Zadanie domowe 4

Type-checking:

W tym zadaniu Waszym celem będzie zaimplementowanie type-checkera dla języka Fun.

Załóżcie nowy katalog **lab4** i skopiujcie do niego całą zawartość katalogu **lab3**. Następnie ściągnijcie plik

`www.math.us.edu.pl/~pgladki/teaching/2013-2014/tk_lab4.zip`

i rozpakujcie go; w szczególności znajdziecie tam pliki typecheck.sml sources.cm compile.sml – zastąpcie starą wersję sources.cm i compile.cm nowymi wersjami.

Jeżeli Wasze wersje **fun.lex** oraz **fun.grm** działają tak, jak powinny, to możecie ich użyć. W przeciwnym razie poproście prowadzącego zajęcia o (w miarę) działającą wersję.

Waszym głównym celem będzie edytowanie pliku **typecheck.sml** celem zaimplementowania funkcji **tc, sub** oraz **join.** Reguły typowania dla języka Fun opisane są w jego specyfikacji, czyli tam gdzie zawsze: a specyfikację języka Fun tam, gdzie ostatnio:

`www.math.us.edu.pl/~pgladki/teaching/2013-2014/tk_fun.html`

Będziecie musieli zaimplementować te reguły. Główny dowcip polega na tym, że reguły te są deklaratywne. Innymi słowy, nie są deterministyczne i tym samym nie specyfikują bezpośrednio algorytmu do type-checkingu. Będziecie musieli zaimplementować deterministyczny algorytm dla type-checkingu, który jest niesprzeczny i zupełny ze względu na dane niedeterministyczne reguły. Wasz algorytm będzie niesprzeczny, jeżeli za każdym razem, gdy określi zgodność typów, będzie istniało jakieś typowanie w niedeterministycznym, deklaratywnym systemie typów dla programu. Wasz algorytm będzie zupełny, jeżeli za każdym razem, gdy istnieje pewne typowanie, Wasz algorytm będzie potrafił je odnaleźć i zgłosić, że zadany program ma zgodne typy. Więcej o niesprzeczności i zupełności znajdziecie w moich notatkach z logiki dla informatyków.

Okazuje się, że opisany problem jest w zasadzie istotny jedynie w przypadku reguły dla if-then-else. Zacznijmy więc od opisania, jak można sobie poradzić w pozostałych przypadkach.

Pierwszy etap definiowania algorytmu do type-checkingu to zdefiniowanie funkcji subtypującej **sub**. Jeśli wywołacie funkcję **sub(t1, t2),** funkcja powinna zwrócić prawdę, gdy **t1** jest podtypem **t2** oraz fałsz, gdy jest inaczej. Reguły subtypowania w języku Fun nie są deterministyczne i nie mogą być zaimplementowane wprost głównie z powodu reguły przechodniości (transitivity), orzekającej co następuje:

$$\frac{t1 < t2 \qquad t2 < t3}{t1 < t3}(transitivity)$$

Gdybyśmy chcieli odczytać powyższą reguły od dołu do góry tak jak funkcję, to orzekałaby, że **t1** jest podtypem **t3**, gdybyśmy potrafili jakoś magicznie odgadnąć typ **t2** taki, że **t1** jest podtypem **t2** oraz **t2** jest podtypem **t3**. Na szczęście możemy trochę przerobić pozostałe reguły subtypowania tak, aby mieć pewność, iż nasz podtyp w relacji jest, jako taki, przechodni – wówczas nie będziemy musieli implemetować reguły przechodniości wprost. W szczególności oznacza to, że będzie konieczne połączenie reguł dla n-ek i sprawdzenie wszystkich warunków na subtypowanie dla n-ek za jednym razem.

Kolejna część naszego algorytmu będzie polegała na zaimplementowaniu algorytmu type-checkingu dla samych wyrażeń. Ten algorytm będzie wywoływał **sub** w razie konieczności. Zauważmy, że wśród reguł deklaratywnych jest jedna, która sprawia wrażenie problematycznej:

$$\frac{G \vdash e : t \qquad t < t'}{G \vdash e : t'}(transitivity)$$

Zaimplementowanie powyższej reguły bezpośrednio w funkcji **tc_exp** będzie problematyczne, albowiem gdybyśmy chcieli odczytać ją jak funkcję od dołu do góry, gdzie argumentami byłby kontekst G, wyrażenie e i typ t', to znów musielibyśmy jakoś magicznie odgadnąć z jakim typem t możemy rekursywnie wywołać **tc_exp.** Więcej, ta reguła zawsze może być użyta; kiepski type-checker może chcieć próbować sprawdzić powyższą regułę raz za razem i tym samym wpaść w nieskończoną pętlę.

Poradzimy sobie z tym problemem wywołując **tc_exp ctxt e** i spodziewając się, że wywołanie albo zakończy się fiaskiem (ze zgłoszeniem ErrorMsg.error), albo sukcesem i zwróci przynajmniej jeden typ **t**, który **e** może miec w kontekście **ctxt.**

Gdy type-checker zgłosi błąd, nie oznacza to, że koniecznie musi się zatrzymać. Powinien próbować wyjść z błędu tak, aby być w stanie zgłosić też inne błędy. Jeden ze sposobów, aby sobie z tym poradzić, to założyć, że wszystkie wyrażenia z błędami mają typ **int** i mieć nadzieję, że się uda.

Jedyne miejsca, gdzie będziemy używać algorytmu subtypowania to te, gdzie nie będziemy mieli wyboru, bo inaczej type-checking zakończy się fiaskiem. Jednym z takich miejsc może być, na przykład, sytuacja, w której wywołujemy funkcję **f(e)**. Jeżeli **f** jest typu **t1 -> t2** oraz **e** jest typu **t1'** to nasz type-checker powinien zadziałać, jeśli **t1'** jest subtypem **t1**. Wtedy typ **t(e)** może być dowolnym podtypem **t2;** aby zgłosić przynajmniej jeden supertyp **t2**, po prostu możemy zwrócić **t2** jako rezultat **tc_exp("f(e)")**.

Type-checker powinien przechodzić przez wyrażenia i jednocześnie zachowywać kontekst. Przykładem innego programu, który przechodzi wyrażenia i zachowuje kontekst jest podany Wam w zadaniu program **eval.sml.** Oczywiście program ten robi zupełnie inne rzeczy z kontekstami, tak więc nie możecie zbyt bezpośrednio go naśladować, ale powinien służyć Wam pomocą w uczeniu się.

Najbardziej skomplikowany problem to type-checking ald if-then-else. Rozważmy następujące wyrażenie języka Fun, w kontekście, gdzie **f** ma typ **int -¿ int:**
if x then <0,f> else <1,2,3>
Dla tego typu wyrażenia spodziewamy się, że zwróci rekord, którego pierwszym polem będzie **int.** Innymi słowy, najmniejszym subtyp dla wszystkich wartości jakie mogą być zwrócone przez takie wyrażenie if-then-else jest **<int>.** Poniżej podamy "high-endowe" rozwiązanie problemu takiej implementacji, a póki co, zanim nie doprowadzicie wszystkiego innego do poprawnego działania, posłużcie się następującym przybliżeniem: przypadki then- oraz else- muszą mieć dokładnie ten sam typ, a więc:
join complain (t1,t2) = if t1=t2 then t1 else (complain "t1 and t2 do not join"; t1)
To nie jest zgodne z semantyką języka Fun, a więc może odrzucić pewne poprawnie napisane programy w Fun! Z drugiej strony, nie zaakceptuje programów napisanych niepoprawnie – innymi słowy, taki algorytm będzie niesprzeczny, ale nie zupełny.

Skompilujcie kod z użyciem **CM.make "sources.cm";** przetestujcie Wasz kod wywołując **Compile.compile** *"nazwa".*

Przed wysłaniem zadania, sprawdźcie:

- czy upewniliście się, że żadne dwie funkcje nie używają tej samej nazwy? (użyjcie wydajnej Symbol.table, nie używajcie algorytmu kwadratowego);
- czy upewniliście się, że main jest int-¿int?
- czy upewniliście się, że pozwoliliście używać takiej samej nazwy parametru i funkcji globalnej?
- czy pozwoliliście, żeby przechodziły głupotki takie, jak (fun f(x:int):<int>=let x=3 in let x=<x> in x)? Chodzi o to, aby wiązanie "let" miało możność ukrywania wiązań zewnętrznych o tej samej nazwie w otaczającym kontekście;

- czy pozwoliliście, żeby przechodziły jeszcze większe głupotki, na przykład (fun g(int:<int>): <int> = int)? Chodzi o to, żeby przestrzeń nazw dla typów (która w podstawowym Fun zawiera tylko "int") była ortogonalna do przestrzeni nazw dla wartości;
- czy zabił ktoś tokarza? czy często się to zdarza?

Wasz type-checker nie powinien poddawać się po pierwszym zgłoszonym błędzie, powinien być w stanie wychodzić z błędu i znajdować kolejne. Najprościej będzie Wam to osiągnąć przez ErrorMsg.error.

Jak już wszystko będzie działało, zaimplementujcie type-checking dla if-then-else opisane poniżej.

Prześlijcie mailem pliki README oraz typecheck.sml. Termin zadania mija w **czwartek, 9 stycznia.** Proszę pamiętać o oznaczeniu maila tagiem `[aghtk]`.

1. Najmniejszy wspólny supertyp

Aby właściwie zaimplementować type-checking dla if-then-else, musicie najpierw zaimplementować algorytm, który oblicza najmniejszy wspólny supertyp (join) dla dwóch typów.

Jeśli wywołacie **join complain (t1,t2)** powinniście otrzymać trzeci typ **t3** taki, że **t1** jest podtypem typu **t3** oraz **t2** jest podtypem typu **t3**. Jeżeli taki typ nie istnieje, należy użyć **complain** do zgłoszenia błędu. Ponadto **join** powinien zwracać "najmniejszy" taki typ **t3**, na przykład

tc_join (<int, (int->int)>) (<int>) = <int>

Nawet jeśli <> jest supertypem dla zarówno <int,(int->int)> jak i <int>, jest również supertypem dla <int>. Tym samym funkcja **join** powinna zwrócić <int> zamiast <>.