Konstrukcja kompilatorów

Projekt nr 2

Termin oddawania: 23 listopada 2018

UWAGA! Przed przystąpieniem do zadania proszę zapoznać się ze stronami w systemie SKOS poświęconymi pracowni. Część ze stron została uaktualniona.

Drugim projektem na pracownię z konstrukcji kompilatorów jest mechanizm sprawdzania typów. W systemie SKOS znajdują się nowe źródła kompilatora, gdzie w katalogu «`source/mod_student`» zamieściliśmy szablon rozwiązania w pliku «`typechecker.ml`». W katalogu «`tests/pracownia2`» znajdą Państwo przygotowany przez nas zestaw testów.
Wszelkie niejasności powinni Państwo rozstrzygnąć obserwując jak radzi sobie nasza wtyczka.

Strategia obliczeń Zaimplementowanie mechanizmu sprawdzania typów wymaga dobrania odpowiedniej strategii przechodzenia abstrakcyjnego drzewa rozbioru programu. Może to być strategia oddolna (bottom-up), gdzie najpierw rekonstruujemy typ podwyrażeń, a następnie decydujemy o poprawności i typie właśnie rozpatrywanego węzła. Może to być też strategia odgórna (top-down), gdzie sprawdzamy czy rozpatrywany węzeł drzewa pasuje do oczekiwanego typu, a następnie przechodzimy do poddrzew wyliczając odpowiednie oczekiwania dla nich. Może to być też podejście hybrydowe, gdzie w zależności od kontekstu przełączamy się pomiędzy dwiema wymienionymi wcześniej strategiami. Rozważmy poniższe reguły typowania dla operatorów dodawania i odejmowania.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 - e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau[] \quad \Gamma \vdash e_2 : \tau[]}{\Gamma \vdash e_1 + e_2 : \tau[]}$$

Regułę dla wyrażenia $e_1 - e_2$ można zaimplementować dowolną strategią. Przy strategii oddolnej byśmy wyliczyli najpierw typy wyrażeń e_1 oraz e_2; sprawdzili czy wyliczone typy to `int`; ostatecznie uznali, że całe wyrażenie też ma też typ `int`. Przy strategii odgórnej widząc wyrażenie $e_1 - e_2$ upewnilibyśmy się, że oczekiwanym typem jest `int`, a następnie byśmy kazali sprawdzić czy podwyrażenia mają też taki typ. W języku Xi operator dodawania jest przeładowany. Może oznaczać dodawanie liczb oraz konkatenację tablic. Możemy na nim zademonstrować podejście hybrydowe, które mogłoby działać tak: w trybie oddolnym zrekonstruuj typ wyrażenia e_1 – nazwijmy go σ; w trybie odgórnym sprawdź czy wyrażenie e_2 ma typ σ; upewnij się że σ jest typem `int` lub tablicą; uznaj że całe wyrażenie ma typ σ. Przy dobieraniu odpowiedniej strategii wygodnie jest posłużyć się dodatkową notacją dla reguł typowania zawierającą informację o przyjętej strategii.

- $\Gamma \vdash e \uparrow \tau$ - strategia oddolna, algorytm rekonstruuje typ τ
- $\Gamma \vdash e \downarrow \tau$ - strategia odgórna, algorytm sprawdza czy wyrażenie spełnia dane oczekiwanie τ

Zapisanie reguł typowania z użyciem powyższej notacji jest bardzo wygodne z punktu widzenia implementacji. Zanim przystąpimy do programowania możemy już ustalić strategię obliczeń oraz zlokalizować kłopotliwe miejsca i podjąć pewne pragmatyczne decyzje. Możemy też przewidzieć jakie błędy będzie potrafiła zgłaszać nasza implementacja. Poniższa reguła obrazuje przykład hybrydowego podejścia podanego wcześniej:

$$\frac{\Gamma \vdash e_1 \uparrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \text{typ } \tau \text{ jest } \texttt{int} \text{ lub tablicą}}{\Gamma \vdash e_1 + e_2 \uparrow \tau}$$

Nasze rozwiązanie oraz przygotowany dla Państwa szablon opierają się o podejście hybrydowe. W szablonie rozwiązania znajdą Państwo procedurę «`infer_expression`» realizującą strategię oddolną oraz procedurę

«check_expression» realizującą strategię odgórną. Kod odpowiadający powyższej regule wyglądałby mniej więcej tak[1]:

```
                    (* zrekonstruuj typ e1 *) let tp = infer_expression gamma e1 in
          (* zweryfikuj czy e2 ma taki typ *) check_expression gamma tp e2;
                (* sprawdzenie kształtu tp *) check_if_type_is_int_or_array tp;
(* zrekonstruowany typ całego wyrażenia *) tp
```

Zdecydowaliśmy się na hybrydowe podejście, ponieważ umożliwia wygodne poradzenie sobie z przeładowanym operatorem dodawania oraz radzenie sobie z wyrażeniami tablicowymi. Rolą odgórnego podejścia w naszym rozwiązaniu jest wykrycie specjalnych przypadków, gdzie warto propagować w dół drzewa pewne oczekiwania. Jeżeli nie zachodzi specjalny przypadek, to przełączamy się na strategię oddolną. Zwróćmy uwagę, że poza kilkoma wyjątkami wyrażenia będą występować zwykle w miejscach, gdzie znamy oczekiwany typ:

- argument polecenia `return` - znamy typ jaki ma funkcja zwrócić;
- argument przekazany do funkcji - z typu wywoływanej funkcji znamy typ formalnego parametru;
- warunek w konstrukcji warunkowej lub pętli - oczekujemy typu `bool`
- wyrażenie po prawej stronie przypisania - możemy najpierw wyliczyć typ wyrażenia po lewej stronie przypisania.

Na podstawie powyższej obserwacji wiemy, że będziemy zaczynać sprawdzanie wyrażeń w programie znając oczekiwany typ. Naszą implementację odgórnego sprawdzania opisują poniższe reguły:

$$\frac{\Gamma \vdash e_1 \downarrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \text{typ } \tau \text{ jest } \texttt{int} \text{ lub tablicą}}{\Gamma \vdash e_1 + e_2 \downarrow \tau} \qquad \frac{\Gamma \vdash e_1 \downarrow \tau \ \cdots \ \Gamma \vdash e_n \downarrow \tau \quad n \geq 0}{\Gamma \vdash \{e_1, \cdots, e_n\} \downarrow \tau[]}$$

$$\frac{\Gamma \vdash e_1 \downarrow \tau[] \quad \Gamma \vdash e_2 \downarrow \texttt{int}}{\Gamma \vdash e_1[e_2] \downarrow \tau} \qquad \frac{\Gamma \vdash e \uparrow s \quad s = t}{\Gamma \vdash e \downarrow t}$$

Pierwsza reguła służy do propagowania oczekiwania w dół przy dodawaniu. Dzięki tej regule nie musimy się martwić czy programista napisał `return {} + f()` czy też `return f() + {}`. Czysta oddolna strategia mogłaby mieć kłopot jaki typ przypisać wyrażeniu `{}`. Druga reguła służy propagacji oczekiwania w dół wyrażeń tablicowych. Gdy znamy oczekiwany typ tablicy, to możemy łatwo wyliczyć oczekiwany typ elementu. Trzecia reguła jest analogiczna. Gdy znamy typ elementu, to możemy łatwo wyliczyć typ całej tablicy. Ostatnia reguła oznacza zmianę strategii, gdy nie można użyć żadnej z poprzednich reguł. Wyliczamy oddolnie typ wyrażenia i sprawdzamy czy jest taki jak oczekiwanie.
Strategia oddolna może w wybranych miejscach znów przełączać się na strategię odgórną, jak w regule dodawania podanej na początku dokumentu czy też w regule sprawdzania wywołania.

$$\frac{\Gamma(f) = \texttt{fn}(\tau_1, \cdots, \tau_n) \to \tau_r \quad \Gamma \vdash e_1 \downarrow \tau_1 \ \cdots \ \Gamma \vdash e_n \downarrow \tau_n \quad n \geq 0}{\Gamma \vdash f(e_1, \cdots, e_n) \uparrow \tau_r}$$

W przygotowanym przez nas szablonie znajdą Państwo szkielet powyższych strategii oraz implementację przykladowych reguł. Reguły reprezentujące naszą strategię znajdują się w załączniku do zadania.

Wyrażenie `length` W wyrażeniu `length` potrzebujemy zrekonstruować typ argumentu i upewnić się, że jest tablicą. Niestety nie mamy żadnego oczekiwania tutaj i musimy zacząć sprawdzanie typu od strategii oddolnej. W tym miejscu obecność wyrażeń tablicowych staje się kłopotliwa. Rozważmy takie wyrażenia jak:

- `{}`
- `{ {}[0], {{}} }`

[1] Zwróć uwagę, że kolejność sprawdzania wyrażeń oraz kształtu typów może mieć wpływ na jakość zwracanych błędów. Zastanów się, czy w podanym przez nas kodzie nie byłoby lepiej wpierw sprawdzić czy wyliczony typ ma pożądany kształt.

- `{ {}, {{}}, {{{f()}}}, {{{}}} }`

Sprawdzenie powyższych wyrażeń czystą strategią oddolną nie jest trywialne. Widząc dany węzeł wyrażenia nie potrafimy ani wyliczyć oczekiwanego typu, ani nie potrafimy przewidzieć zależności pomiędzy typami podwyrażeń. Przykładowo, w ostatnim podanym wyrażeniu na typ całości ma wpływ typ funkcji f.
Związku z tą trudnością podjęliśmy decyzję, że inferencja wyrażeń typowych w naszej implementacji jest bardzo prosta. Potrafimy zrekonstruować typ wyrażenia tablicowego, tylko gdy potrafimy zrekonstruować typ pierwszego podanego elementu. W przeciwnym wypadku nasz algorytm się poddaje. Poniższe reguły opisują naszą strategię w tym wypadku:

$$\frac{\Gamma \vdash e \uparrow \tau[]}{\Gamma \vdash \texttt{length}(e) \uparrow \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \uparrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \cdots \quad \Gamma \vdash e_n \downarrow \tau \quad n \geq 1}{\Gamma \vdash \{e_1, \cdots, e_n\} \uparrow \tau[]}$$

Intencjonalnie przyjęliśmy założenie, że nie potrafimy zrekonstruować typu pustego wyrażenia tablicy `{}`. Moglibyśmy założyć, że jest to jakiś dowolnie wybrany typ tablicowy, np `int[]`. Pozwoliłoby to otypować `length({})`, lecz nie zadziałałoby z ostatnim wyrażeniem z powyższych przykładów. Proszę zwrócić uwagę, że w tych miejscach programów gdzie znamy oczekiwany typ, jesteśmy w stanie poradzić sobie z takimi wyrażeniami. Mogą Państwo się zastanowić, czy są jeszcze inne miejsca w języku Xi gdzie brakuje nam oczekiwania.

Błędy Nasza implementacja stara się zwrócić jak najwięcej błędów. Przygotowany przez nas szablon rozwiązania jest przygotowany na zakończenie sprawdzania typów przy pierwszym napotkanym błędzie. Przygotowane przez nas kody błędów znajdują się w module «`Xi_lib.Typechecker_errors`». Państwa implementacja nie musi zgłaszać dokładnie tych błędów jak nasza wtyczka. Ważne tylko aby nieprawidłowy program zostrzał odrzucony oraz aby zgłoszony błąd był sensowny.

Dodatkowe sprawdzenia

- Przykrywanie nazw (funkcji i zmiennych) jest zakazane. W przeciwieństwie do oficjalnej specyfikacji nie można definiować funkcji, co są zadeklarowane.
- Nie można wywoływać funkcji w komendach, tylko procedury (chodzi o «`STMT_Call`»).
- W funkcjach wszystkie ścieżki sterowania muszą kończyć się returnem.

Dla ciekawskich Hybrydowe podejście do formułowania reguł typowania, gdzie zaznaczamy strategię obliczeń, nazywa się dwukierunkowym sprawdzaniem typów (ang. *bidirectional type-checking*). Pojawiło się ono w kontekście bardzo zaawansowanych systemów typów [6], które miały już więcej wspólnego z logiką niż z językami programowania. W artykule [3] znajdą Państwo samouczek poświęcony opisowi tego podejścia do formułowania reguł w ten sposób.
Moglibyśmy zadać sobie pytanie czy znane są techniki sprawdzania typów, dzięki którym moglibyśmy w pełni zaimplementować oficjalny system typów języka Xi? Bardziej zaawansowaną techniką rekonstrukcji typów jest *algorytm W* [4]. Jest on podstawą systemów typów [5] w takich językach jak OCaml czy Haskell, lecz sposób jego działania znajdzie zastosowanie nie tylko w językach funkcyjnych.
W naszej wtyczce mamy eksperymentalną implementację małego fragmentu wspomnianego algorytmu, można ją włączyć ustawiając zmienną środowiskową `XI_ADVANCE_INFER=yes`. Algorytm operuje na schematach typów $\sigma ::= \alpha \mid \sigma[] \mid \tau$, gdzie mogą pojawiać się zmienne typowe. Naszą implementację da się opisać regułami z sądami postaci $\Gamma \vdash e \Uparrow \sigma, C$ oznaczającymi, że oddolnie zrekonstruowaliśmy schemat typu σ oraz wyliczyliśmy zbiór C więzów reprezentujących zależności.

$$\frac{\Gamma \vdash e_1 \Uparrow \sigma_1, C_1 \quad \Gamma \vdash e_2 \Uparrow \sigma_2, C_2}{\Gamma \vdash e_1 + e_2 \Uparrow \sigma_1, C_1 \cup C_2 \cup \{\sigma_1 = \sigma_2\}} \qquad \frac{\Gamma \vdash e_a \Uparrow \sigma_a, C_a \quad \Gamma \vdash e_i \downarrow \texttt{int} \quad \texttt{fresh}\ \alpha}{\Gamma \vdash e_a[e_i] \Uparrow \alpha, C_a \cup \{\sigma_a = \alpha[]\}}$$

$$\frac{\Gamma \vdash e_1 \Uparrow \sigma_1, C_1 \quad \cdots \quad \Gamma \vdash e_n \Uparrow \sigma_n, C_n \quad \texttt{fresh}\ \alpha}{\Gamma \vdash \{e_1, \cdots, e_n\} \Uparrow \alpha[], \bigcup_n^{i=1} C_i \cup \{\alpha = \sigma_i\}} \qquad \frac{\Gamma \vdash e \uparrow \tau}{\Gamma \vdash e \Uparrow \tau, \emptyset}$$

$$\frac{\Gamma \vdash e \Uparrow \sigma, C \quad \texttt{unify}(C \cup \{\sigma = \alpha[]\}) \quad \texttt{fresh}\ \alpha}{\Gamma \vdash \texttt{length}(e) \uparrow \texttt{int}}$$

Dzięki obecności zmiennych typowych algorytm nie musi podejmować wszystkich decyzji przy rozpatrywaniu danego węzła drzewa. Takie podejście pozwala pozbierać wymagania z całego wyrażenia, a następnie je dopiero rozwiązać za pomocą *algorytmu unifikacji* znanego już Państwu z wykładu *Logika dla informatyków*. Powyższe reguły to tylko przykład demonstrujący ideę stojącą za algorytmem W. Projektując algorytm sprawdzania typów bazujący na tej technice należałoby wszystkie reguły sformułować w ten sposób, co pozwoliłoby uwzględnić wymagania płynące ze wszystkich węzłów. Oryginalne sformułowanie algorytmu [4] nie zbierało więzów, lecz przeprowadzało unifikację na bieżąco. Więcej o sformułowaniu opartym o zbieranie więzów mogą Państwo znaleźć w rozdziale 22.3 książki [1] oraz rozdziale 10 książki [2].

Literatura

[1] Pierce B. C. Types and Programming Languages. The MIT Press ©2002

[2] Pierce B. C. Advanced Topics in Types and Programming Languages. The MIT Press ©2004

[3] Christiansen. D. R. Bidirectional Typing Rules: A Tutorial. http://davidchristiansen.dk/tutorials/bidirectional.pdf

[4] Milner. R. A theory of type polymorphism in programming. Journal of Computer and System Sciences 1978.

[5] Hindley–Milner type system. Wikipedia. https://en.wikipedia.org/wiki/Hindley-Milner_type_system

[6] Coquand. T. An algorithm for type-checking dependent types. Science of Computer Programming 1996.

A Nasza strategia

$$\frac{\Gamma(x) = \texttt{var}\ \tau}{\Gamma \vdash x \uparrow \tau} \qquad \frac{}{\Gamma \vdash \mathit{int} \uparrow \texttt{int}} \qquad \frac{}{\Gamma \vdash \mathit{char} \uparrow \texttt{int}} \qquad \frac{}{\Gamma \vdash \mathit{string} \uparrow \texttt{int}[]}$$

$$\frac{}{\Gamma \vdash \mathit{bool} \uparrow \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 \uparrow \tau[] \quad \Gamma \vdash e_2 \downarrow \texttt{int}}{\Gamma \vdash e_1[e_2] \uparrow \tau}$$

$$\frac{\Gamma \vdash e \uparrow \tau[]}{\Gamma \vdash \texttt{length}(e) \uparrow \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \uparrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \cdots \quad \Gamma \vdash e_n \downarrow \tau \quad n \geq 1}{\Gamma \vdash \{e_1, \cdots, e_n\} \uparrow \tau[]}$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \boxdot \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash e_1 \boxdot e_2 \uparrow \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 \downarrow \texttt{int} \quad \Gamma \vdash e_2 \downarrow \texttt{int} \quad \boxdot \in \{\texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1 \boxdot e_2 \uparrow \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 \downarrow \texttt{int} \quad \Gamma \vdash e_2 \downarrow \texttt{int} \quad \lhd \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \lhd e_2 \uparrow \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 \downarrow \texttt{bool} \quad \Gamma \vdash e_2 \downarrow \texttt{bool} \quad \boxdot \in \{\texttt{\&}, \texttt{|}\}}{\Gamma \vdash e_1 \boxdot e_2 \uparrow \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \text{typ } \tau \text{ jest int lub tablicą}}{\Gamma \vdash e_1 + e_2 \uparrow \tau}$$

$$\frac{\Gamma(f) = \texttt{fn}(\tau_1, \cdots, \tau_n) \to \tau_r \quad \Gamma \vdash e_1 \downarrow \tau_1 \ \cdots \ \Gamma \vdash e_n \downarrow \tau_n \quad n \geq 0}{\Gamma \vdash f(e_1, \cdots, e_n) \uparrow \tau_r}$$

$$\frac{\Gamma \vdash e_1 \downarrow \tau \quad \Gamma \vdash e_2 \downarrow \tau \quad \text{typ } \tau \text{ jest } \texttt{int} \text{ lub tablicą}}{\Gamma \vdash e_1 + e_2 \downarrow \tau} \qquad \frac{\Gamma \vdash e_1 \downarrow \tau \ \cdots \ \Gamma \vdash e_n \downarrow \tau \quad n \geq 0}{\Gamma \vdash \{e_1, \cdots, e_n\} \downarrow \tau[]}$$

$$\frac{\Gamma \vdash e_1 \downarrow \tau[] \quad \Gamma \vdash e_2 \downarrow \texttt{int}}{\Gamma \vdash e_1[e_2] \downarrow \tau} \qquad \frac{\Gamma \vdash e \uparrow s \quad s = t}{\Gamma \vdash e \downarrow t}$$