1 Inledning

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

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

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 Egenskaper av STG-språket

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.

3.4 Semantik över STG

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:

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

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 let-bundna variabler, aven kallade atomer. Vi ser har att map tar tva argument och ater saledes upp tva argument fran argumentstacken. Formellt:

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

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:

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

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.

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

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
$C\ a_1\ \ldots\ a_n$	$CON(C\ a_1\ \ldots\ a_n)$	Konstruktor
$\lambda x_1\ \ldots\ x_n.e$	$FUN(x_1\ \ldots\ x_n \rightarrow e)$	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.

$$f\,;\, Arg\, a_1 : \ldots : Arg\, a_m : Stack\,;\, Heap[f \mapsto FUN(x_1 \ldots x_n \to e) \Rightarrow p\,;\, Stack\,;\, Heap[x \mapsto PAP(f\, a_1 \ldots a_m)]$$

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.

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

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

3.5 Våra utökningar

3.5.1 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.

3.5.2 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.

3.5.3 Print continuation

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.

4 Optimise

4.1 Call by Value semantik

4.2 Case Branches

4.3 Call by Name semantik

4.4 Dödkods eliminering

4.5 Välkänd case-lag

4.6 Afterburner

5 Resultat

6 Diskussion

7 Relaterade Arbeten

8 Framtida Arbeten