1. Denotational Semantics

A direct denotational semantics of a programming language $L$ is defined by:

1. View the syntax of $L$ as an initial $\Sigma_L$-algebra $T_{\Sigma_L}$, and

2. Define a $\Sigma_L$-algebra structure in the category $\mathcal{P}ow(\omega)$ of $\omega$-cpos and $\omega$-continuous functions, which allows defining recursively constructs in $L$ by means of fixpoint operators. Specify this $\Sigma_L$-algebra, let us call it $D$, amounts to specifying:

\[(2.1)\text{ for each sort } s \text{ of } \Sigma_L \text{ an } \omega\text{-cpo } D_s\]

\[(2.2)\text{ for each } f : s_1 \ldots s_n \rightarrow s \text{ in } \Sigma_L \text{ an } \omega\text{-continuous function } f_D : D_{s_1} \times \ldots \times D_{s_n} \rightarrow D_s\]

Then this defines a unique $\Sigma_L$-homomorphism

$$\delta : T_{\Sigma_L} \rightarrow D$$

called the semantic function.
In terms of the categorical semantics of operational theories, \( \hat{D} \) is a product-preserving functor

\[
\hat{D} : \prod_{\Sigma} \to \mathbb{P}_\omega(\omega), \quad \text{and} \quad \Gamma - D \text{ is the unique } \Sigma \text{-homomorphism in } \mathbb{Alg}(\Sigma, \mathcal{P}) : 
\]

\[
\begin{array}{ccc}
\hat{D} & \to & \mathbb{P}_\omega(\omega) \\
\downarrow & & \downarrow U \\
\prod_{\Sigma} & \to & \text{Set}
\end{array}
\]

where \( U : \mathbb{P}_\omega(\omega) \to \text{Set} \) is the forgetful functor.

One first problem with this approach is that the domains (i.e. \( \omega \)-coalg) \( \hat{D} \) may not be computable data types in any reasonable sense. For example, the domain \( \hat{D}_{\text{IntExp}} \) associated by Reynolds to the sort \( \text{IntExp} \) of integer expressions in his simple language is:

\[
[ [X \to \mathbb{Z}] \to \mathbb{Z} ]
\]

where \( X \) is an infinite set of variables, and we get

\[
\Gamma - D : \prod_{\mathcal{Z}_L} \to [ [X \to \mathbb{Z}] \to \mathbb{Z} ]
\]
TD is a set of huge cardinality. Even the set \([X \rightarrow 2]\) has the power of the continuum.

A second, related problem is that there is then a non-trivial story to tell about how this denotational semantics relates to an operational semantics, that should be defined in terms of computable sets and data structures.

For our purposes here we can cut down the extravagant size of \(TD_{\text{itexp}}\) with no real loss of generality by considering as computable states the set

\([X \rightarrow 2]_{\text{ff}}\)

of finite (partial) functions \(f : X \rightarrow 2\), where \(\text{dom}(f) \subseteq X\) is always a finite set. This is what any memory looks like anyway.

Life gets considerably simplified by assuming that when a variable \(x\) has not been initialized, its value is, by default, 0. This amounts to defining the following function:
\[ -[\_] : \left[ x \mapsto \mathbb{Z} \right] \rightarrow \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \]

where \( f[x] = \text{if } x \in \text{dom}(f) \text{ then } f(x) \text{ else } 0 \text{ end} \).

The signature of Reynolds' simple language is, with minor variations, the one in the functional module SIMPLE in Appendix 5, where INT is imported solely for data generation, i.e., to have integer expressions of the form \( i(m) \) for each \( m \in \mathbb{Z} \), but we do not intend to use the operations of \( \mathbb{Z} \) at all. A more awkward way of avoiding INT would have been to use Peano notation with \( 0, s \) and \( p \) functions and get \( \text{Int} \) as the sort of the term algebra for the signature:

![Diagram of term algebra for Int]

which has only constructors and no defined functions whatever.

We can further cut down the complexity of the denotational semantics of SIMPLE by observing that (as pointed out in the homework) we have an equivalence...
*** What follows defines a parameterized theory of finite functions
*** and instantiates it for finite functions from Var to Int .
*** Finite functions are a simple model of memory for a language
without blocks.

fth EQ is protecting BOOL .
   sort Elt .
   op eq : Elt Elt -> Bool .
   vars X Y : Elt .
   eq eq(X,X) = true .
   ceq X = Y if eq(X,Y) = true [nonexec] .
endfth

fth TRIV* is
   sort Elt .
   op * : -> Elt .
endfth

fmod FUN{X :: EQ, Y :: TRIV*} is
   protecting BOOL .
   sorts Cell{X,Y} Magma{X,Y} Fun{X,Y} .
   subsort Cell{X,Y} < Magma{X,Y} .

   op [_,-] : X$Elt Y$Elt -> Cell{X,Y} [ctor] .
   op mt : -> Magma{X,Y} [ctor] .
   op _,- : Magma{X,Y} Magma{X,Y} -> Magma{X,Y} [ctor assoc comm id:
     mt] .
   op {}- : Magma{X,Y} -> Fun{X,Y} [ctor] .
   op error : -> Fun{X,Y} [ctor] .

   var I J : X$Elt .
   vars D D' : Y$Elt .
   var M : Magma{X,Y} .

   eq {{[I,D],[I,D'],M} = error .

   op dbl : Magma{X,Y} -> Bool .  *** detects double cell [I,D],[I,D']
   eq dbl(mt) = false .
   eq dbl([I,D]) = false .
   eq dbl([I,D],[J,D'],M) = eq(I,J) or dbl([I,D],M) or dbl([J,D'],M) .

   op insert : X$Elt Y$Elt Fun{X,Y} -> Fun{X,Y} .
   ceq insert(I,D,[M,[I,D']]) = {M,[I,D]} if dbl(M,[I,D']) = false .
   ceq insert(I,D,[M]) = {M,[I,D]} if dbl(M,[I,D]) = false .
   eq insert(I,D,error) = error .
op _[_] : Fun{X,Y} X$Elt -> Y$Elt .
  ceq \{M, \{I,D]\}\{I\} = D if \text{dbl}(M,\{I,D\}) = false .
  ceq \{M\}\{I\} = * if \text{dbl}(M,\{I,*\}) = false .
  eq error[I] = * .
endfm

fmod VAR is protecting NAT .
  sort Var .
  op x[_] : Nat -> Var [ctor] .
  op eq : Var Var -> Bool [comm] .
  vars N M : Nat .
  eq eq(x\{N\},x\{M\}) = eq(N,M) .
  eq eq(0,0) = true .
  eq eq(0,s(N)) = false .
  eq eq(s(N),s(M)) = eq(N,M) .
endfm

view Var from EQ to VAR is
  sort Elt to Var .
endv

view Int from TRIV* to INT is
  sort Elt to Int .
  op * to \emptyset .
endv

fmod FUN<VAR/INT> is
  protecting FUN{Var,Int} .
endfm
A simple programming language in Chapter 2 of Reynolds' "Theories of PLs"

syntax of SIMPLE. To avoid confusion between built-in INT and Int expressions,
an integer J is wrapped as the expression i(J). Likewise, a
Boolean B is wrapped
as the Boolean expression b(B). Division is excluded for
simplicity.

load fun1.maude.

fmod SIMPLE is protecting VAR . protecting INT .
  sorts IntExp BoolExp Comm .
  subsort Var < IntExp .


op skip : -> Comm [ctor] .
op _:= : Comm Comm -> Comm [ctor assoc] .
op if_then_else_end : BoolExp Comm Comm -> Comm [ctor] .
endfm
of categories, $\text{Pfa} \simeq \text{Set}_\perp$ when $\text{Set}_\perp = \text{Alg}_{\Sigma_{\perp}}$

in the category of "pointed sets", i.e., of unsorted
$\Sigma_{\perp}$-algebras, where $\Sigma_{\perp}$ counts of the single constant
$\perp$. Since we also have an inclusion of categories
$\text{Set}_\perp \rightarrow \text{Pon}(\mathcal{M})$ when we view each pointed
set $A$ as a "flat cone," so that $\forall a \in A$, $A \leq a$,
we can carry our constructions in the simpler
category $\text{Pfa}$.

Therefore, to give a denotational semantics to
$\text{SIMPLE}$, we need to define sets:

$$\text{D}_{\text{IntExp}}, \text{D}_{\text{BoolExp}}, \text{D}_{\text{Comm}}$$

for the three sorts of $\Sigma_{\text{SIMPLE}}$, and partial functions
$f_D$ (which in some cases are total) for each of the
operations $f \in \Sigma_{\text{SIMPLE}}$. Let us do it. Define:

$$\text{D}_{\text{IntExp}} = \left[[X \rightarrow \mathbb{Z} \gg \mathbb{Z}\right]$$

where we choose as $X$ the set of variables in the module $\text{VAR}$ and
$\mathbb{Z}$ denotes the set of integers.
Define similarly,

$$D_{\text{bool exp}} = \lambda [X \to \mathbb{Z}] \to \mathbb{Z}$$

where $\mathbb{Z} = \{\text{true}, \text{false}\}$.

Finally, define

$$D_{\text{comm}} = \lambda [X \to \mathbb{Z}] \to [X \to \mathbb{Z}]$$

i.e., as the set of partial functions from $[X \to \mathbb{Z}]$ to itself.

As we have simplified these, as much as possible to make them agree with actual practice, note that none of the above sets is countable.

All we need to define $\tilde{T} : T_{\mathcal{L}} \to \mathcal{D}$ is to

specified for each $f \in \sum_{\mathcal{L}}$ the "semantic equation" defining $D_f$:

$$D_x = \lambda \sigma . \sigma [x] \quad \text{for each } x \in X$$

$$D_{(n)} = \lambda \sigma . n \quad \text{for each } n \in \mathbb{Z}$$

$$D_{-} = \lambda \gamma . \lambda \sigma . - \sigma (\gamma (\sigma)) \quad \text{where } \gamma : \mathbb{Z} \to \mathbb{Z} \text{ in minus}$$

$$D_{+} = \lambda \gamma . \lambda \sigma . + \sigma (\gamma (\sigma))$$

$$D_{*} = \lambda \gamma . \lambda \sigma . \ast \sigma (\gamma (\sigma))$$

where $\ast, \ast : \mathbb{Z}^2 \to \mathbb{Z}$ are addition and multiplication, and where $\sigma : [X \to \mathbb{Z}]$, $\gamma, \gamma_1, \gamma_2 : D_{\text{bool exp}}$. 
\[ D_{\text{true}} = \lambda \sigma. \text{true} \quad D_{\text{false}} = \lambda \sigma. \text{false} \]

\[ D_{\text{AND}} = \lambda (s_1, s_2), \lambda \sigma. s_1(\sigma) \text{ and } s_2(\sigma) (\text{likewise for } \text{OR}). \quad D_\&(\sigma) = \lambda (s_1, s_2), \lambda \sigma. s_1(\sigma) \mathbin{\&} s_2(\sigma) \]

\[ D_{\text{Ski}} = 1 \quad [x \mapsto Z]_{\text{ff}} \]

\[ D_j = \lambda \langle t_1, t_2 \rangle. \lambda \sigma. t_2(t_1(\sigma)) \]

\[ D_\exists = \lambda x, y. \lambda \sigma. \text{insert}(x, \delta(\sigma), \sigma) \quad (\text{see def. of insert in pg. 5}) \]

\[ D_{\text{if-then-else-end}} = \lambda (s, t_1, t_2). \lambda \sigma. \text{if } s(\sigma) \text{ then } t_1(\sigma) \text{ else } t_2(\sigma) \]

\[ D_{\text{while-do}} = \lambda (s, t). \lambda \sigma. \text{if } s(\sigma) \text{ then } t'(t(\sigma)) \text{ else } \sigma \]

where \( s : [X \rightarrow Z]_{\text{ff}} \rightarrow Z \), \( t, t', t_1, t_2 \in D_{\text{comm}} \), and where it should be noted that since the \( t, t', \) etc. are partial functions, compositions like \( t_1(t_1(\sigma)) \) and \( t'(t(\sigma)) \) are partial function compositions, so that if, say, \( t_1(\sigma) \) or \( t(\sigma) \) is undefined, the entire composition is also undefined.

By universality of \( T_{\Sigma_L} \) (again, consider Nat and Int only as data constructors, not as data constructors), we then have a unique \( \Sigma_L \)-homomorphism:

\[ D : T_{\Sigma_L} \rightarrow D \]

which is the direct denotational semantics of SIMPLE.

(The associativity of \( - \circ - \) is preserved because of the definition of \( D \); since \( T_2(T_1(\sigma)) = (T_1 \mathbin{\circ} T_2)(\sigma) \), with \( - \circ - \) partial function composition).
An Algebraic Deconstruction of Denotational Semantics

In 1980, nine years after the Scott-Strachey paper on denotational semantics (see further partly) two papers, by Mitchell-Wand and by Gopin and Paranjye-Ghosh appeared purporting the alternative of an algebraic specification by means of an equational theory \((\Sigma E, E^2)\) of a (deterministic) program language \(E\). Wand waffled a little about whether the semantics \(E\) of \((\Sigma E, E^2)\) should be the whole class \(Alg((\Sigma E, E^2))\), but his paper contained nonetheless the key idea of using the semantic equational theory as rewrite rules. The Gopin and Paranjye-Ghosh paper leaves no doubt about:

1. The semantics map the initial algebra \(\Sigma E/\Sigma E\) and
2. The agreement between the algebraic denotational semantics \(\Sigma E/\Sigma E\) and the equational semantics by rewrite with the oriented equations \(E^2\). Furthermore, by using the OBJT language it shows how the specification \((E, E^2)\) becomes executable as an interpreter.

Of course, all this notation has been inherited by the "Rewriting Logic Semantics Project" (see required reading), but broadly the scope from:

(i) Deterministic languages only to

possibly concurrent languages; and

(ii) From equational logic to rewriting logic, so that specifications become
reduce theory, \( R_L = (E_L, F_L, R_L) \). The special case of a deterministic language \( L \) then appears as \\
\( R_L = (E_L, F_L, \emptyset) \), i.e., as the standard initial algebra semantics \( \Sigma_L = E_L / E_L \) defined by the equational theory \( (E_L, E_L) \).

So, how do traditional denotational semantics and algebraic semantics compare with each other? Algebraic, initial algebra semantics has a similar spirit, since in fact we have just seen that a denotational semantics is precisely the engine of \( \Sigma_L \). Homomorphism associated to the semantic equation define each \( D_f, \Sigma_L \).Homomorphism associated to the semantic equation define each \( D_f, \Sigma_L \).Homomorphism associated to the semantic equation define each \( D_f, \Sigma_L \).Homomorphism associated to the semantic equation define each \( D_f, \Sigma_L \).

\[ \Gamma : \Sigma_L \rightarrow D \]

although this not necessarily fully apparent and can get lost in a sea of somewhat esoteric types and morph domain constructions. What they have in common goes in the PL argot under the name of "compositional", a notion that goes back to Frege and can be boiled down to a simple, clear word: homomorphic. Both semantics are homomorphic, and therefore give semantics to bigger pieces of language in terms of their smaller pieces (this is what "compositional" really means) by the simple equation:

\[ \Gamma (f(P_1, \ldots, P_n)) = D_f (\Gamma P_1, \ldots, \Gamma P_n) \].

Nevertheless, initial algebra semantics (IAS) have definite advantages over denotational semantics (DS), including:
1. **Compatibility of semantic domains**: in IAS all relevant semantic domains can be defined as computable data types (for example, \( e \_O \_n \_e \)), and all semantic functions become computable functions, computed by rewriting. This is in contrast to denotational semantics, where semantic domains are typically not computable.

2. **Executability**: IAS language specifications are executable, and can be directly used as interpreters. DDS are usually not directly executable in the

3. **Automatic built-in denotation**: an IAS specification is also denotational in an obvious sense; the mathematical model that \( (\Sigma, \Sigma_e, E_e) \) denotes is precisely the initial algebra \( T_{\Sigma, e}/E_e \), and the denotation function in precisely the unique \( \Sigma_e \_e \_n \_u \_m \_n \_o \_y \_n \_s \_m \_h \_a \_n \_y \_l \_i \_n \_e \) (where \( \Sigma \) also specifies states)

\[
[\_I, \_D]: T_{\Sigma, e} \rightarrow T_{\Sigma_e, e}/E_e
\]

so that the denotation is automatic and built-in into the semantics of equational logic, as opposed to depend on some possibly complex choice of domains \( \{ D_S \}_{i \in S} \) in \( B_{\Sigma}(\omega) \) and semantic functions \( D_S \).
4. Complete Agreement between Denotational and Operational Semantics: An IAS specification \( (E_1, E_2) \) has both and simultaneously a denotational semantics \( T_{E_1/E_2} \) and an operational semantics given by \( E_2 \). The rewrite relation \( \rightarrow_{E_2} \) is confluent, or at least ground confluent. This is not at all the case in DS, where one has to prove that a given operational semantics, defined separately, agrees with the denotational semantics DS.

5. Modularity: IAS is not a panacea for modularity problems, which is also a challenge for IAS definitions. However, it seems fair to say that:

(5.1) DS specs are remarkably unmodular. Just for SIMPLE we have seen the appearance of what I call "monkeys", like \( S, S', S'' \), and \( Z \), in semantic definitions. As soon as new features are added, new breeds of monkeys are needed and tend to procreate exponentially (see Chapter 5 of Reynolds' book for example).

(5.2) Within the rewrite logic semantics project very modular IAS and rewrite logic semantic (RLS) definitions can be given (see Mereguia & Braga AMAST 2004; and Serafimova - Rosu - Mereguia; required reading).

An IAS semantics of the SIMPLE language is given in the next page:
load fun1.maude.
load simple-PL.maude.

fmod SIMPLE-SEMANTICS is protecting SIMPLE.
protecting FUN<VAR/INT>.
sort State.
subsort Fun{Var,Int} < State.

op [[_,_]] : IntExp Fun{Var,Int} -> Int.
op [[_,_]] : BoolExp Fun{Var,Int} -> Bool.
op [[_,_]] : Comm State -> State.

var X : Var.
vars n m : Int.
var B : Bool.
var M : Magma{Var,Int}.
vars C C' : Comm.
vars IE IE' : IntExp.
vars BE BE' : BoolExp.
var F : Fun{Var,Int}.

eq [[i(n),F]] = n.
eq [[X,F]] = F[X].
eq [[(- IE),F]] = - [[IE,F]] .
eq [[(IE + IE'),F]] = [[IE,F]] + [[IE',F]] .
eq [[(IE * IE'),F]] = [[IE,F]] * [[IE',F]] .

eq [[b(B),F]] = B .
eq [[~(BE),F]] = not([[BE,F]]). 
eq [[(BE ∧ BE'),F]] = [[BE,F]] and [[BE',F]] .
neq [[(BE ∨ BE'),F]] = [[BE,F]] or [[BE',F]] .
eq [[(IE = IE'),F]] = [[IE,F]] == [[IE',F]] .
eq [[(IE <= IE'),F]] = [[IE,F]] /= [[IE',F]] .
eq [[(IE > IE'),F]] = [[IE,F]] > [[IE',F]] .
eq [[(IE >= IE'),F]] = [[IE,F]] >= [[IE',F]] .

eq [[skip,F]] = F.
eq [[(C ; C'),F]] = [[C',C[[C,F]]]].
eq [[(X := IE),F]] = insert(X,[[IE,F]],F).
eq [[(if BE then C else C' end),F]] = if [[BE,F]] then [[C,F]] else
[[C',F]] fi .
eq [[(while BE do C),F]] = if [[BE,F]] then [[C ; while BE do
C,F]] else F fi .
endfm

red [[(while (i(10) > x{0}) do (x{0} := i(2) * (x{0} + i(1)))),{mt}]]
The obvious questions are:

1. How are the DS semantics of SIMPLE and its IAS
   semantics related?

2. Can the DS semantics be derived from the IAS?

First we need a few observations. I will claim certain
facts without giving detailed proofs, which are left as (mod- trivial)
exercises. For $L = SIMPLE : \{ \text{modulo the A and ACU axioms} \}
\text{B, ground} \}

1. $(Σ_2, E_2|B)$ is congruent. Furthermore, all equations
   except the one defining the semantics of while-do-
   are terminating modulo $B$.

2. A SIMPLE program $P$ terminates from an initial
   state $o$ if and only if there exist a (unique) $o : \text{Fun}(\text{Var}, \text{Int})$
such that $[[ P, o ]] \rightarrow^* o$

3. $Σ_2/E_2|B, \text{Fun}(\text{Var}, \text{Int})] \equiv [X \rightarrow \mathbb{Z}] \cup \{ \text{error} \}$

4. $Σ_2/E_2|B, \text{Nat} \equiv \mathbb{N}$, $Σ_2/E_2|B, \text{Int} \equiv \mathbb{Z}$ (for $b_0 \equiv 2$)

5. $Σ_2/E_2|B, \text{IntExp} = Σ_2, \text{IntExp} = \mathbb{N}$, likewise for BE and CMI

6. Let us denote $Σ_2/E_2|B, \text{State} \equiv \mathbb{S}$. Then, up to bijection,
   $[X \rightarrow \mathbb{Z}] \subset \mathbb{S}$. Furthermore, the only construction terms
   in are precisely the elements of $[X \rightarrow \mathbb{Z}] \mathbb{S}$ and error.
Note that given sets $A, B$ with subsets $A_0 \subseteq A$, $B_0 \subseteq B$, we have a **surjective restriction function**

$$[A \to B] \xrightarrow{r_A^B} [A_0 \to B_0]$$

$$f \xrightarrow{r_A^B} f|_{A_0}^B = \{(a, b) \in f | a \in A_0 \land b \in B_0\}$$

Let us abbreviate the function

$$[\_, \_] : \underbrace{T \prod_{E / X \uparrow B}}_{T \prod_{E / X \uparrow B}} \times \underbrace{T \prod_{E / Y \downarrow B}}_{T \prod_{E / Y \downarrow B}} \to T$$

to just

$$[\_, \_] : BE \times ([X \to Z]_{ff \cup \text{comm}}) \longrightarrow \mathbb{Z}$$

and likewise for Boolean expressions to:

$$[\_, \_] : BE \times ([X \to Z]_{ff \cup \text{comm}}) \longrightarrow \mathbb{I}$$

and for commands to:

$$[\_, \_] : CM \times S \longrightarrow S$$

**Lemma 1.** The function

$$BE \xrightarrow{\land [\_, \_]} \left([([X \to Z]_{ff \cup \text{comm}}) \to \mathbb{Z}\right) \xrightarrow{[X \to Z]_{ff}} \left([X \to Z]_{ff} \to \mathbb{Z}\right)$$

coincides with the function

$$[\_] : BE \longrightarrow ([X \to Z]_{ff} \to \mathbb{Z})$$

in the denotational semantics $D$ of SIMPLE.
Proof. Suppose not. Pick $e \in \mathcal{BE}$ of smallest size possible as a tree so that both functions disagree but agree on terms of smaller size. By the equations for $i(n)$, and $x$ and the definitions of $\mathcal{D}_x$ and $\mathcal{D}_{\text{ann}}$, $e$ cannot be a variable or an integer. So it must be either $e = -e_1$, or $e = e_1 + e_2$, or $e = e_1 \cdot e_2$. Let us reach a contradiction for $e = e_1 + e_2$, the other two cases are similar. We have for each $\sigma \in [X \to Z]_{\text{ff}}$,

$$\lambda [-,-]_{[X \to Z]_{\text{ff}}}^{[X \to Z]_{\text{ff}}} (e_1 + e_2)(\sigma) = [e_1 + e_2, \sigma] = (\text{by equation for })$$

$$= [e_1, \sigma] + [e_2, \sigma] = (\text{by minimality of } e) =$$

$$[e_1]_{\mathcal{D}^+} + [e_2]_{\mathcal{D}^+} = (\text{by definition of } \mathcal{D}^+)$$

$$= [e_1 + e_2)(\sigma), \text{ contradicting the supposed disagreement between both functions. } \square$$

In a completely similar manner we obtain:

**Lemma 2.** The function

$$\mathcal{BE} \xrightarrow{\lambda [-,-]_{[X \to Z]_{\text{ff}}}^{[X \to Z]_{\text{ff}}} \Rightarrow \mathcal{Z} \Rightarrow [X \to Z]_{\text{ff}} \Rightarrow \mathcal{Z}}$$

coincides with the function

$$[-] : \mathcal{BE} \rightarrow [[X \to Z]_{\text{ff}} \Rightarrow \mathcal{Z}]$$

in the denotational semantics $\mathcal{D}$ of SIMPLE.
Let us now consider the function

\[ \lambda [z, \pi] \rightarrow [\pi \rightarrow \pi] \cong [X \rightarrow Z]_{ff} \rightarrow [X \rightarrow Z]_{ff} \].

It looks involved but it is actually quite simple. It maps each \( \pi \in \text{CM} \) to a partial function that maps each \( \sigma_0 \in [X \rightarrow Z]_{ff} \) to either:

\[
\begin{cases}
\text{the unique } \sigma \in [X \rightarrow Z]_{ff} \text{ such that } (p, \sigma_0) \rightarrow! \sigma \\
\text{or is undefined if no such } \sigma \text{ exists.}
\end{cases}
\]

The point is that each \( \text{Ex} \cup \text{B} \)-equivalence class \([p, \sigma_0]],[p, \sigma_0] \) is such that either:

\[
\begin{cases}
1. \exists \sigma \in [X \rightarrow Z]_{ff} \text{ s.t. } \{ \sigma \} = [p, \sigma_0] \cap [X \rightarrow Z]_{ff}, \text{ or} \\
2. [p, \sigma_0] \cap [X \rightarrow Z]_{ff} = \emptyset
\end{cases}
\]

Case (1) is the \underline{terminating} case, and Case (2) is the \underline{non-terminating} case. The uniqueness of \( \sigma \) in case 1 is guaranteed by any \( \sigma \in [X \rightarrow Z]_{ff} \) being in canonical form and \( \sigma \) being \underline{ground confluent} modulo B. Therefore,

\[
\{ x \in \text{term} \} = [X \rightarrow Z]_{ff} \cup \left\{ [\pi, \sigma] \mid \pi \in \text{CM}, \sigma \in [X \rightarrow Z]_{ff} \wedge [p, \sigma_0] \cap [X \rightarrow Z]_{ff} = \emptyset \right\}.
\]
Theorem. The function \( \mu \) coincides with the function

\[ \mu \colon [X\to Z]_{\mathbb{N}} \to [X\to Z]_{\mathbb{N}} \]

in the denotational semantics \( D \) of SIMPLE.

Proof (sketch). Pick again \( p \) smallest possible as a tree so that both functions disagree. Because of the definitions for \( \text{skip} \) and \( := \) in (\( \Sigma F, \exists q \cup i \beta \)) and the functions \( D_{\text{skip}} \) and \( D_{:=} \), \( p \) cannot be \( \text{skip} \) or an assignment. Therefore either (i) \( p = p_1 \cdot p_2 \) or (ii) \( p = \text{if} \text{expr} \text{then} p_3 \text{else} p_2 \cdot \text{end} \), or (iii) \( p = \text{while} \text{expr} \text{do} p_4 \).

Let now \( \sigma_0 \in [X\to Z]_{\mathbb{N}} \) be such that \( \llbracket p \rrbracket (\sigma_0) \neq \bigwedge_{i=1}^n \llbracket X_i \rrbracket_{\mathbb{N}} (\bullet, \sigma_0) \).

In other words, either one side is defined and the other is not, or both sides are defined but different.

Case (i) \( p = p_1 \cdot p_2 \) and we have two cases:

1. \( \llbracket p_1, p_2, \sigma_0 \rrbracket \) undefined, i.e., \( \llbracket p_1, p_2, \sigma_0 \rrbracket \cap [X\to Z]_{\mathbb{N}} = \emptyset \)

but \( \llbracket p_2 \rrbracket (\sigma_0) \) defined. But since, by the definition of \( D \), and \( \mu \) a \( \Sigma \) - homomorphism we have,

\[ \llbracket p_1 ; p_2 \rrbracket (\sigma_0) = \llbracket p_2 \rrbracket (\llbracket p_1 \rrbracket (\sigma_0)) \]

we must have \( \sigma, \sigma' \in [X\to Z]_{\mathbb{N}} \) such that \( \llbracket p_2 \rrbracket (\sigma) = \sigma \) and \( \llbracket p_1 \rrbracket (\sigma) = \sigma' \). But by the minimality of \( p \) this means that \( \llbracket p, \sigma_0 \rrbracket \to \sigma \) and \( \llbracket p_1, \sigma_0 \rrbracket \to \sigma' \). But then we have
\[ [P_1; P_2, σ_0] \rightarrow [P_2, [P_3, σ_1]] \rightarrow [P_2, σ] \rightarrow [σ', 0] \]

Contradict the assumption that \([P_2, P_2, σ] \) was undefined.

(2) \([P_1, P_2, σ_0] \rightarrow [σ']\). But this forces the above
rewrite sequence, and, by the minimality of \(P\), that we
then must have \([P_1, σ_0] = [P_2](σ_0) = σ'\), and
\([P_2, σ] = [P_2](σ) = σ'\), which forces by \(T\), and \(T\)-homom,
\([P_1; P_2, σ_0] \rightarrow [P_2](P_2)(σ_0)) = σ' = [P_1; P_2](σ_0)\), contradicting the
claim that both functions differed for \(P_1; P_2\) and \(σ_0\).

Case (ii) \(P = \text{if } b_0 \text{ then } P_1 \text{ else } P_2 \text{ end }\). Without
loss of generality we may have \([b_0, σ] \rightarrow [σ']\) true,
so that \([P, σ_0] \rightarrow [P_1, σ_0]\). But then we must have
\([P_1](σ_0) = [P_2](σ_0)\). But with the minimality of \(P\),
regardless of whether \([P_1](σ_0)\) is defined or not, both
functions must agree on \(σ_0\), contradicting their supposed disagreement.

Case (iii) \(P = \text{while } b_0 \text{ do } P_1 \). Let us abbreviate notation
and write \([P]_0 = (⊥, [X → Z]_0, [X → Z]_{ff})\). Because of the
equations for while-do- and \(-; -\) in the algebraic semantics
for any \(σ \in [X → Z]_{ff}\) we must have (regardless of whether
the function is defined or not for \(σ\)) that
\[ [[\text{while } \text{loxp do } p_2]^0(\sigma)] = \]

\[ = \text{if } [[\text{loxp}]()] \text{ then } [[p_2, \text{while } \text{loxp do } p_2]^0(\sigma)] \text{ else } \sigma \text{ fi} \]

\[ = \text{by minimality of } p \]

\[ = \text{if } [[\text{loxp}]()] \text{ then } [[\text{while } \text{loxp do } p_2]^0([p_2]()] \text{ else } \sigma \text{ fi}. \]

But this means that the partial function \( [[\text{while } \text{loxp do } p_2]^0] \)

in a fixpoint for the functional

\[(\star) \quad \lambda \sigma'. \lambda \sigma. \quad \text{if } [[\text{loxp}]()] \text{ then } \sigma'( [p_2]()) \text{ else } \sigma \text{ fi.} \]

But since by \( \text{if-then-else} \) - soundness and the definition of \( D \)

\( [[\text{while } \text{loxp do } p_2]] \) is the minimal fixpoint of such a functional, we must have a containment of partial functions,

\( [[\text{while } \text{loxp do } p_2]] \subseteq [[\text{while } \text{loxp do } p_2]^0] \).

This means that the only way in which these functions can differ for the input \( \sigma \) where they presumably differ is when \( [[\text{while } \text{loxp do } p_2]](\sigma) \) is undefined, but \( [[\text{while } \text{loxp do } p_2, \sigma]] \rightarrow! \sigma' \) for some \( \sigma' \).

But since we already know by minimality of \( p \)

that \( [[p_2]] = [[p_2]]^0 \), and we must have states
$\sigma_1, \sigma_2, \ldots, \sigma_n : \sigma'$ such that

$[\text{while loop do } p_1, \sigma_0] \rightarrow^* [\text{while loop do } p_1, \sigma_1]\rightarrow^* [\text{while loop do } p_2, \sigma_2]\rightarrow^* \ldots \rightarrow^* [\text{while loop do } p_n, \sigma_n] \rightarrow^* \sigma'$

with $\sigma_1 = [p_1](\sigma_0)$, $\sigma_2 = [p_1] \sigma_1$, $\ldots$, $\sigma' = [p_n](\sigma_{n-1})$

and $[\text{loop }] (\sigma_0) = \ldots = [\text{loop }] (\sigma_{n-2}) = \text{true}$,
$[\text{loop }] (\sigma_{n-1}) = \text{false}$

The desired contradiction, and therefore agreement of both functions, can easily be shown by applying the functional (得住) repeatedly to the empty partial function $\emptyset$, and then applying the result of partial functions to $\sigma_0$. The remaining details are left as an exercise. □
We have seen in detail an example of a direct denotational semantics. As it is very clearly explained in Chapter 5 of Reynolds' book, the direct semantics approach runs into a serious tour de force when control-intensive features such as fail, input-output, jumps, call-cc, and the like are added to a language. Not only is a new plethora of "monkeys" needed, but the effort of still giving a direct semantics requires a truly, not just Baroque, but indeed Rococo menagerie of increasingly more implausible cpos.

The proper way out is, as pointed out masterfully by Reynolds, to adopt a continuation semantics, capturing what "the rest of the program" after the execution of a command does.

But continuation denotational semantics can also be algebraically "deconstructed." In some ways this has been done by Reynolds himself, and in others by the use of first-order continuations in the IK framework, as discussed in the required reading of the paper by Sato and Poisson.