Innehåll

1 Inledning | Bakgrund och motivering **3**

2 Vårt Sockerspråk **3**
- 2.1 Introduktion . . . 3
- 2.2 BNF . . . 4
- 2.3 Jämförelse med Haskell . . . 4
- 2.4 Definitioner . . . 5
  - 2.4.1 Fria variabler och substitution . . . 5
  - 2.4.2 Partiell evaluering . . . 6

3 Corespråk **6**
- 3.1 Olika maskiner . . . 6
- 3.2 Semantik över STG . . . 6
- 3.3 Avsockring av Sockerspråk till STG . . . 11
  - 3.3.1 Lambdalyftaren . . . 13
- 3.4 Våra utökningar . . . 13
  - 3.4.1 Boxing . . . 13
  - 3.4.2 IO . . . 14
  - 3.4.3 Print continuation . . . 14
- 3.5 Callstack . . . 15
  - 3.5.1 Substitution . . . 15
  - 3.5.2 Localise . . . 15

4 Optimise **18**
- 4.1 Introduktion till optimise . . . 18
- 4.2 Call by Value semantik . . . 20
- 4.3 Optimise ... with 22
  - 4.3.1 Case Branches . . . 23
  - 4.3.2 Inline-räknare . . . 23
- 4.4 Call by Name semantik . . . 23
- 4.5 Dödkods eliminering . . . 23
- 4.6 Välkänd case-lag . . . 24
- 4.7 Afterburner . . . 25
  - 4.7.1 Gemensam caseganskare . . . 25

5 Resultat **26**

6 Diskussion **26**
- 6.1 Implementerbarhet i kompilator . . . 26
- 6.2 Lathet . . . 27
- 6.3 Optimise-with . . . 27
- 6.4 Sharing . . . 27
- 6.5 Call stack . . . 28
- 6.6 Optimise med call stack . . . 29

7 Relaterade Arbeten **29**
7.1 Statiska optimeringar 29
7.2 Optimeringar under körningstid 29
7.3 Funktionella maskiner 29

8 Framtida Arbeten **29**

1 Inledning | Bakgrund och motivering

Vanligtvis optimeras kod endast vid kompileringsfasen. I följande Haskell-exempel skulle en sådan optimering inte ge ett optimalt resultat.

```
power :: Integer -> Integer -> Integer
power 0 x = 1
power n x = x * power (n - 1) x

powers :: Integer -> [Integer] -> [Integer]
powers n xs = map (power n) xs

main = do
    n <- getIntegerFromUserInput
    print (powers n [1..100])
```

Funktionen *powers* tar in ett tal n och en lista xs, och kör funktionen *power* n, som upphöjer ett tal till n, på varje element i listan. Om indatan n till *powers* är känd statiskt (när programmet kompileras) skulle en kompilator kunna göra en specialiserad version av *power* för just det n-värdet. Till exempel, när $n = 4$:

```
power_4 :: Integer -> Integer
power_4 x = x * x * x * x
```

Den specialiserade versionen av *power* är snabbare än originalet, då den inte är rekursiv och inte behöver titta på värdet som n har.

I *main*-funktionen ovan är dock n inte känt när programmet kompileras, utan läses istället in från användaren under körningstid. Detta gör att en kompilator inte har tillräckligt mycket information för att kunna göra en specialiserad version av *power*.

Vi har i det här projektet tagit fram en möjlig lösning på detta problem, där programmeraren kan annotera var optimeringar ska ske under körningstid.

2 Vårt Sockerspråk

2.1 Introduktion

För att undersöka optimeringsmöjligheterna behövs ett språk att arbeta med. Detta språk behöver inte vara lika rikt som Haskell, men tillräckligt bekvämt att arbeta med för att skriva några större program. Språket som används i den här rapporten kommer att refereras till som sockerspråket, för att det innehåller syntatiskt socker. Detta är bekväma sätt att skriva olika saker, fast de skulle gå att skriva mer omständigt. Ett klassiskt exempel är syntaxen för listor i Haskell, *[5,0,4]* som är socker för *5 : 0 : 4 : []* . Det går att översätta Haskell till sockerspråket, och naturligtvis i också i motsatt riktning. Sockerspråket är också designat så, att det är lätt att översätta och avsockras till core-språket som introduceras i nästa kapitel.

2.2 BNF

$$
\begin{aligned}
program &::= && \overline{function} \\
function &::= && v\ \overline{v}{=}expr; \\
expr &::= && v \\
&\mid && v\ \overline{expr} \\
&\mid && \texttt{let}\{\overline{function}\}\texttt{in}\ expr \\
&\mid && \texttt{letrec}\{\overline{function}\}\texttt{in}\ expr \\
&\mid && \texttt{case}expr\texttt{of}\{\overline{branch}\} \\
&\mid && \lambda v.expr \\
&\mid && expr \oplus expr \\
&\mid && K\ \overline{expr} \\
&\mid && primitive \\
branch &::= && K\ \overline{v} \rightarrow expr; \\
&\mid && v \rightarrow expr;
\end{aligned}
$$

Programmet är en lista av funktioner. Dessa funktioner är de som sägs vara på *toppnivå*. Funktioner som är skapade med en lambdaabstraktion eller i en letsats är inte på toppnivå. Funktionerna har en identifierare, dess namn, med noll eller fler variabler som argument. Efter likhetstecknet kommer deras funktionskropp, som är ett uttryck, expr, och de avslutas med ett semikolon. Uttrycken kan antingen vara en variabel, ett funktionsanrop som är ett funktionsnamn applicerat på en eller fler uttryck. Det kan också vara let eller letrec-bundna funktioner (eller variabler i det fallet då de är funktioner som inte tar något argument), eller en casesats som tar ett uttryck och flera grenar. Som uttryck tillåts också primitiva operationer som skrivs infix, tex addition och multiplikation. Att returnera en konstruktor eller en primitiv som exempelvis ett heltal eller decimaltal är också ett uttryck. Grenarna kan antigen vara att mönstermatcha mot en konstruktor och binda dess konstruktionsvariabler, eller att matcha vilken konstruktor som helst.

2.3 Jämförelse med Haskell

En feature som saknas i sockerspråket som ofta används flitigt i Haskell är möjligheten att mönstermatcha närhelst en variabel binds, tex i vänsterledet i en funktionsapplikation. Det är hursomhelst så, att alla mönstermatchningar, även med vakter, går att översätta enbart med enkla casesatser. Vårt sockerspråk har sådana, och de är enkla i det avseendet att man inte kan mönstermatcha i mer än ett djup, och att det inte finns några vakter.

Här är insertmetoden skriven i Haskell, den sätter in ett element i en sorterad lista så att den är sorterad efteråt.

```
insert :: Ord a => a -> [a] -> [a]
```

```
insert v (x:xs)
    | v < x     = v:x:xs
    | otherwise = x:insert v xs
insert v [] = [v]
```

I sockerspråket hade det blivit

```
insert v list = case list of
    Cons x xs -> case v < x of
        True  -> v:x:xs
        False -> x:insert v xs
    Nil -> v:Nil
```

I Haskell definieras användarnas datatyper explicit, som i definitionen av Maybe:

```
data Maybe a = Just a | Nothing
```

I sockerspråket finns ingen sådan konstruktion, utan det är bara att använda konstruktorerna Just och Nothing när så behövs.

Sockerspråket gör åtskillnad på rekursiva och vanliga let-bindningar.

```
repeat x = letrec xs = x : xs in xs
```

Här används xs, som binds av letrec-satsen, även i uttrycket efteråt. Detta finns inte i Haskell, då alla let-bindningar räknas som rekursiva. I sockerspråket får användaren också ha i åtanke att let-bundna variabler binds sekvensiellt, dvs att de måste skrivas i ordning:

```
let { t1 = f x y
    ; t2 = g y t1
    }
in  h t1 t2
```

Här går det inte, till skillnad från i Haskell, att byta plats på t1 och t2.

2.4 Definitioner

2.4.1 Fria variabler och substitution

Låt oss säga att vi har en funktion f som använder sina två argument, x och y. Om vi betraktar funktionskroppsuttrycket för sig är x och y *fria variabler*. Det skulle inte gå att evaluera funktionskroppen utan att känna till vad de är. Vet man om x och y:s värden, låt oss säga att de är True och 9, så kan man *substituera* in dem i uttrycket. Då byts varje x ut mot True och varje y mot 9. Om uttrycket är e skrivs det $e[True/x, 9/y]$. Ett sätt att konkret realisera en substitution är att traversera trädet för uttrycket och jämföra varje variabel och ersätta om det är en av de relevanta. Detta är den effekten man alltid vill eftersträva rent abstrakt. Det går också att tänka sig andra sätt att implementera substitution, genom att tex göra ett uppslag i en tabell varje gång en fri variabel funnes när koden evalueras.

2.4.2 Partiell evaluering

Detta arbete handlar till stor del om partiell evaluering. Men vad innebär det? I funktionella språk är funktioner första-ordningens medlemmar och kan tex skickas med som argument till andra funktioner, eller så kan de returneras från andra funktioner. Tag tex zipWith, här skriven i Haskell:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _      _      = []
```

Vad kan f vara här? Det kan tex vara en unär funktion som returnerar en ny funktion. Det kan vara et funktion som tar tre argument och redan har ett applicerat:

```
let f = (\x y z -> x * y - z)
in  zipWith (f 3) list1 list2
```

Funktioner som har några, men inte alla argument applicerade kallas att vara partiellt applicerade. Att börja evaluera en sådan funktion, partialevaluera, är inget som vanligtvis görs i Haskell, och vad som undersöks i detta projekt.

3 Corespråk

3.1 Olika maskiner

För att evaluera program skrivna i ett lat funktionellt programmeringsspråk behövs en evalueringsmodell. Det finns olika abstrakta maskiner utvecklade för detta ändamål, tex G-maskinen, TIM (Three Instruction Machine), GRIN, och STG. Dessa går att dels tolka, dvs skriva ett program som kopierar maskinens funktionalitet i något programmeringsspråk, exempelvis Haskell. Man kan också kompilera dem till något lågniva-språk som C, assembly eller LLVM. I det här arbetet har vi valt att utveckla en egen tolk. Att skriva en kompilator hade varit ett delikat arbete i sig, även utan optimering. Vi ville fokusera mer på optimeringen och valde därför att skriva en tolk. Av de olika maskiner vi undersökte när vi valde vilken vi skulle tolka så ansåg vi att STG var mest lämpad. I de andra ovan nämnda maskinerna är instruktionerna lågnivå, liknande assemblerinstruktioner men för att manipulera ett tänkt syntaxträd. STG, å andra sidan, ser ut som ett funktionellt programmeringsspråk och det skulle underlätta att arbeta på optimeringen av den anledningen.

3.2 Semantik över STG

[1][usedefault, addprefix=, 1=]$1_1 \ldots 1_n$

[1][usedefault, addprefix=, 1=]$1_1 \ldots 1_m$

Språket som vi har definerat är en utökning till STG-maskinen, med nya regler och tillstånd. För att kunna diskutera dessa behöver man först förstå hur maskinen fungerar. Eftersom läsaren inte förväntas ha stött på begrepp som

används när man beskriver abstrakta maskiner och deras semantik, kommer vi att bygga upp förståelsen via ett exempel och introducera koncepten efter hand det uppstår. En abstrakt maskin kan beskrivas av att den evalueras stegvis under evalueringen, detta kommer bli tydligare med hjälp av genomstegning av evalueringen av följande program.

```
cube x = x * x * x

main = let list = [2,3,5,7,11]
        in map cube list
```

Det som evalueras först är main funktionen och där finns ett let-uttryck. De är verktyget för att binda variabler, i det här fallet variablen list, till listan av de fem första primtalen. Man kan använda let för att associera andra sort värden än listor, t.ex. andra sorters datastrukturer, numeriska värden samt funktioner.

Vi behöver någonstanns att reservera utrymme för listan så därför inför vi en så kallad heap. I ett tradionellt, imperativt språk är det en representation av minnet som är för dynamisk allokering. När maskinen stöter på ett let-uttryck så allokerats det på heapen, detta beskrivs formellt av följande regel.

$$\texttt{let x=e in code}\,;\, Heap \Rightarrow \texttt{code}\,;\, Heap[x \mapsto e]$$

Läs det som att om maskinen är i tillståndet till vänster om $\Rightarrow$så blir nästa tillstånd det som står på högra sidan. Ett tillstånd i maskinen är ett uttryck och en heap, formellt separeras dessa med ett semikolon. I vårt fall så går maskinen från let-uttrycket till det som står efter in, samt att x läggs till i heapen och pekar på e. Heap[] betyder inte att heapen bara innehåller det som är inuti hakparanteserna utan är bara notation för att explicit påvisa vad en variabel binds till.

Maskinens tillstånd är nu:

$$\texttt{map cube list}\,;\, Heap[list \mapsto [2,3,5,7,11], cube \mapsto \lambda x.x \cdot x \cdot x, \ldots]$$

Den observanta läsaren märker att även cube ligger på heapen, det beror på att funktioner på topnivå läggs in i heapen i maskinens initialtillstånd. Alltså ligger även även map och andra kända standardfunktioner där.

Som uttryck ligger nu ett anrop till map funktionen, den vill vi anropa. Här låter vi oss inspereras av imperativ anropsmetodik. Argumenten lägger vi på en stack. Formellt får vi då denna regel:

$$f\ a_1\ \ldots\ a_n\,;\, Stack\,;\, Heap \Rightarrow f\,;\, a_1 : \ldots : a_n : Stack\,;\, Heap$$

Stacken är placerade mellan utrycket och heapen och vi använder : för att separera elementen på stacken, den tomma stacken skrivs som ϵ. Maskinens tillstand ar:

$$\texttt{map}\,;\texttt{cube : list : }\epsilon\,;\, Heap$$

Nu behover vi diskutera koden for map:

```
map f xs = case xs of
    { Cons x xs -> let { a = f x
                       ; b = map f xs
                       } in Cons a b
    ; Nil       -> Nil
    }
```

Anledningen till att argumenten till Cons inte ar funktionsapplikationer ar att maskinen blir mer latthanterlig om funktionsapplikationer enbart far letbundna variabler, aven kallade atomer. Vi ser har att map tar tva argument och ater saledes upp tva argument fran argumentstacken. Formellt:

$$f\,;\, a_1 : \ldots : a_n : \mathit{Stack}\,;\, \mathit{Heap}[f \mapsto \lambda x_1 \ldots x_n.e] \Rightarrow e[a_1/x_1 \ldots a_n/x_n];\, \mathit{Stack}\,;\, \mathit{Heap}$$

Har behover vi ersatta alla funktionens argument $x_1 \ldots x_n$till de atomer $a_1 \ldots a_n$som finns pa stacken. Vi noterar denna substitution med $e[a_1/x_1 \ldots a_n/x_n]$. I vart fall sa substitueras f mot cube och xs till list. Maskinen:

$$\texttt{case list of}\,\{\ldots\};\, \epsilon\,;\, \mathit{Heap}$$

I heapen pekar list på listan [2,3,5,7,11] men detta är socker för:

```
list = Cons 2 temp1
temp1 = Cons 3 temp2
temp2 = Cons 5 temp3
temp3 = Cons 7 temp4
temp4 = Cons 11 nil
nil = Nil
```

De temporära variablerna behövs för att konstruktorer skall kunna allokeras i ett sammanhängande minne i heapen. I vårat fall pekar list variabeln på en Cons konstruktor, som har en motsvarande gren i map funktionens definition. Det som ska hända är att maskinens uttryck blir grenen med x substituerat mot 2 och xs mot temp1. Detta kan vi åstadkomma med denna formella regel.

$$\begin{aligned} \mathit{case}\ v\ \mathit{of}\ \{\ldots; C\ x_1\ \ldots\ x_n\ \rightarrow e; \ldots\}\,;\, \mathit{Stack}\,;\, \mathit{Heap}[v \mapsto C\ a_1 \ldots a_n] \\ \Rightarrow e[a_1/x_1\ \ldots\ a_n/x_n]\,;\, \mathit{Stack}\,;\, \mathit{Heap} \end{aligned}$$

Det som står i granskaren, uttrycket mellan case och of, var i vårt fall en variabel. Det finns också fall då granskaren är andra uttryck t.ex. funktionsapplikation. I så fall vet vi inte vilken gren vi ska ta innan vi har evaluerat granskaren, då måste vi evaluera den först. Vi måste veta efter vi evaluerat klart vilken case-sats den skall gå tillbaka till, det här är snarlikt en returadress. Standard i imperativa språk är att returadresser pushas på stacken. Det vi gör är att lägga ut en case-continuation på stacken som också innehåller grenarna till case-satsen.

Formellt kommer vi beskriva en case-continuation som $case\ \bullet\ of\ \{\ldots\}$ där $\bullet$ står för platsen vi skall lägga in resultatet. Så regeln blir följande:

$$case\ e\ of\ \{\ldots\}\,;\,Stack\,;\,Heap \Rightarrow e\,;\,case\ \bullet\ of\ \{\ldots\} : Stack\,;\,Heap$$

Det här uttrycket kommer att evalueras till ett värde t.ex. en konstruktor. Då kommer case-continuation ligga överst på stacken och vi vill välja rätt gren. Vi vill återanvända grenväljarregeln som vi definerade ovan så därför inför vi denna regel som ställer in maskinen för detta. Formellt blir det:

$$v\,;\,case\ \bullet\ of\ \{\ldots\} : Stack\,;\,Heap \Rightarrow case\ v\ of\ \{\ldots\}\,;\,Stack\,;\,Heap$$

För att återgå till löpande exempel så är koden just nu detta:

```
let { a = cube 2
    ; b = map cube temp1
    } in Cons a b
```

Nu skall a bindas till ett uttryck, f 2, i tradionella språk skulle f 2 först evalueras och dess värde bindas till a. Men eftersom STG är lat så väntar den med att beräkna det, ifall den inte skulle behövas. Ett sådant senarlagt uttryck kallas för en thunk, det som händer är att a binds till thunken `f 2`. På heapen finns flera olika sorters objekt: konstruktorer, lambda funktioner och nu även thunkar. Formellt kommer vi behöva skilja dem åt, så vi inför ny notation för dessa objekt.

Föregående syntax	Faktisk syntax	Beskrivning
$C\ [a]$	$CON(C\ [a])$	Konstruktor
$\lambda[x].e$	$FUN([x] \rightarrow e)$	Lambdafunktion
-	$THUNK\ e$	Senarelagd beräkning

Så här skrivs egentligen map funktionen i STG språket utan något socker:

```
map f xs = case xs of
    { Cons x xs -> let { a = THUNK (f x)
                       ; b = THUNK (map f xs)
                       ; c = CON (Cons a b)
                       } in c
    ; Nil       -> xs
    }
```

Notera att i STG är det bara med let man kan allokera objekt, således måste vi använda en tempvariabel c för att returnera en konstruktor. Numera är vårt exempel med riktig syntax egentligen:

```
let { a = THUNK (cube 2)
    ; b = THUNK (map cube temp1)
    ; c = CON (Cons a b)
    } in c
```

Maskinen kommer stega igenom let-uttrycket och allokera objekten. Därefter returneras en variabel som pekar på en konstruktor i heapen som är en Cons, skapad enligt ovan. När man retunerar något som är en konstruktor sägs den vara i weak head normal form (WHNF). Denna behöver inte vara helt evaluerad, som exempel har vi vårt fall där den innehåller två pekare till olika thunkar. Eftersom STG är ett lat språk så är det upp till koden som kallade på map att fortsätta evaluera dessa thunkar ifall de behövs eller inte. I vårt fall är det main som har kallat på map och då main skriver ut det den har binds till kommer den att evaluera thunkarna. Vi tänker oss fallet då den börjar evaluera variabeln a.

$$x\,;\,Stack\,;\,Heap[x \mapsto THUNK\,e] \Rightarrow e\,;\,Upd\,x\,\bullet\,:\,Stack\,;\,Heap[x \mapsto BLACKHOLE]$$

Evalueringen av en thunk kommer då att flytta uttrycket till den nya koden för evaluering. Den kommer också lägga ut en Upd continuation, detta återigen för att STG är lat. Lathet kräver att utryck bara evalueras en gång, däför kommer vi när e har evaluerats till ett värde att spara det så att x pekar på det nya värdet och om x's värde skulle behövas igen kan man returnera det direkt. Under evalueringen av x sätter vi även x till ett BLACKHOLE, detta sker av två anledningar.

1. Om x beror på sig själv, eller flera variabler beror på varandra utan att producera värden. Om man evaluerade utan att sätta till BLACKHOLE skulle vår maskin komma in i en oändlig loop. Detta är ett sätt att upptäcka det och vi kan berätta för programmeraren att det har blivit ett fel.

2. Garbage Collection, gcn kommer gå igenom alla heap och stack objekt samt koden efter vilka värden som fortfarande kan nås via den nuvarande koden samt stacken. Thunken kan referera till objekt, som inte längre behövs för evalueringen, men gc kommer inte se att dessa objekt skulle kunna garbage collectas. Om man istället sätter det till ett BLACKHOLE kommer gc veta att den inte behöver söka efter vilka objekt som den behöver, då alla som behövs för den finns redan på koden samt stacken.

Det som saknas för thunkar är nu regeln som uppdaterar dem när de är klara, det sker via denna regel.

$$v\,;\,Upd\,x\,\bullet\,:\,Stack\,;\,Heap \Rightarrow v\,;\,Stack\,;\,Heap[x \mapsto Heap[v]]$$

Om v är ett värde så uppdateras x med vad v pekar på.

Det saknas nu bara en feature innan vår beskrivning av STG är komplett. Vad gör vi med funktioner som inte fått alla sina argument?

```
max x y = case x < y of
    { True  -> y
    ; False -> x
    }
```

Om man nu skriver `max 100` så är det en ny funktion som retunerar det maximala talet mellan dess input och 100. För detta kommer ett nytt heap objekt att skapas vid namn PAP som står för partial application. Denna kommer innehålla argumenten samt en pekare till funkitonen.

$$f\,;\, Arg\, a_1 : \ldots : Arg\, a_m : Stack\,;\, Heap[f \mapsto FUN([x] \to e) \Rightarrow p\,;\, Stack\,;\, Heap[x \mapsto PAP(f\,[a])]$$

Där antalet argument m är mindre än antalet parametrar n, alltså är det en sparad funktionsapplikation som kommer utföras när resterande argument har kommit. Så för att avsluta så kommer regeln som verkligen gör applikationen av dessa partiellt applicerade objekt.

$$p\,;\, Arg\, a_{n+1} : Stack\,;\, Heap[p \mapsto PAP(f\,[a]) \Rightarrow f\,;\, Arg\, a_1 : \ldots : Arg\, a_n : Arg\, a_{n+1} : Stack\,;\, Heap$$

Denna regel kommer återigen försöka göra applikationen, antigen lyckas den, eller så kommer den skapa ett nytt PAP objekt med ytterligare argument.

3.3 Avsockring av Sockerspråk till STG

STG är ett ganska primitivt språk med tex att man kan bara anropa funktioner och skapa konstruktorer med argument från atomer, samt att alla objekt måste explicit annoteras om de är thunkar eller konstruktörer. För att underlätta att skriva kod skapade vi ett sockerspråk för att underlätta detta, och ett pass över koden som avsockrar.

```
isPrimer n t = case t * t >= n of
    { True -> True
    ; False -> if (n % t == 0) False (isPrimer n (t + 1))
    };
```

Den här koden kommer från testsviten och är del i ett program som testar primalitet. Här används sockerspråket flitigt för att göra koden läsbar och hanterlig. I exemplet ovan är *isPrimer* är ett funktionsobjekt och i STG och skrivs det i en stil mer lik en lambdafunktion:

```
isPrimer = FUN (n t -> ...)
```

I STG tar funktioner endast atomer som argument. Detta gör att i jämförelsen $t * t >= n$ behövs först kvadreringen sparas bindas till en temporär variabel:

```
let temp = THUNK (t * t)
in  (>=) temp n
```

Liknande för if-satsen: ifs argument *(n % t == 0)* och *isPrimer n (t + 1)* är inte atomer så de behövs bindas och i sin tur är *n % t* och *t + 1* inte atomer till dessa funktionsanrop till *==* och *isPrimer* .

Allt med inledande versal i sockerspråket tolkas som konstruktorer, precis som i Haskell. Så *True* ovan i koden behöver allokeras med en let. Den första grenen blir i så fall *True -> let temp = CON (True) in temp* Faktum är att de nullära konstruktorerna *True* , *False* , *Nil* , finns allokerade på toppnivå i tolken och instantieras endast en gång. Men i alla andra fall behövs de letbindas.

Detta gäller även heltal och andra primitiver. 0 avsockras till *let temp = CON (I# 0) in temp* Notera att alla temporära variabler som skapas är unika.

För att göra det lättare att operera på listor och göra booleska jämförelser finns infixoperatorerna &&, || som alias för && och || respektive, och ++ och : för append och cons. Funnes någon av dessa infixvarianter byts de ut under avsockringen.

I haskell används också $ flitigt som funktionsapplikation med låg högerfixitet, exempelvis i denna kod som testar quicksort med en lista sorterad i fallande ordning:

```
main = qsort $ reverse $ take 3 $ iterate (\x.x + 1) 0;
```

Detta avsockras till motsvarande funktionsanrop:

```
main = qsort (reverse (take 3 (iterate (\x.x + 1) 0)));
```

En annan metod är att göra $ ett alias för apply:

```
main = apply (qsort (apply reverse (apply take3 (iterate (\x.x + 1) 0))));

apply f x = f x;
```

Detta ger så stor overhead för något så simpelt som funktionsapplikation att det är rimligare att göra det till funktionsanrop direkt.

> Notering!
> Detta kanske borde vara i optimise-avsnittet

Avsockring av optimise with

```
let f = optimise foo x y with { caseBranches, inlinings = n * n }
in map f zs
```

avsockras till

```
let { temp1 = THUNK ( n * n )
    ; temp2 = OPT ( foo x y ) with { caseBranches, inlinings = temp1 }
    ; f = THUNK ( case temp1 of _ -> temp2 }
    }
in map f zs
```

I sockerspråket är optimise ett nyckelord som ser ut som en funktionsapplikation, men avsockras till ett OPT-objekt. Dessutom så forceras alla dess inställningar med hjälp av en casesats.

3.3.1 Lambdalyftaren

I tolkandet av koden underlättar det om man inte har några let-bundna funktioner. Dessa behöver lyftas ut tilkl toppnivå. I sockret kan dessa uppstå antingen av att man skriver lambdafunktioner eller let-bundna funktioner. Ett exempel från quicksort:

```
Cons x xs ->
    let { lesser y = y < x
        ; listLess = filter lesser xs
        ; listMore = filter (\y . y >= x) xs
        }
    in  qsort listLess ++ x : qsort ListMore
```

Först översätts let-satsen med hjälp av avsockraren

```
.. let { lesser = FUN (y -> let t = THUNK (y < x) in t)
       ; listLess = THUNK (filter lesser xs)
       ; lambda0 = FUN (y -> let t = THUNK (y >= x) in t)
       ; listMore = THUNK (filter lambda 0 xs)
       } ..
```

När dessa funktioner, lesser och lambda0, lambdalyfts, skapas nya funktioner på toppnivå. En sak att ha i åtanke då är att de fria variablerna i funktionskroppen nödvändigtvis kommer från ursprungsfunktionen. Dessa måste skickas med som extra parametrar. I det här fallet rör det sig om variabeln x i båda fall. Dessa parametrar måste också skickas med när de anropas.

– på toppnivå

```
lesserLifted = FUN(x y -> let t = THUNK (y < x) in t)
lambda0Lifted = FUN(x y -> let t = THUNK (y >= x) in t)
```

– nya letsatsen:

```
 .. let { lesser = THUNK (lesserLifted x)
        ; listLess = THUNK (filter lesser xs)
        ; lambda0 = THUNK (lambda0Lifted x)
        ; listMore = THUNK (filter lambda0 xs)
        } ..
```

En sak att poängtera är att listLess och listMore inte behövs ändras för att de lyftna funktionerna partialevalueras med samma namn. Här lesser som blir lesserLifted partiellt applicerat på x.

3.4 Våra utökningar

3.4.1 Boxing

För att få en uniform representation av användardefinierade datatyper, skapade med en konstruktor, och primitiva datatyper som Int, Double och Char, ges

även de sistanämnda konstruktorer. I# för Int, D# för Double, C# för Char. De har invarianten, till skillnad från egna datatyper, att de inte pekar på en thunk utan är fullt evaluerade. Detta kallas att boxa primitiven. För att utföra addition skapas den här funktionen som avboxar primitiven, forcerar och utför primitiv addition och sen boxar igen:

```
x + y = case x of
    { I# x' -> case y of
        { I# y' -> case x' +# y' of
            { r -> let r' = CON (I# r) in  r'}}};
```

Så + är additionen som användaren av vårt språk får använda och +# är den primitiva additionen som inte exporteras för användaren. Notera att $x' + \#y'$ forceras med hjälp av case för att den ska vara fullt evaluerad.

3.4.2 IO

Tolken har en primitiv form av IO: den har konstant indata vid körning och skriver ut det main returnerar, som behöver vara en konstruktor. Dessutom forceras hela konstruktorn till reducerad normalform.

Funktionerna getInt, getIntList, getDouble, getDoubleList och getString finns till hands under körning om användaren har angett någon eller några av dessa i anropet till tolken. Dessa är konstanta under körningstid och gör det lättare att testa programmen genom att ändra deras indata utan att modifiera källkoden.

3.4.3 Print continuation

> Notering!
> Finns lite arbete och göra på dessa

Utdata fås genom att main evalueras fullt. Två till continuations behövs - Print och PrintCont. Uttrycksdatatypen utökas med en ny typ av värden, sk s-värden, som antingen är en evaluerad primitiv, tex en int, eller en konstruktor. Om någon som ska skrivas ut evalueras till en (partiellt evaluerad) funktion så skrivs <FUN> istället ut. Detta är också ett s-värde.

```
sval ::= n num
       | K C [sval]
       | <FUN>
```

nya Cont:

* Print * PrintCont C [sval] [atom]

Print betyder att det som evalueras ska skrivas ut. PrintCont består av ett konstruktorsnamn C, och först en lista på färdiga s-värden och en lista med atomer som maskinen har kvar att evaluera.

nya regler:

(1)

$$x; Print : s; H[x \mapsto CON(C\ a_1 \ldots a_n)] => \begin{cases} KC[]; s; H, n = 0 \\ a_1; Print : PrintConC[][a_2 \ldots a_n] : s; H, n > 0 \end{cases}$$

Om x har evaluerats till en konstruktor C så finns två fall. Antingen att C är en nullär konstruktor och i det fallet är vi klara och ger s-värdet bestående av konstruktorn C. är det inte en nullär konstruktor behövs alla dess konstruerande atomer evalueras, med start på a1. Den läggs ut på maskinen med en print-continuation och under denna en contiunation som säger att de andra atomerna också ska evalueras.

(2) x ; Print : s ; H , x value other than sval => n x ; s ; H

Om x är en primitiv datatyp, tex en int eller double, så skapa detta s-värde.

(3) x ; Print : s ; H[x -> FUN / PAP] => <FUN> ; s ; H

Om x pekar på en funktion skapa s-värdet <FUN>.

(4) sv ; PrintCon C ps (n : ns) : s ; H => n ; Print : PrintCon C (ps ++ sv) ns : s ; H

Ett s-värde är fullt evaluerat och då kan nästa atom i konstruktorn evalueras.

(5) sv ; PrintCon C ps [] : s ; H => K C (ps ++ sv) ; s ; H

Konstruktorn är helt evaluerad när den kommit till sista konstruerande atomen.

3.5 Callstack

3.5.1 Substitution

Till en början använde vår tolk sig av substitution, vilket innebär att man traverserar syntaxträdet och byter ut variabler mot deras värde, exempelvis vid funktionsapplikation.

Detta är långsamt och orealistiskt. I en riktig kompilator vill man istället använda sig av en stack för de lokala variablerna så att man slipper traversera något träd.

3.5.2 Localise

I vår tolk har vi ett pass där vi går från ett syntaxträd där alla variabler antas substitueras (eller ligga på heapen) till ett träd där lokala variabler har information om sin position på call-framen.

Genom exempel ska vi förklara hur lokaliseringstransformationen går till.

```
map f xs = case xs of
    { Cons y ys -> let
        { a = THUNK (f y)
        ; b = THUNK (map f ys)
        ; c = CON (Cons a b)
        } in c
    ; Nil -> xs};
```

Eftersom argumenten skall ligga på stacken så ger vi dem indicier som anger dess position på stacken. Vi visar detta med <x,n> där x är vad variabeln kallades innan och n är indexet på stacken.

```
map <f,0> <xs,1> = case <xs,1> of
    { Cons y ys -> let
        { a = THUNK (<f,0> y)
        ; b = THUNK (map <f,0> ys)
        ; c = CON (Cons a b)
        } in c
    ; Nil -> <xs,1>};
```

Eftersom case samt let också binder på stacken (pekaren till objektet) så skall även dessa ges indicier.

```
map <f,0> <xs,1> = case <xs,1> of
    { Cons <y,2> <ys,3> -> let
        { <a,4> = THUNK (<f,0> <y,2>)
        ; <b,5> = THUNK (map <f,0> <ys,3>)
        ; <c,6> = CON (Cons <a,4> <b,5>)
        } in <c,6>
    ; Nil -> <xs,1>};
```

Här får vi dock ett problem med att thunkarna kommer att evalueras någon annanstanns, och därmed i en annan stack frame. Vi kommer därför evaluera thunkar i en egen stack frame. Men thunken kan använda sig av de lokala variabler som redan finns. I exemplet har vi till exempel att <a,4> = THUNK (<f,0> <y,2>) där f och y är kommer "utifrån". Därför måste thunken veta vilka lokala variabler i map den använder, så att de kan läggas på dess stack frame. Noterbart är även att när let allokerar kommer den att byta ut alla stackvariabler mot vad de pekar på i stacken. Så att Con (Cons pekar på heap objekt och inte stacken)

```
map <f,0> <xs,1> = case <xs,1> of
    { Cons <y,2> <ys,3> -> let
        { <a,4> = THUNK [<f,0>, <y,2>]  (<f,0> <y,1>)
        ; <b,5> = THUNK [<f,0>, <ys,3>] (map <f,0> <ys,1>)
        ; <c,6> = CON (Cons <a,4> <b,5>)
        } in <c,6>
    ; Nil -> <xs,1>};
```

I <a,4> = THUNK [<f,0>, <y,2>] (<f,0> <y,1>) ser vi att f och y har fått nya index jämfört med vad de hade utanför thunken, vilket beror på att de som sagt kommer att evalueras med en egen stack frame.

Det största värdet på variabelindex + 1 blir funktionens stackstorlek, vilken man alltså kan veta statiskt (vid kompileringstid i en kompilator). mapfunktionens stackstorlek blir här 7.

Alla funktioner har då information om sin stackstorlek, så att den kan allokeras när funktionen ska köras. Detta möjliggör en effektiv implementering som kan använda arrayer av konstant storlek, vilket ger en konstant lookup-tid i call-framesen. Även thunkar vet sin stakstorlek eftersom de också allokeras på sin egna stack frame, dessa räknas ut på liknande sätt som för funktioner, där de utomstående lokala variablerna (de så kallade fria variablerna) kan ses som funktions argument. Thunken THUNK [<f,0>, <y,2>] (<f,0> <y,1>) har stackstorlek 2.

Ett lurigt fall är letrecs.

Följande (ineffektiva) funktion f ger tillbaka en bool som säger om argumentet n är jämnt eller inte. [exempel sockerspråk]

```
f n = letrec
    { even x = if (x == 0) True  (odd  (x - 1))
    ; odd  x = if (x == 0) False (even (x - 1))
    } in even n;
```

Funktionen lambdalyfts (se tidigare avsnitt) först till följande:

```
f n = letrec
    { even = THUNK (even' odd)
    ; odd  = THUNK (odd' even)
    } in even n;
even' odd x = THUNK (if (x == 0) True  (odd  (x - 1)));
odd' even x = THUNK (if (x == 0) False (even (x - 1)));
```

Här måste even ha odds index och odd ha evens index. Vi får följande:

```
f <n,0> = letrec
    { <even,1> = THUNK [<even,1>,<odd,2>] (even' <odd,1>)
    ; <odd,2>  = THUNK [<even,1>,<odd,2>] (odd' <even,0>)
    } in <even,1> <n,0>;
```

Den observanta läsaren ser att i första thunken läggs even till utan att den används. Detta är ett implementations val som tillåter stg att alltid lägga till allt som binds i en letrec i början på callstacken när thunkarna ska evalueras. Att hålla reda på vilka thunkar som behöver vad är svårt.

Nu ska vi visa hur anropsstacken kan se ut vid ett anrop till map.

```
main = map cube [1,2,3];
```

argumenten pushas först på continuation-stacken så att den ser ut typ: Arg cube Arg [1,2,3]

Sen ser maskinen att map tar två argument, och funktionsapplikationen kan då köras, och vi lägger argumenten på argumentstacken och går in i funktionen. Argumentstacken blir: [cube, [1,2,3]] eftersom f i definitionen av map har index 0 och xs har index 1.

4 Optimise

4.1 Introduktion till optimise

Vi börjar med att återigen betrakta powerfunktionsexemplet från introduktionen.

```
power n x = let t1 = Thunk (n==0)
              in case t1 of
                      True -> 1
                      False -> let t2 = n - 1
                                   t3 = power t2 x
                                   t4 = x * t3
                               in t4
```

Vi ser att *power 3 5* kommer upphöja 5 till 3, dvs 5 * 5 * 5

Om vi känner till exponenten men inte basen kan vi göra en optimerad version av power specialicerad för just den exponenten, hur detta går till ska vi nu försöka visa på ett informellt sätt. Misströsta ej alla detaljer ges senare.

```
optimise (power 3)
```

Vi börjar med att byta ut alla n mot 3 i power. Ingeting kan göras med x då den är okänd. Vi 'döper' också om power för att inte förstöra orginalet.

```
power_3 x = let t1 = Thunk (3==0)
               in case t1 of
                     True -> 1
                     False -> let t2 = 3 - 1
                                  t3 = power t2 x
                                  t4 = x * t3
                              in t4
```

Vi börjar nu gå igenom trädet. Först möter vi *let t1 = ... in exp* vilket sparas undan och vi gå vidare in i *exp* som är ett case-uttryck. Case:en kan forcera fram värdet oss t1 då t1 är fri från fria variabler. Då *3==0* är falskt fås nu:

```
power_3 x = case False of
              True -> 1
              False -> let t2 = 3 - 1
                           t3 = power t2 x
                           t4 = x * t3
                       in t4
```

Vi kan nu välja False branchen, vi sparar sedan undan let:arna och går in i t4. Vi inlinar det hela (Egentligen allokeras 3-1 på heapen och forceras fram senare)

```
power_3 x = x * power (3-1) x
```

Vi forstätter med att inline power (3-1) x på samma sätt.

```
power_3 x = x * x * x * 1
```

Vilket är en mycket mindre och snabbare funktion. Men det här är inte hela sanningen, egentlingen boxas varje heltal i vårt språk (står säkert någon annanstans). Och + är som bekant bara socker för att case fram värdet i av intarna och sedan addera det med en primitiv plus operator #+

```
a * b
```

blir efter inlining

```
case a of
    I# a' -> case b of
        I# b' -> case  a' #* b' of
            r' -> let r = I# r'
                      in r
```

Och om vi då inlinarna alla multiplicationer i power_3 får vi följande kod:

```
power_3 x = case (
                    case (
                    case x of
                       I# a -> case x of
                           I# b -> case a #* b of
                                  r' -> let r = I# r'
                                   in r
                       )
                         of
                       I# c -> case x of
                           I# d -> case c #* d of
                                  r' -> let r = I# r'
                                   in r
                  case x of
                       I# a -> case x of
                           I# b -> case a #* b of
                                  r' -> let r = I# r'
                                   in r
                       )
                         of
                       I# c -> case c #* 1 of
                                  r' -> let r = I# r'
                                   in r
```

Vi case:ar flera gånger på x och bygger upp I# för att direkt ta bort de senare. Detta kan naturligtvis också optimeras.

```
power_3 x = case x of
              I# x' -> case x' #* x' of
                       a -> case x' #* a of
                          b -> case x' #* b of
                             c -> case x' #* 1 of
                               r' -> let r = I# r'
                                 in r
```

Detta är hur långt vår optimise funktionalitet kommer. x * 1 skulle naturligtvis kunna optimeras till x, men vi har inte fokuserat på dessa corner cases. I resten av det här avsnittet kommer vi i detalj förklara vart och ett av dessa steg och varför det fungerar som det gör.

4.2 Call by Value semantik

Vi har använd ett par olika semantiker för att beskriva optimeringsskeded i vårt språk, vi ska nu förklara var och ett av dessa i kronologisk ordning.

Detta optimeringspass arbetar sig igenom träded på ett sätt inte helt olikt stg-maskinen, dock kan vi bland annat gå in och evaluera funktion utan att känna till alla argument, s.k. evaluera under lambdat. För att inte duplicera för mycket av stg funktionallitet så anropas optimise stg för dvs uppgifter.

Det finns tre delar i optimeringen

Omega, Ingångspunkten till optimeringsfunktionen, anropas från maskinen när något har blivit annoterat för optimering. Omega bryter ner träded och delegerar vidare vem och vad som ska ske, samt lägger ut kontinuations för att senare bygga upp ett förhoppningsvis ny träd.

$$\begin{aligned}
\Omega_{s;H}(\texttt{case}\, e\, \texttt{of}\, brs) &\Rightarrow & \Omega_{O\texttt{case}\,\bullet\,\texttt{of}\, brs\,:\,s;H}(e)\\
\Omega_{s;H}(f\,\bar{k}) &\Rightarrow & f\,\bar{k};s;H\\
\Omega_{s;H}(f\,\bar{x}\,k\,\bar{y}) &\Rightarrow & f\,\bar{x}\,k\,\bar{y};OInlining\,:\,s;H\\
\Omega_{s;H}(\texttt{let}\, x = THUNK\, e\, \texttt{in}\, e') &\Rightarrow & \Omega_{O\texttt{let}\, x=THUNK\,\bullet\,\texttt{in}\, e'\,:\,s;H}(e)\\
\Omega_{s;H[x\rightarrowtail]}(\bullet) &\Rightarrow & \Phi_{s\,;\,H}(\bullet)
\end{aligned}$$

Irr, kortform för Irreducable. Anropas när Omega eller Psi inte längre kan optimeringa vidare. Irr kan då utifrån nuvarande continuations bygga träded eller byta continuation och fortsätta optimera på en andra delar i uttrycket. Där är här vi kan tillslut fly tillbaka till stgmaskinen.

$$\begin{aligned}
\Phi_{O\texttt{let}\, x=THUNK\,\bullet\,\texttt{in}\, e\,:\,s\,;\,H}(e') &\Rightarrow & \Omega_{O\texttt{let}\, x=THUNK\, e'\,\texttt{in}\,\bullet\,:\,s;H}(e)\\
\Phi_{O\texttt{case}\,\bullet\,\texttt{of}\, brs\,:\,s\,;\,H}(e) &\Rightarrow & \Phi_{s\,;\,H}(ECase\, e\, brs)
\end{aligned}$$

Istället för att skapa en ECase e brs har vi även testat att gå in och optimera varje branch, detta brytter dock semantiken, vi kan fastna i oändliga loopar i brancher vi aldrig hade hamnat i.

```
case x of
    True  -> 5 : Nil
    False -> repeat 1
```

Om x är okänd och vi försöker optimera False branchen kommer vi fastna i en oändlig loop, dock är det möjligt att när programet körs kommer x bara anta värdet True.

$$\Phi_{O\texttt{let}\, x=OBJ\, \texttt{in}\, \bullet\, :\, s\, ;\, H}(e) \Rightarrow \qquad \Phi_{s\, ;\, H}(\texttt{let}\, x = OBJ\, \texttt{in}\, e)$$
$$\Phi_{OFUN\, (\bar{x}\rightarrow\bullet)_\alpha\, :\, s\, ;\, H}(e) \Rightarrow \qquad \alpha; s; H[\alpha \mapsto FUN(\bar{x} \rightarrow e)]$$

Psi, anropas från när optimise har lagt ut något åt maskinen och maskinen inte kan komma längre. Psi kommer använda värdet från maskinen och sedan delegera vidare optimeringen.

$$\Psi_{O\texttt{let}\, x=THUNK\, \bullet\, \texttt{in}\, e\, :\, s\, ;\, H}(v) \Rightarrow \qquad \Psi_{s\, ;\, H}(e[v/x])$$

Här byter vi ut alla x mot värdet v som maskinen har arbetat fram. Här använder vi en substitute funktion som går igenom hela e uttryckträd vilket är väldigt kostsamt. Något som vi bytte ut i senare versioner.

$$\Psi_{O\texttt{let}\, x=OBJ\, \texttt{in}\, \bullet\, :\, s\, ;\, H}(v) \Rightarrow \qquad \Phi_{s\, ;\, H}(\texttt{let}\, x = OBJ\, \texttt{in}\, v)$$
$$\Psi_{O\texttt{case}\, \bullet\, \texttt{of}\, brs\, :\, s\, ;\, H}(v) \Rightarrow \qquad \Omega_{s;H}(\text{instantiate correct brs with v})$$

Instantiate correct branch väljer helt enkelt vilken branch som matchar värdet som maskinen arbetat fram. Detta är troligtvis en av de viktigaste stegen i vår optimering då kan skala bort alla onödiga brancher, minska koden och oftast fortsätta optimera en bit in det nya uttrycket.

Stg, då vi interagerar med stg maskinen måste vi har tydliga regler för när maskien ska sluta evaluera och börja optimera. Detta sker antingen då den stöter på en OPT(t) för första gången, eller när den inte längre kan evaluera ett uttryck och har en Optimise kontinuation på stacken.

$$a; s; H[a \mapsto OPT(t)] \Rightarrow$$
$$t; OPT\, a\ :\ s; H[a \mapsto BLACKHOLE]$$
$$x; OPT\, a\ :\ s; H[x \mapsto PAP(f\, a_1 \cdots a_n)\, ;\, f \mapsto FUN(x_1 \cdots x_m \rightarrow e)] \Rightarrow$$
$$\Omega_{OFUN\, (x_{n+1} \cdots x_m \rightarrow \bullet)_a\, :\, s; H}(e[a_1/x_1 \cdots a_n/x_n])$$
$$v; O.\star\ :\ s; H[v \twoheadrightarrow THUNK\, e] \Rightarrow$$
$$\Psi_{O.\star\, :\, s\, ;\, H}(v)$$

Oturligt nog bröt ovanstående optimise semantik mot stgs egna semantik. Som bekant forceras evaluering av thunkar endast i case-granskaren. Om funktionen som optimeras hade forcerat en thunk under vanlig körning så är opimise också tillåten att göra detta. Det finns ibland anledning att focera även vid andra tillfällen, men vårt naiva tillvägagångsätt höll inte måttet.

```
take n list = if (n == 0) Nil (head list : take (n-1) (tail list))
```

Här vill man först evaluera n == 0 innan någon av argumenten till if. If är defienierad med en case sats på det första argumentet. Även om if inlinas så ligger inte casen-först i funktionen. För att bättre förstå varför kan vi observera hur take koden ser ut osockrad.

```
take n list = let t1 = THUNK (n == 0)
                  t2 = THUNK (head list)
                  t3 = THUNK (n - 1)
                  t4 = THUNK (tail list)
                  t5 = THUNK (take t3 t4)
              in  if t1 Nil t5
```

Vi evaluerar varje ny thunk efter vi passerat den, när vi kommer till t5 görs ett rekursivt anrop till take och processen börjar om. 'if t1 Nil t5' får aldrig en chans att avbryta loopen och optimeringen terminerar aldrig.

Definierades take istället på följande sättet terminerar optimise:

```
take n list = case n == 0 of
                  True -> Nil
                  False -> let t2 = THUNK (head list)
                               t3 = THUNK (n - 1)
                               t4 = THUNK (tail list)
                           in  take t3 t4
```

Och det är oeftersträvansvärt att tvinga användarna att skriva all funktioner de vill optimera med detta i åtanke. Användaren behöver inte bara hålla reda på hur sina funktioner är skrivan, men också hur alla funktioner han använder är skrivna. Detta arbetar emot principen om abstraktion vilket är orimligt. Detta leder oss in på vår strävan efter en lat optimeringssemantik.

4.3 Optimise ... with ..

Det är finns alltid specialfall där brott mot semantiken hade varit fördelaktigt. Sådana fall ytterst svåra att idenfiera maskinellt, därför ombes användaren ge ytterligare argument till optimeringsfunktionen när detta är dessa specialfall uppstår.

```
optimise f x with { options }
```

4.3.1 Case Branches

Ibland finns det möjlighet att optimera i en casebranch. Detta bryter semantiken då detta kan forcera evaluering av uttryck som vanligtvis inte skulle beräknats.

```
f x y z = case g z of
        A -> case h x y of
              R -> t1 z
              S -> t2 z
        B -> case h y x of
              R -> t3 z
              S -> t4 z
```

Om vi nu använder *optimise(f 1 2)* , så är z okänd, men i brancherna så finns det case-sastser som bara beror på x och y och rätt gren kan alltså väljas. Vi har stöd för att optimera i sådana här casegrenar:

```
optimise f x y with { casebranches }
```

Detta kan bryta semantiken, tex om

```
f x z = case z of
    True  -> error -- tex en oändlig loop
    False -> x
```

Om vi här försöker optimera i brancherna då z är okänd kommer optimeringen att loopa, och det skulle programmet inte gjort om t ex. f alltid får False som argument.

4.3.2 Inline-räknare

När man valt en utökning som bryter mot semantiken kan man inte alltid vara säker på att den terminerar, därför kan inline-räknaren användas för att garantera det det högst ser ett visst antal inlines.

Det går att både begränsa det globala antalet inlines och inlines för olika funktioner.

```
f z = case g z of
   A -> repeat 1
   B -> Nil
```

Här hade det varit otrevligt att försöka inlina repeat då den aldrig terminerar.

4.4 Call by Name semantik

4.5 Dödkods eliminering

När diverse optimeringar har körts och det är dags att bygga upp träded igen kommer så kallad död kodätt förekomma. Det vill säga, kod som inte längre behövs eftersom den inte kan nås.

```
main = let t = getValue() in
            case True of
               True  -> 15
               False -> t
```

Detta exemplet är trivialt att optimera då värdet i casen är känd. Vi kan därför initisera den korrekta branchen.

```
main = let t = getValue() in 15
```

Vi ser nu att t inte längre är bunded i uttrycket (15) och därför kan vi betrakta *let t = getValue* som 'död kod'. Död kod tar inte bara onödig plats vilket i sig är ett stort problem, kod ligger på flera pages i minnet och access tiden blir otroligt mycket högre, det tar även längre tid att traversera träded i interperten eller i andra optimeringar. Därför är det viktigt att eftersträva en så kort kod som möjligt.

```
main = 15
```

Denna process kan mer formellt skrivas som

```
t not in freevars(e2)
++++++++++++++++++++++++++
let t = e1 in e2 => e2
```

Detta är en kontroll som utförs varje gång vi bygger upp en let i Irr och Psi. Dock är freevars ganska kostsam (bör stå om det här någon annanstans).

4.6 Välkänd case-lag

Alternative title, case identitet.

Väldigt ofta uppstår en casesats inuti en casesats, vilket är svårt att optimera vidare. Vi utnyttjar i dessa tillfällen en identitet hos case:ar.

```
case (case x of
        A a -> x'
        B b -> y'
     )
  X -> r
  Y -> y
```

$\Leftrightarrow$

```
case x of
    A a -> case x' of
             X -> r
             Y -> y
    B b -> case y' of
             X -> r
             Y -> y
```

Nu är det lättare att applicera flera av våra andra optimeringar, t ex gemensam casegranskare.

4.7 Afterburner

Alla optimeringar som hitills har arbetar tillsammans, dvs att de traverserar trädet endast engång och följer en gemensam semantik för att nå det önskade resultatet. Detta är önskvärt ut tidssynpunkt fast gör det ibland svårt att byta ut enskilda delar utan att behöva ändra på många smådelar.

Det finns även en del optimeringar som behöver se större delar av syntaxträded samtidigt för att kunna vara verksamma. Dessa high-level (vad ska vi kalla det här?) optimeringar är oftast mycket långsamare att utföra.

Vi har valt att utföra ett antal sådana optimeringar efter de vanliga optimeringana är körda, därifrån namnet äfter burner".

4.7.1 Gemensam caseganskare

Att case:a på samma uttryck två gånger kommer alltid i ett funktionellt språk att ge samma resultat då alla variabler egentligen är konstanter. För varje casebranch vi går in i vet vi att alla nästlade case på samma uttryck måste initiera samma branch.

```
test x = x * x
```

Blir efter vanlig optimering

```
test x = case x of
            I# a -> case x of
                I# b -> case a #* b of
                    r' -> let r = I# r'
                            in r
```

Om vi sedan kör gemensam casegranskare erhålls:

```
test x = case x of
            I# a -> case a #* b of
                r' -> let r = I# r'
                    in r
```

Generellt:

```
case x of
    C a b -> e1
    D a b -> e2
```

Vi söker upp *case x* i *e1* och initiera C branchen med a b som värden. I *e2* gör vi samma sak fast initierar D branchen.

5 Resultat

Notering!

- Jämför hur kod ser ut före och efter optimering. Förklara lite vad som har gjorts.
  - Power har använts som löpande exempel, så vi skulle kunna visa det här också.
  - Andra program. Förslagsvis något av de lite större exempelprogrammen om vi får dem att fungera bra och kan presentera det på ett vettigt sätt. Shapes, regexp
- Jämför snabbheten hos program med och utan optimering (benchmarks)
  - Med och utan callstack för att visa hur mycket snabbare resultat det gav oss?
  - scatterplot för en större mängd program (el. körningar)
- Jämför kodstorlek, komplexitet hos optimise
- Hur stor del av tiden går åt till att köra optimise? En graf skulle kunna visa detta på ena axeln och listlängd på andra axeln, för att illustrera att en optimerad funktion måste köras många gånger för att man ska vinna något på det.

6 Diskussion

Notering!
Vad vi har gjort, helhetsbild, metaobservationer, kritik

Notering!
Vissa av dessa passar också in i future work

Vi har gjort en tolk för ett funktionellt programmeringsspråk med en semantik som följer STG-maskinen som används i Haskellkompilatorn GHC. Till denna har vi lagt till möjligheten att optimera partiellt applicerade funktioner under körningstid då vi känner till värdena som applicerades och kan utnyttja dessa till optimering.

6.1 Implementerbarhet i kompilator

Vi valde att göra en tolk eftersom det ofta är enklare än att implementera en kompilator. En tolk innebär att man känner till syntaxträdet hos koden, vilket vanligtvis kompilerad kod inte gör, vilket gör att om det skulle implementeras i en kompilator skulle denna information behöva kompileras ner för att finnas tillgänglig när programmet körs.

6.2 Lathet

Vid optimeringen vill man att koden ska bli så optimerad som möjligt, men man får vara försiktig så att man inte är för aggressiv, eftersom det kan leda till icke-terminering.

```
take n xs = let
    { a1 = THUNK (n == 0)
    ; a2 = THUNK (case xs of
        { Nil -> Nil
        ; Cons x ys -> let
            { b1 = THUNK (n - 1)
            } in take b1 ys
        }
    } in if a1 Nil a2;

main = (optimise (take 5)) [1,2,3,4,5,6,7];
```

I exemplet ovan skulle en sämre designad optimise-funktion gå in och optimera a2 oberoende av värdet på a1, och eftersom värdet på xs inte är känt när funktionen optimeras leder det till icke-terminering om man inlinar det rekursiva anropet till take. Den första semantiken som presenterades led av detta problem, men löstes i den senare med hjälp av en semantik som var latare, inte forcerade evaluering av alla thunkar och närmare semantiken från STG.

6.3 Optimise-with

En annan möjlig lösning på problemet ovan är att låta programmeraren själv avgöra hur många gånger en funktion får inlinas.

```
main = optimise (take 5) with { inline take 5 } [1,2,3,4,5,6,7];
```

I exemplet ovan betyder inline take 5 att take bara får inlinas 5 gånger. En funktionalitet som skulle kunna gynna optimise, eller användandet av den, är att den skulle kunna dektektera när den potentiellt skulle falla in i en oändlig inliningloop. Det har inte gjorts till det här arbetet.

6.4 Sharing

Ett annat problem förknippat med lathet är att optimise måste se till att variabler bara evalueras en gång. Detta ställer till besvär hur man skall optimera let-satser.

```
test x y = let
    { a = THUNK (fib x + y)
    } in (+) a a;

main = optimise (test 10) 5;
```

Med vår senare semantik inlinas först additionen, och sedan inlinas thunkens innehåll som motsvarar a. Det gör att optimise sedan kommer att köras två gånger på samma kod. Med en optimering som tog delning i åtanke skulle dela resultatet av att optimera a och använda det på båda ställen där det används.

En optimerad version är denna, eller en med additionen inlinad. Notera att fib 10 = 55.

```
test_x y = let
    { a = THUNK (55 + y)
    } in (+) a a;
```

6.5 Call stack

I projektets tidigare fas användes substituering för att göra exempelvis funktionsapplikation. Det betyder att syntaxträdet traverseras och argumentvariablerna byts i funktionen mot vad argumenten faktiskt är. Detta är långsamt och inte särskilt realistiskt - speciellt inte om man talar om en kompilator. Kompilatorer kan istället använda call-stacks där man låter funktionsargumenten indexeras in på en stack, så att man slipper gå ner i hela syntaxträdet. Det är vanlightvis inte möjlight i kompilerad kod eftersom det inte finns någon tillgång till syntaxsträdet.

För att göra vår tolk mer realistisk byttes substitueringen till en callstack. Det gav också en hastighetsökning av tolken med ungefär 2500%.

> Notering!
> Varför finns det en till förklaring av callstack här?!

Callstacken fungerar såhär, betrakta funktionen f:

```
f a = case g a b of
    A -> .. a ..
    B -> .. a ..
```

Evaluering av f 5 med substitution består av att ersätta argumentet a mot 5 i funktionskroppen:

```
f_a = case g 5 b of
    A -> .. 5 ..
    B -> .. 5 ..
```

Med en callstack behövs först koden ett pass där f och de andra funktionerna får sina lokala variabler indexerade. Här får då f istället för a 0 för första lokala variabeln. Vi skriver <a,0> för att underlätta läsning. Denna indexgivning utförs för alla lokala variabler, och inte bara för argument, även för let-bundna samt varibler bundna via case blir indexerade.

Detta transformeringspass som ger alla lokala variabler ett index görs tolken körs, så att inget onödigt arbete behöver göras vid körningstid.

Funktionen f ser ut så här med sina variabler indexerade. Notera att g och b är fria variabler och antas således vara definierade på toppnivå. De finns alltså på heapen och inget index krävs.

```
f <a,0> = case g <a,0> b of
    A -> .. <a,0> ..
    B -> .. <a,0> ..
```

När man anropar funktionen behöver man nu bara lägga argumentet a på position 0 på stacken, och överallt där a används kan man sedan indexera in på stacken för att få ut värdet.

6.6 Optimise med call stack

Notering!
Gör semantiken svårare. Shiftup and shiftdown. Situps and pulldowns

7 Relaterade Arbeten

Vårt arbete spänner över två forskningsområden, partiell evaluering och optimering under körningstid. I litteraturen är dessa områden oftas separerade, något som vi håller'

7.1 Statiska optimeringar

Core[?]
Partiell evaluering

7.2 Optimeringar under körningstid

JIT (just in time) compilation är en metod för att dynamiskt optimera de delar av ett program som används mest under körningstid. Val av dessa punkter sker

med hjälp av statistik och heuristik gentemot vår lösning där programmeraren själv har kontroll över angreppspunkterna.

Att annotera optimeringspunkten har gjorts tidigare av Bolz C., då för en Prologvariant.[?] Vårt arbete liknar hans på många punker, men skiljer sig i bl. a. hur vi hanterar förgreningar (i vårt fall case-satser i hans if-satser). Han lägger till en callback medan vi försöker fortsätta optimeringen.

7.3 Funktionella maskiner

Stg....[?]

8 Framtida Arbeten

Extremt informellt?

En avgränsning för vårt projekt var att jobba inom ramen för en interpreter. Detta av flera anledningar, men främst för att det är lättare att prototypa nya regler för optimise. Nu när vi har tagit fram fungerande optimerings semtantik är steget till att implementera vårt språk i en kompilator inte allt för avlägset. Vi har som bekant även byt ut vår substitutions-funktion nästan överallt mot att använda en callstack, vilket är mycket mer kompilator vänligt.

Vårt core språk liknar det som både yhc och ghc arbetar med i sina mellansteg. Det hade varit ytterst intressant att utöka vårt språk så att vi skulle kunna vara backend till något större kompilator (eller runtimesystem?). Vi har alltid haft detta i åtanke och det är delvis därför vi ej har någon typechecker.

[Info bild som visar en pipe line för vart vår kompilator finns i kedjan.]

En typechecker och stöd för t ex algebric datatypes i ett frontend hade också varit trevligt då det är väldigt lätt att skriva fel i testprogramen.

Något annat som föll lite utanför ramen för vårt arbete är att bevisa korrektheten hos optimisefunktionen, för att se att en optimerad funktion gör samma sak som dess ooptimerade motsvarighet. Vi har endast använt oss av en testsvit och andra exempelprogram för att testa optimise. För att vara säker på att den gör vad som förväntas skulle det vara intressant att formalisera reglerna och bevisa att de är korrekta, exempelvis med hjälp av en bevisassistent såsom Agda.

[Sharing skulle kunna vara med här (om vi inte får det att fungera).]