets of pairs \((s, n)\) where \(s\) is a sequence of naturals, and the monoid operation is defined as

\[
\alpha \cdot \beta = \{(s_1 \ldots s_k, n) | \exists (s_1, n_1) \ldots (s_k, n_k) \in \alpha, ([n_1 \ldots n_k], n) \in \beta\}
\]
A model of affine $\lambda$-calculus

- This setup turns out to be good enough to model affine $\lambda$-calculus.
- The model is again based on an encoding of graph-elements $(s, n)$ as natural numbers.
- The Karoubi envelope gives a model of imperative computation.
Why only affine?

- Suppose a term $x \vdash f$ includes the element
  
  \[
  ([0], \text{code}(([1], 2))
  \]

  and $x \vdash a$ includes $([1], 1)$.

- Ignoring the $x$-part, we see that $f$ can take input 1 and return 2, while $a$ can give 1. Hence $f(a)$ should be able to return 2.

- However, the $x$-parts show that $f$ needs to see 0 from $x$, and $a$ needs to see 1 from $x$ in order to produce this behaviour.
Why only affine?

• In the set-based model, this is not a problem: the denotation of $x \vdash f(a)$ would contain $(\{0, 1\}, 2)$.

• In the sequence-based model, however, we cannot just use union: we need to decide in what order the $x$-events happen. We do not have enough information to do this.

• Thus we can only model multiplicative application: a function and its argument cannot have any variables in common.
**Products**

- We can recover some variable-sharing abilities via a *pairing* construction. Define $\langle \alpha, \beta \rangle$ to be

  \[
  \{(s, 2n) \mid (s, n) \in \alpha \} \cup \{(s, 2n + 1) \mid (s, n) \in \beta \}.
  \]

- First projection is given by

  \[
  \{([2n], n) \mid n \in \omega \}
  \]

  and second projection similarly.

- Operations such as addition can now be interpreted using, for example,

  \[
  \{([2n, 2m + 1], n + m) \mid n, m \in \omega \}.
  \]
**Evaluation order**

- A term like $f(0) + f(1)$ will have elements like this in its denotation:

  $$([\text{code}([0], 1), \text{code}([1], 2)], 3).$$

- Note that the left-to-right evaluation order is recorded here.

- The denotation of $f(0) + g(1)$ contains things like

  $$([\text{code}(([0], 1)), [\text{code}(([1], 2))], 3)$$

  and $g(1) + f(0)$ is the same: evaluation order between different variables is ignored.
The exciting bit

So far, all we have done is to break Scott’s model! We will now show what we have gained: we can model

- assignment and dereferencing of variables
- basic imperative constructs like sequential composition, while-loops and so on
- block-scoped allocation of variables.
Imperative variables

- The key things one can do with an assignable variable are write a value into it, and read a value out of it.
- Let us associate these actions with natural numbers via encodings $\text{write}(n)$ and $\text{read}(n)$ with disjoint images.
- A typical graph element can now be seen as something like

$$([\text{write}(0), \text{read}(3), \text{read}(2)], 7).$$
**Simple imperative programming**

- Using 0 to signal termination, a command $x \leftarrow x := 3$ can be interpreted as
  $$\{([\text{write}(3)], 0)\}.$$

- Dereferencing is easy to model, too: $x \leftarrow !x$ is interpreted as
  $$\{([\text{read}(n)], n) \mid n \in \omega\}.$$
**Sequential composition**

- Concretely, sequential composition is interpreted by concatenation of input sequences.
- If $x \vdash C_1$ contains $(s_1, 0)$ and $x \vdash C_2$ contains $(s_2, 0)$ then $x \vdash C_1; C_2$ contains $(s_1s_2, 0)$.
- We can give a compositional account of this as follows. The sequential composition operator has the graph

$\{(0, 1), 0\} = \{([2 \times 0, (2 \times 0) + 1], 0)\}$

Composing this with the pairing $\langle C_1, C_2 \rangle$ gives $C_1; C_2$.
- We are using pairing for sequential operations whose subterms may share variables, and curried functions for operations whose subterms cannot share variables.
**Variable allocation**

- The program $x \vdash x := 0; x := !x + 3; \text{return}(!x)$ will contain entries like
  \[
  ([\text{write}(0), \text{read}(n), \text{write}(n + 3), \text{read}(m)], m).
  \]
- Nothing forces $n$ to be 0 here, or $m$ to be $n + 3$.
- To interpret imperative programming, we want to restrict our attention to those sequences where reads and writes match up in the expected way.
- Say that a sequence $s$ of reads and writes is a *cell trace* if every $\text{read}(-)$ matches the most recent $\text{write}(-)$.
Define a constant \( \text{alloc} \) with interpretation

\[ \{([\text{code}(s,n)], n) \mid s \text{ is a cell trace}\} \]

Given a term \( x \vdash C \), the term \( \vdash \text{alloc}(\lambda x. C) \) will contain \( n \) iff \( C \) has an element \((s, n)\) where \( s \) is a cell-trace.

Thus, applying \( \text{alloc} \) makes \( x \) into a locally allocated good variable!
An example

• The denotation of the term

\[ x \vdash x := 0; x :=! x + 3; \text{return}(!x) \]

contains all pairs of the form

\[ ([\text{write}(0), \text{read}(n), \text{write}(n + 3), \text{read}(m)], m) \].

• The only such entry in which the input-side sequence is a cell-trace is

\[ ([\text{write}(0), \text{read}(0), \text{write}(3), \text{read}(3)], 3) \]

• Thus \( x \vdash \text{alloc}(\lambda x. x := 0; x :=! x + 3; \text{return}(!x)) \) contains only the number 3, as you would expect.
A programming language

If we put together everything we can model so far, we obtain an untyped programming language which contains

• the affine $\lambda$-calculus
• numerals and arithmetic constants
• basic imperative constructs: assignment, dereferencing, sequential composition and while-loops
• local variable allocation.

This is an untyped version of the language at the heart of Reynolds’s *Syntactic Control of Interference*. 
A Type System

- The SCI language is obtained by imposing a type system on this programming language.
- The base types are `comm` for commands, `nat` for natural numbers, and `var` for assignable variables.
- The only type constructor is `→`, the linear (affine) function space.
A Type System

• The “no variable sharing” restriction on application rule is enforced by means of a multiplicative typing rule:

\[
\Gamma \vdash M : A \rightarrow B \quad \Delta \vdash N : A \\
\Gamma, \Delta \vdash MN : B
\]

• Constructs which use their operands sequentially are given additive typing rules:

\[
\Gamma \vdash M : \text{comm} \quad \Gamma \vdash N : \text{comm} \\
\Gamma \vdash M; N : \text{comm}
\]
**Operational semantics**

- This language can be given an operational semantics in the usual way.
- The semantics is based on *stores* $\sigma$ which map $\text{var}$-typed variables to natural number values.
- Judgements of the form
  \[
  \sigma, M \Downarrow \sigma', \text{skip}
  \]

  are defined inductively.
- One can then go on to define equivalence of terms in the standard fashion: $M \equiv N$ iff $M$ and $N$ give the same results in all closed contexts.
Denotational Semantics

- We will give this language a denotational semantics in the Karoubi envelope of our imperative graph model.
- The semantics of terms is as we have already outlined, so we just need to explain the semantics of types.
- A type is interpreted as an object, that is to say, as an idempotent of the monoid.
Semantics of types

- Base types are interpreted using subsets of the “identity” relation

\[ \{ ([n], n) \mid n \in \omega \} . \]

The chosen subset is the set of naturals which encode valid events in the given type.

- \([\text{nat}]\) is the whole relation: we encode numbers in a trivial way.

- \([\text{comm}]\) admits only 0: a term of type \(\text{comm}\) can only produce 0 to signal termination.

- \([\text{var}]\) admits all the \(\text{read}(n)\) and \(\text{write}(n)\) events.
**Function types**

Function types are encoded in the standard manner.

\[ [A \rightarrow B] \text{ is given by the monoid element interpreting} \]

\[ x \vdash \lambda y.[B](x([A](y))). \]
Some categorical structure

- This \( \circ \) construction gives an *internal hom* in the Karoubi envelope.
- Unfortunately, \( A \circ \) does not possess a left adjoint: the category is not monoidal closed.
- Nevertheless there is enough structure to interpret the affine typed \( \lambda \)-calculus and all the imperative constants.
- There is a different category which the same model inhabits which *is* symmetric monoidal closed—phew! Click [here](#) to see it.
Exploiting the universal object

- The Karoubi envelope has a *universal object*: the object corresponding to the identity element of the monoid.

- *Universal* means that every object is a retract of this object.

- The universal object is the denotation of the type `nat`. Can we get anything useful from this?
Soundness and full abstraction

• It is possible to prove a soundness theorem saying that closed terms of type \texttt{nat} are equivalent if and only if they have equal denotation.

• It is desirable for the same to be true for all types; this property is called full abstraction.

• The universal object gives us an approach to proving this: if we can show that the retractions are definable in the programming language, then we are done!
Definable retracts implies full abstraction

Here’s why.

• Suppose $M$ and $N$ are equivalent terms of type $A$.
• Let $F : A \rightarrow \text{nat}$ be the term whose denotation is the section-part of the retraction $[A] \subseteq [\text{nat}]$.
• $FM$ and $FN$ are therefore equivalent terms of type $\text{nat}$.
• By soundness, $[FM] = [FN]$.
• Since $[F]$ has a left-inverse, this means that $[M] = [N]$.
• The converse only requires compositionality of the semantics.
Definability of retracts

- Sadly, the retractions are not definable in the language as it stands.
- However, with two extensions, they become definable.
- The extensions we need are:
  - nondeterminism: erratic choice, random assignment, a coin flip or a random number generator will do
  - a bad-variable constructor.
Defining retractions

The most interesting retraction is \([\text{nat} \to \text{nat}] \trianglelefteq [\text{nat}]\). The section part of this looks like:

\[
\{(\text{code}(s, n), \text{code}(s, n)) \mid s \in \omega^*, n \in \omega\}.
\]

It is defined as follows.

\[
\lambda f. \text{new } x := \text{emptyseq in}
\]

\[
\text{let } y = f(\text{let } z = \text{random in}
\]

\[
x := \text{append}(!x, z); \\
\text{return}(z)
\]

\[
\text{in code}(!x, y)
\]
Dealing with var

For type \( \text{var} \), we need a term \( x : \text{nat} \vdash M : \text{var} \) whose denotation is

\[
\{ ([\text{write}(n)], \text{write}(n)), ([\text{read}(n)], \text{read}(n)) \mid n \in \omega \}.
\]

A variable can be considered as a simple \textit{object} with two methods: a reading method and a writing method:

\[
\begin{align*}
\text{read} & = \lambda x. !x : \text{var} \to \text{nat} \\
\text{write} & = \lambda x. \lambda n. x := n : \text{var} \to (\text{nat} \to \text{comm}).
\end{align*}
\]
A bad-variable constructor

The required map \( \text{nat} \rightarrow \text{var} \) can be defined if we add to our language a constant \( \text{mkvar} \) with typing rule

\[
\Gamma \vdash M : \text{nat} \quad \Gamma \vdash N : \text{nat} \rightarrow \text{comm}
\]

\[
\Gamma \vdash \text{mkvar} \; M \; N : \text{var}
\]

such that

\[
\text{read}(\text{mkvar} \; M \; N) = M
\]

\[
\text{write}(\text{mkvar} \; M \; N) = N.
\]

Thus \( \text{mkvar} \) is the constructor corresponding to the destructors \( \text{read} \) and \( \text{write} \).
Full abstraction and universality

With these additions to the language, the retraction $A \subseteq \mathcal{P}\omega$ is definable for every type $A$. It follows that:

- the model is fully abstract for SCI + nondeterminism + mkvar
- the model is universal: every recursive element of the model is definable by a term.
**Conservativity**

- It turns out that the additions of nondeterminism and mkvar are *conservative extensions* of the basic language.

- This means that for any terms $M_1$ and $M_2$ of the basic language, if there is a context $C[\_]$ in the extended language such that

  \[
  C[M_1] \downarrow \text{skip} \quad C[M_2] \not\downarrow
  \]

  then there is a context $C'[-]$ in the basic language with the same property.

- It follows that the model is also fully abstract for the basic language.
Conservativity of nondeterminism

• Suppose \( C[-] \) is a context employing nondeterminism such that \( C[M_1] \Downarrow \) but \( C[M_2] \nmid \).

• In the course of evaluating \( C[M_1] \), the nondeterminism is resolved in a particular way.

• Create \( C'[\_] \) by replacing all nondeterministic choices in \( C[\_] \) with appropriate predetermined choices. This requires the use of state to remember which choice to make next.

• \( C'[M_1] \) still converges, but \( C'[M_2] \nmid \).
Conservativity of \texttt{mkvar}

- This is much harder to prove.
- It seems to be rather anomalous: minor changes to the basic language (such as moving to call-by-value, or eliminating side-effecting numeric expressions) render it false.
- \texttt{mkvar} is only a conservative extension for observational \textit{equivalence}: the observational \textit{preorder} is altered by the inclusion of \texttt{mkvar}.
- For example, without \texttt{mkvar} we have:

\[
\begin{align*}
\text{if } !x = 3 \text{ then skip else diverge} \\
\sqsubseteq x := 3
\end{align*}
\]
Conclusions

• A simple alteration to Scott’s graph-model yields a model of imperative programming.
• The resulting typed model can be seen as a category of monoids and relations.
• The universal object in this model gives a very easy approach to proving full abstraction.
• This is the first full abstraction result for an interference-controlled language.
Appendix: A category of monoids

All our work can be couched in terms of a category of monoids and relations.
A graph consists of a set of entries \((s, n)\). Such a set can be seen as a function

\[
f : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N}^*)
\]

Viewing \(\mathcal{P}(\mathbb{N}^*)\) as a monoid, this is the same as a homomorphism

\[
f^* : \mathbb{N}^* \rightarrow \mathcal{P}(\mathbb{N}^*)
\]

The monoid operation turns out to be the same as composition in the category \((\text{Mon}_\mathcal{P})^{\text{op}}\).
Structure of \((\text{Mon}_\mathcal{P})^{\text{op}}\)

The relevant structure of this category is easy to establish.

- The product in \(\text{Mon}\) yields a symmetric monoidal structure.
- The coproduct in \(\text{Mon}\) yields products.
- The Kleene monoids \(A^*\) form an exponential ideal.
Exponentials in \((\text{Mon}_\mathcal{P})^{\text{op}}\)

\[
\begin{align*}
(\text{Mon}_\mathcal{P})^{\text{op}}(A \otimes B, C^*) \\
\text{Mon}(C^*, \mathcal{P}(A \times B)) \\
\text{Set}(C, \mathcal{P}(UA \times UB)) \\
\text{Set}(UA \times C, \mathcal{P}UB) \\
(\text{Mon}_\mathcal{P})^{\text{op}}(B, (UA \times C)^*)
\end{align*}
\]
A direct presentation

This category can be presented directly as follows.

- Objects are monoids.
- A map $A \rightarrow B$ is a relation $R$ satisfying the following three conditions:
  - **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 $a = a_1 a_2$. 