0 Introduction

H0 is an extremely simple calculator notation.

1 Syntax

The syntax of H0 is given by the grammar

$$
\begin{aligned}
\mathsf{expression} &= \mathsf{atom}+ \\
\mathsf{atom} &= \mathsf{integer} \mid \mathsf{variable} \mid (\mathsf{expression}) \\
\mathsf{variable} &= \mathsf{identifier}
\end{aligned}
$$

We let E range over expression , I range over integer and V range over identifier .

```
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Token
import Text.ParserCombinators.Parsec.Language


type Scanner a = GenParser Char () a
data Token
     =   INum Integer | FNum Double
       | Varid String | Reserved String
     deriving (Show, Eq)
scan :: Scanner a -> Scanner (a, SourcePos)
scan p = do  pos <- getPosition
             x <- p
             return (x, pos)
scan_integer, scan_varid, hreserved, htoken :: Scanner (Token, SourcePos)
scan_integer = do  (i, pos) <- scan (integer (makeTokenParser haskellDef))
                   return (INum i, pos)
hfloat = do  (i, pos) <- scan (float (makeTokenParser haskellDef))
             return (FNum i, pos)
scan_varid = do  (c, pos) <- scan (hlower)
                 cs <- identifier (makeTokenParser haskellDef)
                 return (Varid (c : cs), pos)
hlower :: Scanner Char
```

```
hlower = lower <|> char '_'
hreserved = do  (cs, pos) ← scan (string "(" <|> string ")")
                return (Reserved cs, pos)
htoken = try hfloat <|> scan_integer <|> scan_varid <|> hreserved
scanall :: String → [(Token, SourcePos)]
scanall cs = let result = parse (many htoken) "" cs
             in case result of
                  Right tkns → tkns
                  Left err → error (show err)
```

These are utility functions to aid parsing.

```
type HParser a = GenParser (Token, SourcePos) () a
item :: HParser (Token, SourcePos)
item = do  tkn ← anyToken
           return tkn
sat :: (Token → Bool) → HParser (Token, SourcePos)
sat p = do  (t, pos) ← item
            if p t then return (t, pos) else pzero
isInteger, isFloat, isVarid :: Token → Bool
isInteger t = case t of
    INum _ → True
    otherwise → False
isFloat t = case t of
    FNum _ → True
    otherwise → False
isVarid t = case t of
    Varid _ → True
    otherwise → False
```

```
data Expression
      = Integerr Integer | Floatt Double
         | Var String | Expression :@ Expression
    deriving (Show, Eq)
type Expr = (Expression, SourcePos)
expression, atom, integerr, floatt, variable :: HParser Expr
expression = do  pos ← getPosition
                 xs ← many1 atom
                 return (foldl1 (f pos) xs)
                     where f x (e, _) (e′, _) = (e :@ e′, x)
```

$$
\begin{aligned}
&atom \\
&\quad = \quad try\ (variable) <|> try\ floatt <|> integerr <|> \\
&\qquad\quad \mathbf{do}\ \{sat\ (\equiv Reserved\ "(");(e,pos) \leftarrow expression; \quad sat\ (\equiv Reserved\ ")");return\ (e,pos
\end{aligned}
$$

$$
\begin{aligned}
integerr = \mathbf{do}\quad & (INum\ i, pos) \leftarrow sat\ isInteger \\
& return\ (Integerr\ i, pos)
\end{aligned}
$$

$$
\begin{aligned}
floatt = \mathbf{do}\quad & (FNum\ x, pos) \leftarrow sat\ isFloat \\
& return\ (Floatt\ x, pos)
\end{aligned}
$$

$$
\begin{aligned}
variable = \mathbf{do}\quad & (Varid\ cs, pos) \leftarrow sat\ isVarid \\
& return\ (Var\ cs, pos)
\end{aligned}
$$

$$
\begin{aligned}
&parseall :: [(Token, SourcePos)] \rightarrow Expr \\
&parseall\ ts = \mathbf{let}\ result = parse\ expression\ ""\ ts \\
&\qquad\qquad\quad \mathbf{in\ case}\ result\ \mathbf{of} \\
&\qquad\qquad\qquad Right\ expr \rightarrow expr \\
&\qquad\qquad\qquad Left\ err \rightarrow error\ (show\ err)
\end{aligned}
$$

2 Semantics

Semantic algebra

1. Integers
   Domain $\mathrm{Int} = \mathbb{Z}$
   Operations
   $-\infty \ldots \infty : \mathrm{Int}$

In order to define H0 completely, a *static semantics* and a *dynamic semantics* must be supplied. The static semantics should describe all context conditions, for example type checking.

Static semantics

We start by defining $\mathcal{E}_s$: expression $\rightarrow$ Type $\rightarrow$ Type . The type Type $\rightarrow$ Type is a map from some set of inherited attributes into a set of derived attributes. Since we are interested in the type of expressions, one of the derived attributes will be of type Type defined as

$$\mathrm{Type} \ = \ \mathrm{Int} \mid \mathrm{Error}$$

On Type we impose the lattice structure:

$$\mathrm{Int} \sqsubseteq \mathrm{Error}$$

An inherited attribute T is the type required by the context, while the corresponding derived attribute $\mathsf{T}' = \mathcal{E}.\mathsf{E}.\mathsf{T}$ delivers the derived type of E. This is captured by the invariant:

$$\mathsf{T}' = \mathcal{E}_s.\mathsf{E}.\mathsf{T} \;\Rightarrow\; \mathsf{T} \sqsubseteq \mathsf{T}'$$

The type derived for literal constants is the smallest type that agrees with the type of the constant and the required type.

$$\mathcal{E}_s.\mathrm{i}.\mathsf{T} \quad = \quad \mathsf{T} \sqcup \mathrm{Int}$$

Dynamic semantics

$$\begin{aligned} \mathbf{E} &\;:\; \mathit{expression} \to \mathrm{Int} \to \mathrm{Int} \\ \mathbf{E}.[\![\mathsf{E}]\!].\mathrm{i} &\;=\; \mathsf{LITERAL}.[\![\mathsf{E}]\!] \end{aligned}$$

$$\begin{aligned} \mathsf{LITERAL} &\;:\; \mathit{integer} \to \mathrm{Int} \\ \mathsf{LITERAL}.[\![\mathrm{I}]\!] &\;=\; \text{(omitted—maps I to corresponding } \mathrm{i} \in \mathrm{Int}) \end{aligned}$$

```haskell
data (Num a) => Value a = VNum a | VFloat Double
     deriving (Show, Eq)
numfns :: Num a => [(String, a -> a -> a)]
numfns = [("_iadd_", (+)), ("_isub_", (-)), ("_imul_", (*))]
fracfns :: Fractional a => [(String, a -> a -> a)]
fracfns = [("_fadd_", (+)), ("_fsub_", (-)), ("_fmul_", (*)), ("_fdiv_", (/))]
eval :: Num a => Expression -> Register -> Value a
eval (Integerr i) r0 = VNum (fromInteger i)
eval (Floatt x) r0 = VFloat x
eval ((Var cs :@ e) :@ e') r0 = case lookup cs numfns of
     Just f -> let VNum i = eval e r0
                   VNum j = eval e' r0
               in  VNum (f i j)
     Nothing -> case lookup cs fracfns of
          Just g -> let VFloat x = eval e r0
                        VFloat y = eval e' r0
                    in VFloat (g x y)
```

$$
\begin{array}{l}
\qquad\quad \textit{Nothing} \rightarrow \textit{error}\ (\texttt{"unknown function: "} +\!\!+\ \textit{cs}) \\
\textit{eval}\ (\textit{Var}\ \texttt{"\_ineg\_"} :@\ e)\ \textit{r0} = \textbf{let}\ \textit{VNum}\ i = \textit{eval}\ e\ \textit{r0} \\
\quad\ \textbf{in}\quad \textit{VNum}\ (-i)
\end{array}
$$

$$\textbf{type}\ \textit{Register} = \textit{Integer}$$

$$\textit{calc}\ \textit{cs} = \textit{eval}\ ((\textit{fst} \circ \textit{parseall} \circ \textit{scanall})\ \textit{cs})\ 0$$

Expression continuations

Explicit control-flow can be introduced into the evaluation of expressions by lifting the order of evaluation of subexpressions to the statement level.

We introduce the assignment statement into our semantics...

This suggests that expressions should be turned into statements by means of the transformation E , $\mathsf{E} : expr \rightarrow var \times stat$; the type of E leaves little choice but taking

$$\mathsf{E}.e.(x, s) = [x := e].s$$

New compile- and run-time operations are extracted by calculating.

$$
\begin{array}{ll}
 & \mathsf{E}.(\mathsf{LITERAL}.i).(x, s) \\
= & \quad \text{unfold} \\
 & [x := \mathsf{LITERAL}.i].s
\end{array}
$$

Referring to the fusion law yields the new semantics:

$$\mathcal{E}[\![i]\!].(x, s) \quad = \quad [x := \mathsf{LITERAL}.i].s$$

The new set of run-time operations is:

$$[x := \mathsf{LITERAL}.i].s.cxt \quad = \quad s.((x := i).cxt)$$