Jak (prawie) bezboleśnie napisać kompilator języka funkcyjnego?

Opracowanie seminarium „Implementacja języków funkcyjnych”

Łukasz Czapliński

9 czerwca 2016

Streszczenie

W pracy „Compiling with Continuations” Andrew W. Appel opisał konstrukcję kompilatora SML/NJ. W trakcie seminarium „Implementacja języków funkcyjnych” napisaliśmy prosty kompilator języka z rodziny ML. W tej pracy opisuję, które idee okazały się szczególnie przydatne i na jakie problemy natrafiliśmy.

1 Wstęp

Wielu programistów uważa, że pisanie kompilatorów jest ciężkie. Do tego stopnia, że podstawowy podręcznik nosi potocznie nazwę „Książki o smokach” (Dragonbook, [1]). Skoro tak ciężkie jest napisanie prostego kompilatora, to jak ciężkie może być napisanie kompilatora języka wysokiego poziomu, z rodziny ML? W ramach seminarium “Implementacja języków funkcyjnych” w Instytucie Informatyki Uniwersytetu Wrocławskiego postanowiliśmy taki kompilator napisać.

Za podręcznik przyjęliśmy „Compiling with continuations” [2], opisujący implementację języka SML. Poniższa praca jest podsumowaniem tego, co udało nam się osiągnąć. Kolejność rozdziałów odpowiada kolejności wykonywanych przez nas prac. Nacisk położony jest na kroki kluczowe dla działania kompilatora.

2 Architektura kompilatora (Języki pośrednie)

Celem kompilatora jest transformacja tekstu napisanego przez użytkownika na kod maszynowy, działający na danej architekturze i wykonujący program zadany przez użytkownika. Większość kompilatorów wyróżnia przynajmniej dwa stadia: parsowanie tekstu do drzewa składniowego (*Abstract Syntax Tree* – AST) – *analiza*, a następnie generowanie kodu maszynowego z AST – *synteza* [1]. Im prostszy język w którym kompilator jest napisany, tym trudniej dodać więcej stadiów. Ponieważ nasz kompilator napisany został w Ocamlu (języku funkcyjnym wysokiego poziomu [3]), kompilacja miała kilka stadiów; składała się z parsowania tekstu do AST, a następnie z ciągu transformacji, które upraszczały, optymalizowały oraz tłumaczyły AST na kolejne języki pośrednie. W finalnej wersji wyróżnić można następujące języki pośrednie:

- RawMiniML – pierwszy język, powstały ze sparsowania tekstu użytkownika. Można powiedzieć, że ten właśnie język jest przez nas kompilowany.
- MiniML – powstały z poprzedniego przez wyprowadzenie typów (zaimplementowany został system inferencji typów) i usunięciu sugestii typów danych przez użytkownika na rzecz informacji o typie każdego węzła AST.
- Lambda – powstały przez usunięcie typów i wszystkich zależnych od nich konstrukcji z MiniML.
- CPS – Continuation-Passing Style – język w którym będzie wykonana większość optymalizacji.
- AbstractM – asembler procesora na jaki będziemy kompilować.

3 Opis kompilowanego języka

MiniML jest uproszczonym językiem z rodziny ML, silnie wzorowanym na Ocamlu [3] oraz SML-u [2, Chapter 1. What is ML?]. Podobnie jak one, jest statycznie typowany i posiada system inferencji typów. Składnia jest podobna do innych języków z tej rodziny, z kilkoma wyjątkami:

```
datatype 'a list = nil | cons of {'a, 'a list}
in

let rec rev_append ['a] (xs ys : 'a list) : 'a list =
  match xs with
  | nil    => ys
  | cons p => rev_append['a] (#1 p) (cons['a] {#0 p, ys})
  end
in fn ['a] (xs : 'a list) => rev_append['a] xs (nil['a])
```

- Deklaracje typów są widoczne tylko w określonym bloku kodu.

- Nie ma ograniczenia w nazewnictwie typów danych i konstruktorów (inaczej niż w Ocamlu, podobnie jak w SML-u).
- Pattern matching nie wprowadza nowych zmiennych – do pól konstruktora trzeba się odwołać przez dekonstruktor: `\#1 p` oznacza drugi element `p` (gdzie `p` jest stworzone przez konstruktor o arności przynajmniej 2).
- Polimorficzne funkcje muszą deklarować parametry typowe przez `['a]`.
- Nie posiada systemu modułów.

4 Optymalizacje typowe dla języka ML

Pierwszym krokiem kompilacji na jaki się zdecydowaliśmy to transformacja do prostego języka opartego na rachunku lambda. Transformację tę można rozważać jako funkcję F, która jako argument przyjmuje Γ- informację o typach wszystkich wyrażeń w programie, oraz A_{ML}- wyrażenie w języku MiniML, a zwraca A_L- wyrażenie w języku Lambda.

Nasz język pośredni Lambda pozbawiony jest typów, tak więc na tym etapie wszystkie konstrukcje, które w jakiś sposób potrzebują systemu typów muszą zostać zamienione na prostsze, które nie potrzebują tej dodatkowej informacji. W językach rodziny ML zwykle do nich należą [2, Chapter 4: ML-specific optimizations]:

- nazwane pola rekordów;
- moduły, sygnatury, funktory;
- typy danych, konstruktory;
- pattern-matching;
- wyjątki.

Dodatkowo uwagi wymagają:

- testowanie równości (to, jaki kod będzie potrzebny do porównania równości dwóch obiektów ściśle zależy od tego co wiemy o ich typie) [2, Chapter 4.3: Equality].,
- optymalizacja dostępu do pamięci (niektóre typy mogą pozwolić na lepszą alokację pewnych komórek pamięci) [2, Chapter 4.4: Unboxed updates].

W naszym języku zaimplementowane zostały jedynie typy danych, pattern-matching oraz wyjątki.

Na tym etapie można przyjąć jedno z kilku ograniczeń tego, co jesteśmy w stanie zrobić w trakcie działania programu [2, Chapter 4.1: Data representation]:

0. Nie potrafimy odróżnić wskaźników od liczb całkowitych.
1. Potrafimy odróżnić małe liczby całkowite od wskaźników.

2. Potrafimy odróżnić wszystkie wskaźniki od liczb całkowitych.

3. Potrafimy odróżnić wskaźniki do rekordów o różnej długości.

Od tego, jakie założenia przyjmiemy zależy ile dodatkowych informacji będziemy musieli przechować w pamięci (np dodając dodatkowy znacznik dla każdej wartości, mówiącej czy jest ona liczbą całkowitą, czy wskaźnikiem – pozwoli to pracować przy założeniu 2, gdy w rzeczywistości maszyna operuje na 0).

W naszym kompilatorze (na tym etapie) przyjęliśmy założenie 1 – jest ono realne, gdyż w rzeczywistości wszystkie „niskie" wskaźniki zwykle są specjalnie pozostawiane wolne, by łatwiej znajdować błędy w programach. Przy takim założeniu typy danych można podzielić na kilka kategorii, zależnie od tego, jakie informacje musimy o jego konstruktorach przechować:

1. „Tagged" – wymagające przechowania numeru konstruktora i wskaźnika na wartość zapisaną w konstruktorze (najogólniejszy),

2. „Constant" – potrzeba tylko numeru konstruktora, żaden nie posiada dodatkowej wartości,

3. „Transparent" – typ danych ma tylko jeden konstruktor, nie potrzeba żadnej informacji zapisywać,

4. „TransB" – typ danych posiada jeden konstruktor z wartością, wszystkie poza nim nie przechowują wartości. Dzięki założeniu 1 możemy użyć jednej komórki pamięci.

Pozbycie się konstruktorów wymaga więc sprawdzenia, do jakiej kategorii należy jego typ danych, a następnie czasem wygenerowania kodu tworzącego rekord (jedno, lub dwuelementowy) o odpowiednich wartościach (stała będąca numerem konstruktora i/lub wartość przechowywana w konstruktorze).

Wprowadzenie nowego typu danych nie generuje żadnego kodu.

Dekonstrukcja instancji typu danych (*pattern matching*) wymaga wygenerowania kodu zależnie od kategorii tego typu danych – żeby dowiedzieć się, z jakim konstruktorem mamy do czynienia może być potrzebne wykonanie kilku różnych czynności:

- odczytanie jednego z jego pól i porównanie z numerami konstruktorów tego typu danych,

- sprawdzenie czy jest to wskaźnik,

- czasem też nie trzeba nic robić.

Sytuacja z wyjątkami jest jeszcze gorsza – wyjątki różnią się od zwykłych typów danych tym, że na etapie kompilacji danej jednostki nie jest wiadome ile będzie typów konstruktorów. Dzieje się tak, ponieważ moduły są kompilowane oddzielnie, i każdy może zdefiniować swoje wyjątki. Z tego powodu wyjątki są reprezentowane jako rekordy dwuelementowe, gdzie pierwszy element to (wskaźnik na) wartość przechowywaną

w konstruktorze, a drugi to wskaźnik na ciąg znaków będący nazwą wyjątku. Rzucenie wyjątku to wówczas zapisanie takiej wartości w odpowiednie miejsce, a złapanie to porównanie wskaźnika na nazwę ze znanymi wartościami. Jest to wydajne, ponieważ musimy jedynie porównać wskaźniki (wiemy że wszystkie instancje wyjątku danego typu będą miały ten sam wskaźnik na nazwę, bo jest on generowany przez kod w miejscu deklaracji wyjątku).

5 Continuation-Passing Style (CPS)

CPS to styl zapisu programu, w którym kontynuacja (sterowanie, kod który ma użyć zwracanej wartości funkcji) jest przekazywana jawnie jako parametr każdej funkcji – zwykle ostatni.

Zakładając, że `*&`, `+&`, `sqrt&` to odpowiedniki funkcji `*`, `+`, `sqrt` w CPS, używane w notacji prefiksowej, to program po transformacji do CPS-u wygląda następująco:

Typowy Program

```
let pyth x y =
 sqrt(x*x + y*y)
```

Ten sam program w CPS

```
let pyth& x y k =
 (*& x x (fn (x2) =>
          (*& y y (fn (y2) =>
                    (+& x2 y2 (fn (x2py2) =>
                                (sqrt& x2py2 k))))))
```

Jak widać, zapisanie programu w CPS wymaga nazwania każdej wartości pośredniej. Dzięki temu analiza ile wartości jest żywych (ile rejestrów jest potrzebnych do przechowywania wartości w danym momencie obliczeń) jest trywialna.

Dodatkowymi zaletami CPS-u są:

- kolejność wykonywania funkcji jest jasno określona i nie zależy od języka zapisu programu – metajęzyka (może on być ewaluowany zarówno strict, jak i lazy – te same obliczenia zostaną wykonane w tej samej kolejności).
- łatwa reprezentacja domknięcia funkcji – w innych językach pośrednich często nie pozwala się na zagnieżdżanie funkcji, co wymusza pozbycie się ich z języka we wcześniejszych stadiach – co może utrudnić wiele optymalizacji [2, Chapter 1.2: Advantages of CPS].

Wszystko to sprawia, że CPS jest zapisem doskonałym do optymalizacji (porównywalnym, o ile nie lepszym, niż SSA wykorzystywane przez LLVM i Clang).

6 Transformacja do CPS-u

Przekształcenie języka Lambda do CPS-u można traktować jako funkcję F, przyjmującą jako argumenty A_L – wyrażenie w języku Lambda oraz c – kontynuację, a zwracającą A_C

– wyrażenie w CPS. Transformację rozpoczyna wywołanie F z wyrażeniem reprezentującym pełen program oraz kontynuacją która zakończy działanie programu gdy zostanie wywołana.

Całość transformacji została opisana szczegółowo w [2, Conversion into CPS]. Na szczególną uwagę zasługują pewne szczególne przypadki.

6.1 Operatory logiczne

Operatory logiczne w językach z rodziny ML zwracają wartości: *prawdę* lub *fałsz* (*true* / *false*). W CPS wybierają jedną z dwóch kontynuacji. Aby temu zaradzić wywołujemy tę samą kontynuację w każdym z dwóch przypadków, ale przekazujemy różne wartości:

```
F(App (Primop i, E), c) =
   F(E, λ v . PRIMOP(i, [v], [], [c(INT 0), c(INT 1)]))
```

Niestety kopiując kontynuację, zwiększamy rozmiar kodu wynikowego – czasem nawet wykładniczo! Możemy tego uniknąć używając pomocniczej funkcji, która przechowa kontynuację:

```
F(App (Primop i, E), c) =
   F(E, λ v . FIX([(k, [x], c(VAR x))],
     PRIMOP(i, [v], [], [APP(VAR k, INT 0), APP(VAR k, INT 1)]))
```

6.2 Operowanie kontynuacjami w języku Lambda

Operatory *callcc* i *throw* (pozwalające używać mechanizmu kontynuacji bezpośrednio w języku Lambda) wydają się idealnie pasować do CPS-u:

```
F(App (Primop throw, E), c) =
   F(E, λ v . FIX([(f, [x, j], APP(v, [VAR x]))], c(VAR f))
```

`throw f` obliczy się do funkcji, która zignoruje swoją kontynuację i zamiast niej użyje `f`.

```
F(App (Primop callcc, E), c) =
  FIX([(k, [x], c(VAR x))], F(E, λ v . APP(v, [VAR k, VAR k])))
```

`callcc f` zapisuje aktualną kontynuację `c` jako funkcję `k : λv . c(v)` i używa funkcji `k` zarówno jako argumentu, jak i kontynuacji `f`. Dzięki temu niezależnie od tego, czy `f` zakończy działanie normalnie, czy wywoła swój argument, sterowanie zostanie przekazane zadanej kontynuacji `c`.

6.3 Wyjątki

Za wyjątki w języku Lambda odpowiadają instrukcje *Handle* i *Raise*. W CPS w tym celu używane są operatory *gethdlr* i *sethdlr*, odpowiadające za trzymanie wskaźnika do funkcji, która aktualnie obsługuje wyjątki:

Raise po prostu wywołuje aktualną funkcję obsługującą wyjątki.

```
F(Raise E, c) = F(E, λ v . PRIMOP(gethdlr, [], [h], [APP(VAR h, [v])]))
```

Handle natomiast musi ją odpowiednio ustawić, i na koniec przywrócić:

```
F(Handle(A, B), c) =
  PRIMOP(gethdlr, [], [h],
    [FIX([(k, [x], c(VAR x)),
            (n, [e], PRIMOP(sethdlr, [VAR h], [],
                          [F(B, λ f . APP(f, [VAR e, VAR k]))]))],
      PRIMOP(sethdlr, [VAR n], [],
        [F(A, λ v . PRIMOP(sethdlr, [VAR h], [] [APP(VAR k, [v])]))])])])
```

Jednakże wzajemne zachowanie *callcc* i *Handle* prowadzi do problemu: w aktualnym rozwiązaniu kontynuacja jest wywoływana z funkcją obsługującą wyjątki wiązaną dynamicznie (taką, jak zostawił wywołujący), natomiast powinna być ona wiązana statycznie (taka, jaką miała funkcja tworząca kontynuację). Z tego powodu *callcc* musi odczytać i przywrócić odpowiednią funkcję obsługującą wyjątki:

```
F(App (Primop callcc, E), c) =
  PRIMOP(gethdlr, [], [h],
    [FIX([(k, [x], c(VAR x)),
          (k', [x'],
             PRIMOP(sethdlr, [VAR h], [], [APP(VAR k, [VAR x'])]))],
      F(E, λ v . APP(v, [VAR 'k, VAR k]))])
```

6.4 Case

Dodatkowo, instrukcja *Case* (ma ona wybrać odpowiedni przypadek ze skończonej listy) może być przekształcona na wiele sposobów – [2, Chapter 5.7: Case statements] sugeruje kilka możliwości optymalizacji: binarne przeszukiwania, drzewa decyzyjne, liniowe porównania, tablicę skoków. W naszej implementacji wykorzystaliśmy najprostszą do napisania: liniowe przeszukiwanie. W większości przypadków lista możliwości będzie bardzo krótka – praktycznie jedynie generatory kodu tworzą więcej niż kilka przypadków.

7 Optymalizacje kodu w CPS-ie

Dotychczas celem naszej kompilacji było uzyskanie kodu w CPS-ie. Stało się tak, ponieważ uznaliśmy CPS za dobry język do optymalizacji. Zaimplementowaliśmy następujące:

- „proste”
  - zwijanie stałych,
  - η-redukcja,
  - uncurrying,
- „zaawansowane”
  - β-ekspansja,

– hoisting (zmiana kolejności wyrażeń),
– eliminacja wspólnych podwyrażeń.

Najwięcej problemów pojawiło się z zaawansowanymi optymalizacjami.

7.1 β-ekspansja

β-ekspansja (wielokrotna β-redukcja) to zamiana wywołania funkcji na ciało funkcji z podstawionymi argumentami. Jest to przydatna optymalizacja (może pozwolić na działanie wielu innymi), ale niebezpieczna (powiększa rozmiar programu, bez gwarancji na poprawę). Niestety nie da się z pewnością stwierdzić, kiedy ta optymalizacja coś da [2, Chapter 7.1: When to do in-line expansion]. Dlatego stosuje się heurystyki, określające czy warto. Starają się one ocenić proste kryterium: czy po zastosowaniu β-ekspansji i kolejnych optymalizacjach rozmiar kodu się zmniejszy. Niestety ponownie nie jest to problem, który da się policzyć z całą pewnością inaczej niż wykonując te działania. Nasza ocena (wzorowana na [2, Chapter 7.2: Estimating the savings], [2, Chapter 7.3: Runaway Expansion]) opiera się na przypisaniu każdej funkcji pewnej oceny jej rozmiaru i porównywaniu szacowanych zmian w rozmiarze programu z pewną stałą, pomniejszoną o głębokość (aby uniknąć rozwijania ciała już rozwiniętej funkcji) oraz numer rundy (aby przypadkiem nie pozostać w cyklu, np w funkcjach rekurencyjnych). Używając takich heurystyk wykonujemy na zmianę β-ekspansję oraz „proste" optymalizacje aż osiągniemy punkt stały.

7.2 Hoisting

Hoisting polega na zmianie kolejności wyrażeń. W szczególności może to pozwolić na połączenie definicji funkcji, lub wyjęcie pewnego wyrażenia z pętli, lub niewykonanie potencjalnie drogiej operacji jeśli jej wynik jest potrzebny tylko w jednej gałęzi instrukcji warunkowej. Definicja, w jakim przypadku optymalizacja ta może być wykonana nie jest ciężka, ale żmudna [2, Chapter 8.2: Rules for hoisting]. Wybór czy należy ją wykonać, podobnie jak w wypadku β-ekspansji, nie jest trywialny. Appel [2, Chapter 8.3: Hoisting optimizations] sugeruje zachłanne podejście: zamiana kolejności tylko jeśli mamy gwarancję zmniejszenia czasu wykonania programu. To oznacza, m.in. że nigdy nie wyjmiemy wyrażenia z pętli (gdyż nie mamy gwarancji że wykona się ona przynajmniej raz). Definicje funkcji będziemy najpierw spychali w dół, a jeśli to się nie uda, to w górę. W ten sposób połączenie dwóch funkcji zawsze zmniejsza czas wykonania.

7.3 Eliminacja wspólnych podwyrażeń

Eliminację wspólnych podwyrażeń (*Common Subexpression Elimination* – CSE) najprościej jest pokazać na przykładzie [2, Chapter 9: Common subexpressions]:

```
val z = a*b*c+a*b*c
```

W CPS będzie on wyglądać następująco:

```
PRIMOP(*,[VAR a, VAR b], [u], [
  PRIMOP(*, [VAR u, VAR c], [v], [
    PRIMOP(*, [VAR a, VAR b], [w], [
      PRIMOP(*, [VAR w, VAR c], [x], [
        PRIMOP(+, [VAR v, VAR x], [z], [ ...
```

Wartość `a*b` jest liczona podwójnie. Możemy więc zastąpić `w` przez `u`:

```
PRIMOP(*,[VAR a, VAR b], [u], [
  PRIMOP(*, [VAR u, VAR c], [v], [
      PRIMOP(*, [VAR u, VAR c], [x], [
        PRIMOP(+, [VAR v, VAR x], [z], [ ...
```

To z kolei pozwala zauważyć, że wartość `u*c` jest liczona podwójnie. Zastępujemy więc `x` przez `v`:

```
PRIMOP(*,[VAR a, VAR b], [u], [
  PRIMOP(*, [VAR u, VAR c], [v], [
        PRIMOP(+, [VAR v, VAR v], [z], [ ...
```

To z kolei odpowiada wyrażeniu

```
val z = let val v = a*b*c in v+v
```

8 Droga do kodu maszynowego

Po otrzymaniu zoptymalizowanego kodu w CPS-ie pozostaje tylko przepisać go na wykonywalny na danej maszynie. Problemy na jakie napotkaliśmy to:

- zagnieżdżone definicje funkcji – wywołanie funkcji w architekturze von Neumanna jest proste, problem pojawia się gdy funkcja posiada zmienne wolne,
- ilość żywych (potrzebnych w danym momencie obliczeń) wartości może przewyższyć ilość rejestrów maszyny na którą generujemy kod,
- brakuje nam automatycznego zarządzania pamięcią.

8.1 Domknięcia

Za rozwiązanie pierwszego problemu odpowiada dodanie domknięć (*closure conversion*) [2, Chapter 10: Closure conversion]. Polega ono na prostej zmianie: każda funkcja (zarówno przekazywana jawnie, jak i kontynuacja) przestaje być „tylko” prostym wskaźnikiem na funkcję, a staje się strukturą danych o znanej strukturze, która przechowuje zarówno adres funkcji, jak i jej zmienne wolne. Struktura ta jest bardzo prosta – jest to rekord, którego pierwszym elementem jest wskaźnik na funkcję, a kolejnymi jego elementami są zmienne wolne.

Wywołanie funkcji wiąże się ze stworzeniem takiej struktury. Z tego powodu przydatne jest znalezienie takich funkcji które nie potrzebują tej dodatkowej struktury. Nazywamy

takie funkcje *znanymi* – są to funkcje, które nie są nigdy przekazywane jako argument. Dzięki temu znamy wszystkie miejsca gdzie dana funkcja jest wywoływana i możemy ją przekształcić – przekazując wszystkie jej zmienne wolne jako argumenty.

8.2 Rozlewanie rejestrów

Aby upewnić się, że ilość potrzebnych żywych zmiennych w danym momencie nie przekracza ilości rejestrów (a zatem kod daje się wykonać) wykonujemy proces zwany rozlewaniem rejestrów (*register spilling*) [2, Register spilling]. Polega on na wykryciu takich sytuacji w czasie kompilacji i rozwiązaniu problemu przez alokację dodatkowych rekordów na stosie. Dzięki temu zamiast potrzebować n rejestrów na zmienne, możemy użyć jednego rejestru wskazującego na rekord n-elementowy. Oczywiście ceną za takie posunięcie jest konieczność tworzenia i ładowania wartości do oraz z takiego rekordu w trakcie wykonania programu.

8.3 Odśmiecanie pamięci

Do korzystania z automatycznego odśmiecania pamięci (*garbage collection*) potrzebujemy ustalić:

- kiedy uruchomić odśmiecanie pamięci, oraz
- które wartości są osiągalne (a zatem które możemy usunąć).

Okazuje się, że odpowiedź na pierwsze pytanie może być prosta: przed wywołaniem każdej funkcji sprawdzamy, ile wolnej pamięci pozostało, i jeśli jest jej za mało uruchamiamy procedurę odśmiecania pamięci.

Pozostaje tylko ustalić, skąd odśmiecanie ma wiedzieć które wartości są osiągalne. W tym celu trzeba przyjąć albo bardzo silne założenia o systemie zarządzania pamięcią, albo dodać do obiektów pewne metadane o ich zawartości. Podobnie jak Appel [2, Chapter 16.4: Runtime data formats] założyliśmy:

- przed każdym punktem wejścia do funkcji znajduje się wskaźnik na jej początek,
- przed początkiem funkcji znajduje się informacja o używanych przez nią rejestrach,
- przed każdym rekordem znajduje się blok metadanych (określający, jego długość oraz czy może zawierać wskaźniki – jeśli tak, to każdy jego element jest albo wskaźnikiem,albo liczbą całkowitą ze znacznikiem (*tagged integer*), w przeciwnym wypadku może zawierać dowolne dane nie będące wskaźnikami).

Zauważmy, że jest to wszystko, co potrzebujemy – ponieważ nie mamy stosu. Razem pozwala to zaimplementować szybkie odśmiecanie pamięci przez kopiowanie generacjami (*generational garbage collection*).

Opiera się ono na dwóch obserwacjach dotyczących języków funkcyjnych:

1. nowe obiekty są częściej usuwane od starych,

2. nowe obiekty trzymają referencje na starsze.

Wynika to z trwałych struktur danych używanych często w programowaniu funkcyjnym. W przypadku zagnieżdżania takich struktur konstruktor przyjmuje jako argument starszy obiekt (a więc musiał on powstać przed tym wywołaniem konstruktora) oraz trzyma do niego referencję (a więc musi zostać usunięty przed nim). Jedynym wyjątkiem są modyfikowalne komórki pamięci (o tym za chwilę).

Generacjami będziemy nazywać obszary pamięci, w których będziemy alokować obiekty. Kiedy *n*-ty obszar się zapełni, przechodzimy do nowego – *n+1* – będziemy nazywać go *generacją n+1*. Numery generacji nigdy się nie zmniejszają. Aby odzyskać pamięć, użyjemy informacji o aktualnie wykonywanej funkcji, by znaleźć wszystkie żywe obiekty w najnowszych generacjach (zwykle wystarczą 2-3), a następnie skopiujemy je w nowy obszar i zwolnimy pamięć ze starego. Dzięki obserwacji 1. wiemy, że usunęliśmy śmieci z obszarów w których było ich najwięcej, a dzięki obserwacji 2. wiemy, że nie zepsuliśmy referencji ze starych generacji. Z wyjątkiem modyfikowalnych komórek pamięci. Problem z nimi możemy rozwiązać na kilka sposobów [2, Chapter 16.3: Generational garbage collection]: poprzez dwupoziomowe wskaźniki (*assignment table*) [4], trzymanie zbioru komórek które mają wskaźniki do nowszych komórek od siebie [5], lub użycie pamięci wirtualnej [6].

9 Ograniczenia

Z różnych przyczyn stworzony przez nas kompilator nie osiągnął poziomu generowania faktycznego kodu maszynowego jednej z aktualnych architektur procesora. Podczas jego pisania rozważane było użycie języka pośredniego LLVM jako ostatniej fazy (z niego kod maszynowy na różne architektury potrafi generować szereg narzędzi składających się na kompilator Clang). Niestety okazało się, że ten język pośredni nie pasuje do założeń przyjętych w książce Appela – pod pewnymi względami jest o wiele silniejszy, pod innymi brakuje mu pewnych rzeczy. Z tego powodu stwierdziliśmy że bardziej kształcące będzie wygenerowanie kodu pewnej abstrakcyjnej maszyny. Z tego też powodu nie ma implementacji zarządzania pamięcią, ani innych zależnych od platformy części.

10 Podsumowanie

Okazuje się, że napisanie kompilatora, nawet wysokiego poziomu, nie jest ciężkie — duża część teorii jest przystępna i łatwa do implementacji. Z drugiej strony, jest też duże pole do rozwoju – pojawiają się problemy nietrywialne, a całość wymaga dobrej organizacji i staranności. Być może napisany na seminarium kompilator zostanie w przyszłości rozwinięty o brakujące mu części. Mimo pewnych braków, uważam że Implementacja Języków Funkcyjnych zakończyła się sukcesem – zaimplementowaliśmy wszystkie główne części kompilatora.

Literatura

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[3] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.03, documentation and user's manual, 2016.

[4] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[5] Robert A. Shaw. Improving garbage collector performance in virtual memory. 1987.

[6] David Michael Ungar. *The Design and Evaluation of A High Performance Smalltalk System*. PhD thesis, EECS Department, University of California, Berkeley, Feb 1986.