Języki i Paradygmaty Programowania
Semantyka

Marcin Benke

MIM UW

10 kwietnia 2017

Plan

- Semantyka ogólnie
- Semantyka statyczna — typy
- Semantyka operacyjna, denotacyjna i interpretery

- Semantyka — opis znaczenia konstrukcji języka.
- Triada semiotyczna (oryginalnie dla języków naturalnych)
  - Składnia: “kształt” języka
  - Semantyka: znaczenie języka
  - Pragmatyka: użycie języka
- W odniesieniu do języków programowania pierwsze dwa elementy przeważnie formalnie, ostatni nieformalnie.

Semantyka statyczna i dynamiczna

W większości języków/implementacji występuje rozróżnienie faz

- statyczna — analiza (poprawności) programu bez jego wykonania
- dynamiczna — wykonanie programu

Stąd mówimy czasem o semantyce statycznej (głównie analiza typów) i dynamicznej

Granice między fazami nie zawsze są ostre

Sama "semantyka" oznacza zwykle semantykę dynamiczną

W językach dynamicznych kontrola typów w fazie dynamicznej

Po co nam typy?

A type system
is a syntactic method
for automatically checking
the absence of certain erroneous behaviors
by classifying program phrases,
according to the kinds of values they compute.

— Benjamin Pierce

Po co nam typy?

Potrzeba rozróżnienia pomiędzy różnymi rodzajami obiektów; operacje wykonywane na napisach są inne od wykonywanych na liczbach.

Typy pozwalają na uniknięcie pewnych błędów w programie, mogą zapewniać niezmienniki.

Typy pozwalają na klasyfikację obiektów. Możliwość definiowania nowych typów zwiększa siłę wyrazu języka.

> *System typów jest syntaktyczną dyscypliną narzucającą poziomy abstrakcji.*
>
> — John C. Reynolds

Po co nam typy?

Zależnie od systemu, kontrola typów może zapobiec

- zastosowaniu funkcji do niewłaściwej liczby argumentów,
- zastosowaniu funkcji całkowitej do napisu,
- użyciu niezadeklarowanych zmiennych,
- użyciu niezainicjalizowanych zmiennych,
- funkcjom, które nie dają wyniku (a powinny),
- dzieleniu przez zero,
- wyjściu poza zakres tablicy,
- algorytmom sortowania, które źle sortują,
- …

Dlaczego nie?

W każdym (rozstrzygalnym) systemie typów są programy, które są dobrze zdefiniowane, ale nie są poprawne typowo, np

```
length ["hello", 'w', False]
```

Czasem jesteśmy zmuszeni napisać program większy lub wolniejszy niż byśmy chcieli aby dopasować go do systemu typów.

"I'm not against types, but I don't know of any type systems that aren't a complete pain, so I still like dynamic typing." — Alan Kay

Typowanie dynamiczne

- Kontrola typów w czasie wykonania
- Daje programiście większą elastyczność, ale nie za darmo.
- Skrajnym przypadkiem jest asembler: bardzo elastyczny, ale żadnych zabezpieczeń.

Pozwala na pisanie funkcji, które zachowują się różnie dla różnych typów wejścia, np:

```
def negate(x):
  if type(x) == int:
    return -x
  else:
    return not(x)
```

Typowanie dynamiczne

- Każda wartość niesie informację o swoim typie.
- Te informacje są przeważnie dostępne dla programisty (np. **type** w Pythonie, **typeof** w Javascript).
- Każda operacja sprawdza typy swoich argumentów.
- Niezgodność typów $\rightsquigarrow$ błąd wykonania (często: wyjątek).
- Przykłady języków: Lisp, Smalltalk, Python, Ruby, JS…
- Wiele języków łączy typowanie statyczne i dynamiczne, np.

```
import Data.Dynamic
d :: Typeable a => a -> Dynamic
d = toDyn
hlist = [d "hello", d 'w', d False]
main = print $ length hlist
```

Typowanie statyczne

- Kontrola typów w czasie kompilacji.
- Niepoprawne typowo programy są odrzucane (przy typowaniu dynamicznym błędy typowe mogą zostać długo niewykryte).
- Mniejsza elastyczność (o ile — to zależy od systemu typów)
- Nowoczesne systemy typów dopuszczają:
  - przeciążanie
  - polimorfizm
  - kontrolowane typowanie dynamiczne
- Prawie takie same możliwości co przy typowaniu dynamicznym.
- Silne systemy typów pozwalają wyrazić niemal dowolne własności programów.

Silne i słabe typowanie

- Silne typowanie (kontroler typów ma władzę)
  - Niezgodność typów uniemozliwia uruchomienie programu
  - Gwarantuje bezpieczeństwo typowe w trakcie wykonania
  - Problemy gdy system typów nie jest zbyty ekspresywny (np. Pascal).
- Słabe typowanie (programista ma władzę)
  - programista może obejść kontrolę typów
  - żadnych gwarancji
  - przykłady:
    - Java: `(String) vector.get(1)`
    - C: `(int *)123`
- **Uwaga:** epitety “silne” i “słabe” nie mają tu znaczenia wartościującego.

Systemy typów

System typów — zbiór typów i reguł wnioskowania o poprawności typowej konstrukcji języka (głównie wyrażeń)

Reguły są zwykle wyrażane w postaci

$$\frac{A_1 \quad \ldots \quad A_n}{B}$$

oznaczającej "jeśli A_1 i ... i A_n to możemy wnioskować B".

Prosty system typów

Typy:

$$\tau ::= \textbf{int} \mid \textbf{bool}$$

Wyrażenia:

$$e ::= n \mid b \mid e_1 + e_2 \mid e_1 = e_2 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$$

Reguły:

$$\frac{}{n : \textbf{int}} \qquad \frac{}{b : \textbf{bool}}$$

$$\frac{e_1 : \textbf{int} \quad e_2 : \textbf{int}}{e_1 + e_2 : \textbf{int}} \qquad \frac{e_1 : \textbf{int} \quad e_2 : \textbf{int}}{e_1 = e_2 : \textbf{bool}}$$

$$\frac{e_0 : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau}$$

Wyprowadzanie typów

Aby wykazać, że wyrażenie e ma typ τ możemy skonstruować *wyprowadzenie typu* (dowód w naszym systemie typów).

$$\dfrac{\dfrac{1 : \mathbf{int} \quad 2 : \mathbf{int}}{1 + 2 : \mathbf{int}} \qquad 3 : \mathbf{int}}{(1 + 2) + 3 : \mathbf{int}}$$

$$\dfrac{\dfrac{1 : \mathbf{int} \quad 0 : \mathbf{int}}{1 = 0 : \mathbf{bool}} \qquad 1 : \mathbf{int} \quad 2 : \mathbf{int}}{\mathbf{if}\ 1 = 0\ \mathbf{then}\ 1\ \mathbf{else}\ 2 : \mathbf{int}}$$

Zmienne

Rozszerzmy nasz język o zmienne:

$$e ::= x \mid n \mid b \mid e_1 + e_2 \mid e_1 = e_2 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$$

Typ zmiennej zależy od kontekstu, rozszerzymy zatem nasze reguły typowania o informacje o kontekście (środowisko).

Używamy notacji

$$\Gamma \vdash e : \tau$$

znaczącej "w środowisku Γ, wyrażenie e ma typ τ".

Środowisko przypisuje zmiennym typy, tzn. jest zbiorem par $(x : \tau)$, gdzie x jest zmienną zaś τ typem.

Reguły typowania w kontekście

Typy zmiennych odczytujemy ze środowiska:

$$\Gamma(x:\tau) \vdash x:\tau$$

$$\Gamma \vdash n : \mathbf{int} \qquad \Gamma \vdash b : \mathbf{bool}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{int} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 + e_2 : \mathrm{int}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{int} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 = e_2 : \mathrm{bool}}$$

$$\frac{\Gamma \vdash e_0 : \mathrm{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

Kontrola typów w językach imperatywnych

Rozważmy mały język imperatywny:

$$e ::= x \mid n \mid b \mid e_1 + e_2 \mid e_1 = e_2$$
$$s ::= x := e \mid \textbf{while } e \textbf{ do } s \mid s; s$$

Wprowadzimy nowy osąd dla programów

$$\Gamma \vdash_P s$$

o znaczeniu “w środowisku Γ, program s jest poprawny”.

Kontrola typów w językach imperatywnych

Niektóre reguły będą używać zarówno $\vdash$ jak $\vdash_P$, np.

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash_P x := e}$$

czy

$$\frac{\Gamma \vdash e : \mathrm{bool} \quad \Gamma \vdash_P p}{\Gamma \vdash_P \textbf{while } e \textbf{ do } p}$$

Deklaracje

Możemy uznać deklarację jako rodzaj instrukcji oraz dodać regułę

$$\frac{\Gamma(x : \tau) \vdash_P p}{\Gamma \vdash_P \textbf{var}\ x : \tau;\ p}$$

Deklaracje

inną możliwością jest wprowadzenie nowego typu osądu, $\vdash_D$:

$$\Gamma \vdash_D (\mathbf{var}\ x : \tau) : \Gamma(x : \tau)$$

$$\frac{\Gamma \vdash_D ds : \Gamma' \quad \Gamma' \vdash_P p}{\Gamma \vdash_P ds;\ p}$$

Deklaracje

Można też pozwolić instrukcjom na modyfikację środowiska.
Deklaracje i instrukcje mogą być wtedy swobodnie przeplatane:

$$\frac{\Gamma \vdash_P s : \Gamma' \quad \Gamma' \vdash_P p : \Gamma''}{\Gamma \vdash_P s;\ p : \Gamma''}$$

Kontrola typów w językach funkcyjnych

Typy:

$$\tau ::= \mathrm{int} \mid \mathrm{bool} \mid \tau_1 \to \tau_2$$

Wyrażenia:

$$\begin{aligned} E ::= {} & x \mid n \mid b \mid e_1 e_2 \mid \lambda(x{:}\tau).e \\ & \mid e_1 + e_2 \mid e_1 = e_2 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \end{aligned}$$

Reguły typowania

$$\frac{\Gamma(x : \tau) \vdash e : \rho}{\Gamma \vdash \lambda(x : \tau).e : \tau \to \rho}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \rho \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \rho}$$

Sprawdzanie vs wyprowadzanie typów

Mamy dwa zagadnienia algorytmiczne:

- **Sprawdzanie** czy wyrażenie e ma typ τ? $\quad \Gamma \vdash e \Leftarrow \tau$
- **Wyprowadzenie:** jaki typ ma wyrażenie e? $\quad \Gamma \vdash e \Rightarrow \tau$

Algorytmy możemy opisywać podobnie do systemów typów:

$$\frac{\Gamma(x:\tau) \vdash e \Rightarrow \rho}{\Gamma \vdash \lambda(x:\tau).e \Rightarrow \tau \to \rho}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \to \rho \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash e_1 e_2 \Rightarrow \rho}$$

$$\Gamma(x:\tau) \vdash x \Rightarrow \tau \qquad\qquad \frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau = \tau'}{\Gamma \vdash e \Leftarrow \tau}$$

Przykłady

$$\frac{\Gamma \vdash a \Leftarrow \mathrm{int} \quad \Gamma \vdash b \Leftarrow \mathrm{int}}{\Gamma \vdash a + b \Rightarrow \mathrm{int}}$$

$$\frac{x : \mathrm{int} \vdash x \Rightarrow \mathrm{int}}{x : \mathrm{int} \vdash x \Leftarrow \mathrm{int}}$$

$$\dfrac{\dfrac{\dfrac{x : \mathrm{int} \vdash x \Leftarrow \mathrm{int} \quad x : \mathrm{int} \vdash 1 \Leftarrow \mathrm{int}}{x : \mathrm{int} \vdash x + 1 \Rightarrow \mathrm{int}}}{\vdash \lambda(x : \mathrm{int}).x + 1 \Rightarrow \mathrm{int} \to \mathrm{int}} \qquad \vdash 7 \Leftarrow \mathrm{int}}{\vdash (\lambda(x : \mathrm{int}).x + 1)\ 7 \Rightarrow \mathrm{int}}$$

Przykłady

$$\dfrac{\dfrac{\dfrac{x : \mathsf{int}, y : \mathsf{int} \vdash x \Leftarrow \mathsf{int} \quad x : \mathsf{int}, y : \mathsf{int} \vdash y \Leftarrow \mathsf{int}}{x : \mathsf{int}, y : \mathsf{int} \vdash x + y \Rightarrow \mathsf{int}}}{x : \mathsf{int} \vdash \lambda(y : \mathsf{int}).x + y \Rightarrow \mathsf{int} \to \mathsf{int}}}{\vdash \lambda(x : \mathsf{int}).\lambda(y : \mathsf{int}).x + y \Rightarrow \mathsf{int} \to \mathsf{int} \to \mathsf{int}}$$

Reguły typowania są sterowane składnią i można je łatwo zakodować:

```
typeOf (EInt _) = return TInt
typeOf (EVar n) = askType n

typeOf (ELam n t e1) = do
  t1 <- local (addVar n t) $ typeOf e1
  return $ t0 :-> t1

typeOf exp@(EApp e1 e2) = do
  t1 <- typeOf e1
  case t1 of
    ta :-> tr | checkType e2 ta
              | otherwise -> throwError ...

checkType e t = do
  t' <- typeOf e
  if t == t' then return t else throwError ...
```

Rekonstrukcja typów

Jeśli typy identyfikatorów nie są znane, musimy *zrekonstruować* pasujące typy.

Reguły typowania pozostają te same; reguła dla funkcji odpowiada zmienionej składni:

$$\frac{\Gamma(x:\tau) \vdash e : \rho}{\Gamma \vdash \lambda x.e : \tau \to \rho}$$

co prowadzi do problemu: skąd wziąć dobre τ?

Możemy uczynić τ niewiadomą (zmienną typową).
Proces typowania da nam typ wraz z układem równań

Przy każdym użyciu reguły aplikacji

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

dodajemy do układu równanie $\tau_1 = \tau_2$.

Przykłady rekonstrukcji typów

Możemy podobnie jak w Haskellu traktować $a + b$ jako aplikację $(+)\ a\ b$.

$$\dfrac{\dfrac{\dfrac{x : \tau_x \vdash x : \tau_x \quad x : \tau_x \vdash 1 : \mathrm{int}}{x : \tau_x \vdash x + 1 : \mathrm{int}} \{\tau_x = \mathrm{int}\}}{\vdash \lambda x.x + 1 : \tau_x \to \mathrm{int}} \qquad \vdash 7 : \mathrm{int}}{\vdash (\lambda x.x + 1)\ 7 : \mathrm{int}}$$

z niemal trywialnym układem równań $\{\tau_x = \mathrm{int}\}$.

Przykłady rekonstrukcji typów

Podobnie mozemy uzyskać

$$\vdash (\lambda f.\lambda x.f(fx))(\lambda y.y)\ 7 : \tau_f'$$

Z równaniami:

$$\tau_f = \tau_x \to \tau_f' \quad (1)$$

$$\tau_f = \tau_f' \to \tau_f' \quad (2)$$

$$\tau_f \to (\tau_x \to \tau_f') = (\tau_y \to \tau_y) \to (\tau_x \to \tau_f') \quad (3)$$

$$\tau_x \to \tau_f' = \mathrm{int} \to \tau_f' \quad (4)$$

Rozwiązywanie równań: unifikacja

Otrzymane układy możemy rozwiązywać przez upraszczanie.

W naszym przykładzie możemy uprościć równanie (4)

$$\tau_x \to \tau_f' = \mathrm{int} \to \tau_f'$$

do

$$\tau_x = \mathrm{int}$$

i podstawić int za x w pozostałych, otrzymując

$$\begin{aligned}
\tau_f &= \mathrm{int} \to \tau_f' \\
\tau_f &= \tau_f' \to \tau_f' \\
\tau_f \to (\mathrm{int} \to \tau_f') &= (\tau_y \to \tau_y) \to (\mathrm{int} \to \tau_f') \\
\tau_x &= \mathrm{int}
\end{aligned}$$

$$
\begin{aligned}
\tau_f &= \text{int} \to \tau_f' && (5)\\
\tau_f &= \tau_f' \to \tau_f' && (6)\\
\tau_f \to (\text{int} \to \tau_f') &= (\tau_y \to \tau_y) \to (\text{int} \to \tau_f') && (7)\\
\tau_x &= \text{int} && (8)
\end{aligned}
$$

Dalej możemy połączyć (5) z (6) otrzymując

$$\text{int} \to \tau_f' = \tau_f' \to \tau_f'$$

co może być uproszczone do

$$\tau_f' = \text{int}.$$

Po podstawieniu int za τ_f', mamy

$$\tau_f = \text{int} \to \text{int} \tag{9}$$

$$\tau_f' = \text{int} \tag{10}$$

$$\tau_f \to (\text{int} \to \text{int}) = (\tau_y \to \tau_y) \to (\text{int} \to \text{int}) \tag{11}$$

$$\tau_x = \text{int} \tag{12}$$

Upraszczając (11) i podstawiając τ_f mamy

$$\text{int} \to \text{int} = \tau_y \to \tau_y$$

Podstawiając **int** za τ_y otrzymujemy

$$\tau_f = \text{int} \to \text{int} \tag{13}$$
$$\tau_f' = \text{int} \tag{14}$$
$$\tau_y = \text{int} \tag{15}$$
$$\tau_x = \text{int} \tag{16}$$

Opisany proces rozwiązywania równań nazywamy *unifikacją*. W przypadku sukcesu wynikiem jest *podstawienie*.

Fakt: unifikacja może być zastosowana do rozwiązywania równań na termach nad dowolną sygnaturą. Rozstrzygalna w czasie liniowym.

Kiedy unifikacja zawodzi

Unifikacja zawodzi, gdy napotka jedno z poniższych:

- Równanie postaci (k_1 i k_2 są różnymi stałymi)

$$k_1 = k_2$$

- Równanie postaci (k — stała):

$$k = t \to t'$$

- Równanie postaci

$$x = t$$

  gdzie x — zmienna a t zawiera x ale różny od x.

Na przykład, próba wyprowadzenia typu dla $\lambda x.xx$ prowadzi do

$$\tau_x = \tau_x \to \rho.$$

Ten term nie jest typowalny (w tym systemie).

Polimorfizm

Z drugiej strony, układ równań może mieć więcej niż jedno rozwiązanie. W efekcie możemy wyprowadzić więcej niż jeden typ dla danego wyrażenia. Na przykład, mamy

$$\vdash \lambda x.x : \tau \to \tau$$

dla każdego typu τ!

Dla opisu tego zjawiska możemy wprowadzić nową postać typu: $\forall\alpha.\tau$, gdzie α jest zmienną typową, oraz dwie nowe reguły:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall\alpha.\tau}\ \alpha \notin FV(\Gamma) \qquad\qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\rho/\alpha]}$$

($\tau[\rho/\alpha]$ oznacza typ τ z ρ podstawionym za α).

Polimorfizm — przykłady i smutna konstatacja

Możemy wyprowadzić

$$\vdash \lambda x.x : \forall \alpha.\alpha \to \alpha$$

Także $\lambda x.xx$ staje się typowalne:

$$\vdash \lambda x.xx : \forall \beta (\forall \alpha.\alpha \to \alpha) \to \beta \to \beta$$

Niestety nowy system nie jest już sterowany składnią: nowe reguły nie odpowiadają żadnym konstrukcjom składniowym i nie wiemy kiedy je stosować. Okazuje się, że rekonstrukcja typów w tym systemie jest **nierozstrzygalna**.

Płytki polimorfizm

Rekonstrukcja typów jest rozstrzygalna jeśli wprowadzimy pewne ograniczenie: kwantyfikatory są dopuszczalne tylko na najwyższym poziomie oraz mamy specjalną składnię dla wiązań polimorficznych:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma(x : \forall\vec{\alpha}.\tau_1) \vdash e : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e : \tau}$$

Taki system jest często wystarczający w praktyce. Na przykład możemy zastąpić konstrukcję **if** funkcją

$$\mathit{if\_then\_else\_} : \forall\alpha.\mathrm{bool} \to \alpha \to \alpha \to \alpha$$

Jest on również podstawą systemów dla ML i Haskella (choć ten ostatni jest znacznie bardziej skomplikowany).

Rekonstrukcja typów w ML

Typując **let** generalizujemy wszystkie możliwe zmienne

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma(x : \forall \vec{\alpha}.\tau_1) \vdash e : \tau \quad \vec{\alpha} = FTV(\tau_1) \setminus FTV(\Gamma)}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e : \tau}$$

(FTV — Free Type Variables — zmienne typowe nie związane kwantyfikatorem)

Dla każdego wystąpienia zmiennej, kwantyfikowane zmienne typowe zastępujemy świeżymi zmiennymi:

$$\Gamma(x : \forall \vec{\alpha}.\tau) \vdash x : \tau[\vec{\nu}/\vec{\alpha}] \qquad \vec{\nu} \text{ — świeże}$$

Jak poprzednio, otrzymujemy typ z układem równań, który rozwiązujemy przez unifikację.

Monada dla rekonstrukcji typów

```
type TCM a = ErrorT String (StateT TcState (Reader Env))

data TcState = TcState {
 tcsNS       :: NameSupply, -- dostawca nazw
 tcsSubst    :: Subst,      -- podstawienie
 constraints :: Constraints -- równania
}

tcmDeplete :: TCM String -- daje świeżą nazwę
```

Rodzaje semantyk

Semantyka może koncentrować się na różnych aspektach:

- co oznacza *wykonanie programu* (jakie *operacje* wykonuje);
- czym program *jest* (jaki obiekt matematyczny *denotuje*);
- jakie *zdania logiczne* są prawdziwe dla programu.

Rodzaje semantyk

Semantyka może koncentrować się na różnych aspektach:

- co oznacza *wykonanie programu* (jakie *operacje* wykonuje);
- czym program *jest* (jaki obiekt matematyczny *denotuje*);
- jakie *zdania logiczne* są prawdziwe dla programu.

Innymi słowy:

- semantyka operacyjna
- sematyka denotacyjna
- semantyka aksjomatyczna

Dla pełności warto jeszcze wymienić semantykę translacyjną — tłumaczenie na inny język o znanej semantyce.

Semantyka operacyjna

- Opisuje **jak** program jest wykonywany.
- Często skoncentrowana na znaczeniu kroków obliczenia.
- W pewnym sensie abstrakcyjny opis interpretera.
- Metoda *małych kroków*: definiujemy relację przejścia wyznaczającą ciąg

$$\langle P, \sigma \rangle \to \langle P_1, \sigma_1 \rangle \to \langle P_2, \sigma_2 \rangle \to \ldots \to \sigma_n$$

  opisujacy wykonanie programu w stanie σ. $\langle P_i, \sigma_i \rangle$ są konfiguracjami pośrednimi do stanu σ_n.
- Metoda *dużych kroków*: definiujemy relację "wykonanie programu P w stanie σ, prowadzi do stanu końcowego σ'."

$$\langle P, \sigma \rangle \Rightarrow \sigma'$$

- Relacje obliczeń przeważnie definiowane przez indukcję

Interpretery

Zauważmy, że

$$\langle P, \sigma \rangle \Rightarrow \sigma'$$

oznacza, że program P dla stanu startowego σ osiąga stan końcowy σ'. Relację $\Rightarrow$ możemy traktować jako funkcję typu

$$(\mathit{Prog}, \mathit{State}) \to \mathit{State}$$

Możemy zatem zapisać tę funkcję w języku programowania, uzyskując interpreter.

- Reguły mówią jak traktować każda kategorię syntaktyczną
- Przykład: przypisanie

$$\frac{\langle e, \sigma \rangle \Rightarrow v}{\langle x := e, \sigma \rangle \Rightarrow \sigma[x = v]}$$

  musimy obliczyć wartość wyrażenia, po czym zmodyfikować środowisko.

Semantyka denotacyjna

Przypisuje konstrukcjom języka obiekty matematyczne (denotacje), np.

$$State = Var \to Value$$

$$STMT = State \to State$$

$$\mathcal{S} : Stmt \to STMT$$

$$\mathcal{S}[\![s_1; s_2]\!] = \mathcal{S}[\![s_2]\!] \circ \mathcal{S}[\![s_1]\!]$$

$$\mathcal{S}[\![x := e]\!] = \lambda\sigma.\sigma[x := \mathcal{E}[\![e]\!]\sigma]$$

$$EXP = State \to Int$$

$$\mathcal{E} : Exp \to EXP$$

$$\mathcal{E}[\![x]\!] = \lambda\sigma.\sigma[x]$$

Semantyka i interpreter TINY

Adaptacja języka z SWP: Exp i BExp razem (dopuszczamy zmienne Bool)

```
data Stmt = Var := Exp
          | Sif Exp Stmt
          | Swhile Exp Stmt
          | Scomp Stmt Stmt

data Exp = Eeq Exp Exp
         | Ele Exp Exp
         | Eplus Exp Exp
         | Enum Int
         | Evar Var

type Var = String
```

Dziedziny semantyczne

$$
\begin{aligned}
\mathit{Value} &= \mathit{Int} + \mathit{Bool} \\
\mathit{State} &= \mathit{Var} \to \mathit{Value} \\
\mathit{EXP} &= \mathit{State} \to \mathit{Error} + \mathit{Value} \\
\mathit{STMT} &= \mathit{State} \to \mathit{Error} + \mathit{State} \\
\mathcal{E} &: \mathit{Exp} \to \mathit{EXP} \\
\mathcal{S} &: \mathit{Stmt} \to \mathit{STMT}
\end{aligned}
$$

alternatywnie

$$
\begin{aligned}
\mathit{IM}\ a &= \mathit{State} \to \mathit{Error} + (a, \mathit{State}) \\
\mathit{EXP} &= \mathit{IM}\ \mathit{Value} \\
\mathit{STMT} &= \mathit{IM}\ ()
\end{aligned}
$$

NB wtedy $\mathit{EXP} = \mathit{State} \to \mathit{Error} + (\mathit{Value}, \mathit{State})$

Wartości i środowisko

Typ dla wartości

```
data Value = Vint Int
           | Vbool Bool
  deriving Eq
```

Pamięć:

```
type State = Store
type Store = Data.Map.Map Var Value
```

Odczyt z i zapis do pamięci: $\sigma[x]$ oraz $\sigma[x := v]$:

```
getVar :: Var -> State -> Value
setVar :: State -> Var -> Value -> State
```

Interpretery eval i exec

Potrzebujemy interpreterów dla wyrażeń i instrukcji:

```
eval :: Expr -> State -> Value
exec :: Stmt -> State -> State
```

W praktyce te typy będą bardziej skomplikowane z uwagi na konieczność obsługi błędów, np. **getVar** może mieć typ

```
getVar ::  Var -> State -> Maybe Value
```

Interpretery monadyczne

Monady zostały wymyślone własnie dla semantyki denotacyjnej

```
type IM = StateT IntState (Either String)

eval :: Expr -> IM Value
exec :: Stmt -> IM ()
getVar :: Var -> IM Value
setVar :: Var -> Value -> IM ()
```

Moga się też przydać pomocnicze

```
evalI :: Expr -> IM Int
evalB :: Expr -> IM Bool
```

eval (1)

```
eval ::  Expr -> IM Value
```

Literały

$$\mathcal{E}[\![i]\!] = \lambda\sigma.inr\,\mathcal{N}[\![i]\!]$$

albo monadycznie

$$\mathcal{E}[\![i]\!] = return\ \mathcal{N}[\![i]\!]$$

```
eval (Enum i) = return (Vint i)
```

Zmienne:

```
eval (Evar v) = getVar v

getVar v = do
  r <- gets (Map.lookup v)
  maybe (throwError $ "Undefined var "++v) return r
```

Arytmetyka

$$\mathcal{E}[\![e1 + e2]\!] = \mathit{lift}\,(+)\,\mathcal{E}[\![e_1]\!]\,\mathcal{E}[\![e_2]\!]$$

```
eval (Eplus e1 e2) = do
   Vint i1 <- eval e1
   Vint i2 <- eval e2
   return $ Vint (i1+i2)
```

Albo krócej

```
eval (Eplus e1 e2) = Vint <$> liftM2
                       (+) (evalI e1) (evalI e2)
```

gdzie

```
liftM2 :: Monad m => (a1 -> a2 -> r)
                  -> m a1 -> m a2 -> m r
```

Porównania

```
eval (Ele e1 e2) = Vbool <$> liftM2
                     (<=) (evalI e1) (evalI e2)

eval (Eeq e1 e2) = Vbool <$> liftM2
                     (==) (eval e1) (eval e2)
```

czyli

```
eval (Eeq e1 e2) = do
   v1 <- eval e1
   v2 <- eval e2
   return (Vbool (v1==v2))
```

Uwaga:

- `v1, v2` mogą być dowolnego typu
- `==` jest tu równością w typie `Value`

Przypisanie

```
exec ::  Stmt -> IM ()

exec (x := e) = do
       v <- eval e
       setVar x v
-- ... = eval e >>= setVar x
-- ... = setVar x =<< eval e

setVar :: Var -> Value -> IM ()
setVar x v = modify (Map.insert x v)
```

exec (2)

```
exec (Sif e s) = do
  b <- evalB e
  when b (exec s)

exec w@(Swhile e s) = do
  b <- evalB e
  when b (exec s >> exec w)
```

Uwaga: nie piszemy jawnie punktów stałych: rekurencja w Haskellu to właśnie punkt stały

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

Sekwencjonowanie instrukcji

```
exec (Scomp s1 s2) = exec s1 >> exec s2
exec (Sblock ss) = mapM_ exec ss
```

Przykład

```
sumto n = Sblock [
        "i" := 1,
        "r" := 0,
         Swhile (Ele (Evar "i") (Enum n))
                (Sblock [
                  "r" := Eplus (Evar "r") (Evar "i"),
                  "i" := Eplus (Evar "i") (Enum 1)])
        ]
```

```
> putStrLn $ runProg (sumto 10)
[("i",11),("r",55)]
```

Prosty język funkcyjny

```
data Exp = EInt Int | EVar Name
         | ELam Name Exp | EApp Exp Exp
data Value = Vint Int | Vclos Exp Env

type Env = Map Name Value

eval :: Exp -> Env -> Value
eval (EInt i) env =  (Vint i)
eval (EVar v) env = env ! v
eval e@(ELam _ _) env = Vclos e env
eval (EApp e1 e2) env = apply (eval e1 env)
                              (eval e2 env)

apply :: Value -> Value -> Value
apply (Vclos (ELam x e1) env) e2 =
  eval e1 (Map.insert x e2 env)
```