Some Notes from the First Meeting

Natalie Ravenhill

July 14, 2016

An example of a grammar for types is the following:

$$A ::= \mathrm{A} \to \mathrm{B} \mid \mathrm{Nat}$$

We can form the type Nat and functions between Nats (eg. Nat → Nat, (Nat → Nat) → Nat, etc..)

Note that there are a countable number of programs we can write, but there are an uncountable number of functions that we can write between the natural numbers (and they may not all be expressible as programs).

1 Semantics of typing contexts

Given a context Γ and an expression e of type A, we have the following semantics, which denotes a function from the denotation of the context to A:

$$[\![\Gamma \vdash e : A]\!] \in [\![\Gamma]\!] \to A$$

The typing context is represented by a tuple of size n, where n is the number of terms in the context:

$$[\![x_1 : A_1,\ \dots\ , x_n : A_n \vdash e : B]\!] \in A_1 \times \ldots \times A_n \to B$$

To get a function from a single element:

We also get $e \mapsto e'$ by using a reduction relation.

2 Adequacy

Adequacy is a property that will only hold on ground types and is the same as soundness and completeness. In our grammar the ground types are natural numbers, so adequacy is phrased as:

Adequacy 1. *If* $\vdash e : Nat$ *(ie. e is a closed term of type Nat) and* $[\![e]\!] = n$ *then* $[\![e]\!] = n \leftrightarrow e \mapsto^* \underline{n}$

For higher types, we cannot easily prove this. If Adequacy holds for all types that we have *Full Abstraction*

Adequacy cannot be proved by induction, so we use a new technique called logical relations.

3 Logical Relations

Logical relations are a proof technique for higher order frameworks and have been used to prove definability problems, type safety and strong normalisation. There are two types

1. Unary relations
   Have a single expression with a property P(e) we are interested in. These are built out of types.
   Using *unary* relations you can prove the strong normalisation of Simply Typed Lambda Calculus and Type Safety (especially of non-terminating languages).
2. Binary relations
   Have two terms with the same type that are possibly related and a property P(e, e')
   Using *binary* relations you can show equivalence of programs, which has applications in showing the correctness of compiler optmisations. You can also prove non-interference in security (that for the same low security level inputs and different high level inputs we get low level outputs that look the same).

To prove adequacy for our language there are three things we must do:

3.1 Define logical relations on closed terms of our types

In our language, we have a unary logical relation for each type. For natural numbers we have

$$R_{Nat} = \{e \mid \vdash e : Nat \land ([\![e]\!] = n \leftrightarrow e \mapsto^* \underline{n})\}$$

This relation contains all closed terms of type Nat for which adequacy holds.

For function types we have the following logical relation:

$$R_{A \to B} = \{e \mid \vdash e : A \to B \land (\forall e' \in R_A.\ (e\ e') \in R_B\}$$

3.2 Define the logical relation for the open terms

The relation for any context Γ is the following, defined for the empty case and the inductive case, where we are adding a new value to the context:

$R_\Gamma =$

$$R_\bullet = \{\bullet\}$$

$$R_{\Gamma, x:A} = \{(\gamma, e/x) \mid \gamma \in R_\Gamma \wedge e \in R_A\}$$

$\bullet$ is the empty context that contains no values, so its relation just gives a set containing the empty substitution.

In the inductive case, γ is a well typed assignment of values for variables. For example we could have:

$$\gamma = 3/x, \lambda x.x + 2/y, 4/z$$

Therefore the relation relates γ which is a valuation that holds for the original context to the substitution where e substitutes x if e is in the type A's relation.

3.3 Prove the Fundamental Theorem

Fundamental Theorem 1. *If* $\Gamma \vdash e : A$ *and* $\gamma \in R_\Gamma$ *then* $\gamma(e) \in R_A$

Which says that if we have a context Γ and a term $x : A$ and we have a valuation γ for the context Γ, which is in the logical relation of Γ, then it must be the case that $\gamma(e)$ is in the relation for this type A, as R_A represents all possible terms of A.

4 Domains

Domains are used to model programs.

Here is the syntax of PCF:

$$\begin{array}{l} A ::= \text{Nat} \mid A \to B \\ e ::= \lambda x : A.e \mid e\ e \mid x \\ \mid z \mid s(e) \mid case\ (e,\ z \to e_0\ ,\ s(n) \to e_s) \\ \mid \text{fix}\ x : A\ .\ e \end{array}$$

x represents identifiers, z is zero and $s(e)$ is the successor function for numbers.

fix is the fixpoint combinator and is used for recursive functions. If we unfold the combinator we get the following:

$$(fix\ x\ e) \mapsto [(fix\ x\ e)/x]e \mapsto [(fix\ x\ e)/x]e \mapsto \dots$$

The following is an example of recursive definition of double:

$$\begin{aligned}&double \;\rightarrow\; fix\; double:\; Nat \;\rightarrow\; Nat\\ &\lambda n:\text{Nat}\;.\;case\;(n,\\ &z \rightarrow z\\ &s(x) \rightarrow s(s(double\; x)))\end{aligned}$$

Given a number we have successor of successor of *double* $(n-1)$.

We can model this function using a domain. A domain is a 4-tuple $(X, \bot \in X, \sqsubseteq, \sqcup)$, which represents the following:

- X is the set of computations of a certain type. For example, in double, we have different amounts of unfolding attempts before the program just gives up and goes into an infinite loop
- $\bot$ is the element of X that gives us the least information about the program. This is usually the behaviour where the program just goes into an infintite loop without doing anything
- $\sqsubseteq$ is a partial order on the set of programs, where programs with more information are higher than those with less information
- $\sqcup$ is the limit of the domain, which is the least upper bound of it

As a simple example, we can model the program

$$fix\; x.F(x)$$

which is the program that just infinitely applies F to an argument x. We get the following chain:

$$\bot \sqsubseteq F(\bot) \sqsubseteq F(F(\bot)) \sqsubseteq F(F(F(\bot))) \sqsubseteq \ldots \sqcup F$$

$\bot$ represents an infinite loop, $F(\bot)$ applies the function once and goes into an infinite loop, $F(F(\bot))$ applies F twice before going into an infinite loop, and so on...

A *chain* is just a sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ where every element is larger than the preceding one. Fromally we define a chain as a function $\mathbb{N} \rightarrow X$ such that if $i \leq j$ then $x_i \leq x_j$.

So for double we have

$$double_0\; n = double_0\; n$$

which is just the infinite loop. Then we can use this $double_0$ for the computation where we only iterate once:

$$double_1\ z = z$$

$$double_1\ s(x) = s(s(double_0))$$

This does successor of succesor of double 0, so only does one iteration of our original function, before going into the infinite loop defined by $double_0$. Then we use $double_1$ to define $double_2$:

$$double_2\ z = z$$

$$double_2\ s(x) = s(s(double_1))$$

Which does an iteration, then calls $double_1$, which does an iteration of double and calls $double_0$ which loops forever. Therefore $double_2$ does 2 iterations of *double* then loops. We can keep defining double for more iterations in this way, using the $double_{(n-1)}$ function, giving us the following chain:

$$double_0 \sqsubseteq double_1 \sqsubseteq double_2 \sqsubseteq \ldots \sqcup double$$

4.1 Maps between domains

If we have two domains then we can write a map $f : (X, \perp_X \sqsubseteq_X) \rightarrow (Y, \perp_y, \leq_y)$ between them. There are two properties that may hold for this map:

Monotonicity For two computations x and x' in X, if $x \sqsubseteq_X x'$ then $f(x) \leq_Y f(x')$. This means that a more defined argument gives a more defined result.

For a chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqcup \underset{x}{\rightharpoonup}$, by monotonicity we have $f(x_0) \leq f(x_1) \leq f(x_2) \leq \ldots f(\sqcup \underset{x}{\rightharpoonup})$.

Continuity Continuity says that $f(\sqcup \underset{x}{\rightharpoonup}) = \sqcup f(\underset{x}{\rightharpoonup})$.

For example, if we have the chain $(\underset{x}{\rightharpoonup}) = \perp \sqsubseteq \perp \sqsubseteq 2 \sqsubseteq 2 \sqsubseteq 2 \ldots$, by monotonicity, $double\ (\underset{x}{\rightharpoonup}) = \perp \sqsubseteq \perp \sqsubseteq 4 \sqsubseteq 4 \sqsubseteq 4 \ldots$.

We can see that $double(\sqcup(\underset{x}{\rightharpoonup})) = 4$, so by continuity $\sqcup double(\underset{x}{\rightharpoonup}) = 4$

4.2 Examples of Domains

Natural Numbers The domain of natural numbers produces either a natural number, or an infinite loop denoted by $\perp$.

$$\begin{tikzcd}
0 \arrow[dr, no head] & 1 \arrow[d, no head] & 2 \arrow[dl, no head] & \cdots \arrow[dll, no head] \\
 & \bot & &
\end{tikzcd}$$

We have the following chains $\bot \sqsubseteq 1 \sqsubseteq 1 \sqsubseteq \ldots$, $\bot \sqsubseteq 2 \sqsubseteq 2 \sqsubseteq \ldots$, $\bot \sqsubseteq 3 \sqsubseteq 3 \sqsubseteq \ldots$, etc. Note that there is no ordering between the numbers themselves, just that they give more information than $\bot$.

Functions The domain of continuous functions is the tuple $(X \to Y, \bot_{X \to X}, \sqsubseteq_{X \to Y})$ where

- $X \to Y$ is the set of continous functions between a set X and a set Y
- $\bot_{X \to Y}$ is the function $\lambda x.\bot y$ that loops on all inputs
- $\sqsubseteq_{X \to Y}$ is the ordering between functions defined from X to Y. For some $f, g : X \to Y$, $f \sqsubseteq g \leftrightarrow g \forall x.f(x) \leq_Y g(x)$

4.3 Proving something is a domain

Given a set X, an element $\bot \in X$ and a relation $\sqsubseteq \subseteq X \times X$ we must prove

- $\forall x \in X.\bot \sqsubseteq X$, that is $\bot$ is below every other element in the set X
- that $\subseteq$ is a partial order. For this we must prove that it is reflexive, antisymmetric and transitive, which are the following properties:
  - $\forall x \in X, x \sqsubseteq x$
  - $\forall x, y \in X.x \sqsubseteq y \land y \sqsubseteq x \to x = y$
  - $\forall x, y, z \in X.x \sqsubseteq y \land y \sqsubseteq z \to x \sqsubseteq z$
- For all chains $\underset{x}{\rightharpoonup}$, $\underset{x}{\rightharpoonup}$ has a limit. To prove this we must prove when $\exists z \in X$:
  - $\forall i.x_i \sqsubseteq z$
  - $\forall y.(if\ \forall i.x_i \sqsubseteq y)\ then\ z \sqsubseteq y$