Metody programowania 2017

Zadanie dodatkowe na pracownię nr 3

Celem zadania dodatkowego jest napisanie interpretera języka HDML, tak aby wraz z rozwiązaniami poprzednich zadań otrzymać kompletny system do weryfikowania prostych układów cyfrowych. Aby napisać interpreter, trzeba najpierw ustalić co programy dokładnie znaczą, czyli zdefiniować semantykę języka. Jest wiele sposobów definiowania semantyki języka. Najczęściej jest ona zadana przez nieformalny opis albo referencyjną implementację, jednak rodzi to pewne problemy dla implementatorów języka: nieformalny opis nie zawsze jest wystarczająco precyzyjny, a referencyjną implementację ciężko się czyta i w dodatku trzeba mieć zdefiniowaną semantykę języka w którym jest ona napisana. Dlatego semantykę języka HDML zadamy w sposób formalny poprzez tak zwaną *semantykę naturalną dużych kroków*[1].

Semantyka języka HDML

Znaczenie programów będzie zadane za pomocą 5-arnej relacji eval, wiążącej ze sobą program P, środowisku ρ (opisujące wartości zmiennych lokalnych), wyrażenie e, wartość v oraz efekty uboczne C. Dla poprawy czytelności $\mathsf{eval}(P, \rho, e, v, C)$ będziemy zapisywać jako $P; \rho \vdash e \Downarrow v/C$. Zapis taki należy intuicyjnie rozumieć jako: „wyrażenie e przy środowisko ρ wylicza się do wartości v powodując efekty uboczne C". Zanim jednak formalnie zdefiniujemy relację eval, powiedzmy co znaczą poszczególne jej składniki.

W definicji relacji będą pojawiać się elementy składni języka HDML, takie jak programy, wzorce, wyrażenia, itp. Przyjmijmy następujące konwencje:

- *liczby całkowite* będziemy oznaczać metazmiennymi n, n_1, n_2, ...,
- *zmienne* będziemy oznaczać metazmiennymi x, x_1, x_2, ...,
- *nazwy funkcji* będziemy oznaczać metazmiennymi f, f_1, f_2, ...,
- *wzorce* będziemy oznaczać metazmiennymi p, p_1, p_2, ...,
- *wyrażenia* będziemy oznaczać metazmiennymi e, e_1, e_2, ...,
- *program* będziemy oznaczać zmienną P. Dodatkowo zapis $(\texttt{def}\ f(p) = e) \in P$ oznacza, że definicja $\texttt{def}\ f(p) = e$ występuje w programie P.

Dodatkowo w semantyce pojawi się nowy rodzaj zmiennych zwany *sygnałami*, odpowiadający połączeniom w opisywanym przez program układzie cyfrowym. Sygnały będziemy oznaczać metazmiennymi a, b oraz c (być może z indeksami dolnymi).

Wartości, środowiska i efekty uboczne

Zbiór *wartości* (wyników obliczeń) w języku HDML to najmniejszy zbiór V taki, że:

- dla dowolnej liczby całkowitej n zachodzi $n \in V$,
- dla dowolnych sygnałów $a_1, \ldots, a_n$ wektor $[a_1, \ldots, a_n]$ należy do V (dotyczy też wektora pustego $[]$),
- jeśli $v_1 \in V$ oraz $v_2 \in V$, to para (v_1, v_2) należy do V.

Wartości będziemy oznaczać metazmiennymi v,v_1,v_2,...

Środowiska oznaczane metazmiennymi ρ,ρ_1,ρ_2,... to dowolne funkcje częściowe ze zbioru zmiennych języka HDML w zbiór wartości V. Puste środowisko zapisujemy jako $\varnothing$. Dla środowiska ρ,

[1] Istnieją też inne sposoby zadawania formalnej semantyki

zmiennej x oraz wartości v definiujemy operację *rozszerzenia środowiska* (zapisywaną jako $\rho[x \mapsto v]$) w następujący sposób:

$$\rho[x \mapsto v](x_1) = \begin{cases} v, & \text{jeśli } x = x_1 \\ \rho(x_1), & \text{jeśli } x \neq x_1 \end{cases}$$

Efektami ubocznymi w języku HDML są zbiory wygenerowanych elementów układu cyfrowego, czyli zbiory zależności pomiędzy sygnałami opisujące poszczególne bramki logiczne. Ponieważ naszym celem jest weryfikowanie układów, zależności te będziemy opisywać za pomocą klauzul rachunku zdań. Na przykład zależność $a = b \wedge c$ opisującą bramkę AND (a jest wyjściem, b oraz c są wejściami) można zakodować za pomocą trzech klauzul:

$$a \vee \sim b \vee \sim c \qquad \sim a \vee b \qquad \sim a \vee c.$$

Podobnie można zakodować bramki OR (przy użyciu trzech klauzul), XOR (przy użyciu czterech klauzul) oraz NOT (przy użyciu dwóch klauzul).

Dopasowanie wzorca

Zdefiniujmy relację opisującą dopasowanie wartości do wzorca, które ma miejsce podczas wywoływania funkcji i obliczania wyrażenia `let`. Będzie to 4-arna relacja wiążąca ze sobą środowisko przed dopasowaniem, wzorzec i wartość ze środowiskiem po dopasowaniu. Wprowadźmy wygodną notację $\rho \vdash p = v \Downarrow \rho'$, którą nieformalnie rozumiemy jako „w środowiku ρ dopasowanie wartości v do wzorca p generuje środowisko ρ'. Formalnie jest to najmniejsza relacja zamknięta na poniższe reguły:

$$\frac{}{\rho \vdash \_ = v \Downarrow \rho} \qquad \frac{}{\rho \vdash x = v \Downarrow \rho[x \mapsto v]} \qquad \frac{\rho_1 \vdash p_1 = v_1 \Downarrow \rho_2 \qquad \rho_2 \vdash p_2 = v_2 \Downarrow \rho_3}{\rho_1 \vdash p_1, p_2 = (v_1, v_2) \Downarrow \rho_3}$$

Semantyka wyrażeń

Teraz mamy wszystkie składniki potrzebne by zdefiniować relację eval. Podobnie jak relacja opisująca dopasowanie wzorca, relacja eval jest zdefiniowana jako najmniejsza relacja zamknięta na pewien zestaw reguł. Zacznijmy od literałów całkowitoliczbowych i pustego wektora, ponieważ reprezentują one konkretne wartości:

$$\frac{}{P; \rho \vdash n \Downarrow n/\varnothing} \qquad \frac{}{P; \rho \vdash [] \Downarrow []/\varnothing}$$

Wartość zmiennej może być odczytana ze środowiska:

$$\frac{}{P; \rho \vdash x \Downarrow \rho(x)/\varnothing}$$

Pojedynczy bit wymaga wcześniejszego obliczenia wyrażenia, oraz generuje kawałek układu cyfrowego. W dodatku trzeba rozważyć dwa przypadki:

$$\frac{P; \rho \vdash e \Downarrow 0/C}{P; \rho \vdash [e] \Downarrow [a]/C \cup C_0} \qquad \frac{P; \rho \vdash e \Downarrow 1/C}{P; \rho \vdash [e] \Downarrow [a]/C \cup C_1}$$

gdzie a jest świeżym sygnałem, a C_0 oraz C_1 oznaczają zbiory klauzul opisujące zależności odpowiednio $a = \bot$ oraz $a = \top$. Operacje wyboru bitów zdefiniowane są następująco (uwaga na kolejność argumentów):

$$\frac{P; \rho \vdash e_1 \Downarrow [a_0, \ldots, a_{m-1}]/C_1 \qquad P; \rho \vdash e_2 \Downarrow n/C_2}{P; \rho \vdash e_1[e_2] \Downarrow [a_n]/C_1 \cup C_2}$$

$$\frac{P; \rho \vdash e_1 \Downarrow [a_0, \ldots, a_{m-1}]/C_1 \qquad P; \rho \vdash e_2 \Downarrow n_1/C_2 \qquad P; \rho \vdash e_3 \Downarrow n_2/C_3}{P; \rho \vdash e_1[e_2 .. e_3] \Downarrow [a_{n_2}, \ldots, a_{n_1}]/C_1 \cup C_2 \cup C_3}$$

Równie prosto jest zdefiniowany jest operator `@` realizujący konkatenację wektorów sygnałów (uwaga na kolejność argumentów) oraz operator # wyznaczający długość wektora:

$$\frac{P;\rho \vdash e_1 \Downarrow [a_1,\ldots,a_n]/C_1 \qquad P;\rho \vdash e_2 \Downarrow [b_1,\ldots,b_m]/C_2}{P;\rho \vdash e_1 @ e_2 \Downarrow [b_1,\ldots,b_m,a_1\ldots,a_n]/C_1 \cup C_2} \qquad \frac{P;\rho \vdash e \Downarrow [a_1,\ldots,a_n]/C_1}{P;\rho \vdash \#e \Downarrow n/C_1 \cup C_2}$$

Operatory arytmetyczne `+`, `-`, `*`, `/` oraz `%` oznaczające odpowiednio dodawanie, odejmowanie, mnożenie, dzielenie oraz modulo zdefiniowane są następująco ($\oplus$ oznacza dowolny operator arytmetyczny):

$$\frac{P;\rho \vdash e_1 \Downarrow n_1/C_1 \qquad P;\rho \vdash e_2 \Downarrow n_2/C_2}{P;\rho \vdash e_1 \oplus e_2 \Downarrow n/C_1 \cup C_2}$$

gdzie n jest wynikiem odpowiedniej operacji. W analogiczny sposób zdefiniowany jest unarny minus. Operatory porównania zdefiniowane są tylko dla liczb i zwracają zawsze liczbę: 1 jeśli relacja zachodzi, 0 jeśli nie zachodzi. Reguły im odpowiadające przypominają reguły dla operatorów arytmetycznych.

Operatory bitowe `&`, `|`, `^` oraz `~` oznaczające odpowiednio operacje AND, OR, XOR oraz NOT mają efekty uboczne. Na przykład dla reguła operatora `&` wygląda następująco:

$$\frac{P;\rho \vdash e_1 \Downarrow [b_1,\ldots,b_n]/C_1 \qquad P;\rho \vdash e_2 \Downarrow [c_1,\ldots,c_n]/C_2}{P;\rho \vdash e_1 \& e_2 \Downarrow [a_1,\ldots,a_n]/C_1 \cup C_2 \cup C}$$

gdzie $a_1,\ldots,a_n$ są świeżymi sygnałami, a C opisuje zależności $a_1 = b_1 \wedge c_1,\ldots,a_n = b_n \wedge c_n$.

Instrukcja warunkowa wymaga dwóch przypadków: kiedy warunek wylicza się do zera i kiedy wylicza się do liczby różnej od zera. Natomiast wyrażenie `let` korzysta z dopasowania wzorca:

$$\frac{P;\rho \vdash e_1 \Downarrow n/C_1 \qquad P;\rho \vdash e_2 \Downarrow v/C_2}{P;\rho \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v/C_1 \cup C_2}\ n \neq 0$$

$$\frac{P;\rho \vdash e_1 \Downarrow 0/C_1 \qquad P;\rho \vdash e_3 \Downarrow v/C_3}{P;\rho \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v/C_1 \cup C_3}$$

$$\frac{P;\rho \vdash e_1 \Downarrow v_1/C_1 \qquad \rho \vdash p = v_1 \Downarrow \rho' \qquad P;\rho' \vdash e_2 \Downarrow v_2/C_2}{P;\rho \vdash \texttt{let } p = e_1 \texttt{ in } e_2 \Downarrow v_2/C_1 \cup C_2}$$

Ostatnią regułą jest reguła dla wywołania funkcji.

$$\frac{(\texttt{def } f(p) = e') \in P \qquad P;\rho \vdash e \Downarrow v/C_1 \qquad \varnothing \vdash p = v \Downarrow \rho' \qquad P;\rho' \vdash e' \Downarrow v'/C_2}{P;\rho \vdash f(e) \Downarrow v'/C_1 \cup C_2}$$

Zadanie dodatkowe (13 pkt.).
Termin zgłaszania w serwisie SKOS: 24 kwietnia 2017 6:00 AM CEST

Napisz moduł eksportujący predykat `run/5` obliczający funkcję w języku HDML. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

```
% Definiujemy moduł zawierający rozwiązanie.
% Należy zmienić nazwę modułu na {imie}_{nazwisko}_eval gdzie za
% {imie} i {nazwisko} należy podstawić odpowiednio swoje imię
% i nazwisko bez wielkich liter oraz znaków diakrytycznych
:- module(imie_nazwisko_eval, [run/5]).

% Główny predykat rozwiązujący zadanie.
% UWAGA: to nie jest jeszcze rozwiązanie; należy zmienić jego
% definicję.
run(Program, FName, Arg, Value, Clauses) :-
  Program = [def('main', wildcard(no), num(no, 42))],
```

```
FName = 'main',
Arg = [a,b,c],
Value = 42,
Clauses = [].
```

zmieniając definicję predykatu `run(+Program, +FName, +Arg, -Value, -Clauses)`. Oto znaczenia poszczególnych parametrów:

`Program`: Term reprezentujący abstrakcyjne drzewo rozbioru programu w języku HDML.

`FName`: Atom reprezentujący nazwę funkcji do obliczenia.

`Arg`: Wartość będąca argumentem dla obliczanej funkcji. Wartości reprezentujemy jako termy prologowe w następujący sposób:

- liczby reprezezentujemy po prostu jako liczby całkowite;
- wektory sygnałów reprezentujemy jako listy atomów (sygnały reprezentujemy jako atomy);
- parę wartości (v_1, v_2) reprezentujemy jako zwykłą parę w prologu (termy oddzielone przecinkiem).

`Value`: Wartość będąca wynikiem obliczonej funkcji. Przyjmujemy taki sam sposób kodowania wartości jak w przypadku argumentu `Arg`.

`Clauses`: Zbiór klauzul wygenerowanych podczas obliczania funkcji. Klauzule zapisujemy tak, jak w pierwszym zadaniu.

Dla programu P, nazwy funkcji f oraz wartości v, predykat `run`(P, f, v, v', C) powinien znaleźć takie v' oraz C, że dla pewnych p, e oraz ρ zachodzi:

- $(\texttt{def}\ f(p) = e) \in P$,
- $\varnothing \vdash p = v \Downarrow \rho$
- $P; \rho \vdash e \Downarrow v'/C$

Zauważ, że nie wszystkie poprawne wyrażenia mają semantykę, np. `[]+1`. Jeśli Twój program napotka takie wyrażenie, to może, podobnie jak w zadaniu 3, albo zawieść, albo zgłosić wyjątek postaci `runtime_error(Reason, Pos)`, gdzie `Pos` jest pozycją w kodzie źródłowym na której pojawił się problem (możesz ją odczytać z abstrakcyjnego drzewa rozbioru), a `Reason` jest dowolnym termem opisującym błąd.

Wskazówka 1: Niektóre reguły, np. te dla operatorów bitowych wymagają generowania świeżych atomów. Możesz założyć, że w wartości `Arg`, którą otrzymał Twój predykat `run/5` nie ma atomów zaczynających się literami `wire`, więc świeże atomy mogą być bezpiecznie generowane za pomocą standardowego predykatu `gensym(wire, X)`.

Wskazówka 2: Zauważ, że we wszystkich regułach wygenerowany zbiór klauzul w konkluzji zawiera wszystkie wygenerowane klauzule w przesłankach. Zatem podczas wykonania programu, klauzule mogą być tylko generowane, ale nigdy nie mogą być usuwane. Zapisując reguły ewaluacji w Prologu, użyj DCG do generowania list klauzul. Dzięki temu Twój program będzie bardziej elegancki i czytelny.

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *imię_nazwisko*`_eval.pl` gdzie za *imię* i *nazwisko* należy podstawić odpowiednio swoje imię i nazwisko bez wielkich liter oraz znaków diakrytycznych. Nadesłany plik powinien być kodem źródłowym napisanym w Prologu, który definiuje moduł eksportujący tylko predykat `run/5` tak jak to opisano w załączonym szablonie. **Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!**