

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

$\langle Exp \rangle$::= n
 | $\#t \mid \#f$
 | $atom$
 | $"string"$
 | $(\langle ExpList \rangle)$

$\langle ExpList \rangle$::= ϵ
 | $\langle Exp \rangle \langle ExpList \rangle$

Dziedzina wartości

$$Env = (Ide \rightarrow Val)^*$$

$$Cont = Env \rightarrow Val \rightarrow Val^*$$

$$Clo = Val \rightarrow Cont \rightarrow Val^*$$

$$Val = Exp \cup Clo$$

$$Val^* \approx ((Env \times Val) + (\{err\} \times \Sigma^*) + (\{typerr\} \times \Sigma^*))_{\perp}$$

Dziedzina wartości

$$Env = (Ide \rightarrow Val)^*$$

$$Cont = Env \rightarrow Val \rightarrow Val^*$$

$$Clo = Val \rightarrow Cont \rightarrow Val^*$$

$$Val = Exp \cup Clo$$

$$Val^* \approx ((Env \times Val) + (\{err\} \times \Sigma^*) + (\{typerr\} \times \Sigma^*))_{\perp}$$

W implementacji:

- ▶ $Val = Exp$
- ▶ $Exp ::= \dots \mid Clo$

Dynamiczne typowanie

$\iota_{num} : Exp \rightarrow Val$

$\iota_{bool} : Exp \rightarrow Val$

$\iota_{str} : Exp \rightarrow Val$

$\iota_{list} : Exp^* \rightarrow Val$

$\iota_{cons} : Val \times Val \rightarrow Val$

$\iota_{clo} : Clo \rightarrow Val$

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

```
typed :: TypeDef repr -> (repr -> Val) -> Cont
typed t cont _ val =
  case (toRepr t) val of
    Just v -> cont v
    Nothing -> TypeError ("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 -> TypeError ("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

$$\llbracket \cdot \rrbracket : Exp \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

$$\llbracket n \rrbracket \eta \kappa = \kappa \, \eta \, (\iota_{num} \, n)$$

$$\llbracket b \rrbracket \eta \kappa = \kappa \, \eta \, (\iota_{bool} \, b)$$

$$\llbracket x \rrbracket \eta \kappa = \kappa \, \eta \, (\eta \, x)$$

$$\llbracket s \rrbracket \eta \kappa = \kappa \, \eta \, (\iota_{str} \, s)$$

$$\begin{aligned} \llbracket (\oplus e_0 e_1) \rrbracket \eta \kappa = & \\ & \llbracket e_0 \rrbracket \eta \, (\lambda \eta_0 n_0 . \\ & \quad \llbracket e_1 \rrbracket \eta \, (\lambda \eta_1 n_1 . \\ & \quad \quad \kappa \, \eta \, (\iota_{num} \, (n_0 \oplus n_1))) \\ & \quad)_{num^*} \\ &)_{num^*} \end{aligned}$$

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

Semantyka

$$\llbracket \cdot \rrbracket : Exp \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

$$\begin{aligned} \llbracket (\oslash e_0 e_1) \rrbracket \eta \kappa = & \\ & \llbracket e_0 \rrbracket \eta (\lambda \eta_0 n_0 . \\ & \quad \llbracket e_1 \rrbracket \eta (\lambda \eta_1 n_1 . \\ & \quad \quad \text{cond}(n_1 = 0, \langle \text{err}, "div. by 0" \rangle, \kappa \eta (\iota_{num} (n_0 \oslash n_1))) \\ & \quad)_{num^*} \\ &)_{num^*} \end{aligned}$$

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

Semantyka

$$\llbracket \cdot \rrbracket : Exp \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

$$\begin{aligned} \llbracket (\oslash e_0 e_1) \rrbracket \eta \kappa = & \\ & \llbracket e_0 \rrbracket \eta (\lambda \eta_0 n_0 . \\ & \quad \llbracket e_1 \rrbracket \eta (\lambda \eta_1 n_1 . \\ & \quad \quad \text{cond}(n_1 = 0, \langle \text{err}, "div. by 0" \rangle, \kappa \eta (\iota_{num} (n_0 \oslash n_1))) \\ & \quad)_{num^*} \\ &)_{num^*} \end{aligned}$$

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

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

Semantyka

$$\llbracket \cdot \rrbracket : Exp \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

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

$$\begin{aligned} \llbracket (\text{cond } e_0 e_1) \rrbracket \eta \kappa = \\ \llbracket e \rrbracket \eta (\lambda \eta' b . \text{cond}(b, \llbracket e_0 \rrbracket \eta \kappa, \llbracket e_1 \rrbracket \eta \kappa))_{bool^*} \end{aligned}$$

$$\begin{aligned} \llbracket (\text{cons } e_0 e_1) \rrbracket \eta \kappa = \\ \llbracket e_0 \rrbracket \eta (\lambda \eta_0 v_0 . \\ \llbracket e_1 \rrbracket \eta (\lambda \eta_1 v_1 . \\ \kappa \eta (\iota_{cons} \langle v_0, v_1 \rangle)) \\) \\) \end{aligned}$$

$$\llbracket (\text{car } e) \rrbracket \eta \kappa = \llbracket e \rrbracket \eta (\lambda \eta' \langle v_1, v_2 \rangle . \kappa \eta v_1)_{cons^*}$$

$$\llbracket (\text{cdr } e) \rrbracket \eta \kappa = \llbracket e \rrbracket \eta (\lambda \eta' \langle v_1, v_2 \rangle . \kappa \eta v_2)_{cons^*}$$

Semantyka

$$\llbracket \cdot \rrbracket : Exp \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

$$\llbracket (\text{quote } (e_0 \dots e_n)) \rrbracket \eta \kappa = \kappa \eta (\iota_{list}(e_0 \dots e_n))$$

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

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

$$\llbracket (e_1 \ e_2 \ e_3 \dots e_n) \rrbracket \eta \kappa = \llbracket (\dots (e_1 \ e_2) \ e_3) \dots e_n \rrbracket \eta \kappa$$

$$\llbracket (\text{letrec } f \ x \ e' \ e) \rrbracket \eta \kappa = \llbracket e \rrbracket \eta [f \mapsto (\iota_{clo}(\text{fix } F))] \kappa$$

gdzie:

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

Semantyka

$$\llbracket \cdot \rrbracket : Exp \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

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

$$\llbracket (\text{begin } e_1 \dots e_n) \rrbracket \eta \kappa = \llbracket e_1 \dots e_n \rrbracket_{block} (\text{push empty } \eta) \kappa$$

$$\llbracket \cdot \rrbracket_{block} : Exp^* \rightarrow Env \rightarrow Cont \rightarrow Val^*$$

$$\llbracket e \rrbracket_{block} \eta \kappa = \llbracket e \rrbracket \eta (\lambda \eta' v . \kappa (\text{pop } \eta') v)$$

$$\llbracket e_1 e_2 \dots e_n \rrbracket_{block} \eta \kappa = \llbracket e_1 \rrbracket \eta (\lambda \eta' v . \llbracket e_2 \dots e_n \rrbracket_{block} \eta' \kappa)$$

Cukier syntaktyczny

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

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

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

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

$$\mathcal{D}[\![\text{let } x \ e' \ e]\!] = ((\text{lambda } x \ \mathcal{D}[\![e]\!]) \ \mathcal{D}[\![e']\!])$$

$$\mathcal{D}[\![\text{let}^* ((x_1 \ e_1) \dots (x_n \ e_n)) \ e]\!] = \mathcal{D}[\![\text{let } x_1 \ e_1 \ (\dots (\text{let } x_n \ e_n \ e) \dots)]\!]$$

Cukier syntaktyczny

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

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

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

$$\begin{aligned} \mathcal{D}[\![\text{lambda } (x_1 \dots x_n) e]\!] = \\ (\text{lambda } x_1 (\dots (\text{lambda } x_n \mathcal{D}[\![e]\!]) \dots)) \end{aligned}$$

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

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

$$\mathcal{D}[\![e]\!] = e$$

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

Materialy

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