SPL-Scheme

Piotr Krzemiński

Wrocław, 11 lutego 2014

Motywacja i cele

- napisać trochę większy projekt w Haskellu z wykorzystaniem Cabala
- wykorzystać w praktyce wiedzę zdobytą na SJP
- bliżej poznać semantykę denotacyjną, kontynuacje, CPS, call/cc

Dlaczego Scheme?

- niebardzo chcieliśmy skupiać się na pisaniu skomplikowanego parsera czy typecheckera
- dobra okazja, aby głębiej poznać rodzinę języków lispowych

Scheme jest ustandaryzowany

- standard organizacji IEEE (The IEEE standard, 1178-1990 (R1995))
- raport R^6RS (Revised6 Report on the Algorithmic Language Scheme)

Nasze podejście

Zacząć od bardzo małego podzbioru, sukcesywnie dodając nowe konstrukcje do języka

Stan obecny

- parser (w Parsecu), pretty printer
- ewaluacja wyrażeń arytmetycznych, logicznych, sterujących
- lambda-abstrakcja i aplikacja (również wieloargumentowa)
- lispowe struktury danych (`cons`, `car`, `cdr`, ...)
- statycznie wiązane definicje zmiennych i funkcji
- rekursja
- konsola interaktywna (REPL)
- interpreter

Szczegóły implementacyjne

- semantyka denotacyjna w stylu kontynuacyjnym
- minimalny core-language, dużo cukru syntaktycznego
- definicje typów języka jako ADT w Haskellu

Składnia

$$
\begin{array}{lcl}
\langle Exp \rangle & ::= & n \\
 & | & \#\mathrm{t} \mid \#\mathrm{f} \\
 & | & \mathrm{atom} \\
 & | & \text{"string"} \\
 & | & \mathbf{(}\ <ExpList>\ \mathbf{)} \\
 & & \\
\langle ExpList \rangle & ::= & \epsilon \\
 & | & <Exp>\ <ExpList>
\end{array}
$$

Dziedzina wartości

$$
\begin{aligned}
&Env = (Ide \to Val)^* \\
&Cont = Env \to Val \to Val^\star \\
&Clo = Val \to Cont \to Val^\star \\
&Val = Exp \cup Clo \\
&Val^\star \approx ((Env \times Val)\ +\ (\{err\} \times \Sigma^*)\ +\ (\{typerr\} \times \Sigma^*))_\perp
\end{aligned}
$$

Dziedzina wartości

$$
\begin{aligned}
&Env = (Ide \to Val)^* \\
&Cont = Env \to Val \to Val^\star \\
&Clo = Val \to Cont \to Val^\star \\
&Val = Exp \cup Clo \\
&Val^\star \approx ((Env \times Val) \;+\; (\{err\} \times \Sigma^*) \;+\; (\{typerr\} \times \Sigma^*))_\perp
\end{aligned}
$$

W implementacji:

- $Val = Exp$
- $Exp \;::=\; \ldots \mid Clo$

Dynamiczne typowanie

$$\begin{aligned}
&\iota_{num} : Exp \to Val \\
&\iota_{bool} : Exp \to Val \\
&\iota_{str} : Exp \to Val \\
&\iota_{list} : Exp^* \to Val \\
&\iota_{cons} : Val \times Val \to Val \\
&\iota_{clo} : Clo \to Val
\end{aligned}$$

Dynamiczne typowanie

```
data TypeDef repr = TypeDef {
  name :: String,
  toRepr :: Expr -> Maybe repr
}
```

Dynamiczne typowanie

```
data TypeDef repr = TypeDef {
  name :: String,
  toRepr :: Expr -> Maybe repr
}

numType :: TypeDef Int
numType = TypeDef "num" extract where
  extract (Num n) = Just n
  extract _ = Nothing

boolType :: TypeDef Bool
boolType = TypeDef "bool" extract where
  extract (Bool b) = Just b
  extract _ = Nothing

-- atom, string, cons, closure
```

Dynamiczne typowanie

```haskell
typed :: TypeDef repr -> (repr -> Val) -> Cont
typed t cont _ val =
  case (toRepr t) val of
    Just v -> cont v
    Nothing -> TypeErr ("Expected type " ++ name t)
```

Dynamiczne typowanie

```
typed :: TypeDef repr -> (repr -> Val) -> Cont
typed t cont _ val =
  case (toRepr t) val of
    Just v -> cont v
    Nothing -> TypeErr ("Expected type " ++ name t)

evalExpr (List [Atom "+", e0, e1]) env k =
  evalExpr e0 env $ typed numType $ \n0 ->
  evalExpr e1 env $ typed numType $ \n1 ->
  k env (Num (n0 + n1))
```

Semantyka

$$[\![\cdot]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^\star$$

$$\begin{aligned}
[\![n]\!]\eta\kappa &= \kappa\ \eta\ (\iota_{num}\ n) \\
[\![b]\!]\eta\kappa &= \kappa\ \eta\ (\iota_{bool}\ b\ ) \\
[\![x]\!]\eta\kappa &= \kappa\ \eta\ (\eta\ x) \\
[\![s]\!]\eta\kappa &= \kappa\ \eta\ (\iota_{str}\ s\ )
\end{aligned}$$

$$\begin{array}{l}
[\![(\oplus\ e_0\ e_1)]\!]\eta\kappa = \\
\quad [\![e_0]\!]\ \eta\ (\lambda\eta_0 n_0\ . \\
\quad\quad [\![e_1]\!]\ \eta\ (\lambda\eta_1 n_1\ . \\
\quad\quad\quad \kappa\ \eta\ (\iota_{num}\ (n_0 \oplus n_1)) \\
\quad\quad)_{num^*} \\
\quad)_{num^*}
\end{array}$$

$$\oplus \in \{+, -, *\}$$

Semantyka

$$[\![\cdot]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^{\star}$$

$$
\begin{aligned}
&[\![(\oslash\ e_0\ e_1)]\!]\eta\kappa = \\
&\quad [\![e_0]\!]\ \eta\ (\lambda\eta_0 n_0\ . \\
&\qquad [\![e_1]\!]\ \eta\ (\lambda\eta_1 n_1\ . \\
&\qquad\quad \mathrm{cond}(n_1 = 0, \text{<err,"div. by 0">}, \kappa\ \eta\ (\iota_{num}\ (n_0 \oslash n_1))) \\
&\qquad)_{num^*} \\
&\quad)_{num^*}
\end{aligned}
$$

$$\oslash \in \{/, \%\}$$

Semantyka

$$[\![\cdot]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^\star$$

$$\begin{array}{l}
[\![(\oslash\ e_0\ e_1)]\!]\eta\kappa = \\
\quad [\![e_0]\!]\ \eta\ (\lambda \eta_0 n_0\ . \\
\quad\quad [\![e_1]\!]\ \eta\ (\lambda \eta_1 n_1\ . \\
\quad\quad\quad \text{cond}(n_1\ = 0, \text{<err,"div. by 0">}, \kappa\ \eta\ (\iota_{num}\ (n_0 \oslash n_1))) \\
\quad\quad)_{num^*} \\
\quad)_{num^*}
\end{array}$$

$$\oslash \in \{/, \%\}$$

Podobnie dla pozostałych operatorów (and, or, =, <, ≤, ...)

Semantyka

$$[\![\cdot]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^\star$$

$$[\![(\text{not } e)]\!]\eta\kappa = [\![e]\!]\ \eta\ (\lambda\eta' b\ .\ \kappa\ \eta\ (\iota_{bool}(\neg b)))_{bool^*}$$

$$\begin{aligned}
&[\![(\text{cond } e\ e_0\ e_1)]\!]\eta\kappa = \\
&\quad [\![e]\!]\ \eta\ (\lambda\eta' b\ .\ \text{cond}(\text{b}, [\![e_0]\!]\eta\kappa, [\![e_1]\!]\eta\kappa)\ )_{bool^*}
\end{aligned}$$

$$\begin{aligned}
&[\![(\text{cons } e_0\ e_1)]\!]\eta\kappa = \\
&\quad [\![e_0]\!]\ \eta\ (\lambda\eta_0 v_0\ . \\
&\qquad [\![e_1]\!]\ \eta\ (\lambda\eta_1 v_1\ . \\
&\qquad\quad \kappa\ \eta\ (\iota_{cons} <v_0, v_1>) \\
&\qquad) \\
&\quad)
\end{aligned}$$

$$[\![(\text{car } e)]\!]\eta\kappa = [\![e]\!]\ \eta\ (\lambda\eta' <v1, v2>\ .\ \kappa\ \eta\ v1)_{cons^*}$$

$$[\![(\text{cdr } e)]\!]\eta\kappa = [\![e]\!]\ \eta\ (\lambda\eta' <v1, v2>\ .\ \kappa\ \eta\ v2)_{cons^*}$$

Semantyka

$$[\![\cdot]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^\star$$

$$[\![(\mathrm{quote}\ (e_0 \ldots e_n))]\!]\eta\kappa = \kappa\ \eta\ (\iota_{list}(e_0 \cdots e_n)))$$

$$[\![(\mathrm{lambda}\ x\ e)]\!]\eta\kappa = \kappa\ \eta\ (\iota_{clo}(\lambda v\kappa'\ .\ [\![e]\!]\ \eta[x \mapsto v]\ \kappa'))$$

$$[\![(e_0\ e_1)]\!]\eta\kappa = [\![e_0]\!]\ \eta\ (\lambda\eta_0 f\ .\ [\![e_1]\!]\ \eta\ (\lambda\eta_1 v\ .\ f\ v\ \kappa))_{clo^*}$$

$$[\![(e_1\ e_2\ e_3 \ldots e_n)]\!]\eta\kappa = [\![(\ldots(e_1\ e_2)\ e_3) \ldots e_n)]\!]\ \eta\ \kappa$$

$$[\![(\mathrm{letrec}\ f\ x\ e'\ e)]\!]\eta\kappa = [\![e]\!]\ \eta[f \mapsto (\iota_{clo}\ (\mathit{fix}\ F))]\ \kappa$$

gdzie:

$$F\ g\ v\ \kappa' = [\![e']\!]\ \eta[f \mapsto (\iota_{clo}\ g)][x \mapsto v]\ \kappa'$$

Semantyka

$$[\![\cdot]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^\star$$

$$[\![(\text{define}\, x\, e)]\!]\eta\kappa = \begin{cases} \text{<err, "x already bound">} & \text{jeśli } x \in \mathit{dom}(\eta) \\ [\![e]\!]\ \eta\ (\lambda\eta' v\ .\ \kappa\ \eta'[x \mapsto v]\ v) & \text{w p.p.} \end{cases}$$

$$[\![(\text{begin}\ e_1 \ldots e_n)]\!]\eta\kappa = [\![e_1 \ldots e_n]\!]_{\mathit{block}}\ (\mathit{push\ empty}\ \eta)\ \kappa$$

$$[\![\cdot]\!]_{\mathit{block}} : \mathit{Exp}^* \to \mathit{Env} \to \mathit{Cont} \to \mathit{Val}^\star$$

$$[\![e]\!]_{\mathit{block}}\ \eta\kappa = [\![e]\!]\ \eta\ (\lambda\eta' v\ .\ \kappa\ (\mathit{pop}\ \eta')\ v)$$
$$[\![e_1\ e_2 \ldots e_n]\!]_{\mathit{block}}\ \eta\kappa = [\![e_1]\!]\ \eta\ (\lambda\eta' v\ .\ [\![e_2 \ldots e_n]\!]_{\mathit{block}}\ \eta'\kappa)$$

Cukier syntaktyczny

$$\mathcal{D} : Exp \to Exp$$

$$\mathcal{D}[\![\text{ true }]\!] = \text{\#t}$$
$$\mathcal{D}[\![\text{ false }]\!] = \text{\#f}$$

$$\mathcal{D}[\![\text{ (if } b\ e_0\ e_1\text{) }]\!] = \text{(cond } \mathcal{D}[\![b]\!]\ \mathcal{D}[\![e_0]\!]\ \mathcal{D}[\![e_1]\!]\text{)}$$

$$\mathcal{D}[\![\text{ (let } x\ e'\ e\text{) }]\!] = \text{((lambda } x\ \mathcal{D}[\![e]\!]\text{) } \mathcal{D}[\![e']\!]\text{)}$$
$$\mathcal{D}[\![\text{ (let* ((} x_1\ e_1\text{) .. (} x_n\ e_n\text{)) } e\text{) }]\!] = \mathcal{D}[\![\text{(let } x_1\ e_1\ \text{(..(let } x_n\ e_n\ e\text{)..))}]\!]$$

Cukier syntaktyczny

$$\mathcal{D} : \mathit{Exp} \to \mathit{Exp}$$

$$\mathcal{D}[\![\ \text{nil}\ ]\!] = (\text{quote}\ ())$$
$$\mathcal{D}[\![\ (\text{nil?}\ e)\ ]\!] = (\text{equals?}\ \mathcal{D}[\![\ \text{nil}\ ]\!]\ \mathcal{D}[\![\ \text{e}\ ]\!])$$

$$\mathcal{D}[\![\ (\text{lambda}\ (x_1 \ldots x_n)\ e)\ ]\!] =$$
$$(\text{lambda}\ x_1\ (\ldots(\text{lambda}\ x_n\ \mathcal{D}[\![e]\!])\ldots))$$

$$\mathcal{D}[\![\ (\text{defun}\ (f\ x_1 \ldots x_n)\ e)\ ]\!] =$$
$$(\text{define}\ f\ \mathcal{D}[\![\ (\text{lambda}\ (x_1 \ldots x_n)\ e\ )\ ]\!]\ )$$

$$\mathcal{D}[\![\ (e_1 \ldots e_n)\ ]\!] = (\mathcal{D}[\![e_1]\!] \ldots \mathcal{D}[\![e_n]\!])$$
$$\mathcal{D}[\![\ e\ ]\!] = e$$

Demo

Czego brakuje?

- rekursja wzajemna (letrec*)
- kontynuacje jako wartości pierwszego rzędu (call/cc)
- moduły ładowane z zewnętrznych plików
- system makr
- uboga biblioteka standardowa
- dokument z formalnym opisem semantyki

Materiały

- Reynolds
- Racket (REPL)
- Write yourself Scheme in 48 hours
- Notatki z SJP