MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY




# Překladač jazyka Scheme

BAKALÁŘSKÁ PRÁCE

**Tomáš Staněk**

Brno, 2007

# Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

**Vedoucí práce:** RNDr. Libor Škarvada

# Poděkování

Rád bych poděkoval RNDr. Liborovi Škarvadovi za možnost dělat tento projekt jako bakalářskou práci a své rodině a přátelům za podporu během její tvorby.

# Shrnutí

Cílem práce je implementovat překladač jazyka Scheme společně s globální statickou analýzou toku a hodnot. Sémantika jazyka má odpovídat standardu r5rs.

## Klíčová slova

Scheme, Optimization, Control flow analysis, Compilation

# Obsah


1 **Úvod** . . . . . 2
2 **Analyzovaný dialekt** . . . . . 3
2.1 Abstraktní syntaxe dialektu . . . . . 3
2.2 CPS forma . . . . . 3
3 **Statická analýza** . . . . . 5
3.1 Abstraktní interpretace . . . . . 5
3.1.1 Abstraktní hodnoty a lokace - value cache . . . 6
Základní hodnoty . . . . . 6
Abstraktní lokace . . . . . 6
Agregátní hodnoty . . . . . 7
Procedury, kontinuace a proměnné . . . . . 8
3.1.2 Call cache . . . . . 8
3.1.3 Větvení a kondicionál . . . . . 9
3.1.4 Vedlejší efekty a generace identifikátorů . . . . . 9
3.1.5 Polyvariantní analýza . . . . . 10
4 **Analýza výsledků abstraktní interpretace** . . . . . 12
4.1 Relace volání proc-kont-calls-to a proc-calls-to . . . . . 12
4.2 Množina called-more-than-once . . . . . 13
4.3 Množiny recursive-procedures a tail-recursive-procedures 13
4.4 Množina indirect-procedures . . . . . 13
4.5 Množina type-tagged . . . . . 14
4.6 Množina eq?-used . . . . . 14
4.7 Přístupy na volné proměnné . . . . . 16
5 **Generování kódu** . . . . . 17
5.1 Zásobníkový způsob versus CPS . . . . . 17
5.2 Způsob volání procedur . . . . . 18
5.3 Inlinování procedur . . . . . 19
5.4 Výstupy analýzy . . . . . 19
6 **Závěr** . . . . . 22
**Literatura** . . . . . 24

## Kapitola 1

# Úvod

V současné době existuje několik aktivně vyvíjených překladačů jazyka Scheme. Jediný volně dostupný překladač, který provádí globální statickou analýzu, je Stalin. Ten bývá často porovnáván při měření rychlosti jednotlivých implementací jazyka, protože ve většině případů generuje nejefektivnější kód. Jak už to u implementací jazyka Scheme bývá, Stalin implementuje vlastní dialekt, který je poměrně zastaralý, překladač obsahuje chyby a k dialektu implementovaného jazyka není k dispozici interpret. Zároveň Stalin není už dlouhou dobu aktivně vyvíjen a tak není přílišná šance na změnu těchto nedostatků. Hlavní motivace této práce je začít práci na volně dostupné implementaci dialektu Scheme, která bude využívat globální statické analýzy.

## Kapitola 2

# Analyzovaný dialekt

## 2.1 Abstraktní syntaxe dialektu

**konstanta** $e$ = 1, "string", ...

**odkaz na proměnnou** $e$ = v

**lambda abstrakce** $e$ = (lambda ($a_1$ $a_2$ ...) $e$)

**aplikace** $e$ = ($a_1$ $a_2$ ...)

**kondicionál** $e$ = (if $e_1$ $e_2$ $e_3$)

**let** $e$ = (let (($v_1$ $e_1$) ($v_2$ $e_2$) ...) $e_3$)

**rekurzivní funkce** $e$ = (letrec (($v_1$ $lambda_1$) ($v_2$ $lambda_2$) ...) $e$)

**přiřazení** $e$ = (set! v $e$)

Tělo programu pak tvoří jeden $e$ výraz.

## 2.2 CPS forma

Program je v první fázi přeložen do CPS tvaru. Každý výraz je značkován lexikálně unikátním identifikátorem.

**konstanta** $a$ = 1, "string", ...

**odkaz na proměnnou** $a$ = $v_{id}$

**lambda abstrakce** $a$ = $lambda$ = (lambda ($a_1$ $a_2$ ...) e)$_{id}$

**aplikace** $e$ = ($a_1$ $a_2$ ...)$_{id}$

**kondicionál** $e$ = (if $a$ $lambda_1$ $lambda_2$)$_{id}$

**rekurzivní funkce** $e$ = (letrec (($v_1$ $lambda_1$) ($v_2$ $lambda_2$) ...) $e$)$_{id}$

**přiřazení** $e$ = (set! v $a_1$ $a_2$)$_{id}$

Odkaz na proměnnou má stejný význam jako v přímé formě. Lambda abstrakce v CPS formě zahrnuje i kontinuace. Ty jsou pro účely překladače odlišeny od uživatelských procedur a přestože větší část chování a reprezentace zůstává pro kontinuace i procedury stejná, v některých ohledech se s nimi zachází jinak. Proto v textu rozlišuji mezi kontinuací a procedurou.

Kondicionál obsahuje místo then a else výrazů výrazy s then a else kontinuacemi. Pokud není $a$ rovno FALSE, tok pokračuje voláním then kontinuace, jinak volá else kontinuaci.

Definice rekurzivních funkcí pomocí letrec se opět neliší od přímé formy.

Operátor přiřazení je opět podobný přímé variantě, jeho poslední argument je však kontinuace.

$a$ výrazy mají dále tu vlastnost, že vždy zastaví a vrátí hodnotu. $e$ výrazy oproti tomu v CPS formě nevrací, ale potenciálně volají předanou kontinuaci se svým výsledkem a pokračují tak v toku.

Let výrazy se v CPS formě nevyskytují, přotože jsou při konverzi přeloženy pomocí lambda abstrakce.

## Kapitola 3

# Statická analýza

U jazyků, které umožňují tvorbu procedur prvního řádu, jsou problémy toku řízení a množin hodnot, které se mohou vyskytovat v proměnných programu, úzce spjaty, protože samotné procedury jsou hodnoty prvního řádu, vznikají za běhu programu a jsou předávány dalším procedurám. Hodnoty tedy silně určují tok programu. Zároveň procedury, které jsou v těchto jazycích tvořeny jsou zpravidla velmi malé a proto jejich lokální analýza přináší často omezené možnosti optimalizace.

Problém přesného toku řízení, stejně jako problém množin hodnot, je obecně nerozhodnutelný. Proto je jediná možnost řešení těchto problémů aproximovat a to tak, aby byly rozhodnutelné, praktické z hlediska optimalizace a zároveň konzervativní, aby byla dodržena sémantika původního programu. Tato konzervativnost je myšlena v tom smyslu, že aproximovaná množina potenciálních hodnot, resp. toků řízení, je vždy nadmnožina množiny hodnot, které se skutečně vyskytnou ve všech možných bězích programu. Množiny skutečných hodnot jsou obecně nekonečné a aby byly zmíněné problémy rozhodnutelné, je třeba tyto nekonečné množiny sjednotit do konečného množství abstraktních hodnot, se kterými se následně pracuje. Přesnost aproximace pak mimo jiné závisí na tom, jak „jemná“ jsou tato sjednocení.

## 3.1 Abstraktní interpretace

Abstraktní interpretace je obdoba konkrétní sémantické funkce programu, která se od ní liší tím, že místo konkrétních hodnot pracuje s hodnotami abstraktními.

Sémantiku abstraktní interpretace pro jazyk Scheme popisuje detailně Olin Shivers [4]

Jednoduchý příklad abstrahování nekonečně mnoha základních konkrétních hodnot do jedné abstraktní, jsou čísla 1,2,3,... abstrahované do jedné symbolické hodnoty INTEGER.

### 3.1.1 Abstraktní hodnoty a lokace - value cache

Při simulaci programu vznikají abstraktní hodnoty, které jsou zaznamenávány v globální value cache. Některé abstrahují konrétní hodnoty, které figurují v bězích programu, jiné jsou použity pro interní účely analýzy. Mimo abstraktních hodnot tvoří value cache abstraktní lokace, což jsou množiny abstraktních hodnot. Value cache tvoří základ pro syntézu typů jazyka C.

#### Základní hodnoty

Analýza pracuje s několika základními abstraktními hodnotami, které jsou pro její účely atomické.

- INTEGER obsahuje všechna přirozená čísla.
- REAL obsahuje všechna čísla v plovoucí desetinné čárce.
- TRUE, FALSE hodnoty odpovídající konkrétním hodnotám TRUE a FALSE.
- STRING obsahuje všechny textové řetězce.
- NULL odpovídá konkrétní hodnotě NULL, která se ve Scheme používá například jako zakončovač seznamu.

#### Abstraktní lokace

Abstraktní lokace jsou množiny abstraktních objektů. Tyto lokace jsou svázány s atributy agregátních objektů a výsledky $a$ výrazů. Každá lokace má unikátní identifikátor, globální pro celou analýzu.

### Agregátní hodnoty

Agregátní hodnoty vyžadují speciální zacházení. Tam, kde například výsledkem součtu dvou čísel je opět číslo, zachycení hodnot do agregátního objektu vrátí objekt „větší“ a rekurzivní volání by porušila podmínku konečného množství hodnot. Je tedy třeba modelovat nekonečné datové struktury konečným způsobem.

Každý agregátní objekt je tvořen identifikátorem odvozeným od místa vzniku a typu agregátu a množinou lokací, které představují jeho atributy. Identifikátory agregátů jsou stejně jako identfikátory lokací unikátní pro všechny objekty, které se vyskytují v analýze.

S každým agregátem, resp. jeho identifikátorem, je navíc svázaná jeho generace. Globální generace je přirozené číslo, které vzrůstá o jedna při tvorbě nového agregátu, nebo změně jeho atributu. O generacích agregátů budu blíže mluvit níže.

Při interpretaci jako hodnota figuruje identifikátor agregátu a při tvorbě agregátních hodnot je tvořena pomyslná gramatika, kde identifikátor hraje roli neterminálu na levé straně pravidla a obsah agregátu pak pravé strany pravidla.

Mějme příklad s páry, kde se rekurzivně staví seznam. Interpretací výrazu (cons $NUMBER\ NULL$) vznikne nový agregát.

$$PAIR_i \rightarrow [l_j l_k]$$

a lokace jeho atributů

$$l_j = \{NUMBER\}$$

$$l_k = \{NULL\}$$

Při rekurzivním volání se bude procedura cons volat s argumenty řekněme (cons $NUMBER\ PAIR_i$). Agregát $PAIR_i$ už existuje, lokace $l_j$ se nemění a lokace $l_k$ se zvětší o novou hodnotu. Nyní tedy

$$l_k = \{NULL, PAIR_i\}$$

Při třetím volání (cons $NUMBER\ PAIR_i$) se už lokace nemění, agregát se dále nerozrůstá a popisuje potenciálně nekonečnou rekurzivní strukturu.

Procedury, kontinuace a proměnné

Scheme je jazyk s lexikálním rozsahem proměnných. To znamená, že procedura může odkazovat proměnnou, kterou sama neváže a tak hrát roli agregátních hodnot. Procedury tedy obsahují všechny komplikace, které ssebou nesou agregátní hodnoty.

Lexikální uzávěr je často implementováno přes takzvané „parent closury“, kde každá procedura ssebou nese odkaz na agregátní objekt, který zachycuje lexikálně bezprostředně nadřazený dynamický kontext, který je tvořen při vstupu do nadřazené procedury. Ten se zároveň odkazuje na kontext dále nadřazený proceduře, která kontext vytvořila.

Při analýze modeluji proměnné stejným způsobem a každá abstraktní procedura ssebou nese abstraktní agregát s nadřazeným kontextem. Tento agregát zároveň obsahuje parent atribut s agregátem, který je nadřazený jemu samému.

Díky CPS konverzi modeluji kontinuace stejně jako procedury. To má důsledky pro polyvariantní dělení procedur, pro generování výsledného kódu a pro lokalizaci proměnných.

### 3.1.2 Call cache

Jediný způsob, jakým může CPS program pokračovat v běhu je volání procedury, či kontinuace. Návrat z primitivních operátorů je realizován voláním návratové kontinuce z interní call-site operátoru. Interpretace zaznamenává každé volání a jemu předané hodnoty. Tyto hodnoty se ukládají do lokací svázaných s výrazy argumentů konkrétního místa volání. Těmito argumenty se rozumí volaná procedura, resp. kontinuace, agregát, který zachycuje lexikální prostředí, a parametry volané procedury, resp. kontinuace. Po cps konverzi procedury nevrací hodnoty, ale místo toho předávají svůj výsledek dalším procedurám, či kontinuacím. Následný tok i předávané hodnoty jsou tedy zcela vyjádřeny hodnotami těchto argumentů. Díky tomu je možné udělat rozhodnutí o zastavení interpretace jen na základě předaných hodnot a předešlých záznamech pro konkrétní místo volání.

Jsou-li hodnoty argumentů volání již obsaženy v záznamu pro konkrétní místo, opětovné volání určitě nepřinese žádnou novou in-

formaci a interpretace se zastaví.

Toto vyžaduje možnost porovnání hodnot práve předávaných s hodnotami zaznamenanými. V případě základních hodnot je situace triviální. V případě procedur se musí rovnat identifikátor procedury a jejich parent agregáty. V případě agregátních hodnot se musí rovnat identifikátor agregátů a všechny lokace atributů uvnitř zaznamenaného agregátu musí byt nadmnožiny lokací atributů porovnávaného agregátu.

### 3.1.3 Větvení a kondicionál

Protože při abstraktní analýze není možné znát přesné hodnoty a tedy výsledky predikátů jako například zero?, volájí predikáty svoje návratové kontinuace ve většině případů dvakrát. Jednou s argumentem TRUE a podruhé s FALSE. To znamená, že kondicionál, do kterého je předáván výsledek predikátu, se bude také volat dvakrát. Jednou s hodnotou TRUE a podruhé s FALSE. To znamená, že interpretece bude sledovat tok then kontinuace i else kontinuace.

### 3.1.4 Vedlejší efekty a generace identifikátorů

Scheme je jazyk s přítomností vedlejších efektů jak proměnných, tak atributů agregátních objektů. To ssebou nese důležité implikace především pro modelování agregátních hodnot (a tím i procedur).

Každý agregát je při vzniku opatřen aktuální generací, která se tímto zvýší. Do call cache se pak ukládají abstraktní hodnoty společně s jejich aktuální generací. Každý agregát si dále uchovává množninu dalších agregátů, v jejichž lokaci, resp. atributu, se vyskytuje. Jinými slovy: které na něj ukazují. Při zvětšení lokace atributu agregátu pak vzrůstá globální generace, která se zároveň propaguje po hranách závislostí. To umožňuje při rozhodování a pokračování interpretace na základě call cache posuzovat jen generaci v call cache a generaci agregátu, který se vyskytuje ve volání místo toho, aby bylo nutné prozkoumávat postupně obsah agregátů. Pokud je generace již zaznamenaného agregátu nižší než právě porovnávaného, tok bude pokračovat, protože je možnost, že se nějaká lokace, na kterou agregát ukazuje, změnila. Toto chování díky modelování proměnných v agregátech uzávěrů postihuje i případy, kdy se porovnává procedura,

která zachycuje proměnnou, která byla změněna.

### 3.1.5 Polyvariantní analýza

Při monovariantní analýze 0-CFA[4] odpovídá každá analyzovaná procedura svému lexikálnímu lambda termu.

V následujícím programu tedy 0-CFA zjistí, že procedura id se volá s argumenty: NUMBER, STRING, FALSE a TRUE.

```
(let ((g  (lambda (h x) (h x)))
      (id (lambda (z)   z)))
  (id 2)
  (id "string")
  (g id #t)
  (g id #f))
```

To je pro účely linearizace a typové inference poměrně nepřesná informace. Zpřesněním je analýza 1-CFA, kde se každá procedura posuzuje odděleně pro každou lexikální call-site. Pro výše zmíněný výraz generuje tři varianty id. Jednu volá s argumentem NUMBER, další s argumentem STRING a třetí s argumenty TRUE a FALSE. Uvážíme-li například funkci map, která je obvyklá ve vysokoúrovňových jazycích a používaná na mnoha místech a která má z hlediska analýzy stejné vlastnosti, jako funkce g, je aproximace poskytnutá 1-CFA opět poměrně nepřesná.

Překladač provádí analýzu podobnou *n*-CFA s tím, že n není konstantní a globální pro celý program, ale procedury se během interpretace dělí na každé call-site. Analýza takto značí jednotlivé varianty podle řetězu takových dělení, který si uchovává jako dynamický kontext. Pokud aktuální řetez obsahuje volanou proceduru, k dělení nedojde, ale uvažuje se předchozí varianta procedury z řetězu.

Pro zmíněný kód tedy analýza generuje čtyři varianty funkce id, která je pokaždé volaná s různým argumentem.

Analýza provádí ještě další dělení. Každý interpretovaný lambda výraz vrací proceduru která ve svém identifikátoru nese variantu procedury, ve které se výraz vyskuje. Dělení při volání pak uvažuje tento dělený identifikátor. V případě, že by byl výše zmíněný výraz obsažen v proceduře, která bude volaná ze dvou míst, bude existovat osm variant funkce id.

Kontinuace podléhají jen druhé formě dělení. Při volání kontinuace neprobíhá žádné další dělení, ale aktuální řetěz aktivací je obnoven na řetěz procedury, ve které byl interpretován výraz, jehož výsledkem byla volaná kontinuace.

Tento způsob dělení odpovídá zásobníkovému volání procedur. Například: procedura $A$ volá proceduru $B$ a jakožto návratovou kontinuaci jí předá kontinuaci nově naalokovanou. Tato kontinuace ssebou nese kontext $A$. Při volání této kontinuace, například jakožto návrat $B$, je její kontext obnoven a aktuální kontext dělení je tedy znovu $A$.

## Kapitola 4

# Analýza výsledků abstraktní interpretace

Po simulaci programu probíhá několik analýz. Zpravidla se jedná o průchody grafů toku, či hodnot. Jednotlivé aspekty a výstupy analýz demonstruji na následujícím programu.

```
(letrec ((loop (lambda (p) ;;loop-1
                 (let ((ignore (p))) ;;lambda-11
                   (loop p)))))
  (loop
   (lambda () ;; lambda-22
     (let ((a 1)) ;; lambda-29
       (let ((b a)) ;; lambda-36
         (let ((c b)) ;; lambda-43
           (let          ;; lambda-70
                ((p (lambda (x) ;; lambda-51
                      (if x
                          (+ (+ x a) c)
                          b))))
             (p 1)))))))))
```

Program *Loop*

## 4.1 Relace volání proc-kont-calls-to a proc-calls-to

Pro všechny varianty procedur a kontinuací se počítá graf proc-kont-calls-to, kde vrcholy tvoří procedury a kontinuace. Z vrcholu A vede hrana do vrcholu B, pokud kontinuace, či procedura A obsahuje výraz, který volá proceduru, či kontinuaci B. Tento graf odpovídá toku CPS formy.

Druhý graf toku proc-calls-to odpovídá zásobníkovému volání procedur. Jeho vrcholy tvoří varianty procedur a z vrcholu A vede

hrana do vrcholu B, pokud výraz v těle procedury A před CPS konverzí volá proceduru B.

## 4.2 Množina called-more-than-once

Množina called-more-than-once obsahuje varianty procedur a kontinuací, které jsou v programu volané potenciálně vicekrát. Jsou to právě ty vrcholy grafu proc-kont-calls-to, které leží na cyklu. Proměnné, vázané proceduramy, které nejsou volané vícekrát, než jednou, je možné překládat ve výsledném kódu do globálních proměnných.
[1]

## 4.3 Množiny recursive-procedures a tail-recursive-procedures

Množiny recursive-procedures, resp. tail-recursive-procedures, obsahují všechny rekurzivní, resp. tail-rekurzivní procedury.

Jsou to právě ty procedury, které nebyly děleny na místě volání, protože se při nějakém volání v kontextu dělených procedur již vyskytovaly.

Procedury jsou tail-rekurzivní, pokud se dovolají samy sebe jen přes tail volání. Značkování procedur v programu loop je vidět na obrázku4.4. Procedury, které nejsou rekurzivní ani tail-rekurzivní, jsou značeny jako simple.

## 4.4 Množina indirect-procedures

Množina indirect-procs-konts obsahuje procedury, jimiž alokovaná kontinuace může být zachycena pomocí procedury call-with-current-

[1] V článku [5], ze kterého moje analýza z velké části vychází, je definovaná podobná relace, ovšem při splňení podmínky, že procedura je volaná z více call-site, nebo je volaná z procedury, která je v called-more-than-once.
Toto není korektní aproximace. Řekněme, že existuje jediná call-site, která volá proceduru A a zároveň je tato call-site v těle procedury B, která je volaná jen jednou. Přesto může být procedura A volaná vícekrát. Pokud bude zachycena kontinuace po vstupu do procedury B a zároveň před voláním procedury A, může být následně vyvolána několikrát a tedy A bude volána několikrát.

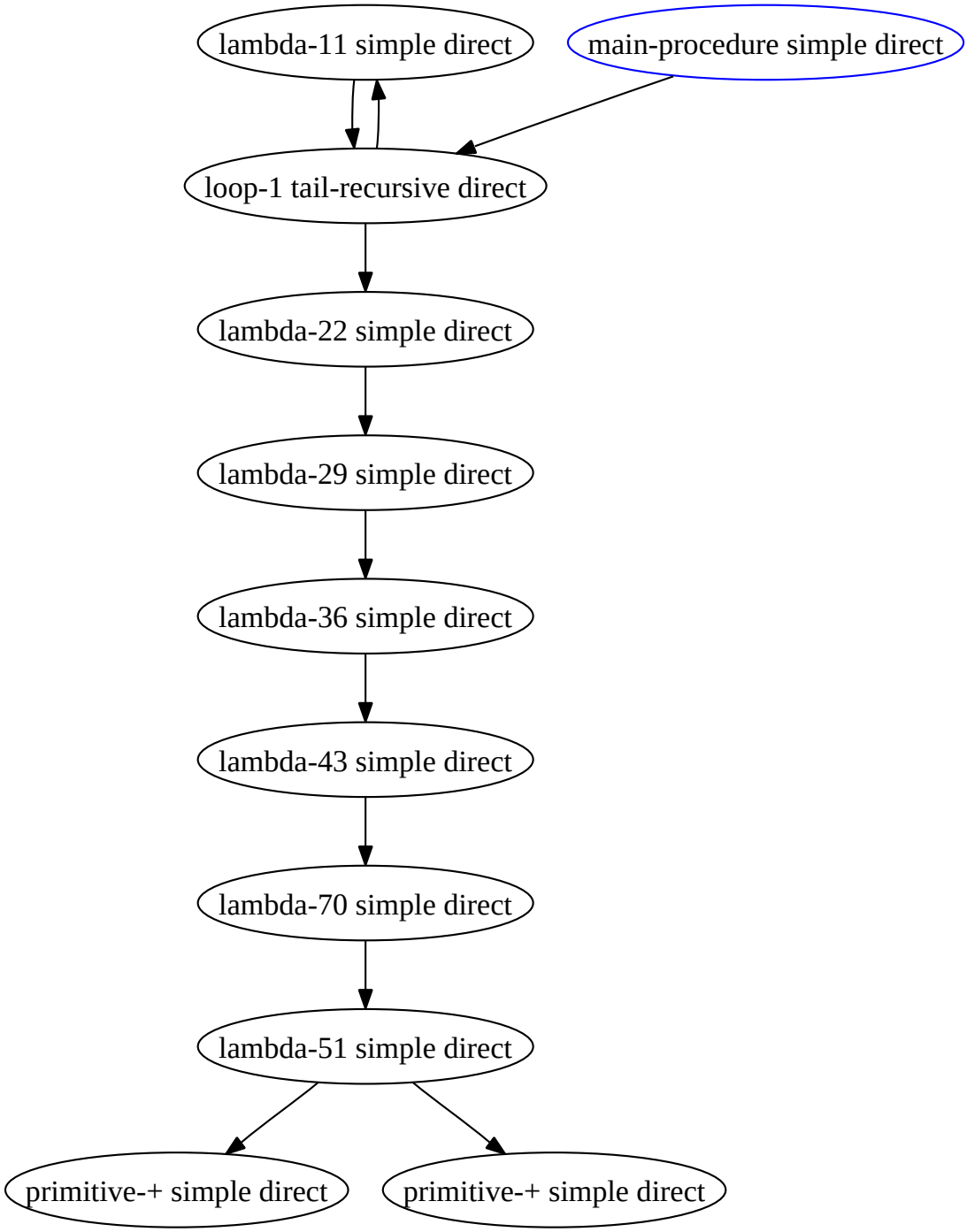
continuation. Tím je myšleno i kontinuace, na které mohou ukazovat volné proměnné v těle jiných kontinuací. Jsou to právě ty vrcholy grafu proc-calls-to, ze kterých je dosažitelný vrchol s procedurou call-with-current-continuation.

## 4.5 Množina type-tagged

Pokud se v nějaké lokaci vyskytnou dvě abstraktní hodnoty různého typu, například INTEGER a STRING, jsou označeny jako type tagged. Takto značené hodnoty mají v generovaném C typ struktury, která na prvním místě obsahuje typový tag, je dynamicky alokovaná a je předávána odkazem. Pokud se tedy v C proměné mají vyskytovat hodnoty více typů, jsou všechny reprezentovány takovouto strukturou a podle tagu je možné zjistit, o kterou se jedná, a provézt přetypování, nebo generovat typovou chybu. Tomuto podléhají všechny hodnoty včetně procedur. Jejich volání v závislosti na typovém tagu je popsáno níže. V případě, že hodnota není v množině type-tagged, nevyžaduje typovou kontrolu za běhu programu. Lokace, které obsahují jen jednu abstraktní hodnotu, která není v type-tagged mohou být otypovány staticky v generovaném C.

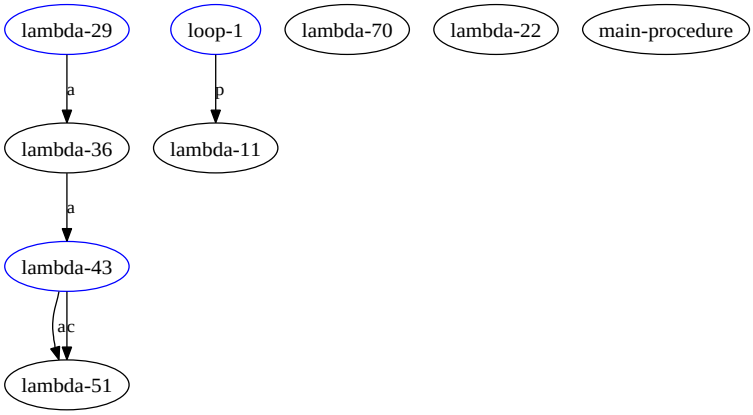
## 4.6 Množina eq?-used

Scheme definuje predikát eq?, který testuje dva předané objekty na identitu. Hodnoty, které jsou takto testovány, jsou dynamicky alokovány a jejich pointr slouží pro test identity. V množině eq?-used jsou právě ty hodnoty, které byly předány jako argumenty proceduře eq? a zároveň toto volání vrátilo hodnotu TRUE i FALSE. Abstraktní procedura eq? pokračuje v toku s oběma hodnotami TRUE a FALSE jen v případě, že jako argumenty dostala dva stejné abstraktní objekty. V případě, že eq? dostává dva různé abstraktní objekty, pokračuje vždy s výsledkem FALSE, protože totožná konkrétní hodnota se nemůže vyskytnout ve dvou různých abstraktních hodnotách.




Obrázek 4.1: proc-calls-to graf programu Loop

Obrázek 4.2: Uspořádání výsledných closur




## 4.7 Přístupy na volné proměnné

Procedury ssebou obecně nesou svůj lexikální uzávěr a typový tag. Pokud nejsou v množině type-tagged, typový tag je eliminován a jsou tvořeny samotným uzávěrem. Pokud zároveň nepřistupují na volné proměnné, je možné eliminovat i lexikální uzávěr. Analýza počítá graf closur a použití volných proměnných. Na obrázku 4.7 je výstup analyzovaného grafu. Modré vrcholy váží nějakou proměnnou, která je přistupována ze zanořené procedury a zároveň analýza nezjistila, že je možné ji globalizovat. Vrcholy, do kterých nevede hrana, označují procedury, které se neodkazují volně na žádnou proměnnou, uloženou v lexikálním uzávěru. Nepotřebují tedy argument s tímto uzávěrem. Zároveň černé vrcholy neváží žádnou volně odkazovanou proměnnou a tedy nemusí při své aktivaci alokovat uzávěr.

## Kapitola 5

# Generování kódu

## 5.1 Zásobníkový způsob versus CPS

Jazyk Scheme umožňuje zachytit aktuální kontinuaci, která se stává objektem první třídy. Tato kontinuace může být následně vyvolána z libovolného místa programu libovolně mnohokrát. To má za následek, že volání procedur nedodržuje obvyklé zásobníkové schéma First In - First Out, což je vlastnost důležitá pro implementaci zásobníkem. Dvě základní metody implementace volání procedur jsou přímá, pomocí zásobníku, a nepřímá, pomocí CPS konverze. V případě přímé metody zachycení, resp. volání zachycené kontinuace, obvykle znamená kopírování aktivačních záznamů ze zásobníku na heap.

Nepřímá metoda pak veškeré aktivační záznamy rovnou alokuje na heap. První metoda penalizuje nepřímé přenosy řízení přes kontinuace, ale běžné volání procedur je zpravidla efektivní. Druhá varianta pak penalizuje všechna volání s tím, že nepříme přenosy řízení jsou stejně efektivní, jako jakékoli jiné volání.

Díky informacím z tokové analýzy je možný hybridní přístup. Generované procedury jsou rozděleny do dvou primárních částí. Veškeré procedury, které se dovolají do procedury call-with-current-continuation, jsou překládány jako CPS, kde volání jak procedury, tak kontinuace, je realizováno přes goto a aktivační záznamy jsou ukládány ny heap.

Ostatní procedury je možné překládat jako přímé C funkce.

Jinými slovy nepřímá procedura může volat přímé procedury, naopak nikoli. Tato vlastnost zaručuje, že je korektní překládat volání zachycené kontinuace z přímé procedury jako longjmp, protože jen volání přímých procedur zvětšuje zásobník.

## 5.2 Způsob volání procedur

Konkrétní procedura je tvořena párem obsahujícím značku procedury a její lokální lexikální uzávěr. Přístupy na volné proměnné jsou pak uvnitř těla procedury realizovány právě přes tento uzávěr.

Uzávěr je implementován C strukturou, která obsahuje hodnoty proměnných, vázaných nadřazenou procedurou, kde konkrétní procedura vznikla a odkazem na její uzávěr. Tento řetěz struktur, pak zachycuje všechny volné proměnné konkrétní procedury.

Samotné volání procedury je realizováno switch výrazem, kde je značka volané procedury porovnávána se všemi značkami procedur, které mohou být z daného místa volány. V každé větvi switch výrazu je pak obsaženo volání příslušné funkce s příslušnými argumenty a jejím lexikálním uzávěrem.

```
switch (proc->tag) {
  case PROCEDURE1:
    result = p1(proc->parent, a1, a2, ...);
    break;
  case PROCEDURE2:
    result = p2(proc->parent, a1, a2, ...);
    break;
  case PROCEDURE3:
    result = p3(proc->parent, a1, a2, ...);
    break;
  ...
  default: error
}
```

To se může zdát na první pohled neefektivní, ale díky tokové analýze, která identifkuje všechny potenciálně volané procedury pro konkrétní místo volání, je počet case výrazu obvykle nízký a díky možné predikci skoků je tento způsob zpravidla efektivnější, než nepřímy skok přes odkaz na funkci. Tento způsob je totožný se způsobem, který používá překladač Stalin.

V případě, že procedura nepřistupuje na volné proměnné, které nejsou lokalizovatelné, globalizovatelné, nebo nepoužité a to samé

platí pro všechny v ní vnořené procedury, agregát s lexikální uzávěrem se negeneruje, ani nepředává.

V případě, že z daného místa je volaná jen jedna abstraktní procedura, místo switch výrazu se generuje přímé volání této procedury.

V případě, že call-site je tail-call, místo plnění proměnné výsledkem procedury, je výraz přeložen na return. [1]

Volání nepřímé procedury je překládáno jako naplňení vstupních argumentů volané funkce a výrazem goto na příslušné návěští. Žádný další aparát není zapotřebí, protože kontinuace nepřímých funkcí se překládají stejně, jako samotné nepřímé funkce a při volání jsou předávány argumentem.

## 5.3 Inlinování procedur

Překladač inlinuje všechny procedury, které nejsou rekurzivní nebo tail-rekurzivní v místě jejich volání. To teoreticky vede k explozi kódu. Analýza momentálně nedělá žádné rozhodování o tom, které procedury se vyplatí inlinovat a které nikoli. Následující graf demonstruje inlinovací rozhodnutí pro program Loop. Vrcholy představují jednotlivé scheme procedury. Z vrcholu A vede hrana do vrcholu B, pokud je B inlinována v A.

Jediná rekurzivní, či tail-rekurzivní procedura v programu je procedura loop. Scheme procedury budou tedy přeloženy do dvou C funkcí. Funkci odpovídající toplevel výrazu, který tvoří program, a funkci odpovídající proceduře loop.

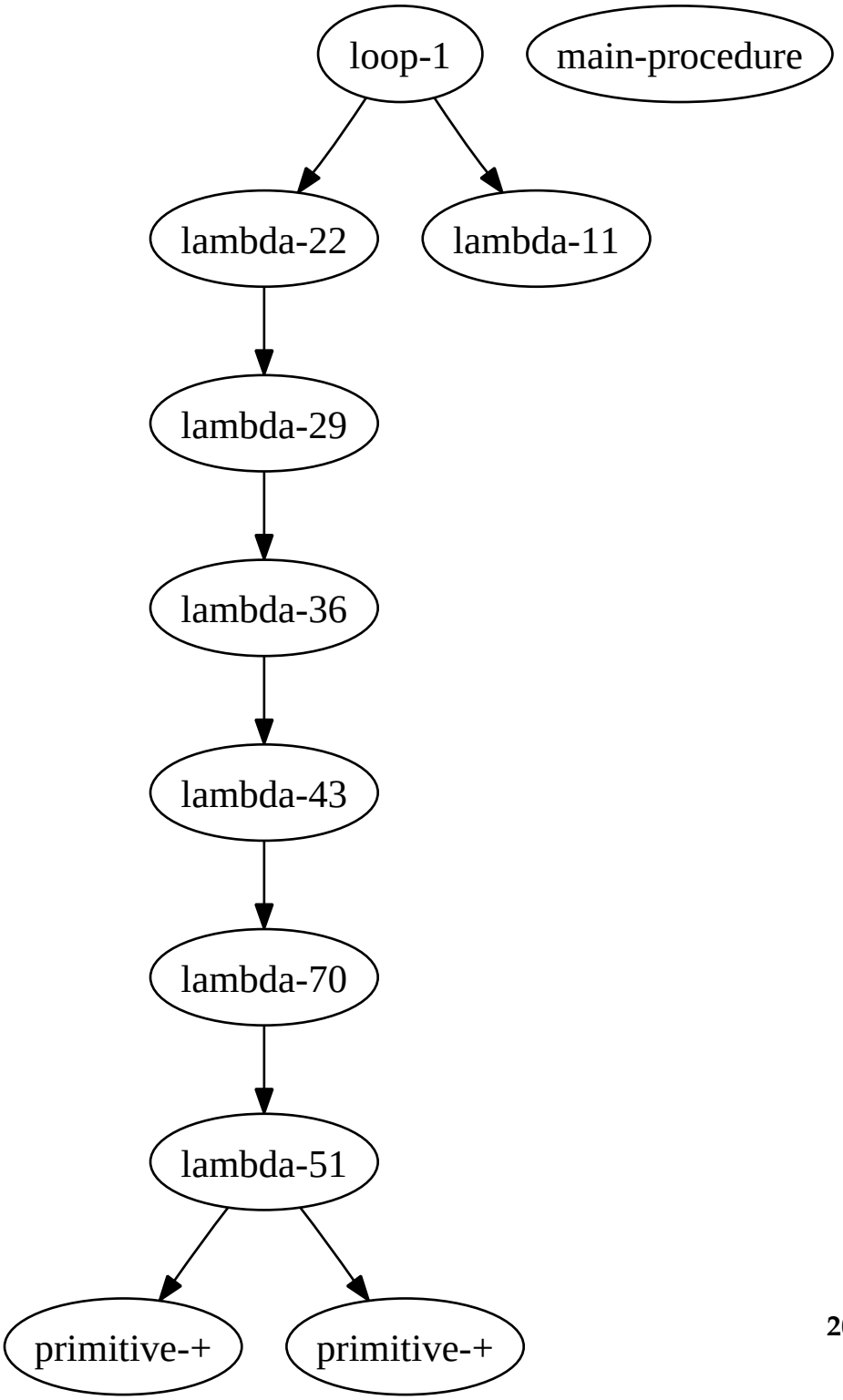
## 5.4 Výstupy analýzy

Program je psán pro interpret scheme Gauche a byl vyvíjen s verzí 0.8.12, která je volně dostupná. Po zavolání skriptu

```
analyze program výstupní-adresář
```

[1] Jazyk scheme vyžaduje garanci tail-call optimalizace. Podle standardu C nemusí být volání funkcí v tail pozici implementovány jako tail-call. GNU gcc však tuto optimalizaci provádí v případě, že volaná funkce je známá v době překladu, volající funkce neodkazuje pointrem na žádnou svoji lokální proměnnou a ve funkci se nevyskytují volání funkcí alloca a setjmp. Generovaný kód tyto podmínky splňuje.

Obrázek 5.1: Graf inlinovaných procedur



analýza vytvoří několik souborů.

- cps, anotated-cps
  vstupní výraz do CPS formy společně s identifkátory a značením typů volání.
- globalizable variables
  identifkátory lokací proměnných, které je možné přeložit do globálních proměnných.
- procedure-blocks
  popis přímých procedur a jejich kontinuací, společně s předávanými abstraktními hodnotami a abstraktními hodnotami, které byly procedurou vráceny.
- indirect-procedure-blocks
  popis nepřímých procedur
- inlined-in.dot
  graf inlinovaných procedur pro vysázení programem dot z balíku graphviz
- proc-calls-to.dot
  graf proc-calls-to
- proc-kont-calls-to
  graf proc-kont-calls-to
- slotted-variable-dependancies.dot
  graf procedur vzhledem k jejich nutným lexikálním uzávěrům
- values.dot
  graf všech hodnot, které se vyskytly při interpretaci
- proc-bound-locations
  seznam procedur a kontinuací, jejich vázaných proměnných společně s metodami jejich alokace

## Kapitola 6

# Závěr

Implementace kompletního překladače společně s analýzou se ukázala velmi pracná a časově náročná. Převážně pak v momentech, kdy je nutné z informací získaných anlýzou generovat cílový kód.

Během práce na překladači se několikrát ukázala nutnost explicitního pojmenování a přístupu k elementům analýzy, zejména lokacím, lexikálních uzávěrům, apod, které byly původně v anlýze obsaženy jen implicitně. To ve výsledku zapříčinilo, že poslední fáze generování kódu je neuplná, negeneruje tedy korektní C kód a nedá se prezentovat jako finální překladač. Analýza zpracovává všechny zásadní a z hlediska efektivní implementace problematické aspekty jazyka Scheme tak, aby odpovídaly jejich definici v r5rs, nicméně analyzuje jen jeho syntakticky normalizovanou formu a nepodporuje některé aspekty, jako variabilní počet argumentů, vracení více hodnot atp. Nejsou kladena žádná omezení na použití zachycených kontinuací první třídy, na použití vedlejších efektů včetně přiřazování do lokací s primitivnímy operátory a na použití procedur prvního řádu.

Aktuální výsledky analýzy, které překladač provádí, poskytují dostatečné informace k výše popsané metodě generování kódu. Program tyto výsledky prezentuje ve formě grafů, které je možné vysázet v balíku graphviz, a dodatečných souborů s informacemi o proměnných a procedurách.

Informace z abstraktní interpretace zároveň umožňují výrazně silnější optimalizace, než jsou zde popsány a vyžadují další zpracování. Aktuální interpretace pro ně ale dává dobrý základ.

Mimo samotné optimalizace poskytuje statická analýza informace hodnotné i v dalších aplikacích. Například informace o hodnotách, které se vyskytují v proměnných, by mohly velmi dobře sloužit pro introspekci kódu při vývoji, kde by bylo možné ve vývojovém pro-

středí, jako například GNU Emacs, ukázat na proměnnou a zobrazit všechny potenciální hodnoty, které se v ní vyskytnou. Stejně tak by ta samá informace mohla sloužit pro chytřejší navigaci po zdrojovém kodu podobně, jako pomocí etags.

Další velmi zajímavá možnost rozvoje překladače je dialekt jazyka, jehož typový systém bude založen na této analýze. Nepovinné typové anotace by byly unifikovány s abstraktními hodnotami, které poskytuje analýza. To by umožňovalo kombinaci statického a dynamického typování v jednom jazyce a to pravděpodobně nenásilným a plynulým přechodem od dynamicky typovaného kódu, jehož vlastnosti jsou žádoucí v ranných fázích vývoje, k částečně, či plně typovanému kódu v pozdějších fázích vývoje, kdy se systém rozrůstá a typová kontrola je velmi cenná.

# Literatura


[1] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, California, 1995.

[2] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised$^5$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, pages 26–76, 1998.

[3] Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, USA, 1991.

[4] Olin Grigsby Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, New Haven, CN, June 1991.

[5] J. Siskind. Flow-directed lightweight closure conversion, 1999.