Programming language semantics

Romuald THION

October 8, 2010

Abstract

Haskell implementation of the denotational semantics for the toy programming language in David A. SCHMIDT's [Sch97], available at `http://people.cis.ksu.edu/~schmidt/papers/CRC.chapter.ps.gz`

1 Introduction

This program is written in the Literate Haskell style. It compile with both a LaTeXand a Haskell compiler with the literate features turned on :

- The Glasgow Haskell Compiler do supports this source style (`.lhs` file extension). The slogan is "everything is comment, please use $>$ before a line of code".
- *lhs2TeX* is used as a frontend for *pdflatex*. A config file is used to typeset the code according to SCHMIDT's mathematical notational conventions.

The document is a quite direct implementation ofDavid A. SCHMIDT's semantics for a toy programming language [Sch97], freely available on the internet [1]. The paper focuses on denotional semantics, its includes a detailled example for a toy imperative language.

The interpreter is built quite closely from the mathematical definitions in pages 6 to 11 of [Sch97]. The main differences with the paper are:

- the `Maybe` monad is used for $\perp$,
- the intepreter returns a store instead of a single integer,
- coproduct $+$ is used instead of $\mathbb{N} \cup \{\mathbf{tt}, \mathbf{ff}\}$,
- some extra tests has been added for borderline cases,
- fixpoint combinator *fix* is used for denotation of while construction. Actually, it's the only tricky part of the code.

It has been written in a hurry an should be improved! The document is meant to compile without warning with full strictures turned on.

[1] `http://people.cis.ksu.edu/~schmidt/papers/CRC.chapter.ps.gz`

2 Syntax

Notational conventions.

- composition is $(\circ) :: (b \to c) \to (a \to b) \to a \to c$
- on types:
  - boolean domain is written $\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$,
  - integers are written $\mathbb{N}$,
  - A_∇ is an optional A that is $\mu X.1 + A$. In **Set**, it is $A_\nabla = A \cup \{*\}$

The syntax of the toy imperative language

$\mathbf{data}\ \mathit{Prog} = \mathit{Prog}\ \mathit{Comm}$

$\mathbf{data}\ \mathit{Comm} = \mathit{Affect}\ \mathit{Id}\ \mathit{Expr}$
$\quad \mid \mathit{Sequence}\ \mathit{Comm}\ \mathit{Comm}$
$\quad \mid \mathit{Begin}\ \mathit{Decl}\ \mathit{Comm}$
$\quad \mid \mathit{Call}\ \mathit{Id}$
$\quad \mid \mathit{While}\ \mathit{Expr}\ \mathit{Comm}$

$\mathbf{data}\ \mathit{Decl} = \mathit{ProcDef}\ \mathit{Id}\ \mathit{Comm}$

$\mathbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathbb{N}$
$\quad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$
$\quad \mid \mathit{NotEq}\ \mathit{Expr}\ \mathit{Expr}$
$\quad \mid \mathit{Var}\ \mathit{Id}$

3 Domain

A triple to store values of variables "X","Y" and "Z"

$\mathbf{type}\ \mathit{Store} = (\mathbb{N}, \mathbb{N}, \mathbb{N})$
$\mathbf{data}\ \mathit{Loc} = A \mid B \mid C$
$\mathit{look} :: \mathit{Loc} \to \mathit{Store} \to \mathbb{N}$
$\mathit{look}\ A\ (x, \_, \_) = x$
$\mathit{look}\ B\ (\_, y, \_) = y$
$\mathit{look}\ C\ (\_, \_, z) = z$
$\mathit{update} :: \mathit{Loc} \to \mathbb{N} \to \mathit{Store} \to \mathit{Store}$
$\mathit{update}\ A\ i\ (\_, y, z) = (i, y, z)$
$\mathit{update}\ B\ i\ (x, \_, z) = (x, i, z)$
$\mathit{update}\ C\ i\ (x, y, \_) = (x, y, i)$
$\mathit{initStore} :: \mathbb{N} \to \mathit{Store}$
$\mathit{initStore}\ n = (n, 0, 0)$

Environment, that maps identifiers to either a location or a function that modifies the store. The Maybe monad is used to capture bottom, *check* function is *bind* $(\gg=)$ with parameters reversed.

$\mathit{check} :: (\mathit{Store} \to \mathit{Store}_\nabla) \to \mathit{Store}_\nabla \to \mathit{Store}_\nabla$
$\mathit{check}\ f\ s = s \gg= f$
$\mathbf{data}\ \mathit{Denotable} = \mathit{Mem}\ \mathit{Loc} \mid \mathit{Fun}\ (\mathit{Store} \to \mathit{Store}_\nabla)$
$\mathbf{type}\ \mathit{Env} = [(\mathit{Id}, \mathit{Denotable})]$
$\mathit{find} :: \mathit{Id} \to \mathit{Env} \to \mathit{Denotable}$
$\mathit{find}\ \_\ [\,] = \mathit{error}$ `"find: Empty environnement"`

$$find\ i\ ((j,d) : es) \mid (i \Leftrightarrow j) \quad = d$$
$$\qquad\qquad\qquad\qquad \mid otherwise = find\ i\ es$$

$$bind :: Id \to Denotable \to Env \to Env$$
$$bind = ((\circ) \circ (\circ))\ (:)\ (,)$$

bind is written in a cryptic way. Please consider this definition $bind\ i\ d\ e = (i,d) : e$.

$$initEnv :: Env$$
$$initEnv = (\texttt{"X"}, Mem\ A) : (\texttt{"Y"}, Mem\ B) : (\texttt{"Z"}, Mem\ C) : [\,]$$

4 Denotation

Semantic mappings for each level of the syntax

- $[\![\cdot]\!]_P$ for Prog(rams)
- $[\![\cdot]\!]_D$ for Decl(arations)
- $[\![\cdot]\!]_C$ for Comm(ands)
- $[\![\cdot]\!]_E$ for Expr(essions)

4.1 Programs

Piece of cake.

$$[\![\cdot]\!]_P :: Prog \to \mathbb{N} \to Store_\bigtriangledown$$
$$[\![(Prog\ c)]\!]_P = \lambda n \to ([\![c]\!]_C)\ initEnv\ (initStore\ n)$$

4.2 Declarations

A declaration is mapped to an endo function of the environment.

$$[\![\cdot]\!]_D :: Decl \to Env \to Env$$
$$[\![(ProcDef\ \ i\ c)]\!]_D = \lambda e \to bind\ i\ (Fun\ ([\![c]\!]_C\ e))\ e$$

4.3 Commands

$$[\![\cdot]\!]_C :: Comm \to Env \to Store \to Store_\bigtriangledown$$
$$\begin{array}{ll}
[\![(Affect\ i\ x)]\!]_C & = \lambda e\ s \to \textbf{case}\ (find\ i\ e)\ \textbf{of} \\
& \quad Mem\ l \to \textbf{case}\ ([\![x]\!]_E\ e\ s)\ \textbf{of} \\
& \qquad\qquad \iota_L\ v \ \to \eta\ (update\ l\ v\ s) \\
& \qquad\qquad \iota_R\ _ \to fail\ \texttt{"denotC: Nat expected"} \\
& \quad Fun\ _ \ \to fail\ \texttt{"denotC: Location expected"} \\
[\![(Sequence\ x\ y)]\!]_C & = \lambda e\ s \to ([\![x]\!]_C\ e\ s) \gg\!= ([\![y]\!]_C\ e) \\
[\![(Begin\ d\ c)]\!]_C & = \lambda e\ s \to [\![c]\!]_C\ ([\![d]\!]_D\ e)\ s \\
[\![(Call\ i)]\!]_C & = \lambda e \to \textbf{case}\ (find\ i\ e)\ \textbf{of} \\
& \qquad\quad Mem\ _ \to const\ (fail\ \texttt{"denotC: Fun expected"}) \\
& \qquad\quad Fun\ f \ \ \to f \\
[\![(While\ x\ c)]\!]_C & = \lambda e \to \textbf{let}
\end{array}$$
$$\quad f :: (Store \to Store_\bigtriangledown) \to (Store \to Store_\bigtriangledown)$$
$$\quad f\ h = \lambda s \to \textbf{case}\ ([\![x]\!]_E\ e\ s)\ \textbf{of}$$

$(\iota_R\ \mathbf{tt}) \rightarrow ([\![c]\!]_C\ e\ s) \gg\!= h$
$(\iota_R\ \mathbf{ff}) \rightarrow \eta\ s$
$(\iota_L\ \_) \rightarrow \mathit{fail}$ `"denotC: Bool expected"`
$\mathbf{in}\ \mathit{fix}\ f$

4.4 Expressions

Function $\mathit{fix} :: (a \rightarrow a) \rightarrow a$ defined as $\mathit{fix}\ f = \mathbf{let}\ x = f\ x\ \mathbf{in}\ x$ is the fixed point[2] combinator of Haskell

$[\![\cdot]\!]_E :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Store} \rightarrow \mathbb{N} + \mathbb{B}$
$[\![(\mathit{Val}\ i)]\!]_E = \lambda\_\ \_ \rightarrow \iota_L\ i$
$[\![(\mathit{Var}\ x)]\!]_E = \lambda e\ s \rightarrow \mathbf{case}\ (\mathit{find}\ x\ e)\ \mathbf{of}$
$\quad \mathit{Mem}\ l \rightarrow \iota_L\ \$\ \mathit{look}\ l\ s$
$\quad \mathit{Fun}\ \_ \rightarrow \mathit{error}$ `"denotE: Location expected"`
$[\![(\mathit{Add}\ x\ y)]\!]_E = \lambda e\ s \rightarrow \mathbf{case}\ ([\![x]\!]_E\ e\ s, [\![y]\!]_E\ e\ s)\ \mathbf{of}$
$\quad (\iota_L\ x', \iota_L\ y') \rightarrow \iota_L\ (x' + y')$
$\quad \_ \rightarrow \mathit{error}$ `"denotE: Nat expected"`
$[\![(\mathit{NotEq}\ x\ y)]\!]_E = \lambda e\ s \rightarrow \mathbf{case}\ ([\![x]\!]_E\ e\ s, [\![y]\!]_E\ e\ s)\ \mathbf{of}$
$\quad (\iota_R\ x', \iota_R\ y') \rightarrow \iota_R\ (x' \not\Leftrightarrow y')$
$\quad (\iota_L\ x', \iota_L\ y') \rightarrow \iota_R\ (x' \not\Leftrightarrow y')$
$\quad \_ \rightarrow \mathit{error}$ `"denotE: Bool VS Nat"`

5 Toy sample

The toy sample of the paper : a function that squares a natural number

$\mathit{myDecl} :: \mathit{Decl}$
$\mathit{myDecl} = \mathit{ProcDef}$ `"INCR"` $\mathit{aComm}\ \mathbf{where}$
$\quad \mathit{aComm}, \mathit{comm1}, \mathit{comm2} :: \mathit{Comm}$
$\quad \mathit{aComm} = \mathit{Sequence}\ \mathit{comm1}\ \mathit{comm2}$
$\quad \mathit{comm1} = \mathit{Affect}$ `"Z"` $(\mathit{Add}\ (\mathit{Var}$ `"Z"`$)\ (\mathit{Var}$ `"X"`$))$
$\quad \mathit{comm2} = \mathit{Affect}$ `"Y"` $(\mathit{Add}\ (\mathit{Var}$ `"Y"`$)\ (\mathit{Val}\ 1))$

$\mathit{myBody} :: \mathit{Comm}$
$\mathit{myBody} = \mathit{Sequence}\ \mathit{initP}\ \mathit{aLoop}\ \mathbf{where}$
$\quad \mathit{initP}, \mathit{aLoop} :: \mathit{Comm}$
$\quad \mathit{initP} = \mathit{Sequence}\ (\mathit{Affect}$ `"Y"` $(\mathit{Val}\ 0))\ (\mathit{Affect}$ `"Z"` $(\mathit{Val}\ 0))$
$\quad \mathit{aLoop} = \mathit{While}\ \mathit{cond}\ \mathit{inn}$
$\quad \mathit{cond} :: \mathit{Expr}$
$\quad \mathit{cond} = \mathit{NotEq}\ (\mathit{Var}$ `"Y"`$)\ (\mathit{Var}$ `"X"`$)$
$\quad \mathit{inn} :: \mathit{Comm}$
$\quad \mathit{inn} = \mathit{Call}$ `"INCR"`

$\mathit{myProg} :: \mathit{Prog}$
$\mathit{myProg} = \mathit{Prog}\ (\mathit{Begin}\ \mathit{myDecl}\ \mathit{myBody})$

Instances of class *Show* are defined in the source files (pretty printing).

[2] http://en.wikibooks.org/wiki/Haskell/Denotational_semantics

$$show\ myProg =$$

```
"begin proc INCR = Z:=Z + X; Y:=Y + 1 in Y:=0; Z:=0; while Y != X do call INCR od enc
```

One can use $[\![\cdot]\!]_P$ as an interpreter for the programming language

$$[\![myProg]\!]_P\ 9 = \eta\ (9, 9, 81)$$

More generally, for $x \geqslant 0$, $[\![myProg]\!]_P\ x = \eta\ (x, x, x \uparrow 2)$.

References

[Sch97] David A. Schmidt. Programming language semantics. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2237–2254. CRC Press, 1997.