Sip2Sep

DAT2-projekt, Aalborg Universitet
Institut for Datalogi

Gruppe d206a
Jakob Svane Knudsen
Mikkel Larsen Pedersen
Simon Nicholas M Tinggaard
Rune Leth Wejdling

Institut for Datalogi

Titel:
Sip2Sep

Projektperiode:
DAT2: 1. februar – 29. maj 2007

Projektgruppe:
d206a

Deltagere:
Jakob Svane Knudsen
Mikkel Larsen Pedersen
Simon Nicholas M. Tinggaard
Rune Leth Wejdling

Vejleder:
Hans Hüttel

Antal Kopier::
6

Sideantal:
67

Sideantal for bilag:
23

Total sideantal:
101

Synopsis:

Denne rapport giver et bud på regler, der kan opstilles for at parallelisere sekventiel kode, og derved opnå implicit parallelitet. De to sprog **Sip** og **Sep** med hhv. implicit og eksplicit parallelitet, danner grundlag for en analyse af parallelitet baseret på en fuldstændig operationel semantik og en formel beskrivelse af parallelisering betragtet som et transistionssystem. Der implementeres en oversætter fra **Sip** til **Sep**. Slutteligt udføres forsøg med oversættelse og fortolkning af sekventielle programmer skrevet i **Sip** og deres tilsvarende parallele programmer oversat til **Sep**.

Forord

Denne rapport er udarbejdet af gruppe d206a i forbindelse med projektforløbet ved Institut for Datalogi på Aalborg Universitet i perioden 1. februar til 29. maj 2007.
Kildekoden til vores oversætter og fortolker, samt en elektronisk udgave af denne rapport kan findes på `http://www.cs.aau.dk/~runelw/dat2/`. Kildekoden er udformet som en solution i Visual Studio 2005 og der er skrevet en makefile til Mono. Det er vigtigt at læse readme-filerne for at forstå output fra fortolkeren.
Vi vil gerne takke vores vejleder Hans Hüttel for et godt samarbejde.

Aalborg d. 29. maj 2007.

Jakob Svane Knudsen	Mikkel Larsen Pedersen
Simon Nicholas M. Tinggaard	Rune Leth Wejdling

INDHOLD

Del I

Introduktion

KAPITEL

1

Indledning

Igennem flere år har hastigheden på processorer fulgt Moores lov, der siger at hastigheden på processorer bliver fordoblet hver 18. måned. Men de senere år er grænsen, ifølge [Gor07], for hvor høj en clockfrekvens man kan køre over en enkelt processor nået, da varmeudviklingen bliver for stor til, at den kan køles med almindelig køling. Tendensen går derfor imod at bygge flere kerner ind i hver processorchip og lade dem køre ved en lavere frekvens. Dette sænker både strømforbruget og nedsætter varmeudviklingen. Der findes allerede processorchips på markedet, til privat brug, med fire eller flere kerner og den nye Playstation 3 spilkonsol er udstyret med 9 processorer. Man forventer at udviklingen fortsat vil følge Moores lov, blot med antallet af kerner per processor chip, i stedet for clockfrekvens.

Dette stiller nye krav til systemudviklere, der er vant til at udvikle sekventielle programmer, da det ikke længere kun er store servere og mainframes der har flere processorer. Man mente tidligere, at det ikke kunne betale sig at udvikle parallelle programmer, medmindre det kunne løse opgaven på under den halve tid. Dette skyldtes at man blot kunne vente 18 måneder og købe en processor, der var dobbelt så hurtig. En anden årsag er at parallelle programmer typisk er mere problematiske at skrive, end sekventielle programmer, til dels fordi mennesker tænker sekventielt.

Der er gennem tiderne blevet gjort flere forsøg på at indføre sprog med det man kalder implicit parallelitet, hvor programmer skrives som om de var beregnet til sekventiel udførsel, men at specielt opstillede regler i oversættelsen, identificerer konstruktioner, der kan køre parallelt. Implicit parallelitet stiller krav til en mere generel form for parallelitet, til forskel fra diverse løsninger der kan sikre parallelitet som implementeres ad hoc. Af aktuelle sprog med implicit parallelitet har vi f.eks. Fortress og Cω.

Fortress er et nyt sprog under udvikling af Sun Microsystems. Størstedelen af sproget bliver udgivet som open source, under BSD-licensen[1].

I Fortress har man valgt at implementere både implicit og eksplicit parallelitet [ACH+07]. Implicit parallelitet er implementeret som strukturer i sproget, hvor bestanddelene bliver afviklet parallelt. For eksempel vil `for`-løkker altid blive paralleliseret, så man kan derfor ikke vide i hvilken rækkefølge iterationerne vil blive afviklet. Derudover indfører de nye konstruktioner som tupel-udtryk og '`also do`'-blokke. I tupel-udtryk bliver de enkelte elementer i tuplen evalueret i separate tråde. I '`also do`'-blokke gives mulighed for at definere, hvilke dele der kan afvikles parallelt, uden eksplicit at skulle oprette nye tråde og samle disse, når de er færdige.

Cω[2] er en videreudvikling af C# fra Microsoft, der indeholder asynkrone

[1] Udviklingen af Fortress kan følges på hjemmesiden `http://fortress.sunsource.net/`, hvor man også kan søge om at blive Sun Contributor og bidrage til udviklingen.

[2] Udviklingsstatus kan følges her: `http://research.microsoft.com/Comega/`

metodekald fra Polyphonic C# og XML-datatyper fra Xen[3]. Sproget bliver udviklet med det formål at gøre det nemmere at skrive datatunge programmer til distribuerede systemer. De asynkrone metodekald fungerer ved, at metoder erklæres som `async`, hvilket gør at de fra kalderens synspunkt blot er `void`-metoder, der returnerer med det samme. Metoderne bliver afviklet parallelt med den kaldende tråd. Fordelen ved dette er, at man ikke fra den kaldende tråd, skal oprette tråde og sikre at disse terminerer. Omvendt har man heller ikke mulighed for at se, om en tråd er termineret og modtage returværdier fra denne.

1.1 Anvendelser af parallelitet

Internettets udbredelse har bevirket, at der kan samles enorme netværk af forbundne processorer, og hvis disse kan samarbejde om at løse et problem, har man en enorm regnekraft til rådighed. Forskellige sådanne initiativer er allerede startet. Eksempler på dette er BOINC[4], jagten på stadigt større primtal[5] og analyse af proteinkæder[6]. Dette har også givet anledning til løsning af kendte matematiske problemer vha. computer, men denne form for "bevis" bliver dog diskuteret. Dette ses f.eks. ved projektet ZetaGrid[7], hvor et stort netværk af brugere vil af- eller bevise Riemann-hypotesen[8] ved at kontrollere om fundne ikke-trivielle nulpunkter ligger indenfor den såkaldte "critical line".

GRID-strukturen bruges også til mere generelle formål. Fx det skandinavisk startede NorduGrid[9], der samler clusters fra mange universiteter i et stort GRID-netværk, og derpå giver brugerne adgang til den processorkraft, de ikke selv udnytter.

Programmering af opgaver til GRID-netværk, stiller store krav til strukturen af delt data imellem de opgaver der afvikles parallelt, da kommunikation imellem enkelte knuder i et GRID-netværk kan være meget langsom. Man har derfor valgt ikke at have delt hukommelse mellem trådene, som man har, hvis de afvikles på en maskine med delt hukommelse. Delt hukommelse imellem tråde introducerer dog problemer med adgang til den delte data. Hvis man ikke strukturerer denne adgang med låse og tildeling af adgang, kan det ikke forudsiges, hvad resultatet vil blive, da man ikke kan forudsige i hvilken rækkefølge trådene bliver afviklet. Det er derfor vigtigt at vide hvilke dele af programmet, der skal have adgang til hvilke data og i hvilken rækkefølge, inden man afvikler flere dele parallelt. Denne problemstilling uddybes yderligere i afsnit 4.2.

[3] Xen er også kendt som X#.

[4] Et projekt (`http://boinc.berkeley.edu/`), der stiller regnekraft til rådighed for andre projekter, som fx SETI@home (`http://setiathome.berkeley.edu/`) der analysere radiobølger modtaget fra rummet.

[5] Fx GIMPS-projektet (`http://www.mersenne.org/`), der udnytter alle forbundne processorer til at finde stadig større primtal.

[6] Fx Folding@Home (`http://folding.stanford.edu/`), der udnytter den samlede processorkraft til beregne, hvad der sker, når proteiner foldes, for bedre at kunne forstå sygdomme, der opstår deraf.

[7] Mere information kan findes på `http://www.zetagrid.net`

[8] Mere information kand findes på `http://en.wikipedia.org/wiki/Riemann_hypothesis`

[9] Mere information kan findes på `http://www.nordugrid.org/`

1.2 Problemformulering

Vi vil i dette projekt designe et sprog med implicit parallelitet, samt tilhørende oversætter og fortolker. Implicit skal i vores tilfælde forstås på den måde, at en programmør, der skriver et program i vores sprog, ikke skal planlægge programmets afvikling over flere tråde, da sproget ikke indeholder konstruktioner til eksplicit at angive dette. Programmer skrevet i sproget, oversættes til et sprog med eksplicit parallelitet og fortolkes af en flertrådet fortolker.

En stor del af arbejdet, for os som udviklere af denne oversætter, består således i at opstille regler for hvordan et program kan paralleliseres. Oversætteren skal kunne bruge disse regler til at identificere konstruktioner i programmet der kan afvikles i parallel, og derfra konstruere et tilsvarende program, der indeholder parallelle konstruktioner.

Et mål for hvor meget et program er blevet paralleliseret i oversættelsen, vil blive søgt. Både i form af graden af parallelitet, samt et mål for længden af programmet, hvis det terminerer.

Vi rejser derfor følgende spørgsmål.

- Hvordan kan parallelitet beskrives formelt?
- Hvilke regler for parallelitet skal og kan opstilles?
- Hvilke regler og metoder for parallelisering skal og kan opstilles og hvordan kan disse bruges i en oversætter?
- Hvordan implementeres en operationel semantik i en fortolker?
- Hvilke mål for parallelitet i et program kan bruges til at sammenligne to programmer?

1.3 Afgrænsning

Vi vil i koncentrere os om analysen af dataafhængigheder mellem kommandoer, samt implementere en fortolker ud fra en operationel semantik.

Da det ikke er muligt at lave en dynamisk analyse af et program skrevet i et Turing-fuldstændigt sprog [Sip06], for alle mulige starttilstande, vil vi bygge analysen til graden af parallelitet i et program, på statisk analyse af programmets kildekode. Graden af parallelitet er derfor en approksimeret værdi.

Del II

Sprogdesign

KAPITEL

2

Sproget **Sip**

Vi vil i dette afsnit introducere sproget **Sip** [1]. Formålet med **Sip** er at give programmøren mulighed for at skrive programmer uden at skulle tænke på, hvad der kan afvikles parallelt, men samtidig opfordre programmøren til at programmere på en måde, der gør det nemmere at parallelisere.

Det primære formål med **Sip** er at det skal kunne bruges til at danne eksempler på programmer. Et designmål for **Sip** er derfor, at det skal have høj læselighed, frem for skrivelighed. Dette medfører at dele af syntaksen vil indeholde ord, der kunne forkortes, men dette er ikke gjort, da det ville sænke læseligheden. Sproget indeholder, af samme grund, ord der ville kunne udelades, som for eksempel `uses` i funktionserklæringer.

Det ville være hensigtsmæssigt hvis **Sip** var et Turing-fuldstændigt sprog. I [KAM82] vises at en meget lille delmængde af Pascal, kaldet **while**programs, er et Turing-fuldstændigt sprog. Kommandoer og udtryk i dette sprog består af konstruktionerne vist i bilag A. Sammen med opbygningsreglerne fra [KAM82, s. 22] er givet ækvivalente **Sip** -konstruktioner, konstrueret med opbygningsreglerne givet i afsnit 2.2. Alle **while**programs kan altså skrives med ækvivalente **Sip** -kommandoer, hvilket er vores argument for at **Sip** er et Turing-fuldstændigt sprog.

2.1 Uformel beskrivelse af Sip

Programmer i **Sip** består af en følge af funktionserklæringer. For at et program er gyldigt, skal en funktion ved navn `main` være erklæret. Denne funktion bliver implicit kaldt ved afvikling af programmet og skal have en tom parameter blok, samt returtypen `int`.

Der er ingen regler for hvor i implementationsrækkefølgen funktionen `main` skal være placeret. Dette giver programmøren frihed til at vælge, hvad der passer bedst til det pågældende program.

2.1.1 Navne, datatyper og erklæring af variable

Navne i **Sip** består af sekvenser af bogstaver og tal. Navne må dog ikke starte med et tal. For eksempel er navnet `kuppel42` lovligt, men ikke navnet `42kupler`.

Variable kan have en af følgende typer:

[1] **Sep** med **I**mplicit **P**arallelitet

`int`	:	Kan antage værdier fra mængden $\mathbb{Z}$.
`bool`	:	Kan antage værdier fra mængden {`true`, `false`}.
`string`	:	En tekstreng er en sekvens af elementer fra mængden {`a`, .., `Z`, `0`, .., `9`, ' '} afgrænses af ", hvor ' ' betegner et mellemrum.
`int list (` v `)`	:	Benævner lister bestående af v `int` værdier.
`bool list (` v `)`	:	Benævner lister bestående af v `bool` værdier.
`string list (` v `)`	:	Benævner lister bestående af v `string` værdier.

Følgende eksempel er en erklæring af en variabel ved navn `a`, som kan antage heltalsværdier, og en variabel ved navn `b`, der er en liste bestående af 10 sandhedsværdier.

```
a : int ;
b : bool list(10) ;
```

Som man kan se er der ingen initialisering af variablene. Vi introducerer derfor en standardværdi for alle typer. Denne standardværdi er givet ved funktionen *default* som er defineret i afsnit 2.3.1.

Sip har en flad blokstruktur, hvilket vil sige at der er et globalt scope som indeholder flere lokale scopes, men lokale scopes kan ikke indeholde scopes. Alle funktioner eksisterer i det globale scope, og alle variable eksisterer i de lokale scopes. Man kan altså ikke lave globale variable i **Sip** .

2.1.2 Udtryk

De aritmetiske udtryk i **Sip** svarer til deres almindeligt kendte modstykker i matematikken. Der er tale om addition, subtraktion, multiplikation og modulo. Boolske udtryk kan skrives med operatorene `and`, `or`, `<`, `>`, `=`. Med `=`, `<` og `>` operatorene kan man sammenligne to aritmetiske udtryk, men ikke to boolske udtryk. I udtryk kan der sættes balancerede parenteser, dog skal der findes en venstreparentes til hver højreparentes.

Et udtryk kan også være et opslag af en variabel. Hvis man vil have variablens værdi skriver man bare dens navn, men hvis variablen `a` har en listetype og man vil have værdien af det n'te element skriver man `a [` n `]`.

Bemærk at lister er 0-indekserede sådan at det første element i en liste `a` med n elementer er `a [ 0 ]` og det sidste element er `a [` n `- 1 ]`.

Alle funktioner i **Sip** returnere en værdi, da det er meningen at man skal bruge dem i udtryk. Et funktionskald er et funktionsnavn efterfulgt af en sekvens af udtryk som udgør de aktuelle parametre. Disse udtryk skrives i en kommasepareret liste afgrænset af paranteser. For eksempelt skrives et funktionskald til en funktion ved navn `f` med de aktuelle parametre `2` og `a` som `f(2,a)`.

Det følgende eksempel viser et boolsk udtryk i **Sip**

```
(( a + b ) < f(5)) and ( 10 = g(a,b,5) )
```

2.1.3 Kommandoer

Sip er et imperativt sprog og den væsentligste kommando er derfor tilskrivning af en værdi til en variabel. En tilskriving foregår med konstruktionen N `:=` U `;` hvor N er en variabel eller en henvisning til et element i en liste som tilskrives

værdien af udtrykket U. En tilskriving til det første element i en liste `a` skrives `a[0] :=` U `;`.

Konstrolstrukturer i **Sip** skrives `if` U `then` K_1 `else` K_2, hvor U er et udtryk som forventes at evaluere til en sandhedsværdi, som afgør hvilken af de to kommandoer K_1 og K_2, der skal udføres. Man kan ikke skrive `if` U `then` K, altså en konstrolstruktur uden `else`. Man kan i stedet bruge den specielle kommando `skip;`, som ikke gør noget.

Der findes to løkkekonstruktioner i **Sip** . `while`-løkker skrives på formen `while` U `do` K. Som betyder at kommandoen K udføres så længe udtrykket U evalueres til `true`.

`for`-løkker kan skrives `for` N `from` U_1 `to` U_2 `do` K. Denne konstruktion betyder at K skal udføres på hinanden følgende gange, hvor variablen N første gang har værdien af udtrykket U_1. De efterfølgende gange har N stigende værdier, indtil den rammer værdien af udtrykket U_2.

De to løkker er eksemplificerede i følgende **Sip** kommando.

```
for i from 3 to 3 + 4 do
  x := x + i;

while (i < 5) do
{
  a := f ( i );
  i := i + 1;
}
```

2.1.4 Erklæring af funktioner

En funktion erklæreres med følgende: dens returtype, typerne af dens formelle parametre, en kommandoblok som skal udføres ved kald af funktionen og de variable som bruges i dennes kommandoblok. En funktion ved navn `sum`, som returnerer summen af to heltal kan skrives, som i følgende eksempel:

```
function sum ( a:int, b:int ) : int
does
{
  sum := a + b;
}
```

Bemærk at returværdien angives ved at tildele en værdi til en variabel med samme navn som metoden, kendt som alias-variablen. Den specielle funktion `main`, som skal erklæres, kan for eksempel erklæres som i følgende eksempel, omend dette program kun tjener et illustrativt formål.

```
function main() : int
main := 42;
```

2.2 Abstrakt syntaks

Ved definitionen af den operative semantik i afsnit 2.3, typereglerne i afsnit 2.4 og paralleliseringen i afsnit 4, får vi brug for en abstrakt syntaks for **Sip** .

Sip deles op i en række syntaktiske kategorier som er vist i tabel 2.1. Konstruktionsregler for udvalgte konstruktioner er vist i tabel 2.2.

Bemærk at den abstrakte syntaks læner sig en smule mere op ad den konkrete syntaks end hvad umiddelbart virker nødvendigt. For eksempel kunne den sidste af disse to regler for funktionserklæringer

$$\texttt{function}\ N\ (\ E_{v1}, .., E_{vn}\ ) : T\ \texttt{uses}\ E_v\ \texttt{does}\ K$$
$$\texttt{function}\ N\ (\ E_{v1}, .., E_{vn}\ ) : T\ \texttt{does}\ K$$

spares væk ved gøre det muligt for E_v at være tom, ligesom semikolonet bag kommandoer som N `:=` U `;` kunne fjernes og sammensætning af kommandoer kunne laves som K `;` K istedet. Vi har dog istedet valgt at lade den abstrakte syntaks være en lettere læst udgave af den konkrete syntaks, sådan at sammenhængen mellem den operationelle semantik og implementationen af denne fremstår tydeligere.

2.3 Semantik

I dette afsnit beskrives semantikken for **Sip** . Semantikken tager udgangspunkt i en model af environments og køretidsstakke som er defineret i afsnit 2.3.1. Denne model benytter sig i høj grad af partielle funktioner, og vi får flere gange brug for at kunne opdatere disse. Derfor introducerer vi en gang for alle en notation til dette.

Definition 1.1 (Opdateringssyntaks for partielle funktioner)**.**
Lad f_1, f_2 være partielle funktioner.
Den opdaterede partielle funktion $f_3 = f_1[f_2]$ er da givet ved

$$f_3(a) = \begin{cases} f_2(a) \text{ hvis } f_2(a) \text{ er defineret} \\ f_1(a) \text{ ellers.} \end{cases}$$

For at kunne definere specifikke partielle funktioner på en behagelig måde introduceres også en listeforskrift med dette formål.

Definition 1.2 (Listeforskrift for partielle funktioner)**.**
En partiel funktion $f : A \rightharpoonup B$ hvor $Dm(f)$ er endelig og har n elementer, kan skrives som

$f = [a_1 \mapsto b_1, .., a_n \mapsto b_n]$

Hvor $a_i \in A$, $b_i \in B$ og $f(a_i) = b_i$ for alle i fra 1 til n.

Vi er nu klar til at beskrive vores model.

2.3.1 Environment-store-modellen

Lageret Store

Lageret er en mængde bindinger mellem adresser og værdier. Et øjebliksbillede af disse bindinger betragtes som en partiel funktion fra funktionsrummet $\textbf{Store} = \mathbb{N}_0 \rightharpoonup \mathbb{V}$, hvor $\mathbb{V} = \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\} \cup \mathbb{S}$, hvor $\mathbb{S}$ er mængden af alle tekststrenge. Et øjebliksbillede af lageret er da en funktion $sto \in \textbf{Store}$ hvor $sto(a) = v$ hvis adressen a er bundet til værdien v.

Funktionen $default : \mathbf{T} \rightarrow \mathbb{V}$ introduceres. Denne giver os en værdi for enhver type.

P	:	Programmer. Et program består af en eller flere funktionserklæringer. Enkelte programmer betegnes med metavariablen P.
K	:	Kommandoer. Enkelte kommandoer betegnes med metavariablen K.
U	:	Udtryk. Disse kan være aritmetiske udtryk $\mathbf{U}_a$, boolske udtryk $\mathbf{U}_b$, strengudtryk $\mathbf{U}_s$. Altså er $\mathbf{U} = \mathbf{U}_a \cup \mathbf{U}_b \cup \mathbf{U}_s$. Enkelte udtryk betegnes med metavariablene U, U_a eller U_s.
E	:	Erklæringer, kan være erklæringer af variable $\mathbf{E}_v$ eller erklæringer af funktioner $\mathbf{E}_f$. Enkelte erklæringer betegnes med metavariablene E_v eller E_f.
N	:	Navne er $\mathbf{N} = \{\mathtt{a},..,\mathtt{z},\mathtt{A},..,\mathtt{Z}\}^+ (\{\mathtt{0},..,\mathtt{9},\mathtt{a},..,\mathtt{z},\mathtt{A},..,\mathtt{Z}\})^*$. Enkelte navne betegnes med metavariablen N.
T	:	Typenavne. Enkelte typenavne betegnes med metavariablen T.
L	:	Literale, kan være numeriske literale $\mathbf{L}_l = \{-,\epsilon\}\{\mathtt{0},..,\mathtt{9}\}^+$, boolske literale $\mathbf{L}_b = \{\mathtt{true},\mathtt{false}\}$ eller strengeliterale $\mathbf{L}_s$ som er mængden af ord over alfabetet $\{\mathtt{A},..,\mathtt{Z},\mathtt{a},..,\mathtt{z},' '\} \cup \{\mathtt{0},..,\mathtt{9}\}$, hvor $' '$ betegner et mellemrum. Enkelte literale betegnes med metavariablene l, l_l, l_b eller l_s.

Tabel 2.1: Syntaktiske kategorier

P	::=	E_f
E_v	::=	$E_v\ E_v \mid N\ \mathtt{:}\ T\ \mathtt{;}$
E_f	::=	$E_f\ E_f \mid$ $\mathtt{function}\ N\ \mathtt{(}\ E_{v1},..,E_{vn}\ \mathtt{)}\ \mathtt{:}\ T\ \mathtt{uses}\ E_v\ \mathtt{does}\ K \mid$ $\mathtt{function}\ N\ \mathtt{(}\ E_{v1},..,E_{vn}\ \mathtt{)}\ \mathtt{:}\ T\ \mathtt{does}\ K$
T	::=	$\mathtt{int} \mid \mathtt{bool} \mid \mathtt{string} \mid$ $\mathtt{int\ list\ (}\ L_a\ \mathtt{)} \mid \mathtt{bool\ list\ (}\ L_a\ \mathtt{)} \mid \mathtt{string\ list\ (}\ L_a\ \mathtt{)}$
K	::=	$K\ K \mid N\ \mathtt{:=}\ U\ \mathtt{;} \mid N\ \mathtt{[}\ U_a\ \mathtt{]}\ \mathtt{:=}\ U\ \mathtt{;} \mid \mathtt{\{}\ K\ \mathtt{\}} \mid \mathtt{skip\ ;} \mid$ $\mathtt{if}\ U_b\ \mathtt{then}\ K\ \mathtt{else}\ K \mid$ $\mathtt{for}\ N\ \mathtt{from}\ U_a\ \mathtt{to}\ U_a\ \mathtt{do}\ K \mid$ $\mathtt{while}\ U_b\ \mathtt{do}\ K$
U_a	::=	$U_a\ \mathtt{+}\ U_a \mid U_a\ \mathtt{-}\ U_a \mid U_a\ \mathtt{*}\ U_a \mid U_a\ \mathtt{/}\ U_a \mid U_a\ \mathtt{mod}\ U_a \mid \mathtt{(}\ U_a\ \mathtt{)} \mid$ $N\ \mathtt{(}\ U,..,U\ \mathtt{)} \mid N \mid N\ \mathtt{[}\ U_a\ \mathtt{]} \mid L_a$
U_b	::=	$U_a\ \mathtt{<}\ U_a \mid U_a\ \mathtt{>}\ U_a \mid U_a\ \mathtt{=}\ U_a \mid U_b\ \mathtt{and}\ U_b \mid U_b\ \mathtt{or}\ U_b \mid \mathtt{(}\ U_b\ \mathtt{)} \mid$ $N\ \mathtt{(}\ U,..,U\ \mathtt{)} \mid N \mid N\ \mathtt{[}\ U_a\ \mathtt{]} \mid L_b$
U_s	::=	$N\ \mathtt{(}\ U,..,U\ \mathtt{)} \mid N \mid N\ \mathtt{[}\ U_a\ \mathtt{]} \mid L_s$

Tabel 2.2: Konstruktionsregler for udvalgte konstruktioner i Sip

$$
\begin{array}{ll}
default(\texttt{int}) & = 0 \\
default(\texttt{bool}) & = \texttt{false} \\
default(\texttt{string}) & = \texttt{""} \\
default(T\ \texttt{list (} n \texttt{)}) & = (default(T)_1, .., default(T)_n)
\end{array}
$$

Bindinger af variabelnavne $\mathbf{Env}_v$

Bindinger mellem variabelnavne $\mathbf{N}$ og lokationer $\mathbb{N}_0$ beskrives af et variabelenvironment.

Øjebliksbilleder af dette betragtes som en partiel funktion fra funktionsrummet $\mathbf{Env}_v = \mathbf{N} \cup \{next\} \rightharpoonup \mathbb{N}_0$, hvor $next$ er et specielt element der altid er bundet til den næste frie plads i lageret. $next$ opdateres løbende i semantikreglerne.

Bindinger af funktionsnavne $\mathbf{Env}_f$

Funktionsnavne er bundet til 4-tupler $\mathbf{K} \times \mathbf{E}_v \times \mathbf{Env}_v \times \mathbf{Env}_f$. Øjebliksbilleder af disse bindinger er funktioner fra funktionsrummet $\mathbf{Env}_f = \mathbf{N} \rightharpoonup \mathbf{K} \times \mathbf{E}_v \times \mathbf{Env}_v \times \mathbf{Env}_f$.

Køretidsstakke Envl

Idet vi beskriver en smallstep-semantik for kommandoer i Sip og vi har statisk binding af variable og funktioner, bliver det nødvendigt at kunne gemme midlertidige bindinger. Dette gøres i en køretidsstak, hvor elementerne i stakken er tupler fra $\mathbf{Env}_v \times \mathbf{Env}_f$. Den øverste tupel i den aktuelle stak indeholder de bindinger, som er gældende.

Øjebliksbilleder af køretidsstakken er stakke ud af mængden $\mathbf{Envl} = (\mathbf{Env}_v \times \mathbf{Env}_f)^*$, som er mængden af alle lister bestående af tupler $\mathbf{Env}_v \times \mathbf{Env}_f$.

Vi benytter notationen $envl = (env_v, env_f) : envl'$ til at beskrive at et øjebliksbillede $envl$ er stakken hvor (env_v, env_f) er øverst og resten af stakken er $envl'$.

Når vi møder et nyt lokalt scope tilføjer vi en tupel (env_v, env_f) øverst i den aktuelle stak $envl$ så denne bliver $envl' = (env_v, env_f) : envl$. Når vi forlader dette scope igen, fjerner vi det øverste element fra den aktuelle stak $envl' = (env_v, env_f) : envl$ så $envl$ bliver den aktuelle stak. Sådan modellerer vi de lokale scopes som er beskrevet i afsnit 2.1.1.

2.3.2 Erklæringer af variable

Den operationelle semantik for erklæring af variable $\mathbf{E}_v$ er givet ved et transitionssystem $(\Gamma_v, \Rightarrow_v, \mathrm{T}_v)$ hvor

$$
\begin{array}{l}
\Gamma_v = \mathbf{E}_v \times \mathbf{Store} \times \mathbf{Envl} \cup \mathbf{Store} \times \mathbf{Envl} \\
\mathrm{T}_v = \mathbf{Store} \times \mathbf{Envl}
\end{array}
$$

Transitionsreglerne for $\Rightarrow_v$ er givet på en af følgende former, idet en erklæring af en variabel både ændrer lageret og bindingerne mellem variabelnavne og adresser i lageret.

$$\langle E_v, sto, envl\rangle \Rightarrow_v \langle E'_v, sto', envl'\rangle$$
$$\langle E_v, sto, envl\rangle \Rightarrow_v \langle sto', envl'\rangle$$

En erklæring af en variabel ved navn N består i at oprette en binding mellem N og en ny adresse a i lageret. Adressen a er i vores model den adresse, som det specielle navn *next* er bundet til. Efter at det aktuelle variabelnavn er bundet til a, skal det næste vi støder på selvfølgelig ikke bindes til dette. Ved at opdatere modellen sådan at *next* er bundet til $a+1$, sørger vi for at variable ikke overskriver hinanden.

Adressen a bindes desuden til en standardværdi for den aktuelle type ved hjælp af $default(T)$. Alt dette fremgår af reglen [var].

[var] $\langle N \ : \ T, sto, envl\rangle \Rightarrow_v \langle sto[a \mapsto default(T)], envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $a = env_V(next)$
og $envl' = (env_v[N \mapsto a][next \mapsto a+1], env_f) : envl''$

Ved erklæring af en liste reserveres en sammenhængende følge af adresser.

[list-2] $\langle N \ : \ T \ \texttt{list} \ (v), sto, envl\rangle \Rightarrow_v \langle sto', envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env'_v = env_v[N \mapsto a][next \mapsto a+v]$
og $sto' = sto[a \mapsto default(T)][a+1 \mapsto default(T)]...$
$[a+v \mapsto default(T)]$
og $envl' = (env'_v, env_f) : envl''$

Resten af transitionsreglerne for $\Rightarrow_v$ er givet i bilag C.3.

2.3.3 Erklæringer af funktioner

Den operationelle semantik for erklæringer af funktioner $\mathbf{E}_f$ er givet ved et transitionssystem $(\Gamma_f, \Rightarrow_f, \mathrm{T}_f)$ hvor

$$\Gamma_f = (\mathbf{E}_f \times \mathbf{Store} \times \mathbf{Envl}) \cup (\mathbf{Store} \times \mathbf{Envl})$$
$$\mathrm{T}_f = \mathbf{Store} \times \mathbf{Envl}$$

Transitionerne er givet ved reglerne opstillet på en af følgende former, idet funktionserklæringer udelukkende ændrer funktionsbindingerne i *envl*.

$$\langle E_f, sto, envl\rangle \Rightarrow_e \langle sto, envl'\rangle$$
$$\langle E_f, sto, envl\rangle \Rightarrow_e \langle E'_f, sto, envl'\rangle$$

Vi bruger her hjælpefunktionen $upd : \mathbf{Env}_f \Rightarrow \mathbf{Env}_f$ til at opdatere øjebliksbilleder af funktionsbindingerne. Intuitivt forstås denne funktion sådan at, den opdaterer funktionsbindingerne i et øjebliksbillede, sådan at alle funktioner kender til hinanden. Den er defineret som

$$\begin{aligned} upd(env_f) = env'_f \quad & \text{hvor } env_f(n) = (K, E_v, env_v, env''_f) \\ & \text{medfører at } env'_f(n) = (K, E_v, env_v, env'_f) \end{aligned}$$

Den spændende transition i denne forbindelse er selvfølgelig den regel [erk], som beskriver erklæringen af en enkelt funktion.

Konstruktionen `function` $N_f(N_1 : T_1, .., N_n : T_n)$ `uses` E_v `does` K betyder at navnet N_f skal bindes til funktionen beskrevet ved tuplen $(K, E_v, (N_1, .., N_n), env_v, env_f)$. Dette beskrives som denne regel.

$$\begin{aligned} [\text{erk}] \quad & \langle \texttt{function } N_f(N_1 : T_1, .., N_n : T_n) \texttt{ uses } E_v \texttt{ does } K, sto, envl \rangle \\ & \quad \Rightarrow_{ef} \langle sto, envl' \rangle \\ & \text{hvor } envl = (env_v, env_f) : envl'' \\ & \text{og } env'_f = env_f[N_f \mapsto (K, (N_1, .., N_n), env_v, env_f] \\ & \text{og } envl' = (env_v, upd(env'_f)) : envl'' \end{aligned}$$

Bemærk at $upd(env'_f)$ også opdaterer den tupel som N_f er bundet til, og at env_v altid vil være tom, idet der ikke erklæres variable i globalt scope, hvor funktionserklæringerne finder sted.

Resten af transitionreglerne for $\Rightarrow_f$ er givet i bilag C.1.

2.3.4 Aritmetiske udtryk

Den operationelle semantik for arimetiske udtryk $\mathbf{U}_a$ er givet ved et transitionssystem $(\Gamma_a, \Rightarrow_a, \mathrm{T}_a)$, hvor

$$\begin{aligned} \Gamma_a &= (\mathbf{U}_a \cup \mathbf{U}_{pseudo} \cup \mathbb{V}) \times \mathbf{Store} \times \mathbf{Envl} \\ \mathrm{T}_a &= \mathbb{V} \times \mathbf{Store} \times \mathbf{Envl} \end{aligned}$$

Bemærk at vi her introducerer pseudoudtryk $\mathbf{U}_{pseudo}$. Vi benævner enkelte pseudoudtryk ved metavariablen U_{pseudo}. Pseudoudtryk er defineret ved følgende opbygningsregler.

$P_{pseudo} \quad ::= \quad \|U\| \mid \|E_v\| \mid \|K\|$

Transitionsreglerne for $\Rightarrow_a$ er givet på en af af følgende former. Selvom evalueringen af udtryk ikke medfører sideeffekter som kunne ændre på lageret eller bindingerne, så vil et funktionskald medføre midlertidige ændringer af stakken *envl* og lageret *sto*.

$$\begin{aligned} \langle U_a, sto, envl \rangle &\Rightarrow_a \langle U'_a, sto', envl' \rangle \\ \langle U_a, sto, envl \rangle &\Rightarrow_a \langle v, sto', envl' \rangle \end{aligned}$$

Udvalgte transitionsregler

Idet aritmetiske udtryk altid opererer på heltal, udfører vi heltalsdivision, som det fremgår af reglen [div-3] hvor det fremgår at resultatet af divisionen rundes ned til nærmeste heltal.

$$\text{[div-3]} \quad \langle v_1 \ / \ v_2, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle \quad \text{hvor } v = \left\lfloor \tfrac{v_1}{v_2} \right\rfloor$$

Literale oversættes til værdier med hjælpefunktionen $\mathcal{N} : \mathbf{L}_a \to \mathbb{Z}$, som f.eks. for literalen `42` giver os værdien 42. Dette viser sig i reglen [lit].

$$\text{[lit]} \quad \langle L_a, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle \quad \text{hvor } \mathcal{N}(L_a) = v$$

Opslag i lageret sker gennem de variabelnavne som er bundet til addresser i det øverste variabelenvironment i stakken, som angivet i reglen [var].

$$\text{[var]} \quad \langle N, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle \quad \text{hvor } envl = (env_v, env_f) : envl' \quad \text{og } sto(env_V(N)) = v$$

I tilfælde af konstruktionen N `[` v `]` lægges værdien v til adressen bundet til N, som i reglen [opslag-2].

$$\text{[opslag-2]} \quad \langle N[v_1], sto, envl\rangle \Rightarrow_a \langle v_2, sto, envl\rangle \quad \text{hvor } envl = (env_v, env_f) : envl' \quad \text{og } sto(env_v(N) + v_1) = v_2$$

I reglerne for funktionskald, [kald-1] til [kald-8], benyttes en speciel notation.

Idet en funktion evalueres opstår der pseudoudtryk, -kommandoer og -erklæringer som evalueres i et udtryk. Disse pseudokonstruktioner betegnes henholdsvis $\|U\|$, $\|K\|$ og $\|E_v\|$.

I reglen [kald-1] introduceres disse pseudokonstruktioner i udtrykket. De indsatte udtryk er de aktuelle parametre til funktionen, og erklæringerne og kommandoen er bundet til funktionsnavnet N i env_f.

$$\text{[kald-1]} \quad \langle N(U_{p_1}, .., U_{p_n}), sto, envl\rangle \Rightarrow_a \langle \|U_{p_1}\|; ..; \|U_{p_n}\|; \|E_v\|; \|K_N\|, sto, envl\rangle \quad \text{hvor } envl = (env_v, env_f) : envl'' \quad \text{og } env_f(N) = (K_N, E_v, (p_1, .., p_n), env'_v, env'_f)$$

Efter at alle udtryk er evalueret til værdier tildeles de aktuelle parametre plads i lageret på nye addresser som bindes til navnene på de formelle parametre.

En variabel til funktionens returværdi bindes ligeledes til en ny adresse i lageret. Toppen af stakken er nu (env'_v, env'_f) hvor env'_v er det variabelenvironment som er bundet til N, modificeret med førnævnte bindinger, og env'_f er det funktionsenvironment som er bundet til N.

Alt dette sker i reglen [kald-4].

$$
\begin{array}{ll}
\text{[kald-4]} & \langle \|v_{p_1}\|; ..; \|v_{p_n}\|; \|E_v\|; \|K_N\|, sto, envl\rangle \Rightarrow_a \langle \|E_v\|; \|K_N\|, sto', envl'\rangle \\
& \text{hvor } envl = (env_v, env_f) : envl'' \\
& \text{og } env_f(N) = (K, E_v, env'_v, env'_f) \\
& \text{og } envl' = (env'_v, env'_f) : envl \\
& \text{og } a = env_v(next) \\
& \text{og } env'_v = env_v[p_1 \mapsto a + 0] \ldots \\
& \qquad\qquad [p_n \mapsto a + n - 1][N \mapsto a + n][next \mapsto a + n + 1] \\
& \text{og } sto' = sto[a + 0 \mapsto v_{p_1}] \ldots [a + n - 1 \mapsto v_{p_n}]
\end{array}
$$

I reglerne [kald-5] og [kald-6] evalueres variabelerklæringerne bundet til N, så der oprettes nye variable til brug under evalueringen af kommandoen K_N som evalueres i reglerne [kald-7] og [kald-8].

Efter evalueringen af K_N bruges værdien på den addresse i lageret, som er bundet til N i det aktuelle variabelenvironment, som funktionskaldets værdi. Den øverste tupel på stakken smides væk, idet den repræsenterer de midlertidige bindinger som blev brugt under evalueringen af funktionskaldet, og en eventuel næste kommando skal evalueres med de bindinger som eksisterede inden funktionskaldet.

$$
\begin{array}{ll}
\text{[kald-8]} & \dfrac{\langle K_N, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle}{\langle \|K_N\|, sto, envl\rangle \Rightarrow_a \langle v, sto', envl'\rangle} \\
& \text{hvor } envl = (env_v, env_f) : envl' \\
& \text{og } v = sto'(env_v(N))
\end{array}
$$

Resten af transitionsreglerne for $\Rightarrow_a$ er givet i bilag C.4 sammen med de få som er gengivet i dette afsnit.

2.3.5 Boolske udtryk

Den operationelle semantik for boolske udtryk $\mathbf{U}_b$ er givet ved et transitionssystem $(\Gamma_b, \Rightarrow_b, \mathrm{T}_b)$ hvor

$$
\begin{aligned}
\Gamma_b &= (\mathbf{U}_b \cup \mathbf{U}_{pseudo} \cup \mathbb{V}) \times \mathbf{Store} \times \mathbf{Envl} \\
\mathrm{T}_b &= \mathbb{V} \times \mathbf{Store} \times \mathbf{Envl}
\end{aligned}
$$

Vi bruger igen $\mathbf{U}_{pseudo}$ som defineret i afsnit 2.3.4. Idet funktionskald og håndtering af variable og lister fungerer helt ligesom de tilsvarende konstruktioner i de arimetiske udtryk.

Transitionsreglerne for $\Rightarrow_b$ er givet på en af følgende former, idet boolske udtryk på lige fod med aritmetiske udtryk kan indføre midlertidige bindinger ved funktionskald, og derved ændre lageret og stakken.

$$\langle U_b, sto, envl\rangle \Rightarrow_b \langle U_b', sto', envl'\rangle$$
$$\langle U_b, sto, envl\rangle \Rightarrow_b \langle v, sto', envl'\rangle$$

Transitionsreglerne for $\Rightarrow_b$ er givet i bilag C.5.

2.3.6 Kommandoer

Den operationelle semantik for kommandoer **K** er givet ved et transitionssystem $(\Gamma_k, \Rightarrow_K, \mathrm{T}_k)$ hvor

$$\Gamma_k = \mathbf{K} \times \mathbf{Store} \times \mathbf{Envl}$$
$$\mathrm{T}_k = \mathbf{Store} \times \mathbf{Envl}$$

Transitionerne er givet ved reglerne opstillet på en af følgende former, idet kommandoer kun ændrer på indeholdet af *sto*.

$$\langle K, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle$$
$$\langle K, sto, envl\rangle \Rightarrow_k \langle K', sto', envl\rangle$$

Udvalgte transitionsregler

Rækkefølgen i en sekvens af kommandoer som skal udføres er dikteret i de to regler [sekvens-1] og [sekvens-2], hvor konstruktionen $K_1\ K_2$ betyder at K_1 skal evalueres før K_2.

$$\text{[sekvens-1]}\quad \frac{\langle K_1, sto, envl\rangle \Rightarrow_k \langle K_1', sto', envl\rangle}{\langle K_1\ K_2, sto, envl\rangle \Rightarrow_k \langle K_1'\ K_2, sto', envl\rangle}$$

$$\text{[sekvens-2]}\quad \frac{\langle K_1, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle}{\langle K_1\ K_2, sto, envl\rangle \Rightarrow_k \langle K_2, sto', envl\rangle}$$

Ændringerne i *sto* sker i kraft af tildeling af værdier til variable, som i regelen [tildel-2] hvor værdien v i *sto* bindes til adressen variabelnavnet N er bundet til i env_v.

$$\text{[tildel-2]}\quad \langle N\ \texttt{:=}\ v\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle sto[a \mapsto v], envl\rangle$$
$$\text{hvor } envl = (env_V, env_F) : envl'$$
$$\text{og } a = env_V(N)$$

Tildelinger af værdier til elementer i lister fungerer næsten ligesådan, men den tildelede adresse er summen af adressen bundet til N og det indeks v_1 som er givet.

$$\text{[tildel-5]}\quad \frac{\langle U, sto, envl\rangle \Rightarrow_u \langle U', sto, envl\rangle}{\langle N\ \texttt{[}\ v_1\ \texttt{]}\ \texttt{:=}\ v_2\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle sto[a \mapsto v], envl\rangle}$$
, hvor $envl = (env_V, env_F) : envl'$
og $a = env_V(N) + v_1$

Der er to løkkekonstruktioner i **Sip** . `while`-løkker er rekursivt defineret i reglen [while].

$$\text{[while]}\quad \langle \texttt{while}\ U_b\ \texttt{do}\ K, sto, envl\rangle \Rightarrow_k \langle \texttt{if}\ U_b\ \texttt{then}\ \texttt{\{}\ K\ \texttt{while}\ U_b\ \texttt{do}\ K\ \texttt{\}}\ \texttt{else skip}, sto, envl\rangle$$

Sip indeholder også `for`-løkker. Dette er dog udelukkende en konstruktion, som skal gøre programmere lettere at læse og skrive. Den tilføjer ikke ny funktionalitet til **Sip** , hvilket tydeligt fremgår af transitionsreglen [for], hvor den skiftes ud med en ækvivalent kommando.

$$\text{[for]}\quad \langle \texttt{for}\ N\ \texttt{from}\ U_{a1}\ \texttt{to}\ U_{a2}\ \texttt{do}\ K, sto, envl\rangle \Rightarrow_k \langle N\ \texttt{:=}\ U_{a1}\ \texttt{;}\ \texttt{while}\ N < U_{a2}\ \texttt{do}\ \texttt{\{}\ K\ N\ \texttt{:=}\ N+1\ \texttt{\}}, sto, envl\rangle$$

Resten af transitionreglerne for $\Rightarrow_k$ er givet i bilag C.2.

2.4 Typeregler

Vi vil i dette afsnit beskrive typereglerne for **Sip** . Disse regler beskriver hvad man må og ikke må når man sammensætter typer med operatorer.

Alle konstruktioner i **Sip** siges at have en type fra mængden $\mathbf{T} \cup \{\checkmark\} \cup (\mathbf{T}^* \times \mathbf{T})$, hvor $\mathbf{T}$ er mængden af alle lister bestående af elementer fra $\mathbf{T}$, og typen $\checkmark$ angiver at en konstruktion er "veltypet".

En type fra mængden $\mathbf{T}^* \times \mathbf{T}$, angives med notationen $(T_1, .., T_n) \rightarrow T$, og bruges til at angive typen af en funktion, hvor $T_1, .., T_n$ er typerne af de formelle parametre og T typen af funktionens returværdi.

Vi definerer type-environmentet som en partiel funktion

$$env_t : \mathbf{N} \cup \mathbf{L} \cup \mathbf{P} \cup \mathbf{E} \cup \mathbf{U} \cup \mathbf{K} \rightharpoonup \mathbf{T} \cup \mathbf{Env}_t$$

sådan at $env_t(N) = T$ hvis navnet N har typen T. Typereglerne kan nu gives med følgende notation

$$\text{[navn]}\quad \frac{præmisser}{env_t \vdash \langle a\rangle : T}$$

Hvor $a \in \mathbf{N} \cup \mathbf{L} \cup \mathbf{P} \cup \mathbf{E} \cup \mathbf{U} \cup \mathbf{K}$[2] og $env_t(a) = T$ hvis alle reglerne i *præmisser* gælder. Bemærk at enkelte konstruktioners typer er defineret med notationen $env_t(K) = T$, hvor K er en konstruktionen som har typen T.

Vi er desuden nødt til at udvide opdateringssyntaksen fra definition 1.1 i afsnit 2.3, idet vi gerne vil kunne opdatere typeenvironments i forhold til konstruktionerne E_v og E_f. Disse ændringer defineres således for erklæringer af variable

$$env_t[N \ : \ T \ ;] = env_t[N \mapsto T]$$

$$env_t[E_{v1} \ E_{v2}] = env_t[E_{v1}][E_{v2}]$$

og således for erklæringer af funktioner

$$env_t[\texttt{function } N \texttt{ (} N_1 \texttt{ : } T_1, .., N_n \texttt{ : } T_n \texttt{) : } T \texttt{ uses } E_v \texttt{ does } K] = env_t[N \mapsto (T_1, .., T_n) \to T]$$

$$env_t[\texttt{function } N \texttt{ (} N_1 \texttt{ : } T_1, .., N_n \texttt{ : } T_n \texttt{) : } T \texttt{ does } K] = env_t[N \mapsto (T_1, .., T_n) \to T]$$

$$env_t[E_{f1} \ E_{f2}] = env_t[E_{f1}][E_{f2}]$$

2.4.1 Udvalgte eksempler på typeregler

Bemærk at der ikke er typeregler for erklæring af variable. Idet der ikke indgår kommandoer eller udtryk kan der ikke gå noget galt. Derfor typecheckes erklæringer af variable aldrig. Erklæringer af variable er altså altid "veltypede".

Hovedreglen er at en konstruktion er veltypet hvis dens umiddelbare bestanddele er veltypede. I nogle tilfælde er der dog yderligere krav. Udvalgte typeregler er gengivet i de følgende afsnit, men de fuldstændige sæt af typeregler findes i bilag D.

Erklæring af funktioner

En funktionserklæring er veltypet relativt til et typeenvironment env_t hvis den tilknyttede kommando er veltypet relativt til evn_t ændret i forhold til de tilknyttede erklæringer af variable. Dette udmønter sig i typereglen [erk-uses]. Der findes også en typeregel for erklæringer uden de tilknyttede erklæringer af variable [erk-nouses], men da den er meget lig [erk-uses] er den ikke gengivet her.

$$\text{[erk-uses]} \quad \frac{env_t[E_v] \vdash \langle K \rangle : \checkmark}{env_t \vdash \langle \texttt{function } N \texttt{ (} N_1 \texttt{ : } T_1, .., N_n \texttt{ : } T_n \texttt{) : } T \texttt{ uses } E_v \texttt{ does } K \rangle : \checkmark}$$

[2] Eller, hvis man er mere munter, $a \in \mathbf{K} \cup \mathbf{U} \cup \mathbf{P} \cup \mathbf{L} \cup \mathbf{E} \cup \mathbf{N}$

Typereglen [sekvens] dikterer at en sekvens af funktionserklæringer $E_{f1}\ E_{f2}$, er veltypet relativt til et typeenvironment env_t hvis de to sider er veltypede relativt til et env_t hvor de to siders ændringer er tilføjet.

$$\text{[sekvens]}\quad \frac{env_t[E_{f1}\ E_{f2}] \vdash \langle E_{f1}\rangle : \checkmark \qquad env_t[E_{f1}\ E_{f2}] \vdash \langle E_{f2}\rangle : \checkmark}{env_t \vdash \langle E_{f1}\ E_{f2}\rangle : \checkmark}$$

Kommandoer

En tildeling af et udtryks værdi til en variabel er veltypet relativt til et typeenvironment env_t hvis variabelnavnet og udtrykket har samme type i env_t.

Hvis der er tale om en tildeling til et element i en liste skal listens type være en T `list` hvor T er udtrykkets type i env_t. Desuden skal udtrykket U_a som bestemmer den tilskrevne plads i listen have typen `int`.

Disse to regler er givet i [tildel-1] og [tildel-2].

$$\text{[tildel-1]}\quad \frac{env_t \vdash \langle N\rangle : T_1 \qquad env_t \vdash \langle U\rangle : T_2}{env_t \vdash \langle N\ \texttt{:=}\ U\ \texttt{;}\rangle : \checkmark},\quad \text{hvor } T_1 = T_2$$

$$\text{[tildel-2]}\quad \frac{env_t \vdash \langle N\rangle : T_l \qquad env_t \vdash \langle U_b\rangle : \texttt{int} \qquad env_t \vdash \langle U\rangle : T_e}{env_t \vdash \langle N\ \texttt{[}\ U_a\ \texttt{]}\ \texttt{:=}\ U\ \texttt{;}\rangle : \checkmark},\quad \text{hvor } T_l = T_e\ \texttt{list (}\ v\ \texttt{)}$$

En løkkekonstruktion `for` N `from` U_{a1} `to` U_{a2} `do` K er veltypet relativt til env_t hvis variabelnavnet N og udtrykkene U_{a1} og U_{a2} har typen `int`. Kommandoen K skal desuden være veltypet.

$$\text{[for]}\quad \frac{\begin{array}{c} \qquad\qquad\qquad\qquad env_t \vdash \langle N\rangle : \texttt{int} \\ env_t \vdash \langle U_{a1}\rangle : \texttt{int} \qquad env_t \vdash \langle U_{a2}\rangle : \texttt{int} \qquad env_t \vdash \langle K\rangle : \checkmark \end{array}}{env_t \vdash \langle \texttt{for}\ N\ \texttt{from}\ U_{a1}\ \texttt{to}\ U_{a2}\ \texttt{do}\ K\rangle : \checkmark}$$

Udtryk

Både addition-, substraktion-, multiplikation-, division- og modulokonstruktionerne har samme krav til de umiddelbare bestanddele. Både højre og venstre side skal have typen `int`. For eksempel er typereglen for additionskonstruktioner, [add], gengivet her.

De boolske udtryk U_a `<` U_a, U_a `>` U_a og U_a `=` U_a har enslydende krav til de umiddelbare bestanddele, dog har disse udtryk typen `bool`.

$$[\text{add}] \quad \frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \ \texttt{+} \ U_{a2} \rangle : \texttt{int}}$$

Et funktionskald N `(` $U_1, .., U_n$ `)` har typen T hvis funktionsnavnet N har typen $(T_1, .., T_n) \rightarrow T$ hvor alle udtryk $U_1, .., U_n$ har typen med samme plads i følgen $T_1, .., T_n$.

$$[\text{kald}] \quad \frac{env_t \vdash \langle U_1 \rangle : T_1, .., env_t \vdash \langle U_n \rangle : T_n}{\begin{array}{l} env_t \vdash \langle N \ \texttt{(} \ U_1, .., U_n \ \texttt{)} \ \rangle : T \\ \text{hvis } env_t(N) = (T_1, .., T_n) \rightarrow T \end{array}}$$

Del III

Parallelitet

KAPITEL

3

Hvad er parallelitet?

Vi møder paralleliteten ofte i hverdagen; f.eks. kasselinjerne i supermarkedet, flersporede motorveje og uddelegering af arbejde i en gruppe, sådan at alle kan arbejde på emner, der ikke overlapper.

I datalogi består udtrykket parallelitet dog i det at opdele en opgave i flere delopgaver og tildele disse opgaver til flere processer, der kan køre samtidigt.

3.1 Parallelitet betragtet som fletning

Vi vil i første omgang betragte det, at to programmer afvikles parallelt, som at de to programmer hver især bliver delt op i mindre programmer, som derefter afvikles i en ikke-defineret rækkefølge, hvor de to programmers dele indbyrdes bevarer deres rækkefølge. Dette kaldes fletning, som beskrevet i [Hüt07, s. 84].

Vi introducerer nu sproget **Sep** [1] som er **Sip** med de tilføjelser til syntaksen som er vist i tabel 3.1. Et eksempel på en kommando i **Sep** kan ses på figur 3.1.

$$K \quad ::= \quad \ldots \mid \{K\} \text{ par } \{K\}$$

Tabel 3.1: Tilføjelsen til den abstrakte syntaks for **Sip**

```
x := 0;
y := 0;
{ x := x + 1; x := y + 1; } par
{ y := y + 1; y := x + 1; }
```

Figur 3.1: Eksempel på en kommando skrevet i **Sep**

Konstruktioner som $\{\{K_1\}$ `par` $\{K_2\}\}$ `par` $\{K_3\}$, betyder at K_1, K_2 og K_3 skal afvikles parallelt.

Transistionsreglerne for `par`, som ses i tabel 3.2, giver os ingen indsigt i rækkefølge af kommandoer. Effekten af dette ses i eksemplerne i tabel 3.3. Man kan se i eksemplet kan sluttilstanden for variablene afhænge af rækkefølgen af de kommandoer, som modificerer den. Derfor kan vi ikke forudsige en sådan tilstand, når vi ikke kan forudsige rækkefølgen af disse kommandoer.

3.2 Mål for parallelisering

For at give et mål for, hvor meget man har fået ud af at parallelisere en given opgave bruges udtrykket **speedup** S_n, som beskrevet i [GGKK03, s. 198], hvor

[1] **Sip** med **E**ksplicit **P**arallelitet

$$\text{[Par-1]} \quad \frac{\langle K_1, sto, envl\rangle \Rightarrow \langle K_1', sto', envl\rangle}{\langle \{K_1\} \text{ par } \{K_2\}, sto, envl\rangle \Rightarrow \langle \{K_1'\} \text{ par } \{K_2\}, sto', envl\rangle}$$

$$\text{[Par-2]} \quad \frac{\langle K_1, sto, envl\rangle \Rightarrow sto', envl}{\langle \{K_1\} \text{ par } \{K_2\}, sto, envl\rangle \Rightarrow \langle K_2, sto', envl\rangle}$$

$$\text{[Par-3]} \quad \frac{\langle K_2, sto, envl\rangle \Rightarrow \langle K_2', sto', envl\rangle}{\langle \{K_1\} \text{ par } \{K_2\}, sto, envl\rangle \Rightarrow \langle \{K_1\} \text{ par } \{K_2'\}, sto', envl\rangle}$$

$$\text{[Par-4]} \quad \frac{\langle K_2, sto, envl\rangle \Rightarrow sto', envl}{\langle \{K_1\} \text{ par } \{K_2\}, sto, envl\rangle \Rightarrow \langle K_1, sto', envl\rangle}$$

Tabel 3.2: Smallstep-semantik for nye kommandoer i **Sep**

n er antallet af tilgængelige processorer. Det er nødvendigt at $S_n > 1$, for at det kan betale sig at parallelisere.

Definition 1.3 (Speedup)**.** *Givet at den hurtigste algoritme til at løse et problem P sekventielt tager tiden T_1 og en algoritme, som kan løse problemet parallelt tager tiden T_n, hvor n er antallet af processorer, defineres speedup som:*

$$S_n(P) = \frac{T_1}{T_n}.$$

Et ideelt speedup opnås hvis $S_n = n$, men dette opnås i praksis ikke ofte, da forskellige faktorer spiller ind:

- Overhead: ventetid, tråd-til-tråd-kommunikation, opstart af tråde
- Afhængigheder: hvis opgaver i det sekventielle program afhænger af andre opgaver, fås ventetid. Dette diskuteres senere i afsnit 4.2.

$$
\begin{array}{ll}
 & \langle x := 0 \;;\; x := 0 \;;\; \{x := x+1 \;;\; x := y+1 \;;\}\ \texttt{par}\ \{x := x+1 \;;\; x := y+1 \;;\}\;, sto, evnl\rangle \\
\Downarrow^* & \langle\ \{x := x+1 \;;\; x := y+1 \;;\}\ \texttt{par}\ \{x := x+1 \;;\; x := y+1 \;;\}\;, sto[a_x \mapsto 0, a_y \mapsto 0], envl\rangle \\
\Downarrow^* & \langle\ \{x := y+1 \;;\}\ \texttt{par}\ \{x := x+1 \;;\; x := y+1 \;;\}\;, sto[a_x \mapsto 1, a_y \mapsto 0], envl\rangle \\
\Downarrow^* & \langle\ \{x := y+1 \;;\}\ \texttt{par}\ \{x := y+1 \;;\}, sto[a_x \mapsto 1, a_y \mapsto 1], envl\rangle \\
\Downarrow^* & \langle\ \{x := y+1 \;;\}\;, sto[a_x \mapsto 2, a_y \mapsto 1], envl\rangle \\
\Downarrow^* & sto[a_x \mapsto 2, a_y \mapsto 2], envl
\end{array}
$$

$$
\begin{array}{ll}
 & \langle x := 0 \;;\; x := 0 \;;\; \{x := x+1 \;;\; x := y+1 \;;\}\ \texttt{par}\ \{x := x+1 \;;\; x := y+1 \;;\}\;, sto, envl\rangle \\
\Downarrow^* & \langle\ \{x := x+1 \;;\; x := y+1 \;;\}\ \texttt{par}\ \{x := x+1 \;;\; x := y+1 \;;\}\;, sto[a_x \mapsto 0, a_y \mapsto 0], envl\rangle \\
\Downarrow^* & \langle\ \{x := x+1 \;;\; x := y+1 \;;\}\;, sto[a_x \mapsto 1, a_y \mapsto 0], envl\rangle \\
\Downarrow^* & sto[a_x \mapsto 1, a_y \mapsto 2], envl
\end{array}
$$

Tabel 3.3: To forskellige sluttilstande for kommandoen i figur 3.1 med semantikken i tabel 3.2. Det gælder at $envl = (env_v, env_f) : envl'$ og $a_x = env_v(x)$, $a_y = env_v(y)$, hvor env_v er defineret for x og y.

KAPITEL

4

Hvad er parallelisering?

Når vi vil oversætte et sekventielt program til et parallelt program, ønsker vi at identificere dele af sekventiel kildekode, der kan køre samtidigt, sådan at vi stadig kan være sikre på at det parallelle program er semantisk ækvivalent[1] med det sekventielle program. Dette kan medføre, at vi i visse situationer ikke kan køre flere dele samtidig. Derfor vælger vi at omtale, at et program kan være mere parallelt end et andet, hvilket efter nogle definitioner vil blive formelt defineret. I følgende definition præsenteres en dynamisk analyse af et program baseret på semantikken for **Sep** beskrevet i tabel 3.1:

Definition 1.4. *Lad $\gamma_0(X)$ være antallet af kommandoer der udføres parallelt for en given* **Sep** *-konstruktion X.*

$$\begin{aligned} \gamma_0(K_1\ K_2) &= \gamma_0(K_1) \\ \gamma_0(\{K_1\}\ \texttt{par}\ \{K_2\}) &= \gamma_0(K_1) + \gamma_0(K_2) \\ \gamma_0(X) &= 1,\ \textit{ellers} \end{aligned}$$

Definition 1.5. *Til dynamisk analyse af parallelitet i et* **Sep** *-program, lad da $\gamma_s(P)$ være den sande værdi for det maksimale antal af kommandoer K, der udføres parallelt i programmet P.*

$$\gamma_s(P) = \max \left\{ \begin{array}{l} \gamma_0(X') | \langle X, s\rangle \Rightarrow^* \langle X', s'\rangle, \\ \textit{hvor } s, s' \in \textbf{\textit{Store}} \times \textbf{\textit{Envl}} \\ \textit{og } X, X' \textit{ er } \textbf{Sep} \textit{ -konstruktioner.} \end{array} \right\}$$

For at vi kan bestemme $\gamma_s(P)$ for et givet program, kræver det at vi kan aflæse enhver tilstand til alle input. Hvis $\gamma_s(P)$ er beregnbar, kan vi bruge $\gamma_s(P)$ på dette eksempel:

$$P' = P \texttt{ \{ x := 42 ; \} par \{ x := 42 ; \} } ,$$

hvor P er et sekventielt program. Vi får nu at

$$\gamma_s(P') = \begin{cases} 1 & \text{hvis hvis } P \text{ kører i uendelig løkke} \\ & \text{for ethvert input.} \\ 2 & \text{ellers} \end{cases} .$$

Men da **Sep** er et Turing-fuldstændigt sprog, er det uafgørbart om et sekventielt program P kører i uendelig løkke for ethvert input [Sip06, s. 175], hvorved $\gamma_s(P)$ ikke er beregnbar. Vi er derfor nødt til at finde en måde at bestemme den statiske grad af parallelitet ved analyse af programmets kildekode. Dette leder til indførelsen af γ_a, en approksimeret grad af parallelitet. Da γ_a er en approksimation, hvor vi tager den mindste værdi i tilfælde af et valg, stiller vi som krav at $\gamma_a(P) \leq \gamma_s(P)$.

[1] Semantisk ækvivalens defineres senere i afsnit 4.1

Definition 1.6. *Lad $\gamma_0'(K)$ være en approksimerede værdi for det maksimale antal samtidige kommandoer, ved statisk analyse af kommandoer i* **Sep** *. $\gamma_0'(K)$ er defineret ved:*

$$\begin{aligned}
\gamma_0'(\texttt{function } N(E_v,...,E_v) \texttt{ uses } E_v \texttt{ does } K) &= \gamma_0'(K) \\
\gamma_0'(K_1\ K_2) &= \max\{\gamma_0'(K_1), \gamma_0'(K_2)\} \\
\gamma_0'(\{K_1\} \texttt{ par } \{K_2\}) &= \gamma_0'(K_1) + \gamma_0'(K_2) \\
\gamma_0'(\texttt{if } U \texttt{ then } K_1 \texttt{ else } K_2) &= \min\{\gamma_0'(K_1), \gamma_0'(K_2)\} \\
\gamma_0'(\texttt{for } N \texttt{ from } U_1 \texttt{ to } U_2 \texttt{ do } K) &= \gamma_0'(K) \\
\gamma_0'(\texttt{while } U \texttt{ do } K) &= \gamma_0'(K) \\
\gamma_0'(K) &= 1, \ \textit{ellers}
\end{aligned}$$

Definition 1.7. *Lad $\gamma_a(P)$ være den approksimerede sande grad af parallelitet baseret på statisk analyse.*

$$\gamma_a(P) \quad = \max\{\gamma_0'(E_f) | E_f \in erk(P)\},$$

hvor $erk(P)$ er mængden af funktionserklæringer i P.

$\gamma_a(P)$ er imidlertid en meget grov approksimation, og den opfylder ikke altid vores krav om at $\gamma_a(P) \leq \gamma_s(P)$. I situationer som

```
while false do
{
  { {K1} par {K2} } par {K3};
}
```

vil den approksimerede parallelitet medregne kroppen af `while`-løkken, mens den sande parallitet ikke vil blive berørt, da løkken ikke vil blive kørt. Dog mener vi at dette er et så ekstremt eksempel at der stadig vil være et formål ved at arbejde videre med definitionen.

Vi vil dog ikke udelukkende bruge γ_a til at vurdere om et program er mere parallelt end et andet, da vi også ønsker at se på hvor jævnt paralleliteten uddelt over udførelsen. Dette kan ses som at vores grad af parallelitet belønner en lige fordeling af udførte kommandoer over trådene.

Vi indfører derfor den gennemsnitlige parallelitet [GGKK03, s. 90]. Til dette benyttes to funktioner, $H(P)$ der bestemmer højden af programmet P og $A(P)$ der bestemmer det samlede antal kommandoer i P ved statisk analyse. Med disse funktioner kan vi bestemme den gennemsnitlige parallelitet af et program ved $A(P)/H(P)$. Vi ønsker en høj gennemsnitlig parallelitet.

Definition 1.8. *Lad $H(P)$ være den approksimerede, maksimale højde af pro-*

grammet P:

$$H(P) = \sum_{E_f \in erk(P)} H(E_f)$$

$$H(\texttt{function } N(E_{v1}, ..., E_{vj}) \texttt{ uses } E_v \texttt{ does } K) = H(K)$$

$$H(\{K_1\} \texttt{ par } \{K_2\}) = \max\{H(K_1), H(K_2)\}$$

$$H(K_1\ K_2) = H(K_1) + H(K_2)$$

$$H(\texttt{if } U \texttt{ then } K_1 \texttt{ else } K_2) = H(U) + H(K_j),$$

$$\textit{hvor } K_j = \begin{cases} K_1 \textit{ hvis } \gamma_0'(K_1) \leq \gamma_0'(K_2) \\ K_2 \textit{ ellers} \end{cases}$$

$$H(\texttt{for } N \texttt{ from } U_1 \texttt{ to } U_2 \texttt{ do } K) = H(U_1) + H(U_2) + H(K)$$

$$H(\texttt{while } U \texttt{ do } K) = H(U) + H(K)$$

$$H(U) = 1$$

$$H(K) = 1, \textit{ ellers}$$

Bemærk at selvom der kan forekomme funktionskald i et udtryk U, som vil medføre evaluering af kommandoer, så er $H(U) = 1$. Dette skyldes at der kan forekomme rekursive kald i funktioner. Det samme gør sig gældende i definition 1.9.

Definition 1.9. *Lad $A(P)$ være det samlede antal af kommandoer i programmet P. For $A(P)$ gælder de samme regler for kommandokonstruktioner som for $H(P)$, med undtagelse af $(K_1 \texttt{ par } K_2)$, der defineres således:*

$$A(\{K_1\} \texttt{ par } \{K_2\}) = A(K_1) + A(K_2)$$

$$A(K) = H(K) \textit{ ellers.}$$

Som tidligere nævnt ønsker vi en så høj gennemsnitlig parallelitet som muligt. Dette er et ønskværdigt scenarie da vi så har en så lille H som muligt. Samtidigt ønsker vi en høj værdi for γ_a, idet at jo højere γ_a er for et program P, des flere samtidige kommandoer vil i værste fald blive udført på et tidspunkt under evaluering af P.

For at understrege hvad vi er kommet frem til, har vi følgende definition:

Definition 1.10. *Lad P_1 og P_2 være semantisk ækvivalente og lad P_1 være fremkommet ved en parallelisering af P_2. P_1 er mere parallelt end P_2 hvis:*

$$\gamma_a(P_1) > \gamma_a(P_2) \textit{ og } \frac{A(P_1)}{H(P_1)} > \frac{A(P_2)}{H(P_2)}$$

4.1 Semantisk ækvivalens og parallelitet

Vi vil i dette afsnit definere semantisk ækvivalens mellem et **Sip** -program og et **Sep** -program. Vi har ikke tænkt os at vise ækvivalensen mellem **Sip** - og **Sep** -konstruktioner, men vi mener at denne diskussion er relevant mht. vores parallelisering.

Der er flere overvejelser der skal tages i betragtning. Som man kan se i afsnit 4.2.1, kan man fjerne afhængigheder i kildekoden ved at indføre midlertidige variable. Dette gør at det sekventielle program og det dertilhørende paralleliserede program med fjernede afhængigheder muligvis vil ende i forskellige tilstande. En anden overvejelse er at selvom vi har omtalt at `par`-konstruktionen kan resultere i non-determinisme, er dette emne ikke interessant, når vi kun paralleliserer uafhængighede delmængder af den sekventielle kildekode.

Vi ønsker at for enhver tilstand for det sekventielle program, skal det parallelle program kunne bringes i samme tilstand. Med samme tilstand menes at variable med samme navn er bundet til de samme værdier, for de to programmer, når man ser bort fra eventuelle overskydende variable, introduceret i fjernelse af afhængigheder.

Vi vil kun benytte denne form for ækvivalens mellem et sekventielt program og dets paralleliserede udgave, da vi i andre tilfælde ikke kan være sikker på at de samme variable eksisterer.

Vi har valgt at benytte definitionen af en svag bisimulering, som er præsenteret i definition 1.11:

Definition 1.11. *En binær relation R kaldes en svag bisimulering, hvis $(x, y) \in R$ medfører:*

1. $x \Rightarrow^* x'$ *medfører* $\exists y' : \ y \Rightarrow^* y'$ *og* $(x', y') \in R$
2. $y \Rightarrow^* y'$ *så* $\exists x' : \ x \Rightarrow^* x'$ *så* $\exists y'' : \ y' \Rightarrow^* y''$ *og* $(x', y'') \in R$

x og y er bisimuleringsækvivalente, hvis der eksisterer en bisimulering R og $(x, y) \in R$. Dette skrives $x \sim y$.

Vi vil nu give vores bud på en relation R, som siger at to tilstande er relaterede hvis deres fælles variable har samme værdi.

Definition 1.12.

$$R = \left\{ (x,y) \;\middle|\; \begin{array}{l} x = \langle K, sto_x, (env_{vx}, env_{fx}) : envl \rangle \\ \text{og } y = \langle K, sto_y, (env_{vy}, env_{fy}) : envl \rangle \\ \text{så } \exists V : \ V = Dm(env_{vx}) \text{ eller } V = Dm(env_{vy}) \\ \text{og } sto_x(env_{vx}(v)) = sto_y(env_{vy}(v)) \text{ for alle } n \in V \end{array} \right\}.$$

Hvis vi ikke havde det sidste krav i definition 1.11, ville vi ikke kunne finde en relation som gælder for et sekventielt program x og et parallelt program y. Betragt følgende eksempel:

$$x = \langle \texttt{m := 42; n := 42;}, [a_m \mapsto 0][a_n \mapsto 0], ([m \mapsto a_m][n \mapsto a_n], env_f) \rangle$$
$$y = \langle \texttt{\{m := 42;\}par\{n := 42;\}}, [a_m \mapsto 0][a_n \mapsto 0], ([m \mapsto a_m][n \mapsto a_n], env_f) \rangle$$

Følger man transitionsreglerne i semantikken fra tabel 3.2 gælder følgende:

$$y \Rightarrow^* y' = \langle \ \texttt{m := 42 ;} \ , [a_m \mapsto 0][a_n \mapsto 42], ([m \mapsto a_m][n \mapsto a_n], env_f) \rangle$$

Men tilstanden, hvor $\texttt{m} = 0$ og $\texttt{n} = 42$ kan x ikke nå. Derimod gælder følgende.

$$x \Rightarrow^* x' = \langle [a_m \mapsto 42][a_n \mapsto 42], ([m \mapsto a_m][n \mapsto a_n], env_f)\rangle$$
$$y' \Rightarrow^* y'' = \langle [a_m \mapsto 42][a_n \mapsto 42], ([m \mapsto a_m][n \mapsto a_n], env_f)\rangle$$

Bemærk at $\mathtt{m} = 42$ og $\mathtt{n} = 42$ i både x' og y''. Dette opfylder vores intuitive idé om semantisk ækvivalens for parallellisering.

Bemærk dog at ækvivalensbegrebet ikke opfylder de velkendte pæne egenskaber for en ækvivalensrelation mht. symmetri.

4.2 Parallelisering af sekventiel kode

Vi vil i dette afsnit præsentere en indledende analyse af sekventiel kode med henblik på at identificere dele af koden, der kan afvikles parallelt.

Da en programmørs måde at skrive kildekode på, kan skabe "falske" afhængigheder i kildekoden, kan en afhængighedsanalyse udføres for at eliminere disse. En sådan defineres i afsnit 4.2.1.

Ifølge Amdahls lov ([Kri01, s. 289]) vil man med et program, hvor F er procentdelen af kode ($0 \leq F \leq 1$), der kun kan udføres sekventielt, og n processorer maksimalt kunne opnå et speedup på

$$\frac{1}{F + \frac{1-F}{n}}.$$

Da $F = 0$ giver et maksimalt speedup er afhængighedsanalysens formål at minimere F.

4.2.1 Afhængigheder

Når vi vil afvikle blokke af kode i flere processer, må vi være opmærksomme på at kommandoer, der skal afvikles i en bestemt rækkefølge stadig afvikles i denne rækkefølge. Som eksempel kan ses en tilskrivning og en aflæsning af en variabel; hvis tilskrivningen ikke kommer før aflæsningen, er det udefineret om man får den rigtige værdi ved aflæsningen.

Vi vil kategorisere forskellige typer af afhængigheder:

Flow-afhængighed

$K1$: `a := ...`
`...`
$K2$: `... := ... a ...`

Variablen `a` tilskrives før den læses. Der siges at være en flow-afhængighed fra $K1$ til $K2$.

Antiafhængighed

$K1$: `... := ... a ...`
`...`
$K2$: `a := ...`

Den første kommando skal udføres først, da den anden kommando ændrer `a`. Vi siger, at der er en antiafhængighed fra $K2$ til $K1$.

Output-afhængighed

$K1$: `a := ...`
`...`
$K2$: `a := ...`

Vi må bibeholde ordenen af tildelinger for at være sikre på at programmet udføres korrekt. Vi siger at der er en output-afhængighed fra $K2$ til $K1$.

Input-afhængighed, hvor den samme variabel læses flere gange, behandles ikke, da den ikke ændrer noget i variablernes tilstande uanset rækkefølgen af læsninger.

Flow-afhængighed er den eneste "ægte" afhængighed. Output- og antiafhængighed fremkommer ved genbrug af variable, hvilket motivere en teknik til fjernelse af disse.

Fjernelse af afhængigheder

Vi vil i dette afsnit præsentere teknikker til fjernelse af afhængigheder. Teorien er hentet fra [PW86] og [CDRV98].

Definition 1.13. *Vi lader U være et uspecificeret udtryk og lader U_v være et udtryk der involverer variablen v. I følgende stykke kildekode haves en antiafhængighed:*

$$
\begin{array}{ll}
 & \ldots \\
K1: & var_1 \texttt{ := } U_{var_2} \texttt{ ;} \\
 & \ldots \\
K2: & var_2 \texttt{ := } U \texttt{ ;} \\
 & \ldots
\end{array}
$$

Ved at transformere kildekoden fjernes antiafhængigheden:

$$
\begin{array}{ll}
 & \ldots \\
K1: & var_2' \texttt{ := } var_2 \texttt{ ;} \\
K2: & var_1 \texttt{ := } U_{var_2'} \texttt{ ;} \\
 & \ldots \\
K3: & var_2 \texttt{ := } U \texttt{ ;} \\
 & \ldots
\end{array}
$$

Betragt følgende eksempel:

Eksempel 1.1. *I følgende kildekode er der en antiafhængighed (og en outputafhængighed fra K3 til K1) fra K3 til K2.*

```
K1:  a := b + c ;
K2:  d := a + 1 ;
K3:  a := d + 2 ;
K4:  e := a + 17 ;
```

Afhængigheden fjernes ved at omdøbe `a` *i første forekomst.*

```
K1:  a2 := b + c ;
K2:  d := a2 + 1 ;
K3:  a := d + 2 ;
K4:  e := a + 17 ;
```

Som tidligere nævnt fremkommer output- og antiafhængigheder ved genbrug af variable. De to stykker kildekode er ækvivalente, men vi bliver i det sidstnævnte nødt til at allokere hukommelse til endnu en variabel.

Dette afføder en transformation, der vil fjerne output-afhængigheder.

Definition 1.14. *Vi lader U være et uspecificeret udtryk og lader U_a være et udtryk der involverer variablen a. I følgende stykke kildekode haves en output-afhængighed:*

$$\begin{array}{ll} & \ldots \\ K1: & var_1 \texttt{ := } U \texttt{ ;} \\ & \ldots \\ K2: & var_1 \texttt{ := } U \texttt{ ;} \\ & \ldots \end{array}$$

Ved at transformere kildekoden fjernes antiafhængigheden:

$$\begin{array}{ll} & \ldots \\ K1: & var_1' \texttt{ := } U \texttt{ ;} \\ & \ldots \\ K2: & var_1 \texttt{ := } U \texttt{ ;} \\ & \ldots \end{array}$$

Hvis kommandoer mellem $K1$ og $K2$ tidligere brugte var_1 ændres disse til at bruge var_1'.

Vi vil nu definere en afhængighedsgraf:

Definition 1.15. *Knuderne i den orienterede afhængighedsgraf G_A består af knuder repræsenteret af kommandoer . Der findes en kant fra knuderne u og v hvis der er en output-, flow- eller antiafhængighed fra u til v.*

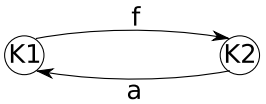
Vi vil i det følgende mærke kanter i afhængighedsgrafer med o, f eller a for hhv. output-, flow- eller antiafhængigheder. Vi er interesserede i at eliminere eventuelle kredse i afhængighedsgrafen. En kreds udgør en række kommandoer der kun kan udføres sekventielt; en bestemt rækkefølge er bestemt på forhånd.

De følgende eksempler illustrerer metoderne beskrevet tidligere:

Eksempel 1.2. *Betragt følgende stykke kildekode:*

```
for i from 0 to n-1 do
{
  K1 : a[i] := b[i] + c[i] ;
  K2 : d[i] := a[i] + a[i+1] ;
}
```

Vi har her en flow-afhængighed fra $K1$ til $K2$. Desuden må $K1$ ikke udføres i næste iteration før `a[i+1]` *i $K2$ læses. Dette giver en antiafhængighed fra $K2$ til $K1$. Vi har derfor følgende afhængighedsgraf med en kreds:*

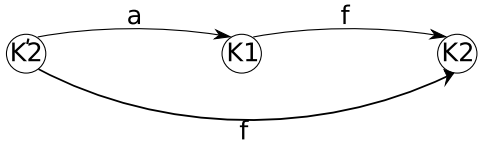


Figur 4.1: K1 og K2 som knuder i en afhængighedsgraf

Vi kan fjerne kredsen ved at bruge et midlertidigt array:

```
for i from 0 to n-1 do
{
  K2' :  temp[i] := a[i+1] ;
  K1 :  a[i] := b[i] + c[i] ;
  K2 :  d[i] := a[i] + temp[i] ;
}
```

På denne måde kan $K1$ udføres, da værdien af `a[i+1]` *gemmes. Dette giver følgende afhængighedsgraf:*



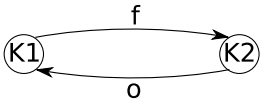
Figur 4.2: Figur 4.1 uden kreds

Tilsvarende kan vi fjerne output-afhængigheder.

Eksempel 1.3. *Betragt følgende stykke kildekode:*

```
for i from 0 to n-1 do
{
  K1 : a[i] := b[i] + c[i] ;
  K2 : a[i+1] := a[i] + d[i] ;
}
```

Vi har her en output-afhængighed fra $K2$ til $K1$, fordi `a[i+1]` *tilskrives og derefter tilskrives igen i næste iteration. Dette giver følgende afhængighedsgraf med en kreds:*

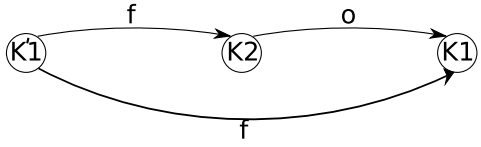


Figur 4.3: $K1$ og $K2$ som knuder i en afhængighedsgraf

Vi kan igen fjerne kredsen ved at bruge et midlertidigt array:

```
for i from 0 to n-1 do
{
  K2' : temp[i] := b[i] + c[i] ;
  K1 : a[i+1] := temp[i] + d[i] ;
  K2 : a[i] := temp[i] ;
}
```

Dette giver følgende afhængighedsgraf:



Figur 4.4: Figur 4.3 uden kreds

Afhængigheder og non-determinisme

Som omtalt i afsnit 3.1 kan man ved bruge af `par`-konstruktionerne opnå forskellige sluttilstande. Lad os definere $L(K)$ og $S(K)$ som mængden af variable

$uses(K_1\ K_2)$	$=$	$uses(K_1) \cup uses(K_2)$
$uses(N$ `:=` U `;`$)$	$=$	$\{(N, w)\} \cup uses(U)$
$uses(N$ `[` U_a `]` `:=` U `;`$)$	$=$	$\{N, w\} \cup uses(U_a) \cup uses(U)$
$uses($`{`K`}`$)$	$=$	$uses(K)$
$uses($`skip;`$)$	$=$	$\emptyset$
$uses($`if` U_b `then` K_1 `else` $K_2$$)$	$=$	$uses(U_b) \cup uses(K_1) \cup uses(K_2)$
$uses($`for` N `from` U_{a1} `to` U_{a2} `do` K	$=$	$\{(N, w)\} \cup uses(U_{a1})$
		$\cup uses(U_{a2}) \cup uses(K)$
$uses($`while` U_b `do` $K$$)$	$=$	$uses(U_b) \cup uses(K)$

Tabel 4.1: Værdier af $uses(K)$ for alle $K \in \mathbf{K}$

der hhv. læses og skrives i udførelsen af kommandoen K. Vi vil i vores implementation afgrænse os til at udføre K_1 par K_2 hvor $L(K_1) \cap S(K_2) = \emptyset$, $S(K_1) \cap L(K_2) = \emptyset$ og $S(K_1) \cap S(K_2) = \emptyset$; delene af et programmet kan ikke påvirke hinanden og vi er sikret en entydig sluttilstand[2]. Dette vil iht. afhængigheder betyde, at der ikke vil være output-, flow og antiafhængigheder mellem K_1 og K_2.

4.2.2 Metode for afhængighedsanalyse

For at kunne finde ud af om to kommandoer $K_1, K_2 \in \mathbf{K}$ afhænger af hinanden, dvs. om afviklingen af de to parallelt muligvis vil give et non-deterministisk resultat, vil det være behageligt at have en funktion

$$depends : \mathbf{K} \times \mathbf{K} \to \{tt, ff\},$$

som kan fortælle om to kommandoer afhænger af hinanden.

Før vi definerer denne vil vi dog gerne introducere hjælpefunktionen

$$uses : \mathbf{K} \cup \mathbf{U} \to (\mathbf{N} \times \{r, w\})^*,$$

hvor $(\mathbf{N} \times \{r, w\})^*$ betegner mængden af alle par af navne og enten et r eller w, som hhv. angiver læsning og skrivning.

Hvis der i en kommando ændres på værdien bundet til et variabelnavn n i en kommando K medfører det at $(n, w) \in uses(K)$. Hvis en kommando K læser værdien bundet til et variabelnavn n i en kommando K medfører det at $(n, r) \in uses(K)$.

Funktionen $uses$ er defineret for alle $K \in \mathbf{K}$ i tabel 4.1 og alle $U \in \mathbf{U}$ i tabel 4.2.

Vi kan nu bruge $uses$ til at angive afhænghed mellem to kommandoer $K_1, K_2 \in \mathbf{K}$ med funktionen $depends$ som kan defineres sådan:

$$depends(K_1, K_2) = \begin{cases} tt \text{ hvis } \exists n \in \mathbf{N} : \begin{pmatrix} ((n, w) \in uses(K_1) \wedge (n, r) \in uses(K_2)) \\ \vee\, ((n, r) \in uses(K_1) \wedge (n, w) \in uses(K_2)) \\ \vee\, ((n, w) \in uses(K_1) \wedge (n, w) \in uses(K_2)) \end{pmatrix} \\ ff \text{ ellers.} \end{cases}$$

[2]Vi kan dog i visse tilfælde godt have en entydig sluttilstand for K_1 par K_2, hvis betingelserne ikke er opfyldt.

$uses(U_{a1} + U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1} - U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1} * U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1}/U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1}$ `mod` $U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1} < U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1} > U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{a1} = U_{a2})$	=	$uses(U_{a1}) \cup uses(U_{a2})$
$uses(U_{b1}$ `and` $U_{b2})$	=	$uses(U_{b1}) \cup uses(U_{b2})$
$uses(U_{b1}$ `or` $U_{b2})$	=	$uses(U_{b1}) \cup uses(U_{b2})$
$uses($ `(` U `)` $)$	=	$uses(U)$
$uses(N$ `(` $U_1, .., U_n$ `)` $)$	=	$uses(U_1) \cup ... \cup uses(U_n)$
$uses(N$ `[` U_a `]` $)$	=	$\{N, r\} \cup uses(U_a)$
$uses(N)$	=	$\{(N, r)\}$
$uses(L)$	=	$\emptyset$

Tabel 4.2: Værdier af $uses(U)$ for alle $U \in \mathbf{U}$

Det vil sige at *depends* for to kommandoer er *ff*, hvis de to ikke har indeholder kommandoer, der er afhængige af hinanden på noget måde. Dette kan illustreres med et eksempel. Betragt kommandoerne K_1 og K_2 givet ved

$K1:$ `x := 0; y := 0; z := 0;`
$K2:$ `z := x + y;`

Idet der er flow-afhængihed fra $K1$ til $K2$ (mht. til `x` og `y`) og output-afhængighed fra K_2 og K_1 (mht. `z`) vil $depends(K_1, K_2) = tt$.

4.2.3 Metode til parallelisering

Vi kan nu definere en metode til parallelisering. Idet parallelisering af et program $P_{\mathbf{Sip}}$ kan betragtes som en transition fra dette program til et andet program $P_{\mathbf{Sep}}$ kan parallelisering beskrives som et transitionssystem $(\Gamma, \rightarrow_{par}, \mathrm{T})$, hvor $\Gamma = \mathrm{T}$ som er mængden af programmer i **Sep** og $\rightarrow_{par}$ er givet ved transitionsreglerne i tabel 4.3 på formen

$$\frac{præmisser}{X \rightarrow_{par} X'} \quad , \quad sidebetingelser,$$

hvor X er en **Sep** -konstruktion og *præmisser* er transitionsregler, som bruges til at danne X'.

For at kunne afvikle to kommandoer K_1 og K_2 parallelt, kræver vi at der ikke er afhængigheder mellem de to, altså at $depends(K_1, K_2) = ff$.

Dette vil være grundlaget for vores implementation af afhængighedsanalyse i oversætteren.

$$\frac{E_{f1} \rightarrow_{par} E'_{f1} \quad E_{f2} \rightarrow_{par} E'_{f2}}{E_{f1}\ E_{f2} \rightarrow_{par} E'_{f1}\ E'_{f2}}$$

$$\frac{K \rightarrow_{par} K'}{\texttt{function}\ N(E_{v1}, .., E_{vn}) : T\ \texttt{uses}\ E_v\ \texttt{does}\ K \rightarrow_{par} \texttt{function}\ N(E_{v1}, .., E_{vn}) : T\ \texttt{uses}\ E_v\ \texttt{does}\ K'}$$

$$\frac{K_1 \rightarrow_{par} K'_1 \quad K_2 \rightarrow_{par} K'_2}{K_1\ K_2 \rightarrow_{par} K'_1\ \texttt{par}\ K'_2}, \text{ hvis } depends(K_1, K_2) = ff$$

$$\frac{K_1 \rightarrow_{par} K'_1 \quad K_2 \rightarrow_{par} K'_2}{K_1\ K_2 \rightarrow_{par} K'_1\ K'_2}, \text{ hvis } depends(K_1, K_2) = tt$$

$$N\ \texttt{:=}\ U \rightarrow_{par} N\ \texttt{:=}\ U$$

$$\frac{K \rightarrow_{par} K'}{\texttt{\{}\ K\ \texttt{\}} \rightarrow_{par} \texttt{\{}\ K'\ \texttt{\}}}$$

$$\texttt{skip} \rightarrow_{par} \texttt{skip}$$

$$\frac{K_1 \rightarrow_{par} K'_1 \quad K_2 \rightarrow_{par} K'_2}{\texttt{if}\ U_b\ \texttt{then}\ K_1\ \texttt{else}\ K_2 \rightarrow_{par} \texttt{if}\ U_b\ \texttt{then}\ K'_1\ \texttt{else}\ K'_2}$$

$$\frac{K \rightarrow_{par} K'}{\texttt{for}\ N\ \texttt{from}\ U_a\ \texttt{to}\ U_a\ \texttt{do}\ K \rightarrow_{par} \texttt{for}\ N\ \texttt{from}\ U_a\ \texttt{to}\ U_a\ \texttt{do}\ K'}$$

$$\frac{K \rightarrow_{par} K'}{\texttt{while}\ U_b\ \texttt{do}\ K \rightarrow_{par} \texttt{while}\ U_b\ \texttt{do}\ K'}$$

Tabel 4.3: Transitionsregler for paralleliseringstransitionen $\rightarrow_{par}$.

Del IV

Implementation

KAPITEL

5

Oversætteren

Vi vil i dette afsnit beskrive, hvordan vi har implementeret en oversætter og en fortolker.

Da vi, som man kan læse i afsnit 5.2, har valgt at benytte SableCC til lexer- og parser-generering, vil afsnittet om syntaktisk analyse primært handle om teorien bag denne form for analyse baseret på [WB00] og kun i mindre grad implementationen. Ved den konstekstuelle analyse begynder den egentlige implementation, hvor afsnittene vil gå i en mere kildekodeorienteret retning og afsluttes med forsøg udført med implementeringen.

5.1 Syntaktisk analyse

De følgende afsnit vil gennemgå delaktiviteter i syntaktisk analyse. Vi omtaler primært de overordnede principper.

5.1.1 Leksikalsk analyse

Den leksikalske analyse er i oversætteren repræsenteret som en klasse ved navn *lexer*. Det er denne klasse som analyserer kildefilen tegn for tegn og genkender de ord i kildefilen som repræsenterer atomare ord som `while`, `do`, literale og navne, kaldet tokens. Disse tokens bliver repræsenteret som objekter af *Token*-klassen i en tokenstrøm.

Følgende eksempel viser, hvordan en streng er klassificeret som tokens.

Eksempel 1.4. *Et eksempel på en leksikalsk analyse af* `z := x + y`.

Identifier	Assignment	Identifier	Addition	Identifier
z	:=	x	+	y

Figur 5.1: Tokens i kommandoen `z := x + y`

Lexeren ved hvilke tokens den skal genkende ud fra en konkret syntaks for **Sep** , skrevet efter den abstrakte syntaks fra afsnittene 2 og 3.1. Da **Sip** er en delmægde af **Sep** vil en lexer skrevet til **Sep** også kunne genkende **Sip** . Den konkrete syntaks og beskrivelse af denne, kan findes i bilag B.

5.1.2 Parsing

Parseren i oversætteren bestemmer om et givent program er gyldigt mht. den konkrete syntaks for sproget og generere et abstrakte syntakstræ for et givent program.

Vi illustrerer Bottom-up og Top-down metoderne til parsing ved at parse strengen "the man sees a computer scientist ." efter følgene konkrete syntaks:

Sentence	::=	Subject Verb Object .
Subject	::=	**I** | **a** Noun | **the** Noun
Object	::=	**me** | **a** Noun | **the** Noun
Noun	::=	**computer scientist** | **mathematician** | **man**
Verb	::=	**sees** | **love**

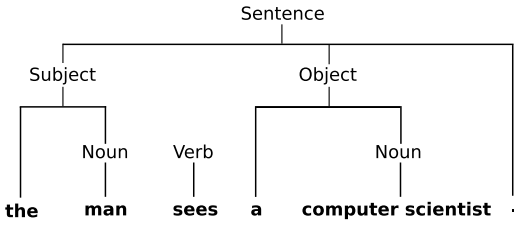
Venstre mod højre, bottom-up

Ved input prøver parseren at konstruere et abstrakt syntakstræ fra venstre mod højre og fra de enkelte tokens, og op mod roden. Givet en produktionsregel $N ::= X_1 \dots X_n$, hvor N er en nonterminal og $X_1, \dots, X_n$ er terminaler eller nonterminaler, vil parseren konstruere et N-træ over en streng af terminalsymboler hvis den matcher $X_1 \dots X_n$.

Først ses ordet 'the'; parseren ved intet endnu, så den læser videre og opdager 'man'. 'man' er Noun og den genkender at 'the man' er en Subject-konstruktion. Parseren laver nu et Subject-træ over 'the man' som det ses på figur 5.2:

Figur 5.2: 'the man' genkendt som Subject

Parseren læser videre fra venstre mod højre og har til sidst konstrueret det abstrakte syntakstræ som det ses på figur 5.3.



Figur 5.3: Det endelige abstrakte syntakstræ

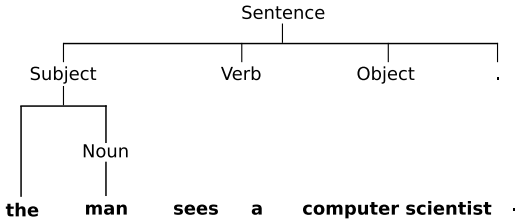
Grunden til at 'a computer scientist' ikke parses som Subject er at dette vil bryde med reglen for Sentence, og parseren har allerede læst Subject-konstruktionen "the man" og Verb-konstruktionen "sees" og ved derfor at den er igang med at læse en Sentence-konstruktion.

Venstre mod højre, top-down

Parseren starter fra startsymbolet og ser på alle bestanddele fra venstre mod højre. Den vil hele tiden se på udvidelsen længst til venstre som ikke er forbundet til strengen:

Parseren starter med Sentence. Kun én produktionsregel er mulig så Sentence får børnene Subject, Verb, Object og . i nævnte rækkefølge. Parseren tager nu

fat i Subject og kan se at den skal have børnene 'the' og Noun. Dette ses på figur 5.4:



Figur 5.4: 'the man' genkendt som Subject

Slutteligt får parseren genereret samme abstrakte syntakstræ som i buttom-up.

5.2 SableCC

I stedet for at skrive vores egen lexer og parser, har vi valgt at bruge værktøjet SableCC. Det skyldes at vi ønsker at lægge vores koncentration i implementeringen af en fortolker og da konstrueringen af en lexer og en parser ifølge SPO-forelæser Bent Thomsen er en "robotic activity which can be automated"[Tho07].

Det har vist sig at mængden af dokumentation til SableCC er begrænset, så vi vil i de følgende afsnit give en beskrivelse af SableCC og demonstrere, hvordan vi bruger SableCC.

SableCC[1] er et værktøj der ud fra en grammatik, der er en specialiseret form for Backus-Naur-formalisme, kan generere en lexer og en parser. En sådan grammatik kan ses på bilag B.1

Oprindeligt genererer SableCC lexeren og parseren i Java[2]-kildekode, men vi benytter en udvidelse [Man04], som genererer C#-kode, da dette sprog er kendt af alle i gruppen.

Før vi beskriver implementationen af funktionalitet til kontekstuel analyse og kodegenerering, vil vi først gennemgå de værktøjer SableCC stiller til rådighed.

5.2.1 Abstrakt syntakstræ

Fra SableCC's parser bliver vi givet et abstrakt syntakstræ. Dette træ vil vi bruge i den kontekstuelle analyse. Det abstrakte syntakstræ konstrueres på følgende måde:

1. Først oprettes en strøm ud fra kildefilen.
2. Denne strøm gives som input til *Lexer*-objektet.
3. Herefter oprettes et *Parser*-objekt ud fra *Lexer*-objektet.
4. Til sidst genereres et abstrakt syntakstræ, hvor *Start* er roden, ved at køre metoden *Parse()* på den nygenererede parser.

[1] Se `http://sablecc.org`
[2] http://java.com

```
TextReader inputFile = new StreamReader(args[0]);
Lexer lexer = new Lexer(inputFile);
Parser parser = new Parser(lexer);
Start ast = parser.Parse();
```

Til at bearbejde dette syntakstræ stiller SableCC metoder til rådighed, en såkaldt dybdeførst-adapter og metoder til dekorering af træet.

5.2.2 Dybdeførst-adapter

Fra SableCC er vi givet klassen *DepthFirstAdapter*, som stiller en række metoder til rådighed. Disse metoder bliver kaldt når en dybdeførst-traversering af syntakstræet møder de forskellige knuder der er knyttet til de forskellige metoder.

```
//det abstrakte syntakstræ
Start ast = parser.Parse();
//træet gennemløbes dybdeførst
ast.Apply(new DepthFirstAdapter());
```

I kodeeksemplet ovenfor kan det ses at der startes en dybdeførst-traversering på det abstrakte syntakstræ.

Følgende opstilling viser de forskellige typer af metoder som stilles til rådighed fra *DepthFirstAdapter*. Opstillingen tager udgangspunkt i produktionsreglen `factor = {mult} factor mult term`, som ses i vores grammatik til SableCC.

- *CaseAMultFactor*: Denne funktion kaldes når vi møder en *AMultFactor*-knude. De to nedennævnte metoder bliver kaldt hhv. på vej ind i og på vej ud af knuden og udfører, ligesom ovenfor, den indeholdende klasse på knudens bestanddele.

  ```
  InAMultFactor(node);

  if(node.GetFactor() != null)
  {
    node.GetFactor().Apply(this);
  }
  if(node.GetMult() != null)
  {
    node.GetMult().Apply(this);
  }
  if(node.GetTerm() != null)
  {
    node.GetTerm().Apply(this);
  }

  OutAMultFactor(node);
  ```

  Metoderne *GetFactor*, *GetMult* og *GetTerm* returner knudens bestanddele.

- *InAMultFactor*: Bliver kaldt på vej ind i knuden.

- *OutAMultFactor*: Bliver kaldt på vej ud af knuden.

- *CaseTMult*: Kaldes når vi møder tegnet `*`.

For at gøre denne traverseringsmetode endnu klarere gennemgåes her et eksempel på en del af traverseringen af den følgende **Sip** -kode:

```
function main () : int
uses
  a : int;
does
{
  a := 2;
  main := f(2);
  ...
```

Hvis man udfører dybdeførst-traverseringen på eksemplets abstrakte syntakstræ, vil følgende metoder bliver kaldt. For at simplificere lidt viser vi kun et udpluk af de metoder der kaldes og kun *Case*-varianten, ikke de to andre varianter (*In*, *Out*).

1. *CaseAUsesImplSingle*, kaldes når traverseringen møder funktionen, da denne har en `uses`-blok. Hvis der ingen `uses`-blok var i erklæringen skulle *CaseANoUsesImplSingle* kaldes.
2. *CaseAIntTypeSpecifier*, kaldes når traverseringen når typespecificeringen mødes.
3. *CaseADefSingle*, kaldes når traverseringen møder erklæringen af variable i `uses`-blokken.
4. *CaseAExprAssignment*, kaldes når traverseringen møder tildelinger.
5. *CaseAFunccall*, kaldes når traverseringen møder et funktionskald.

5.2.3 Hjælpefunktioner

SableCC stiller også to hash-tabeller til rådighed, hvori vi kan gemme en knude som værdi og et objekt som nøgle. Dette kaldes at dekorere knuden med værdien, og gøres med de følgende fire funktioner:

- *void SetIn(Node node, Object inobj)*
- *Object GetIn(Node node)*
- *void SetOut(Node node, Object inobj)*
- *Object GetOut(Node node)*

SetIn-funktionen indsætter en knude og et objekt i en hash-tabel som hhv. nøgle og værdi, hvorefter dette objekt kan hentes ud igen ved hjælp af *GetIn()*-funktionen. *SetOut* og *GetOut* fungerer på samme måde, dog med en anden hash-tabel.

5.3 Kontekstuel analyse

Under en kontekstuel analyse verificeres at det parsede program opfylder de såkaldte kontektuelle krav i sproget, som i **Sep** er scope-regler under identifikationen af variabler og funktioner og typeregler under typetjeket.

Vi har skrevet to nye klasser[3], *GlobalAnalysis* og *LocalAnalysis*, som begge nedarver fra *DepthFirstAdapter*. Disse får hver ansvar for en del af typetjek og identifikation. *GlobalAnalysis* køres som en første traversering af det abstrakte syntakstræ og *LocalAnalysis* som en anden traversering.

Vi benytter i typetjeket typeregler, som vi så dem i afsnit 2.4. Metoderne *SetOut* og *GetOut* vil blive benytte som et type-environment, hvori par af literaler/identifiers/udtryk samt type gemmes. Bemærk at vi ikke registrerer om konstruktioner er veltypede, da dette i implementationen blot skal generere en fejl.

5.3.1 Første traversering

Hvis en traversering af funktionskroppe påbegyndes før vi kender alle funktionsnavne og deres returtyper, vil det ikke være mulig at afgøre om et givent udtryk er gyldigt, da vi kunne støde på et kald til en ukendt funktion. På grund af dette vil denne første traversering stå for at indsamle funktionsnavne, undersøge om navnene er brugt flere gange og aflæse returtyperne. Funktionernes navne, formelle parametre og returtype gemmes. Til at gemme denne information omkring funktioner og variable benyttes klassen *Identifier*, som vil blive brugt i begge traverseringer.

```
class Identifier
{
  private TIdent id;
  private Type type;
  private Boolean listtype;
  private Int32 listsize;
  private List<Identifier> plist = new List<Identifier>();
  ...
```

Identifierens navn gemmes i *id* og dens type i *type*. Derudover indeholder klassen en boolsk værdi til at signalere om dette det givne objekt er en listetype, størrelsen på en evt. liste og nederst en generisk liste, der kan indeholde eventuelle formelle parametre, hvis identifieren er en funktion.

For hver funktion bliver et objekt med denne information gemt i en hashtabel for det globale scope med funktionens navn som nøgle og det tilhørende *Identifier*-objekt som værdi. Først kontrolleres dog, at der ikke allerede findes en nøgle i hash-tabellen med samme navn som den identifier man prøver at tilføje. I så fald returneres en fejl, hvilket følgende eksempel illustrerer.

I metoden *InAUsesImplSingle* skal vi se om knuden *node* som er en instans af *AUsesImplSingle*, eksisterer blandt de gemte globale funktions navne.

```
if(symbol_table_global.ContainsKey(node.GetIdent().Text))
{
  throw new Exception(...);
}
else
{
  checkingNow = node.GetIdent().Text;
  symbol_table_global.Add(node.GetIdent().Text,
                          new Identifier(node.GetIdent()));
}
```

Det noteres hvilken funktion der arbejdes på, hvilket benyttes til at identificere den pågældende funktion, når der skal identificeres type og parametre.

For at give et kort overblik foregår indsamlingen af funktioner således:

[3] I filen ContextualAnalysis.cs

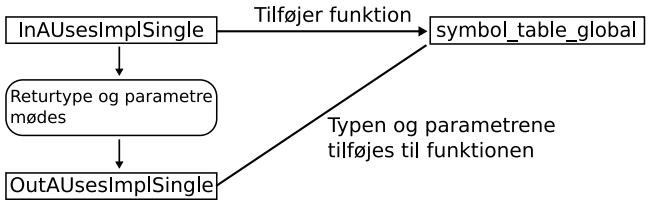
- På vej ind i en funktionsknude instantieres et *Identifier*-objekt med *TIdent*.
- Formelle parametre tilføjes til den pågældende funktions *Identifier*-objekt.
- På vej ud af knuden sættes returtypen på *Identifier*-objektet.

Grunden til at vi opretter et *Identifier*-objekt på vej ind i funktionsknuden er at dette skal være tilgængeligt når formelle parametre skal tilføjes.

Nedenfor ses et eksempel på første traversering af syntakstræet. Her møder dybdeførst-traverseringen en funktionserklæring, som traverseres ifølge figur 5.5.

```
function kuplen (a : int) : int
uses
...
does
...
```

Først kaldes *InAUsesImplSingle*, hvori funktionen bliver tilføjet bliver tilføjet til den globale symboltabel som et *Identifier*-objekt. Herefter mødes parametre og type, hvorefter disse tilføjes textitIdentifier-objektet, når *OutAUsesImplSingle*



Figur 5.5: Eksempel på traversering

5.3.2 Anden traversering

Vi har nu funktionernes navne, type og deres formelle parametre. Der skal nu udføres to opgaver:

- Når formelle parametre og variabelerklæringer mødes skal disse tilføjes til en hash-tabel for det lokale scope. Ved ethvert brug af en variabel skal det ses om denne findes i denne lokale tabel og derved det pågældende scope. Hvis ikke den gør det, er den ikke defineret og en fejl skal rapporteres. Derudover skal der ved funktionskald verificeres at funktionen eksisterer i den globale tabel og om den overholder den givne type.
- Det skal kontrolleres om de givne typeregler overholdes. Vi benytter her en hash-tabel givet fra SableCC til at gemme konstruktioner med en knude som nøgle og den tilhørende type som værdi.

I første traversering mødte vi klassen *Type* i klasse *Identifier*. *Type* indeholder et navn og en metode der kan benyttes til at se om to objekter er ens.

```
public abstract class Type
{
  public abstract string Name();
  public abstract bool Equal(Type t);
}
```

I vores implementation nedarver følgende klasser fra *Type*-klassen: *IntType*, *StringType*, *BoolType*, *StringListType*, *IntListType* og *BoolListType*.

Anden traversering skal som sagt indsamle alle lokale variable. Dette gøres ligesom i den første traversering af træet dog med den ændring at vi nu tjeker op mod både den lokale og den globale symboltabel for at se om et givet variabelnavn allerede er defineret. Et eksempel på indsamlingen ses her:

```
if (symbol_table_global.ContainsKey(node.GetIdent().Text))
{
  Console.WriteLine("Variable already used as function");
}
else if (symbol_table_local.ContainsKey(node.GetIdent().Text))
{
  Console.WriteLine("Variable already defined");
}
else
{
  symbol_table_local.Add(node.GetIdent().Text,
                         new Identifier(node.GetIdent(), leftType));

  SetOut(node.GetIdent(), GetOut(node.GetTypeSpecifier()));
}
```

I dybdeførst-traversering af det abstrakte syntakstræ tilføjes knuderne og deres type, som det blev nævnt tidligere, til en hash-tabel som SableCC stiller til rådighed. Dette gøres vha. metoden *void SetOut(Node node, Object inobj)*. På laveste niveau vil vi, hvis vi møder tal, strenge, lister og boolske værdier tilføje dem til tabellen med deres type som f.eks.

```
public override void OutALiteralTerm(ALiteralTerm node)
{
  SetOut(node, new IntType());
}
```

Hvis vi møder en *AAddExprArit*-knude (addition) trækker vi typerne af bestanddelene *factor* og *expr_arit* ud fra hash-tabellen vha. *object GetOut(Node node)*:

```
Type leftType = (Type) GetOut(node.GetFactor());
Type rightType = (Type) GetOut(node.GetExprArit());
```

Hvis typerne begge er heltal og den identifier, værdien tildeles til er markeret som et heltal, ved vi at typereglerne er overholdt.

Det er vigtigt at vi benytter *OutA⟨navn⟩*-funktionerne af følgende årsager. Besøges knuden `z := 2 + 3;` skal vi først aflæse typen af bestanddelene, før vi kan verificere tildelingen:

- Først findes typen af `3` og derefter typen af `2`. Da de begge er heltal er deres samlede type et heltal ifølge vores typeregler.

- Knuden `z` er af heltalstype og da `2 + 3` og er af heltalstype, er typereglerne overholdt.

5.4 Parallelisering af sekventiel kode

Vi har i klasse *DependencyAnalysis*[4] implementeret en analyse af afhængigheder som beskrevet i afsnit 4.2.2. Klassen nedarver som i identifikation og typecheck fra *DepthFirstAdapter*.
Lad os først præsentere et par hjælpekonstruktioner.

Structen *Usage* bruges til at illustrere hvor og hvordan en given variabel bliver brugt. I koden nedenfor ses de fire medlemmer, typen *UsageType* som kan have værdien *Read* eller *Write* som bruges til at identificere hvilken type den pågældende afhængighed er.

Heltallet *listindex* bliver brugt, hvis den pågældende variabel er et element i en liste (hvis værdien er -1, er der afhængighed over hele listen). Strengen *varname* bruges til variablens navn. Den sidste type *PCmdSingle* bruges til at identificere kommandoen som variablen er indeholdt i.

```
struct Usage
{
  UsageType type;
  string varname;
  int listIndex;
  PCmdSingle command;

  public Usage( string varname, UsageType type, PCmdSingle cmd )
  {
    this.varname = varname;
    this.type = type;
    this.command = cmd;
    this.listIndex = -1;
  }
  public Usage(string varname, UsageType type,
               PCmdSingle cmd, int listIndex)
    : this(varname, type, cmd)

  {
    this.listIndex = listIndex;
  }
  ... Properties til de forskellige members.
}
```

Lad os nu se på klassen *DependencyAnalysis*. I metoden *void DefaultIn(Node node)* angives, hvad der skal udføres på vej ind i en knude, hvis vi ikke angiver andet. I vores tilfælde skal knuden tilføjes til en hash-tabel som nøgle og en tom liste af *Usage*-objekter som værdi:

```
public override void DefaultIn(Node node)
{
   base.DefaultIn(node);
   this.SetIn(node, new List<Usage>());
}
```

Metoden *void SetIn(Node node, Object inobj)* lægger *node* i en hash-tabel som nøgle og *inobj* som værdi. For at tilgå *inobj* bruges metoden *object GetIn(Node node)*.

Ligeledes har vi i en metode angivet, hvad der skal ske når man går ud en en vilkårlig knude.

[4] Filen DependencyAnalysis.cs

```
public override void DefaultOut(Node node)
{
    List<Usage> parentList, myList;
    base.DefaultOut(node);

    // pass on the usages to the parent
    if( node.Parent() != null )
    {
        parentList = (List<Usage>)this.GetIn(node.Parent());
        myList = (List<Usage>)this.GetIn(node);

        foreach( Usage u in myList )
            parentList.Add(u);
    }
}
```

Hvis den pågældene knude har en forælder tilføjes dennes *Usage*-objekter til forælderens liste af *Usage*-objekter.

Vi udfører afhængighedsanalyse når en identifier bliver læst eller tilskrevet. Dette giver os fem typer af knuder som vi skal behandle. I hvert tilfælde udføres behandlingen på vej ud af knuden:

1. *AExprAssigment*: I dette tilfælde bliver knudens identifier tilskrevet. Et *Usage*-objekt med identifierens navn, type og den indeholdende *PCmdSingle* tilføjes til knudens liste:

   ```
   public override void  OutAExprAssignment(AExprAssignment node)
   {
       List<Usage> list = (List<Usage>)this.GetIn(node);
       string varname = node.GetIdent().Text;
       PCmdSingle cmd = containingSingleCommand(node);

       list.Add( new Usage( varname, UsageType.Write, cmd) );

       base.OutAExprAssignment(node);
   }
   ```

   Vi finder den indeholdende *PCmdSingle* for at vide hvilken kommando der "ejer" den pågældende knude. Dette findes via funktionen *PCmdSingle containingSingleCommand(Node node)* som kravler op i træet knudens indeholdende *PCmdSingle*.

2. *ListItemAssignment*: Samme som før da indgangen i listen aflæses. Hvis indekset er en konstant, gemmes dette også[5]; ellers sættes indekset til -1.

3. *AForLoop*: Iteratoren i for-løkken tilskrives. Samme princip som *AExprAssigment*.

4. *AIdentTerm*: Her aflæses identifieren. Samme princip som ovenfor men med *UsageType.Read*.

5. *ListItemTerm*: Adgangen i listen aflæses. Indekset gemmes hvis det er konstant.

Vi har nu indsamlet alle afhængigheder. Vha. metoden *List<Dependency> Dependencies(Node init, Node term)* kan vi få en liste med *Dependency*-objekter. Et *Dependency*-objekt er defineret som:

[5]Vi kan på dette tidspunkt ikke genkende f.eks. `2 + 2`, da vi ikke kan udregne resultatet.

```
struct Dependency
{
   DependencyType type;
   string varname;
   int listIndex;
   ...
```

hvor *DependencyType* er defineret som:

```
    enum DependencyType { Flow, Anti, Output };
```

Vi identificere afhængigheder i følgende klasse:

```
public List<Dependency> Dependencies(Node init, Node term)
{
  List<Dependency> deps = new List<Dependency>();
  List<Usage> initUsages = (List<Usage>)this.GetIn(init);
  List<Usage> termUsages = (List<Usage>)this.GetIn(term);

  foreach (Usage termU in termUsages)
    foreach (Usage initU in initUsages)
      if (termU.Varname == initU.Varname &&
         (termU.ListIndex == initU.ListIndex ||
          termU.ListIndex == -1 || initU.ListIndex == -1))

          // flow dep - write then read
          if (initU.Type == UsageType.Write &&
              termU.Type == UsageType.Read)

              deps.Add(new Dependency(initU.Varname,
                                      DependencyType.Flow));

          // anti dep - read then write
          else if (initU.Type == UsageType.Read &&
                   termU.Type == UsageType.Write)

                   deps.Add(new Dependency(initU.Varname,
                                           DependencyType.Anti));

          // output dep - write then write
          else if (initU.Type == UsageType.Write &&
                   termU.Type == UsageType.Write)

                   deps.Add(new Dependency(initU.Varname,
                                           DependencyType.Output));
  return deps;
}
```

Først hentes listerne af enhederne indeholdt i de to knuder, i form af *Usage* listerne for hver knude. Derefter sammenholdes hvert element i den ene liste, med hvert element i den anden liste for at bestemme eventuelle afhængigheder og deres type. Disse afhængigheder lagres i *deps*-listen, som *Dependency*-objekter og returneres når alle elementerne er sammenlignet.

5.5 Opdeling af sekventiel kode

I klassen *Parallelizer*[6], som også nedarver fra *DepthFirstAdapter*, bliver oplysningerne om afhængigheder udnyttet til at identificere dele af kildekode, der kan afvikles parallelt.

Parallelizer instantieres med et *DependencyAnalysis*-objekt, som vi blive udfyldt med afhængigheder i en traversering:

[6] I filen Parallelizer.cs

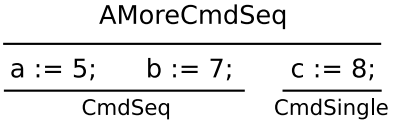
```
public override void InStart(Start node)
{
    node.Apply(this.depAnalysis);
    base.InStart(node);
}
```

Som det ses udføres afhængighedsanalysen på træet på vej ind i roden *Start*.

Som omtalt i afsnit 4.2.3 har vi valgt kun at parallelisere dele af sekventiel kildekode, hvis de er uafhængige. Dette er implementeret i metoden *void CaseAMoreCmdSeq(AMoreCmdSeq node)*. Når vi møder en sekvens af flere kommandoer (*AMoreCmdSeq node*) ser vi på bestandelene *ACmdSeq* og *ACmdSingle* som det ses i figur 5.6. Vi ser på to mulige tilfælde:

- *ACmdSeq* består af en kommando
  Det ses om kommandoen i *ACmdSeq* og *ACmdSingle*-knuden er afhængige af hinanden vha. *DependencyAnalysis*. Hvis ikke, køres disse parallelt vha. `par`-konstruktionen. Herefter erstattes *ACmdSeq*-knuden med kommandoerne kørt i parallelt.

- *ACmdSeq* består af flere kommandoer
  I figur 5.6 består *ACmdSeq* af flere kommandoer.



Figur 5.6: Opsplitning af en sekvens af kommandoer

  Det ses om den sidste kommando i *ACmdSeq* og *ACmdSingle*-knuden er afhængige; hvis de ikke er afhængige, køres disse i parallel og de to kommandoer udskiftes med en `par`-konstruktion.

5.6 Udskrift af kode

Vi har i implementationen af vores oversætter koncentreret os om analysen og paralleliseringen af koden eftersom vi, bortset fra en enkelt konstruktion, oversætter fra og til samme sprog.

Udskrift af kildekode håndteres af en klasse ved navn *Printer*, som ligesom de tidligere implementationer er specialiseringer af klassen *DepthFirstAdapter*. Under en dybdeførst-traversering af det abstrakte syntakstræ, skrives knuderne til en given fil.

Vi starter i implementationen med at initialisere de værktøjer der skal bruges, hvilket kan ses i følgende kode.

I følgende del af *Printer*-klassen initialiseres en *StreamWriter*, der bruges til at skrive til en fil og et heltal som bruges til at holde styr på indrykningen.

```
class Printer : DepthFirstAdapter
{
  System.IO.StreamWriter file;
  int currentIndentLevel;
  string indent;

public Printer(string path)
{
  this.file = System.IO.File.CreateText(path);
  this.indent = "\t";
  this.currentIndentLevel = 0;

  this.Print("#This Sep code is generated by the Sip2sep Printer#");
  this.PrintBreak();
}
...
```

Som det ses i metoden *DefaultCase(Node node)*, vist herunder, er den normale fremgangsmåde at knuden skrives, hvis det er et token.
CaseTLBrace(TLBrace node) sørger for at skrive krøllede parenteser, og inkrementere indrykningen. Hertil er der selvfølgelig en tilsvarende metode der dekrementerer indrykningen, og skriver en afsluttende krøllet parentes. Den sidste metode der vises, *CaseTPar*, bruges til at skrive `par`-kommandoer.

```
public override void DefaultCase(Node node)
{
  if( node is Token )
    this.Print(node);
}

public override void CaseTLBrace(TLBrace node)
{
  this.PrintBreak();
  base.CaseTLBrace(node);
  this.currentIndentLevel += 1;
}

public override void CaseTPar(TPar node)
{
  this.PrintBreak();
  base.CaseTPar(node);
}
...
```

KAPITEL

6

Fortolkeren

Vi har i de tidligere afsnit set på, hvordan vores oversætter er implementeret, og i dette afsnit vil vi se på fortolkeren.

6.1 Environment-store-modellen

Vores fortolker er en direkte implementation af semantikken præsenteret i afsnit 2.3. Vi starter ud med at beskrive modellen for den abstrakte maskine som udgør vores fortolker.

6.1.1 Lageret Store

Vi har implementeret **Store** som en *Dictionary*, hvori man slår en adresse op og får et *Value*-objekt.

```
Interpreter : DepthFirstAdapter
{
  Dictionary<int, Value> sto;
  Envl envl;

  public Interpreter()
  {
    this.envl = new Envl();
    this.sto = new Dictionary<int, Value>();
  }
  ...
```

6.1.2 Køretidsstakke Envl

Som det kan ses i kildekoden nedenfor repræsenterer *EnvlItem* en tupel bestående af et variabel- og funktionsenviroment. Denne tupel er konstrueret så vi har et samlet element at tilføje til køretidsstakken. Variabelenviroments $\mathbf{Env}_v$ er som det kan ses i koden nedenfor konstrueret som *Dictionary*-objekter, hvor man bruger variabelnavnet som nøgle og et heltal som værdi.

```
struct EnvlItem
{
  Dictionary<string,int> varEnv;
  Dictionary<string,Function> funcEnv;

  public EnvlItem( Dictionary<string,int> varEnv,
                   Dictionary<string,Function> funcEnv )
  {
    this.varEnv = varEnv;
    this.funcEnv = funcEnv;
  }
  ...
```

Implementationen af køretidstakke **Envl** benytter *EnvlItem*-objekter. Foruden stakke er *next*-elementet repræsenteret som metoden *GetNext* som er beskyttet af en binær semafor, så flere tråde ikke får samme plads i **Store**.

```
class Envl
{
  Stack<EnvlItem> stack;
  static Mutex next_lock = new Mutex();
  static int next = 0;

  public Envl()
  {
    this.stack = new Stack<EnvlItem>();
    EnvlItem top = new EnvlItem(new Dictionary<string, int>(),
                                new Dictionary<string, Function>());
    this.stack.Push(top);
  }
  ...
```

6.1.3 Funktionsenviroment

Klassen **Function** repræsenterer den 5-tupel som udgør en funktion:

$$(K, E_v, (p, .., p), Env_v, Env_f)$$

Klassen er opbygget at en erklæringsknude og en kommandoknude, som indeholder en kopi af den erklæring og kommando som er knyttet til funktionsnavnet. Derudover indeholder klassen også listen af parameternavne og variabel- og funktionsenviromentet knyttet til funktionsnavnet.

```
class Function
{
  Node command;
  Node declarations;
  List<string> parameterNames;
  Dictionary<string, int> varEnv;
  Dictionary<string, Function> funcEnv;

  public Function( Node command, Node declarations,
                   List<string> parameterNames,
                   Dictionary<string,int> varEnv,
                   Dictionary<string,Function> funcEnv )
  {
    this.command = command;
    this.declarations = declarations;
    this.parameterNames = parameterNames;
    this.varEnv = varEnv;
    this.funcEnv = funcEnv;
  }
  ...
```

6.2 Fortolkning

Fortolkningen vil ligesom den tidligere analyse foregå som en dybdeførst-traversering af det abstrakte syntakstræ.
Her er fremgangsmåde igen, at vi overskriver de forskellige metoder der kaldes under traverseringen. Dog overskriver vi kun *CaseA*-metoderne, da vi her kan kontrollere, hvordan de forskellige knuders bestanddele traverseres. Vi vil i dette afsnit gennemgå en delmængde af de implementerede metoder.

6.2.1 Funktionserklæring

Metoden *CaseAUsesImplSingle* som ses i koden nedenfor, bliver kaldet når traverseringen af syntakstræet møder en funktionserklæring. Det er en implementation af [kald]-reglerne fra semantikken.

Implementationen er som følger:

1. Navnet på den pågældende funktion hentes.
2. Funktionens kommandokrop tildeles til knuden *command*.
3. Funktionens deklarationskrop tildeles til knuden *declarations*.
4. Herefter apply'es traverseringen på parameterblokken, hvilket kalder en metode der sørger for at ligge informationerne omkring parametrene i en hash-tabel, så de kan hentes ud med *GetOut(Node)*
5. Nu tildeles den parameterliste der lige er blevet konstrueret i det forrige kald til listen *parameterNames*.
6. I de næste 2 kald oprettes der et variabel- og funktionsenviroment, som er en kopi af det øverste enviroment på enviromentstakken.
7. Alle disse informationer sammensættes nu til den 5-tupel der udgør en funktion og ligges på enviromentstakken.
8. Til sidst kaldes *updateFuncEnv* som svarer til funktionen $upd : \mathbf{Env}_f \Rightarrow \mathbf{Env}_f$ defineret i afsnit 2.3.3.

```
public override void CaseAUsesImplSingle(AUsesImplSingle node)
{
  string name = node.GetIdent().Text;
  Node command = node.GetCmdBlock();
  Node declarations = node.GetDefBlock();

  node.GetFormParamBlock().Apply(this);
  List<string> parameterNames = (List<string>)this.
                                GetOut(node.GetFormParamBlock());

  Dictionary<string, int> varEnv = new Dictionary<string, int>
                                        (this.envl.TopVarEnv);
  Dictionary<string, Function> funcEnv = new Dictionary
                                        <string, Function>
                                        (this.envl.TopFuncEnv);

  this.envl.TopFuncEnv.Add(name, new Function(command, declarations,
                                        parameterNames,
                                        varEnv, funcEnv));
  Interpreter.updateFuncEnv(this.envl.TopFuncEnv);
}

private static void updateFuncEnv(Dictionary<string, Function> funcEnv)
{
  foreach (Function f in funcEnv.Values)
  f.FuncEnv = new Dictionary<string,Function>(funcEnv);
}
```

6.2.2 Funktionskald

Da funktionskald er en ret stor implemtation er den her delt i to dele. Først vises den indledende del, der indsamler de nødvendige data.

1. Apply kaldes på parameterblokken, så denne bliver traverseret og dekoreret.
2. Parameterne hentes ud fra hash-tabellen

3. Navnet på funktionen gemmes.
4. Funktionensenvironmentet hentes ud fra environmentstakken.
5. Der hentes store adresser til funktionsparametrene og funktionens aliasvariabel.
6. Herefter instantiseres funktionens variabel- og funktionsenvironment ud fra det allerede eksisterende environment.

```
public override void CaseAFunccall(AFunccall node)
{
  node.GetAktParamBlock().Apply(this);
  List<Value> actualParameters = (List<Value>)
                                  this.GetOut(node.GetAktParamBlock());

  string name = node.GetIdent().Text;
  Function func = this.envl.TopFuncEnv[name];
  int addr = Envl.GetNext( actualParameters.Count + 1 );
  Dictionary<string, int> varEnv = new Dictionary<string, int>
                                                  (func.VarEnv);
  Dictionary<string, Function> funcEnv = new Dictionary
                                        <string, Function>
                                        (func.FuncEnv);
```

1. Nu skal vi have funktionens parametre tilføjet til funktionens variabelenviroment, men først checkes om det aktuelle parametre er en liste; hvis ikke, tilføjes variablen værdi til **Store**.
2. Hvis parameteren er en liste, hentes der adresser i store til alle listens elementer, hvorefter alle listens værdier og type kopieres over i **Store**.

```
  for (int i = 0; i < actualParameters.Count; i++)
  {
    varEnv.Add(func.ParameterNames[i], addr + i);
    if (actualParameters[i] is ListVal)
    {
      ListVal origList = (ListVal)actualParameters[i];
      int newElementsAddr = Envl.GetNext(origList.Count);
      ListVal newList = new ListVal(newElementsAddr, origList.Count);
      for (int j = 0; j < origList.Count; j++)
           this.addToSto(newElementsAddr + j, (Value)this.sto[origList.
                                                FirstAddr + j].Clone());

      this.addToSto(addr + i, newList);
     }
     else
       this.addToSto(addr + i, actualParameters[i]);
  }
```

1. Funktionens alias-variabel tilføjes til variabelenviromentet og funktionens værdi i **Store** sættes til *null*.
2. Funktionens variabel- og funktionsenviroment tilføjes til stakken.
3. Herefter traverseres variabelerklæringer og kommandoblokken, hvorved disse evalueres.
4. Til sidst dekoreres funktionens knude med dens værdi, hvorefter funktionens environment fjernes fra stakken.

```
  varEnv.Add(name, addr + actualParameters.Count);
  this.addToSto(addr + actualParameters.Count, null);
  this.envl.Push(varEnv, funcEnv);
  if (func.Declarations != null)
      func.Declarations.Apply(this);

  func.Command.Apply(this);
  addr = this.envl.TopVarEnv[name];
  Value value = this.sto[addr];
  this.SetOut(node, value);
  this.envl.RemoveTop();
}
```

6.2.3 `for`-løkke-konstruktionen

For at implementere transitionsreglen [for] fra semantikken, skal en del information indsamles og forskellige udtryk og kommandoer konstrueres.

Først konstrueres N `:=` U_{a1}`;`, så $N < U_{a2}$ og N `:=` $N+1$ `;`, hvorefter vi til sidst skal konstruere `while` $N < U_{a2}$ `do {` K N `:=` $N+1$ `; }`
Dette har vi implementeret på følgende måde:

1. Først konstrueres et *AExprAssignment*-objekt for N `:=` U_{a1}`;`, hvilket gøres ved at klone identifieren og expression-delen af kommando.

2. Herefter konstrueres en faktor-identifier som bruges i konstruktionen af en expression-identifier som så igen bruges i kontruktionen af et boolsk udtryk.

3. Nu skal vi så konstruere forøgelsen af N, hvilket er gjort ved følgende skridt. Først konstrueres en literal, som bruges til at konstruere en faktor, som igen bruges til at konstruere et aritmetisk udtryk som til sidst sammensættes til en addition, hvis værdi skal tildeles N i tildelingen *identIncrement*, som repræsenterer N `:=` U_{a1}. Bemærk at der ikke er semikolon efter denne tildeling, så det er endnu ikke en komplet kommando.

```
private void CaseAForLoopSemantiks(AForLoop node)
{
  AExprAssignment exprAss = new AExprAssignment((TIdent)node.
                                GetIdent().Clone(), new TAssign(":="),
                                (PExpr)node.GetLeft().Clone());

  ATermFactor factorIdent = new ATermFactor(new AIdentTerm(
                                (TIdent)node.GetIdent().Clone()));

  AOneExprArit exprIdent = new AOneExprArit(factorIdent);
  AExprBool boolExpr = new AExprBool(new TLPar("("), exprIdent, new
                           TBoolOpt("<"), ((AAritExpr)node.GetRight().
                           Clone()).GetExprArit(), new TRPar(")"));

  ALiteralTerm literalOne = new ALiteralTerm(new TLiteralNumb("1"));
  ATermFactor factorOne = new ATermFactor(literalOne);
  AAritExpr exprOne = new AAritExpr(new AOneExprArit(factorOne));

  AAddExprArit addExpr = new AAddExprArit((ATermFactor)factorIdent.
                             Clone(),new TAdd("+"), exprOne);
  AAritExpr expr = new AAritExpr(addExpr);

  AExprAssignment identIncrement = new AExprAssignment((TIdent)node.
                                     GetIdent().Clone(),
                                     new TAssign(":="), expr);
```

Nu da vi har indsamlet det nødvendige data og konstrueret bestanddelene til `while`-konstruktionen, kan vi konstruere denne.

1. Først konstruerer vi en kommando N `:=` U_{a1} `;` ud fra tildelingen *identIncrement*, hvorefter der tjekkes om `for`-løkkens kommandoblok er en *AMoreCmdBlock* eller en *AOneCmdBlock*.

2. Hvis kommandoen er en *AMoreCmdBlock* konstrueres en ny sekvens af kommandoer *cmdSeq* med kommandoen N `:=` U_{a1} `;` .

3. Hvis kommandoen er en *AOneCmdBlock*, konverteres denne først til en kommandosekvens, hvorefter N `:=` U_{a1} `;` tages med.

```
AAssignmentCmdSingle assCmd = new AAssignmentCmdSingle(
                                identIncrement, new TSemicolon(";"));
AMoreCmdSeq cmdSeq = null;

if ((PCmdBlock)node.GetCmdBlock() is AMoreCmdBlock)
    cmdSeq = new AMoreCmdSeq(((AMoreCmdBlock)node.GetCmdBlock().
                                Clone()).GetCmdSeq(), assCmd);
else if ((PCmdBlock)node.GetCmdBlock() is AOneCmdBlock)
{
  AOneCmdSeq oneCmdSeq = new AOneCmdSeq(((AOneCmdBlock)node.
                          GetCmdBlock().Clone()).GetCmdSingle());
  cmdSeq = new AMoreCmdSeq(oneCmdSeq, assCmd);
}
```

Herefter konstrueres der en ny kommandoblok, som indeholder den nylige konstruerede kommandosekvens *cmdSeq*. Som afslutning på denne `for`-løkke-konstruktion konstrueres en `while`-løkke ud fra de elementer vi har konstrueret, hvorpå denne traverseres for at fortolke `while`-løkken.

```
AMoreCmdBlock cmdBlock = new AMoreCmdBlock(new TLBrace("{"), cmdSeq,
                                            new TRBrace("}"));

AOneExprArit boolOneExprArit = new AOneExprArit(new
                                ABoolExprFactor(boolExpr));
AAritExpr boolExprArit = new AAritExpr(boolOneExprArit);
AWhileLoop whileLoop = new AWhileLoop(new TWhile("while"),
                        boolExprArit, new TDo("do"), cmdBlock);

exprAss.Apply(this);
whileLoop.Apply(this);
}
```

6.2.4 Aritmetisk udtryk

I de forrige eksempler har vi flere gange hentet en værdi ud af en hash-tabel SableCC stiller til rådighed, for at belyse hvad der sker og hvordan denne værdi er placeret i hash-tabellen gennemgåes her implementationen af det aritmetiske udtryk $U_1 + U_2$. Fremgangsmåden er følgende:

1. Først hentes de to værdier U_1 og U_2 ud af en hash-tabel via *GetOut* på den pågældende knude.

2. Herefter adderes de to tal.

3. Til sidst gemmes den nye værdi i hash-tabellen med knuden som nøgle.

```
public override void OutAAddExprArit(AAddExprArit node)
{
  int v1, v2;
  IntVal value;

  v1 = ((IntVal)this.GetOut(node.GetFactor())).Value;
  v2 = ((IntVal)this.GetOut(node.GetExpr())).Value;

  value = new IntVal(v1 + v2);

  this.SetOut(node, value);

  base.OutAAddExprArit(node);
}
```

6.2.5 Listeopslag

For at belyse hvordan lister er implementeret, vises her et eksempel på hvordan et listeelement tildeles en værdi.

1. Først aflæses de to værdier ud af en hash-tabel via *GetOut* på den pågældende knude.
2. Herefter aflæses navnet på listen og elementets placering i listen.
3. Listens navn bruges til at hente listens adresse i **Store**.
4. Til sidste bindes værdien *value* til elementets adresse i **Store**.

```
public void OutAListitemAssignment(AListitemAssignment node)
{
  Value value = (Value)this.GetOut(node.GetExpr());

  AListIdent ident = (AListIdent)node.GetListIdent();
  string name = ident.GetIdent().Text;
  int index = ((IntVal)this.GetOut(ident.GetExprArit())).Value;

  int addr = this.envl.TopVarEnv[name];
  addr = ((ListVal)this.sto[addr]).FirstAddr;
  this.sto[addr+index] = value;

  base.OutAListitemAssignment(node);
}
```

6.2.6 `par`-kommando

Til at afvikle `par`-kommandoer har vi udvidet *Interpreter*-klassen med en *CreateSibling*-metode, der returnerer et nyt *Interpreter*-objekt med en kopi af den oprindelige **Envl**, og som peger på det oprindelige **Store**. Til at håndtere tråde, har vi implementeret en *InterpreterThread*-klasse, der aggregerer C#'s *Thread*-klasse, men er bygget til at håndtere afvikling af en knude i vores AST. Den nye klasse skal derfor vide, hvilken knude den skal afvikle, og hvilket *Interpreter*-objekt der skal bruges.

Metoden *CaseAParCmdSingle* opretter en ny tråd, ved hjælp af *InterpreterThread*-klassen og afvikler den venstre kommandoblok i denne med et nyt *Interpreter*-objekt, ved at kalde metoden *Start*. Derefter afvikler den selv den højre kommandoblok og venter til tråden med den venstre kommandoblok er afsluttet, ved at kalde metoden *Join*, inden den returnerer. Implementeringen af `par` kan ses i nedenstående kildekode.

```
public override void CaseAParCmdSingle(AParCmdSingle node)
{
  InAParCmdSingle(node);

  Interpreter sibling = this.CreateSibling();
  InterpreterThread thread = new InterpreterThread(node.GetLeftCmd(),
                                                   sibling);

  thread.Start();

  node.GetRight().Apply(this);

  thread.Join();

  OutAParCmdSingle(node);
}
```

6.3 Forbedret fortolkning

Da vi i første omgang har fulgt semantikken slavisk i implementationen af fortolkeren, er nogle af konstruktionerne implementeret på en uhensigtsmæssig måde hvad angår effektivitet. Dette er forsøgt forbedret, som beskrevet i dette afsnit.

6.3.1 `for`-løkke

Vores tidligere implementation af en `for`-løkke følger reglen [for] i semantikken, og konverterer derfor en `for`-løkke til en ækvivalent `while`-løkke, som derefter foldes ud til et antal `if`-konstruktioner med indlejrede `while`-konstruktioner. I fortolkeren har det haft den effekt, at vi udvider det abstrakte syntakstræ under fortolkning, hvilket resulterer i et voksende brug af hukommelse. For at forbedre dette har vi brugt C#'s `for`-løkke til at implementere vores egen løkke med. Dette er gjort på følgende måde:

1. Traverseringen køres på *from*- og *to*-udtrykkene, som svarer til hhv. U_{a1} og U_{a2} i den abstrakte syntaks.
2. Herefter indsættes resultaterne i en *for*-løkke fra C#.
3. I løkkens krop, indsættes i's værdi i **Store** på den plads der er bundet til variablen i den oprindelige `for`-løkken fra **Sep** -konstruktionen, hvorefter kommandoen fra den oprindelige løkke fortolkes.

```
private void CaseAForLoopFast(AForLoop node)
{
  node.GetLeft().Apply(this);
  int from = ((IntVal)this.GetOut(node.GetLeft())).Value;

  node.GetRight().Apply(this);
  int to = ((IntVal)this.GetOut(node.GetRight())).Value;

  for (int i = from; i < to; i++)
  {
    int addr = this.envl.TopVarEnv[node.GetIdent().Text];
    this.sto[addr] = new IntVal(i);
    node.GetCmdBlock().Apply(this);
  }
}
```

KAPITEL
7

Forsøg med fortolker

Vi vil i dette afsnit præsentere og diskutere forsøg udført på den implementerede oversætter og fortolker. Formålet med disse forsøg er at måle en eventuel effektivitetsforøgelse i programmer ved afvikling af oversatte programmer med fortolkeren.

Forsøgene udføres ved at køre en algoritme skrevet i **Sip** igennem oversætteren, hvorefter de to programmer (fra hhv. **Sip** og **Sep**) køres igennem fortolkeren, mens kørselstiden måles. De to kørselstider vil blive sammenlignet og analyseret, hvorefter resultatet vil blive diskuteret i en konklusion.

Forsøgene i dette afsnit et udført på en computer med to processorkerner.

7.1 Algoritmer til forsøg

I dette afsnit præsenteres de algoritmer der vil danne grundlag for forsøget, samt beskrivelse af resultaterne.

- **Bubblesort:** Dette er en bubblesort-algoritme implementeret i **Sip** . Algoritmen tager en heltalsliste af størrelsen n som input, og returnerer denne sorteret i stigende orden. Da **Sip** ikke understøtter dynamiske listestørrelser er returtypens listestørrelse i funktionserklæringen også sat til n. Koden for algoritmen kan ses i bilag E.1.

  Når **Sip** koden køres igennem oversætteren, modificeres følgende del af algoritmen,

  ```
  A[i+1] := tmp;
  swapped := true;
  ```

  til dette:

  ```
  {
    A [ i + 1 ] := tmp ;
  }
  par
  {
    swapped := true ;
  }
  ```

- **Udregning af primtal:** Dette er en implementation af en algoritme der finder primtal ud fra en given startværdi. Funktionen tager et heltal *startCandidate* som input og udregner derefter de første *maxPrimes* primtal større end dette heltal. Tallene returneres i en liste af størrelse *maxPrimes*. Koden for algoritmen kan ses i bilag E.2.
  Når algoritmen oversættes, modificeres følgende del af algoritmen,

```
maxPrimes := 100;
candidate := startCandidate;
primesFound := 0;
```

til dette,

```
{

  {
    maxPrimes := 100 ;
  }
  par
  {
    candidate := startCandidate ;
  }
}
par
{
  primesFound := 0 ;
}
```

og følgende,

```
prime := 0;
whileLoopIter := candidate + 1;
```

til dette:

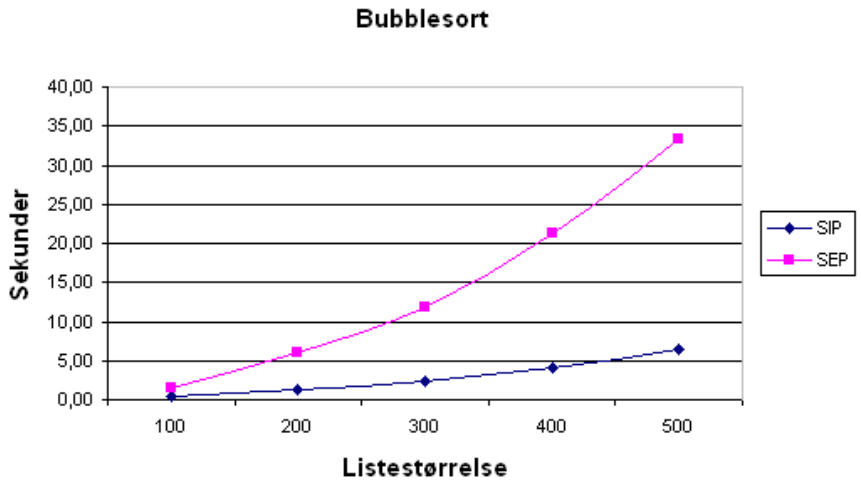
```
{
  prime := 0 ;
}
par
{
  whileLoopIter := candidate + 1 ;
}
```

7.2 Køretidssammenligning

Vi har i vores forsøg fortolket de ovenstående programmer. Outputtet er overført til en fil.

7.2.1 Bubblesort

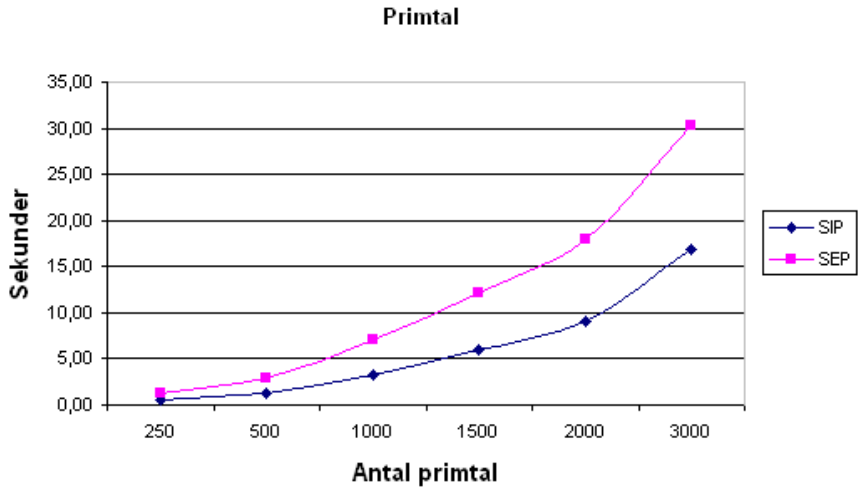
Nedenstående graf viser resultaterne fra fem forsøg på sortering af en liste af heltal på hhv. 100, 200, 300, 400 og 500 tal med bubblesort-algoritmen.



Tidsforskellen mellem **Sep** - og **Sip** -udgaven er voksende for voksende problemstørrelser, hvor **Sip** -udgaven i alle tilfælde er hurtigst.

7.2.2 Udregning af primtal

Den nedenstående graf viser resultatet af de to forsøg, hvor hver version af algoritmen til udregning af primtal er brugt til at finde hhv. 250, 500, 1000, 1500, 2000 og 3000 primtal.



Det kan ses at **Sep** -udgaven af algoritmerne tager væsentligt længere tid, end den samme implementation i **Sip** .

7.2.3 Parallele kald af bubblesort

Vi vil i dette forsøg kalde bubblesort-algoritmen to gange i træk for at se hvordan tidsmålingen vil blive, hvis vi kalder bubblesort-algoritmen to gange og tilskriver returværdien til to forskellige variable, for at undgå dataafhængig-

hed. Formålet med dette er at forsøge at parallelisere nogle større opgaver uden dataafhængighed.

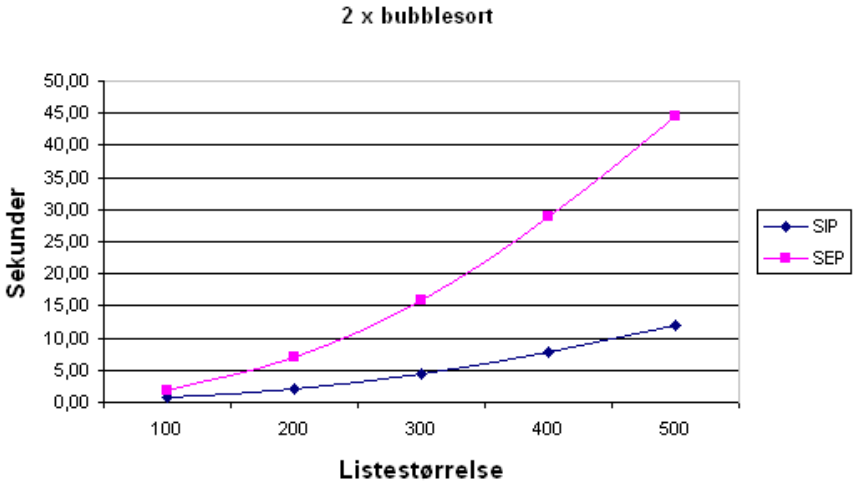
Udover de tidligere nævnte dele af programmet, er følgende del også blevet modificeret i oversættelsen:

```
L := bubbleSort ( L ) ;
M := bubbleSort ( M ) ;
```

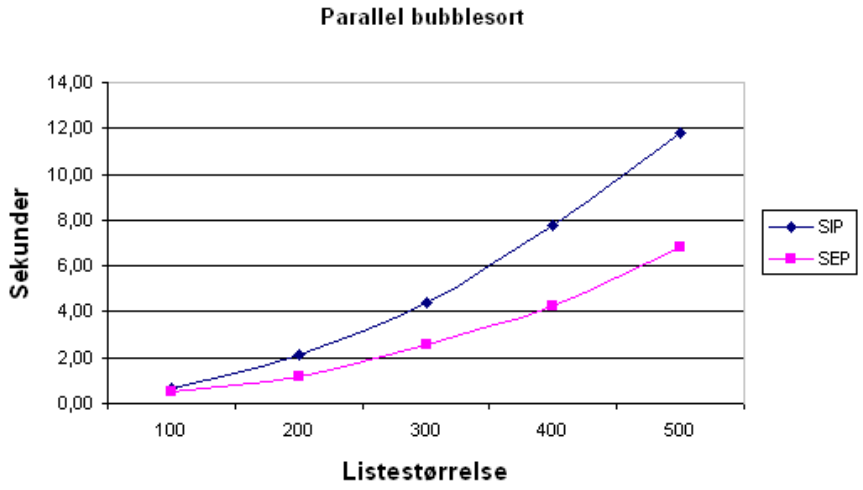
Til dette:

```
{
  L := bubbleSort ( L ) ;
}
par
{
  M := bubbleSort ( M ) ;
}
```

Nedenstående graf viser forsøgets resultater, ved voksende listestørrelser.



Sep -udgaven er stadig langsommere end **Sip** -udgaven. Vi har en teori om at det er de små parallelle konstruktioner, der er tilføjet inde i selve bubblesort-algoritmens løkke, der skaber stor forsinkelse. Baggrunden for denne teori er at fortolkeren opretter en ny tråd til en enkelt tilskrivning, for hver iteration af løkken. Da vi ikke mener dette er hensigtsmæssigt pga. af stor ventetid ift. størrelsen af opgaven, har vi lavet et forsøg hvor vi manuelt har fjernet denne `par`-kommando (beskrevet i afsnit 7.1). Resultatet af dette forsøg, med samme voksende listestørrelser, kan ses i nedenstående graf, hvor man tydeligt ser at **Sep** -programmet nu løser den samme opgave en del hurtigere end **Sip** -programmet.


Parallel bubblesort
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Sekunder
100
200
300
400
500
Listestørrelse
SIP
SEP

For yderligere at sandsynliggøre vores teori, har vi implementeret endnu et tænkt eksempel, hvor vi lader oversætteren parallelisere to simple variable tilskrivninger. Programmet, inkluderet i bilag E.3, indeholder følgende løkke:

```
for i from 0 to 20000 do
{
  a := i;
  b := i;
}
```

Der bliver oversat til følgende i **Sep** :

```
for i from 0 to 20000 do
{
  {
    a := i ;
  }
  par
  {
    b := i ;
  }
}
```

Resultatet af dette forsøg kan ses i følgende tidsmålinger, der viser hvor stor en effekt forsinkelsen på oprettelse af nye tråde kan have på simple opgaver.

Sip : *Interpreter.exe badParTest.sip* > *output.sip* 0,218750 sek

Sep : *Interpreter.exe badParTest.sep* > *output.sep* 4,437500 sek

Som det ses tager **Sep** -udgaven af algoritmen omkring 20 gange så lang tid som **Sip** -algoritmen, hvilket ikke er hensigtmæssigt.

7.3 Konklusion

Vi mener at vi efter det sidste forsøg har fået bekræftet vores påstand om, at det ikke altid kan betale sig at starte en ny tråd til en opgave, hvis denne er lille. Som paralleliseringen er implementeret nu, vil vi på den bekostning desværre ikke få den hastighedsforøgelse, som vi kunne have ønsket.

En løsning kunne være at opdele kildekoden i større dele som kan udføres samtidigt. Det vil kunne give et bedre forhold mellem opstartstiden for en tråd og køretiden for den kildekode der skal paralleliseres.

En anden mulighed er at udtænke en ny model for hvordan samtidighed håndteres i fortolkeren. Vi kunne f.eks. starte et antal tråde med det formål at aftage opgaver fra en kø. En sådan mængde af tråde kaldes en *thread pool*. Dette vil reducere tidsforbruget for opstart af tråd betydeligt.

Vi kunne fastlægge ordenen af opgaver i køen gennem afhængighedsanalysen eller man kunne implementere afhængighedsanalyse i run-time ved at lægge en opgave bagest i køen hvis den afhænger af noget der først udføres senere.

Kombineret med opdelingen af kildekode i større dele, formoder vi, at dette vil være et bedre bud på en effektiv fortolker, end den vi har implementeret nu.

Del V

Konklusion og perspektivering

7.4 Konklusion og perspektivering

Parallelitet er ikke en magisk egenskab ved multiprogrammering, som gør programmer hurtigere, hvilket vi har prøvet at belyse i denne rapport.

Den mest avancerede disciplin i forbindelse med parallelisering er afhængighedsanalyse, hvilket også forklarer hvorfor vi ikke har implementeret nogen form for fjernelse af afhængigheder. Vi berører kun toppen af isbjerget, mht. disciplinen at identificere dele af et program der kan afvikles parallelt. Vi har beskrevet og implementeret en metode til dette som med få ændringer vil kunne effektiviseres, ligesom vi på baggrund af metoderne i afhængighedsanalysen kan implementere en simpel form for afhængighedsfjernelse.

Det har voldt os en del problemer at definere et formelt mål for graden af parallelitet. Selv efter flere iterationer har vores nuværende bud stadig den svaghed, at det er så grov en approksimation, at det ikke har kunnet opfylde de egenskaber vi har ønsket for det.

Der er flere overvejelser, der taler imod parallelitet. Som vi tidligere har nævnt er det svært at programmere parallelt. Vi kunne så ty til det næstbedste, nemlig implicit parallelitet, men vores fornemmelse er at det som regel ikke er lige så hurtigt som eksplicit parallelitet, da programmøren stadig skal gøre sig tanker om hvad der kan udføres parallelt for ikke at tilføje falske afhængigheder. Vi mener derfor at udelukkende implicit parallelitet er utopi, hvilket også er tendensen blandt andre sprog med implicit parallelitet som f.eks. Fortress og Cω.

Vi har ved forsøg med fortolkeren, set at det nødvendigvis ikke er hensigtsmæssigt at implementere en semantik slavisk. Semantikken tager ikke forbehold for ineffektive genbrug af konstruktioner, som f.eks. brug af `while`-løkke i en `for`-løkke.

Desuden fandt vi ud af at vores model for fortolkeren er ineffektiv pga. det store overhead ved opstart af tråde, men den tjener det formål, at kunne afvikle programmer skrevet i **Sip** og **Sep** .

Del VI

Bilag

BILAG

A

Whileprograms i forhold til **Sip**

I dette bilag vil en given opbygningsregel først blive præsenteret i **while**programs gramatik, hvorefter en ækvivalent **Sip** -konstruktion bliver vist.

Tildelinger

Whileprogram

$$\langle \text{asgt} \rangle ::= \langle \text{variable} \rangle := 0 \mid \langle \text{variabel} \rangle := \mathit{succ}(\langle \text{variable} \rangle) \mid \langle \text{variable} \rangle := \mathit{pred}(\langle \text{variable} \rangle)$$

Sip

$$\langle \text{asgt} \rangle ::= N \texttt{ := 0 ; } \mid N_1 \texttt{ := } N_2 \texttt{ + 1 ; } \mid \texttt{if } N_2 \texttt{ > 0 then } N_1 \texttt{ := } N_2 \texttt{ - 1 ; else } N_1 \texttt{ := 0 ;}$$

Udtryk

Whileprogram

$$\langle \text{test} \rangle ::= \langle \text{variable} \rangle \neq \langle \text{variabel} \rangle$$

Sip

$$\langle \text{test} \rangle ::= N_1 \texttt{ > } N_2 \texttt{ or } N_1 \texttt{ < } N_2$$

Kommandoer

Whileprogram

$$\langle \text{statement} \rangle ::= \langle \text{asgt} \rangle \mid \textbf{while } \langle \text{test} \rangle \textbf{ do } \langle \mathit{statement} \rangle \mid \langle \mathit{program} \rangle$$

Sip

$$\langle \text{statement} \rangle ::= \langle \text{asgt} \rangle_{\textbf{Sip}} \mid \texttt{while } \langle \text{test} \rangle_{\textbf{Sip}} \texttt{ do } \langle \mathit{statement} \rangle_{\textbf{Sip}} \mid \langle \mathit{program} \rangle_{\textbf{Sip}}$$

Programmer

Whileprogram

$$\langle \text{program} \rangle ::= \textbf{begin end} \mid \textbf{begin } \langle \text{statement sequence} \rangle \textbf{ end}$$

Sip

$$\langle \text{program} \rangle ::= \texttt{\{ \}} \mid \texttt{\{ } \langle \text{statement sequence} \rangle_{\textbf{Sip}} \texttt{ \}}$$

Kommandosekvens

Whileprogram

$$\langle \text{statement sequence} \rangle ::= \langle \text{statement} \rangle \mid \langle \text{statement sequence} \rangle \langle \text{statement} \rangle$$

Sip

$$\langle \text{statement sequence} \rangle ::= \langle \text{statement} \rangle_{\textbf{Sip}} \mid \langle \text{statement sequence} \rangle_{\textbf{Sip}} \langle \text{statement} \rangle_{\textbf{Sip}}$$

BILAG

B

Grammatik

Vi definere i dette afsnit en kontekstfri grammatik, efterflugt af den konkrete syntaks for **Sep** , som SableCC-grammatik.

Definition 1.16. *En kontekstfri grammatik G består af 4 elementer:*

- *Terminal-symboler: symboler som kan indtastes*
- *Nonterminal-symboler: syntaktiske klasser*
- *Startsymbol: den ledende syntaktiske klasse*
- *Produktionsregler: regler for hvordan syntaktiske klasser kan konstrueres af terminalsymboler.*

Lad os se et eksempel på en kontekstfri grammatik.

Eksempel 1.5. *Lad os betragte en kontekstfri grammatik.*

Program	::=	`begin` *Kommando* `end`
Kommando	::=	*Navn := Udtryk* | *Kommando ; Kommando*
Navn	::=	*Bogstav* | *Navn Ciffer* | *Navn Bogstav*
Udtryk	::=	*HeltalLiteral* | *HeltalLiteral + HeltalLiteral*
HeltalLiteral	::=	*Ciffer* | *HeltalLiteral Ciffer*
Bogstav	::=	*'a'* | *'b'* | ... | *'z'*
Ciffer	::=	*'0'* | *'1'* | ... | *'9'*

Sammenholdt med definition 1.16 fås:

- *Terminal-symboler:* `begin`, `end` *og* `;`
- *Nonterminal-symboler: Program, Kommando, Navn, Udtryk, HeltalLiteral, Bogstav og Tal*
- *Startsymbol: Program*
- *Produktionsregler: reglerne i tabellen*

F.eks. vil strengen `begin a := 2 ; b := 2 end` *være gyldig i grammatikken.*

Strengen `begin a := 2 ; b := 2 end` fra eksemplet kan *deriveres* som følger:

Program	::=	`begin` Kommando `end`
	::=	`begin` Kommando ; Kommando `end`
	::=	`begin` Navn := Udtryk ; Navn := Udtryk `end`
	::=	`begin a := 2 ; b := 2 end`

Hvis vi kan derivere en streng på flere forskellige måder, siges grammatikken at være tvetydig.

Produktionsregler i eksemplet er skrevet i *Backus-Naur-formalisme* (BNF) som følger formen '$N ::= a|b$' som betyder at nonterminal-symbolet N kan skrives som a eller b, hvor a og b er en streng terminal- og nonterminal-symboler.

Vi kan også udtrykke grammatikken i *udvidet* BNF (EBNF), hvilket blander BNF og regulære udtryk. Vi kan så udtrykke Navn fra ovenstående eksempel som

$$\text{Navn} ::= \text{Bogstav}^{+}(\text{Ciffer} \mid \text{Bogstav})^{*},$$

hvor * og $^{+}$ følger de sædvanlige definitioner.

I omskrivningen af Navn fra BNF til EBNF har vi fjernet såkaldt *venstrerekursion*, hvilket findes i udtryk på formen '$N ::= X|NY$'. Dette kan substitueres af udtrykket $N ::= X(Y)^{*}$. Vi kan også *venstrefaktorisere*; udskifte udtryk på formen '$N ::= XY|XZ$' med udtrykket $N ::= X(Y|Z)$'.

B.1 SableCC

Vi skal benytte en speciel form for notation når vi opbygger grammatikken til SableCC. Vi skal i nedenstående rækkefølge tilføje de konkrete definitioner for sproget:

- *Package*: navnet på det *Namespace* (eller Java package) klasserne skal oprettes i.
- *Helpers*: simple konstruktioner, som f.eks. tal, bogstaver og kommentarer, der kan bruges i definitionen af tokens.
- *Tokens*: Reserverede ord, operatorer, tegn osv.
- *Ignored Tokens* Tokens der skal ignoreres. I **Sep** er det whitespaces, EOL og kommentarer.
- *Productions* Produktionsregler. Ved tilfælde med flere cases for en syntaktisk kategori, skal navnet for hvert tilfælde have et navn skrevet i {}.

Følgende er den konkrete syntaks for **Sep** i SableCC grammar (også inkluderet i filen SEP.GRAMMAR).

```
Package sip;

Helpers
  all = [1..127];
  letter = ['a'..'z'] | ['A'..'Z'];
  digit = ['0'..'9'];
  nonzero_digit = [digit - '0'];
  sign = ('+' | '-' | '*' | '/' | '<' | '>' | '=');

  cr = 13;
  lf = 10;
  tab = 9;
  eol = cr | lf | cr lf;

  not_hash = [all - '#'];

Tokens
  function = 'function';
  uses = 'uses';
  does = 'does';


  if = 'if';
  then = 'then';
  else = 'else';

  while = 'while';
  for = 'for';
  from = 'from';
  to = 'to';
  foreach = 'foreach';
  in = 'in';
  do = 'do';
  true = 'true';
  false = 'false';
  skip = 'skip';
  par = 'par';

  neg = 'not';

  semicolon = ';';
  comma = ',';
  colon = ':';
  doublequote = '"';
  l_par = '(';
  r_par = ')';
  l_bracket = '[';
  r_bracket = ']';
  l_brace = '{';
  r_brace = '}';

  int = 'int';
  bool = 'bool';
  string = 'string';
  list = 'list';

  and = 'and';
  or = 'or';
  add = '+';
  sub = '-';
  mult = '*';
  div = '/';
  mod = 'mod';
  bool_opt = ('<' | '>' | '=');

  assign = ':=';
  literal_string = '"' (digit | letter | ' ')* '"';
  literal_numb = '-'? digit+;
  ident = ( letter ) ( letter | digit )*;
  nonzero_digits = (nonzero_digit) (digit)*;

  blank = ( eol | tab | ' ')+;
  comment = '#' not_hash* '#';
```

```
Ignored Tokens
  blank,
  comment;

Productions
  program = impl_seq;

  impl_single = {uses}   function ident form_param_block type_specifier
                         uses def_block does cmd_block |
                {nouses} function ident form_param_block type_specifier
                         does cmd_block;

  impl_seq = {one}  impl_single |
             {more} impl_seq impl_single;

  type_specifier = {int}         colon int |
                   {bool}        colon bool |
                   {string}      colon string |
                   {int_list}    colon int list l_par literal_numb
                                 r_par |
                   {bool_list}   colon bool list l_par literal_numb
                                 r_par |
                   {string_list} colon string list l_par literal_numb
                                 r_par;

  funccall = ident akt_param_block;

  def_single = ident type_specifier semicolon;

  def_seq = {one}  def_single |
            {more} def_seq def_single;

  def_block = {one}  def_single |
              {more} l_brace def_seq r_brace;

  form_param_single = ident type_specifier;

  akt_param_single = expr;

  form_param_seq = {one}  form_param_single |
                   {more} form_param_seq comma form_param_single;

  form_param_block = {empty}   l_par r_par |
                     {noempty} l_par form_param_seq r_par;

  akt_param_seq = {one}  akt_param_single |
                  {more} akt_param_seq comma akt_param_single;

  akt_param_block = {empty}   l_par r_par |
                    {noempty} l_par akt_param_seq r_par;

  cmd_single = {assignment} assignment semicolon |
               {statement}  statement |
               {skip}       skip semicolon |
               {loop}       loop |
               {par}        [ll_brace]:l_brace [left_cmd]:cmd_seq
                            [lr_brace]:r_brace par [rl_brace]:l_brace
                            [right]:cmd_seq [rr_brace]:r_brace;

  cmd_seq = {one}  cmd_single |
            {more} cmd_seq cmd_single;

  cmd_block = {one}  cmd_single |
              {more} l_brace cmd_seq r_brace;

  assignment = {expr}     ident assign expr |
               {listitem} list_ident assign expr;

  statement = if expr then [tt]:cmd_block else [ff]:cmd_block;

  loop = {while} while expr do cmd_block |
         {for}   for ident from [left]:expr to [right]:expr do cmd_block;
```

```
expr = {arit}   expr_arit |
       {string} literal_string;

expr_seq = {one}  expr |
           {more} expr_seq comma expr;

expr_arit = {add} factor add expr |
            {sub} factor sub expr |
            {and} factor and expr |
            {or}  factor or expr |
            {one} factor;

expr_bool = l_par [left]:expr_arit bool_opt [right]:expr_arit r_par;

factor = {term}      term |
         {mult}      factor mult term |
         {div}       factor div term |
         {mod}       factor mod term |
         {bool_expr} expr_bool ;

term = {ident}          ident |
       {literal}        literal_numb |
       {expr}           l_par expr r_par |
       {funccall}       funccall |
       {bool_lit_false} false |
       {bool_lit_true}  true |
       {listitem}       list_ident ;

list_ident = ident l_bracket expr_arit r_bracket;
```

BILAG

C

Transitionsregler fra semantikken

I dette afsnit præsenteres den fuldstændige semantik for **Sip** .

C.1 Transitionsregler funktionserklæringer $\Rightarrow_f$

[sekvens] $$\frac{\langle E_{f1}, sto, envl\rangle \Rightarrow_f \langle sto', envl'\rangle}{\langle E_{f1}\ E_{f2}, sto, envl\rangle \Rightarrow_f \langle E_{f2}, sto', envl'\rangle}$$

[erk-m-uses] $\langle$`function` $N_f(N_1 : T_1, \ldots, N_n : T_n)$ `uses` E_v `does` $K, sto, envl\rangle \Rightarrow_f \langle sto, envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env'_f = env_f[N \mapsto (K, E_v, (N_1, \ldots, N_n), env_v, env_f]$
og $envl' = (env_v, upd(env'_f)) : envl''$

[erk-u-uses] $\langle$`function` $N_f(N_1 : T_1, \ldots, N_n : T_n)$ `does` $K, sto, envl\rangle \Rightarrow_f \langle sto, envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env'_f = env_f[N \mapsto (K, \emptyset, (N_1, \ldots, N_n), env_v, env_f]$
og $envl' = (env_v, upd(env'_f)) : envl''$

C.2 Transitionsregler for kommandoer $\Rightarrow_k$

[sekvens-1] $$\frac{\langle K_1, sto, envl\rangle \Rightarrow_k \langle K_1', sto', envl\rangle}{\langle K_1\ K_2, sto, envl\rangle \Rightarrow_k \langle K_1'\ K_2, sto', envl\rangle}$$

[sekvens-2] $$\frac{\langle K_1, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle}{\langle K_1\ K_2, sto, envl\rangle \Rightarrow_k \langle K_2, sto', envl\rangle}$$

[tildel-1] $$\frac{\langle U, sto, envl\rangle \Rightarrow_u \langle U', sto, envl\rangle}{\langle N\ \texttt{:=}\ U\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle N\ \texttt{:=}\ U'\ \texttt{;}\ , sto, envl\rangle}$$

[tildel-2] $\langle N\ \texttt{:=}\ v\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle sto[a \mapsto v], envl\rangle$
hvor $envl = (env_V, env_F) : envl'$
og $a = env_V(N)$

[tildel-3] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U_a', sto, envl\rangle}{\langle N\ \texttt{[}\ U_a\ \texttt{]}\ \texttt{:=}\ U\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle N\ \texttt{[}\ U_a'\ \texttt{]}\ \texttt{:=}\ U\ \texttt{;}\ , sto, envl\rangle}$$

[tildel-4] $$\frac{\langle U, sto, envl\rangle \Rightarrow_u \langle U', sto, envl\rangle}{\langle N\ \texttt{[}\ v\ \texttt{]}\ \texttt{:=}\ U\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle N\ \texttt{[}\ v\ \texttt{]}\ \texttt{:=}\ U'\ \texttt{;}\ , sto, envl\rangle}$$

[tildel-5] $$\frac{\langle U, sto, envl\rangle \Rightarrow_u \langle U', sto, envl\rangle}{\langle N\ \texttt{[}\ v_1\ \texttt{]}\ \texttt{:=}\ v_2\ \texttt{;}\ , sto, envl\rangle \Rightarrow_k \langle sto[a \mapsto v], envl\rangle}$$
, hvor $envl = (env_V, env_F) : envl'$
og $a = env_V(N) + v_1$

[paran-1] $$\frac{\langle K, sto, envl\rangle \Rightarrow_k \langle K', sto', envl\rangle}{\langle\texttt{\{}\ K\ \texttt{\}}, sto, envl\rangle \Rightarrow_k \langle\texttt{\{}\ K'\ \texttt{\}}, sto', envl\rangle}$$

[paran-2] $$\frac{\langle K, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle}{\langle\texttt{\{}\ K\ \texttt{\}}, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle}$$

[if] $$\frac{\langle U_b, sto, envl\rangle \Rightarrow_b \langle U_b', sto, envl\rangle}{\langle\texttt{if}\ U_b\ \texttt{then}\ K_1\ \texttt{else}\ K_2, sto, envl\rangle \Rightarrow_k \langle\ \texttt{if}\ U_b'\ \texttt{then}\ K_1\ \texttt{else}\ K_2, sto, envl\rangle}$$

[if-true] $\langle\texttt{if true then}\ K_1\ \texttt{else}\ K_2, sto, envl\rangle \Rightarrow_k \langle K_1, sto, envl\rangle$

[if-false] $\langle\texttt{if false then}\ K_1\ \texttt{else}\ K_2, sto, envl\rangle \Rightarrow_k \langle K_2, sto, envl\rangle$

[for] $\langle\texttt{for}\ N\ \texttt{from}\ U_{a1}\ \texttt{to}\ U_{a2}\ \texttt{do}\ K, sto, envl\rangle \Rightarrow_k$
$\langle N\ \texttt{:=}\ U_{a1}\texttt{;}\ \ \texttt{while}\ N < U_{a2}$
$\texttt{do \{}\ K\ N\ \texttt{:=}\ N+1\texttt{; \}}, sto, envl\rangle$

[while] $\langle\texttt{while}\ U_b\ \texttt{do}\ K, sto, envl\rangle \Rightarrow_k$
$\langle\texttt{if}\ U_b\ \texttt{then \{}\ K\ \texttt{while}\ U_b\ \texttt{do}\ K\ \texttt{\} else skip;}, sto, envl\rangle$

C.3 Transitionsregler for variabelerklæringer $\Rightarrow_v$

[sekvens] $$\frac{\langle E_{v1}, sto, envl\rangle \Rightarrow_v \langle sto', envl'\rangle}{\langle E_{v1}\ E_{v2}, sto, envl\rangle \Rightarrow_v \langle E_{v2}, sto', envl'\rangle}$$

[var] $\langle N\ :\ T, sto, envl\rangle \Rightarrow_v \langle sto[a \mapsto default(T)], envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $a = env_V(next)$
og $envl' = (env_v[N \mapsto a][next \mapsto a+1], env_f) : envl''$

[list-1] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle N\ :\ \texttt{list}\ (U_a), sto, envl\rangle \Rightarrow_v \langle N\ :\ \texttt{list}\ (U'_a), sto, envl\rangle}$$

[list-2] $\langle N\ :\ T\ \texttt{list}\ (v), sto, envl\rangle \Rightarrow_v \langle sto', envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env'_v = env_v[N \mapsto a][next \mapsto a+v]$
og $splitsto' = sto[a \mapsto default(T)][a+1 \mapsto default(T)] \ldots$
$[a+v \mapsto default(T)]$
og $envl' = (env'_v, env_f) : envl''$

C.4 Transitionsregler for arimetiske udtryk $\Rightarrow_a$

[add-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto, envl\rangle}{\langle U_{a1} + U_{a2}, sto, envl\rangle \Rightarrow_a \langle U'_{a1} + U_{a2}, sto, envl\rangle}$$

[add-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle v + U_a, sto, envl\rangle \Rightarrow_a \langle v + U'_a, sto, envl\rangle}$$

[add-3] $\langle v_1 + v_2, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $v = v_1 + v_2$

[sub-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto, envl\rangle}{\langle U_{a1} - U_{a2}, sto, envl\rangle \Rightarrow_a \langle U'_{a1} - U_{a2}, sto, envl\rangle}$$

[sub-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle v - U_a, sto, envl\rangle \Rightarrow_a \langle v - U'_a, sto, envl\rangle}$$

[sub-3] $\langle v_1 - v_2, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $v = v_1 - v_2$

[mul-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto, envl\rangle}{\langle U_{a1} \texttt{ * } U_{a2}, sto, envl\rangle \Rightarrow_a \langle U'_{a1} \texttt{ * } U_{a2}, sto, envl\rangle}$$

[mul-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle v \texttt{ * } U_a, sto, envl\rangle \Rightarrow_a \langle v \texttt{ * } U'_a, sto, envl\rangle}$$

[mul-3] $\langle v_1 \texttt{ * } v_2, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $v = v_1 \cdot v_2$

[div-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto, envl\rangle}{\langle U_{a1} \texttt{ / } U_{a2}, sto, envl\rangle \Rightarrow_a \langle U'_{a1} \texttt{ / } U_{a2}, sto, envl\rangle}$$

[div-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle v \texttt{ / } U_a, sto, envl\rangle \Rightarrow_a \langle v \texttt{ / } U'_a, sto, envl\rangle}$$

[div-3] $\langle v_1 \texttt{ / } v_2, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $v = \left\lfloor \frac{v_1}{v_2} \right\rfloor$

[mod-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto, envl\rangle}{\langle U_{a1} \texttt{ mod } U_{a2}, sto, envl\rangle \Rightarrow_a \langle U'_{a1} \texttt{ mod } U_{a2}, sto, envl\rangle}$$

[mod-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle v \texttt{ mod } U_a, sto, envl\rangle \Rightarrow_a \langle v \texttt{ mod } U'_a, sto, envl\rangle}$$

[mod-3] $\langle v_1 \texttt{ mod } v_2, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $v = v_1 - v_2 \left\lfloor \frac{v_1}{v_2} \right\rfloor$

[paran-1] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto', envl\rangle}{\langle (U_a), sto, envl\rangle \Rightarrow_a \langle (U'_a), sto', envl\rangle}$$

[paran-2] $\langle (v), sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$

[opslag-1] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto, envl\rangle}{\langle N[U_a], sto, envl\rangle \Rightarrow_a \langle N[U_a], sto, envl\rangle}$$

[opslag-2] $\langle N[v_1], sto, envl\rangle \Rightarrow_a \langle v_2, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl'$
og $sto(env_v(N) + v_1) = v_2$

[var] $\langle N, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl'$
og $sto(env_V(N)) = v$

[lit] $\langle L_a, sto, envl\rangle \Rightarrow_a \langle v, sto, envl\rangle$
hvor $\mathcal{N}(L_a) = v$

[kald-1-m-uses] $\langle N(U_{p_1}, \ldots, U_{p_n}), sto, envl\rangle \Rightarrow_a$
$\langle \|U_{p_1}\|; \ldots; \|U_{p_n}\|; \|E_v\|; \|K_N\|, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env_f(N) = (K_N, E_v, (p_1, \ldots, p_n), env'_v, env'_f)$
og $E_v \neq \emptyset$

[kald-1-u-uses] $\langle N(U_{p_1}, \ldots, U_{p_n}), sto, envl\rangle \Rightarrow_a$
$\langle \|U_{p_1}\|; \ldots; \|U_{p_n}\|; \|K_N\|, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env_f(N) = (K_N, \emptyset, (p_1, \ldots, p_n), env'_v, env'_f)$

$$\langle U_{p_1}, sto, envl\rangle \Rightarrow \langle U'_{p_1}, sto', envl'\rangle$$

[kald-2]
$$\frac{}{\langle \|U_{p_1}\|;\ldots;\|U_{p_n}\|;\|E_v\|;\|K_N\|, sto', envl'\rangle \Rightarrow_a \langle \|U'_{p_1}\|;\|U_{p_2}\|,\ldots,\|U_{p_n}\|;\|K_N\|, sto', envl'\rangle}$$

$$\langle U_{p_{m+1}}, sto, envl\rangle \Rightarrow \langle U'_{p_{m+1}}, sto', envl'\rangle$$

[kald-3]
$$\frac{}{\langle \|v_{p_1}\|;\ldots;\|v_{p_m}\|;\|U_{p_{m+1}}\|;\ldots;\|U_{p_n}\|;\|E_v\|;\|K_N\|, sto, envl\rangle \Rightarrow_a \langle \|v_{p_1}\|;\ldots;\|v_{p_m}\|;\|U'_{p_{m+1}}\|;\|U_{p_{m+2}}\|,\ldots,\|U_{p_n}\|;\|E_v\|;\|K_N\|, sto', envl'\rangle}$$

[kald-4]
$$\frac{\langle \|v_{p_1}\|;\ldots;\|v_{p_n}\|;\|E_v\|;\|K_N\|, sto, envl\rangle \Rightarrow_a \langle \|E_v\|;\|K_N\|, sto', envl'\rangle}{\begin{array}{l}\text{hvor } envl = (env_v, env_f) : envl'' \\ \text{og } env_f(N) = (K, E_v, env'_v, env'_f) \\ \text{og } envl' = (env'_v, env'_f) : envl \\ \text{og } a = env_v(next) \\ \text{og } env'_v = env_v[p_1 \mapsto a+0]\ldots[p_n \mapsto a+n-1][N \mapsto a+n][next \mapsto a+n+1] \\ \text{og } sto' = sto[a+0 \mapsto v_{p_1}]\ldots[a+n-1 \mapsto v_{p_n}]\end{array}}$$

$$\langle E_v, sto, envl\rangle \Rightarrow_v \langle E'_v, sto', envl'\rangle$$

[kald-5]
$$\frac{}{\langle \|E_v\|;\|K_N\|, sto, envl\rangle \Rightarrow_a \langle \|E'_v\|;\|K_N\|, sto', envl'\rangle}$$

$$\langle E_v, sto, envl\rangle \Rightarrow_v \langle sto', envl'\rangle$$

[kald-6]
$$\frac{}{\langle \|E_v\|;\|K_N\|, sto, envl\rangle \Rightarrow_a \langle \|K_N\|, sto', envl'\rangle}$$

$$\langle K_N, sto, envl\rangle \Rightarrow_k \langle K'_N, sto', envl\rangle$$

[kald-7]
$$\frac{}{\langle \|K_N\|, sto, envl\rangle \Rightarrow_a \langle \|K'_N\|, sto', envl\rangle}$$

$$\langle K_N, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle$$

[kald-8]
$$\frac{}{\langle \|K_N\|, sto, envl\rangle \Rightarrow_a \langle v, sto', envl'\rangle}$$
hvor $envl = (env_v, env_f) : envl'$
og $v = sto'(env_v(N))$

C.5 Transitionsregler for boolske udtryk $\Rightarrow_b$

[mindre-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto', envl'\rangle}{\langle U_{a1} < U_{a2}, sto, envl\rangle \Rightarrow_b \langle U'_{a1} < U_{a2}, sto', envl'\rangle}$$

[mindre-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto', envl'\rangle}{\langle v < U_a, sto, envl\rangle \Rightarrow_b \langle v < U'_a, sto', envl'\rangle}$$

[mindre-3] $\langle v_1 < v_2, sto, envl\rangle \Rightarrow_b \langle \texttt{true}, sto, envl\rangle$
hvis $v_1 < v_2$

[mindre-4] $\langle v_1 < v_2, sto, envl\rangle \Rightarrow_b \langle \texttt{false}, sto, envl\rangle$
hvis $v_1 \geq v_2$

[større] $\langle U_{a1} > U_{a2}, sto, envl\rangle \Rightarrow_b \langle U'_{a2} < U_{a1}, sto, envl\rangle$

[lig-1] $$\frac{\langle U_{a1}, sto, envl\rangle \Rightarrow_a \langle U'_{a1}, sto', envl'\rangle}{\langle U_{a1} = U_{a2}, sto, envl\rangle \Rightarrow_b \langle U'_{a1} = U_{a2}, sto', envl'\rangle}$$

[lig-2] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_a \langle U'_a, sto', envl'\rangle}{\langle v = U_a, sto, envl\rangle \Rightarrow_b \langle v = U'_a, sto', envl'\rangle}$$

[lig-3] $\langle v_1 = v_2, sto, envl\rangle \Rightarrow_b \langle \texttt{true}, sto, envl\rangle$
hvis $v_1 = v_2$

[lig-4] $\langle v_1 = v_2, sto, envl\rangle \Rightarrow_b \langle \texttt{false}, sto, envl\rangle$
hvis $v_1 \neq v_2$

[og-1] $$\frac{\langle U_{b1}, sto, envl\rangle \Rightarrow_b \langle U'_{b1}, sto', envl'\rangle}{\langle U_{b1} \texttt{ and } U_{b2}, sto, envl\rangle \Rightarrow_b \langle U'_{b1} \texttt{ and } U_{b2}, sto', envl'\rangle}$$

[og-2] $$\frac{\langle U_b, sto, envl\rangle \Rightarrow_b \langle U'_b, sto', envl'\rangle}{\langle v \texttt{ and } U_b, sto, envl\rangle \Rightarrow_b \langle v \texttt{ and } U'_b, sto', envl'\rangle}$$

[og-3] $\langle \texttt{true and true}, sto, envl\rangle \Rightarrow_b \langle \texttt{true}, sto, envl\rangle$

[og-4] $\langle \texttt{true and ff}, sto, envl\rangle \Rightarrow_b \langle \texttt{false}, sto, envl\rangle$

[og-5] $\langle \texttt{false and } v, sto, envl\rangle \Rightarrow_b \langle \texttt{false}, sto, envl\rangle$

[eller-1] $$\frac{\langle U_{b1}, sto, envl\rangle \Rightarrow_b \langle U'_{b1}, sto', envl'\rangle}{\langle U_{b1}\ \texttt{or}\ U_{b2}, sto, envl\rangle \Rightarrow_b \langle U'_{b1}\ \texttt{or}\ U_{b2}, sto', envl'\rangle}$$

[eller-2] $$\frac{\langle U_b, sto, envl\rangle \Rightarrow_b \langle U'_b, sto', envl'\rangle}{\langle v\ \texttt{or}\ U_b, sto, envl\rangle \Rightarrow_b \langle v\ \texttt{or}\ U'_b, sto', envl'\rangle}$$

[eller-3] $\langle \texttt{true or } v, sto, envl\rangle \Rightarrow_b \langle \texttt{true}, sto, envl\rangle$

[eller-4] $\langle \texttt{false or true}, sto, envl\rangle \Rightarrow_b \langle \texttt{true}, sto, envl\rangle$

[eller-5] $\langle \texttt{false or false}, sto, envl\rangle \Rightarrow_b \langle \texttt{false}, sto, envl\rangle$

[paran-1] $$\frac{\langle U_b, sto, envl\rangle \Rightarrow_b \langle U'_b, sto', envl'\rangle}{\langle (U_b), sto, envl\rangle \Rightarrow_b \langle (U'_b), sto', envl'\rangle}$$

[paran-2] $\langle (v), sto, envl\rangle \Rightarrow_b \langle v, sto, envl\rangle$

[opslag-1] $$\frac{\langle U_a, sto, envl\rangle \Rightarrow_b \langle U'_a, sto', envl'\rangle}{\langle N\ \texttt{[}\ U_a\ \texttt{]}, sto, envl\rangle \Rightarrow_b \langle N\ \texttt{[}\ U'_a\ \texttt{]}, sto', envl'\rangle}$$

[opslag-2] $\langle N\ \texttt{[}\ v_1\ \texttt{]}, sto, envl\rangle \Rightarrow_b \langle v_2, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl'$
og $sto(env_v(N) + v_1) = v_2$

[var] $\langle N, sto, envl\rangle \Rightarrow_b \langle v, sto, envl\rangle$
hvor $envl = (env_V, env_F) : envl'$
og $sto(env_V(N)) = v$

[kald-1-m-uses] $\langle N(U_{p_1}, \ldots, U_{p_n}), sto, envl\rangle \Rightarrow_b$
$\langle \|U_{p_1}\|; \ldots; \|U_{p_n}\|; \|E_v\|; \|K_N\|, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env_f(N) = (K_N, E_v, (p_1, \ldots, p_n), env'_v, env'_f)$
og $E_v \neq \emptyset$

[kald-1-u-uses] $\langle N(U_{p_1}, \ldots, U_{p_n}), sto, envl\rangle \Rightarrow_b$
$\langle \|U_{p_1}\|; \ldots; \|U_{p_n}\|; \|K_N\|, sto, envl\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env_f(N) = (K_N, \emptyset, (p_1, \ldots, p_n), env'_v, env'_f)$

[kald-2] $$\frac{\langle U_{p_1}, sto, envl\rangle \Rightarrow \langle U'_{p_1}, sto', envl'\rangle}{\langle \|U_{p_1}\|; \ldots; \|U_{p_n}\|; \|E_v\|; \|K_N\|, sto', envl'\rangle \Rightarrow_b \langle \|U'_{p_1}\|; \|U_{p_2}\|, \ldots, \|U_{p_n}\|; \|K_N\|, sto', envl'\rangle}$$

[kald-3] $$\frac{\langle U_{p_{m+1}}, sto, envl\rangle \Rightarrow \langle U'_{p_{m+1}}, sto', envl'\rangle}{\begin{array}{r}\langle \|v_{p_1}\|; \ldots; \|v_{p_m}\|; \|U_{p_{m+1}}\|; \ldots; \|U_{p_n}\|; \|E_v\|; \|K_N\|, sto, envl\rangle \Rightarrow_b \\ \langle \|v_{p_1}\|; \ldots; \|v_{p_m}\|; \|U'_{p_{m+1}}\|; \|U_{p_{m+2}}\|, \ldots, \|U_{p_n}\|; \|E_v\|; \|K_N\|, sto', envl'\rangle\end{array}}$$

[kald-4] $\langle \|v_{p_1}\|; \ldots; \|v_{p_n}\|; \|E_v\|; \|K_N\|, sto, envl\rangle \Rightarrow_b \langle \|E_v\|; \|K_N\|, sto', envl'\rangle$
hvor $envl = (env_v, env_f) : envl''$
og $env_f(N) = (K, E_v, env'_v, env'_f)$
og $envl' = (env'_v, env'_f) : envl$
og $a = env_v(next)$
og $env'_v = env_v[p_1 \mapsto a + 0] \ldots [p_n \mapsto a + n - 1][N \mapsto a + n][next \mapsto a + n + 1]$
og $sto' = sto[a + 0 \mapsto v_{p_1}] \ldots [a + n - 1 \mapsto v_{p_n}]$

[kald-5] $$\frac{\langle E_v, sto, envl\rangle \Rightarrow_v \langle E'_v, sto', envl'\rangle}{\langle \|E_v\|; \|K_N\|, sto, envl\rangle \Rightarrow_b \langle \|E'_v\|; \|K_N\|, sto', envl'\rangle}$$

[kald-6] $$\frac{\langle E_v, sto, envl\rangle \Rightarrow_v \langle sto', envl'\rangle}{\langle \|E_v\|; \|K_N\|, sto, envl\rangle \Rightarrow_b \langle \|K_N\|, sto', envl'\rangle}$$

[kald-7] $$\frac{\langle K_N, sto, envl\rangle \Rightarrow_k \langle K'_N, sto', envl\rangle}{\langle \|K_N\|, sto, envl\rangle \Rightarrow_b \langle \|K'_N\|, sto', envl\rangle}$$

[kald-8] $$\frac{\langle K_N, sto, envl\rangle \Rightarrow_k \langle sto', envl\rangle}{\langle \|K_N\|, sto, envl\rangle \Rightarrow_b \langle v, sto', envl'\rangle}$$
hvor $envl = (env_v, env_f) : envl'$
og $v = sto'(env_v(N))$

BILAG

D

Typeregler

D.1 Erklæring af funktioner

[erk-uses]
$$\frac{env_t[E_v] \vdash \langle K \rangle : \checkmark}{env_t \vdash \langle \texttt{function}\ N\ \texttt{(}\ N_1\ \texttt{:}\ T_1, .., N_n\ \texttt{:}\ T_n\ \texttt{)}\ \texttt{:}\ T\ \texttt{uses}\ E_v\ \texttt{does}\ K \rangle : \checkmark}$$

[erk-nouses]
$$\frac{env_t \vdash \langle K \rangle : \checkmark}{env_t \vdash \langle \texttt{function}\ N\ \texttt{(}\ N_1\ \texttt{:}\ T_1, .., N_n\ \texttt{:}\ T_n\ \texttt{)}\ \texttt{:}\ T\ \texttt{does}\ K \rangle : \checkmark}$$

[sekvens]
$$\frac{env_t[E_{f1}\ E_{f2}] \vdash \langle E_{f1} \rangle : \checkmark \quad env_t[E_{f1}\ E_{f2}] \vdash \langle E_{f2} \rangle : \checkmark}{env_t \vdash \langle E_{f1}\ E_{f2} \rangle : \checkmark}$$

D.2 Kommandoer

[sekvens] $$\frac{env_t \vdash \langle K_1 \rangle : \checkmark \quad env_t \vdash \langle K_2 \rangle : \checkmark}{env_t \vdash \langle K_1\ K_2 \rangle : \checkmark}$$

[tildel-1] $$\frac{env_t \vdash \langle N \rangle : T \quad env_t \vdash \langle U \rangle : T}{env_t \vdash \langle N\ \texttt{:=}\ U\ \texttt{;} \rangle : \checkmark}$$

[tildel-2] $$\frac{env_t \vdash \langle N \rangle : T_l \quad env_t \vdash \langle U_b \rangle : \texttt{int} \quad env_t \vdash \langle U \rangle : T_e}{env_t \vdash \langle N\ \texttt{[}\ U_a\ \texttt{]}\ \texttt{:=}\ U\ \texttt{;} \rangle : \checkmark}$$
hvor $T_l = T_e\texttt{list (} v \texttt{)}$

[paran] $$\frac{env_t \vdash \langle K \rangle : \checkmark}{env_t \vdash \langle \texttt{\{}\ K\ \texttt{\}} \rangle : \checkmark}$$

[skip] $$env_t \vdash \langle \texttt{skip ;} \rangle : \checkmark$$

[if] $$\frac{env_t \vdash \langle U_b \rangle : \texttt{bool} \quad env_t \vdash \langle K_1 \rangle : \checkmark \quad env_t \vdash \langle K_2 \rangle : \checkmark}{env_t \vdash \langle \texttt{if}\ U_b\ \texttt{then}\ K_1\ \texttt{else}\ K_2 \rangle : \checkmark}$$

[for] $$\frac{env_t \vdash \langle N \rangle : \texttt{int} \quad env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int} \quad env_t \vdash \langle K \rangle : \checkmark}{env_t \vdash \langle \texttt{for}\ N\ \texttt{from}\ U_{a1}\ \texttt{to}\ U_{a2}\ \texttt{do}\ K \rangle : \checkmark}$$

[while] $$\frac{env_t \vdash \langle U_b \rangle : \texttt{bool} \quad env_t \vdash \langle K \rangle : \checkmark}{env_t \vdash \langle \texttt{while}\ U_b\ \texttt{do}\ K \rangle : \checkmark}$$

D.3 Aritmetiske udtryk

[add] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ + } U_{a2} \rangle : \texttt{int}}$$

[sub] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ - } U_{a2} \rangle : \texttt{int}}$$

[mul] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ * } U_{a2} \rangle : \texttt{int}}$$

[div] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ / } U_{a2} \rangle : \texttt{int}}$$

[mod] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ mod } U_{a2} \rangle : \texttt{int}}$$

[paran] $$\frac{env_t \vdash \langle U_a \rangle : \texttt{int}}{env_t \vdash \langle \texttt{ (} U_a \texttt{) } \rangle : \texttt{int}}$$

[kald] $$\frac{env_t \vdash \langle U_1 \rangle : T_1, .., env_t \vdash \langle U_n \rangle : T_n}{env_t \vdash \langle N \texttt{ (} U_1, .., U_n \texttt{) } \rangle : T}$$
hvis $env_t(N) = (T_1, .., T_n) \rightarrow T$

[var] $env_t \vdash \langle N \rangle : \texttt{int}$
hvis $env_t(N) = \texttt{int}$

[opslag] $$\frac{env_t \vdash \langle U_a \rangle : \texttt{int}}{env_t \vdash \langle N[U_a] \rangle : \texttt{int}}$$
hvis $env_t(N) = \texttt{int list (}v\texttt{)}$

[Lit] $env_t \vdash \langle L_a \rangle : \texttt{int}$

D.4 Boolske udtryk

[mindre] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ < } U_{a2} \rangle : \texttt{bool}}$$

[større] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ > } U_{a2} \rangle : \texttt{bool}}$$

[lig] $$\frac{env_t \vdash \langle U_{a1} \rangle : \texttt{int} \quad env_t \vdash \langle U_{a2} \rangle : \texttt{int}}{env_t \vdash \langle U_{a1} \texttt{ = } U_{a2} \rangle : \texttt{int}}$$

[og] $$\frac{env_t \vdash \langle U_{b1} \rangle : \texttt{bool} \quad env_t \vdash \langle U_{b2} \rangle : \texttt{bool}}{env_t \vdash \langle U_{b1} \texttt{ and } U_{b2} \rangle : \texttt{bool}}$$

[eller] $$\frac{env_t \vdash \langle U_{b1} \rangle : \texttt{bool} \quad env_t \vdash \langle U_{b2} \rangle : \texttt{bool}}{env_t \vdash \langle U_{b1} \texttt{ or } U_{b2} \rangle : \texttt{bool}}$$

[paran] $$\frac{env_t \vdash \langle U_b \rangle : \texttt{bool}}{env_t \vdash \langle \texttt{ (} U_b \texttt{) } \rangle : \texttt{bool}}$$

[kald] $$\frac{env_t \vdash \langle U_1 \rangle : T_1, .., env_t \vdash \langle U_n \rangle : T_n}{env_t \vdash \langle N \texttt{ (} U_1, .., U_n \texttt{) } \rangle : T}$$
hvis $env_t(N) = (T_1, .., T_n) \rightarrow T$

[var] $env_t \vdash \langle N \rangle : \texttt{bool}$
hvis $env_t(N) = \texttt{bool}$

[opslag] $$\frac{env_t \vdash \langle U_a \rangle : \texttt{int}}{env_t \vdash \langle N[U_a] \rangle : \texttt{bool}}$$
hvis $env_t(N) = \texttt{bool list (}v\texttt{)}$

[lit] $env_t \vdash \langle L_b \rangle : \texttt{bool}$

D.5 Strengeudtryk

[kald] $$\frac{env_t \vdash \langle U_1 \rangle : T_1, .., env_t \vdash \langle U_n \rangle : T_n}{env_t \vdash \langle N \texttt{ (} U_1, .., U_n \texttt{) } \rangle : \texttt{string}}$$
hvis $env_t(N) = (T_1, .., T_n) \rightarrow \texttt{string}$

[var] $env_t \vdash \langle N \rangle : \texttt{string}$
hvis $env_t(N) = \texttt{string}$

[opslag] $$\frac{env_t \vdash \langle U_a \rangle : \texttt{int}}{env_t \vdash \langle N[U_a] \rangle : \texttt{string}}$$
hvis $env_t(N) = \texttt{string list (}v\texttt{)}$

[lit] $env_t \vdash \langle L_s \rangle : \texttt{string}$

Algoritmer

Kildekode til programmerne der er brugt i forsøgsafsnittet.

E.1 Bubblesort

```
function main() : int
uses
{
  L : int list(300);
  i : int;
}
does
{
  # Initiate arrays #

  for i from 0 to 300 do
    L[i] := 300-i;

  # Sort arrays #
  L := bubbleSort(L);
}


function bubbleSort(A : int list(300) ) : int list(300)
uses
{
  swapped : bool;
  i : int;
  tmp : int;
}
does
{
  swapped := true;

  while swapped do
  {
    swapped := false;

    for i from 0 to (300 - 1) do
    {
      if (A[i] > A[i+1]) then
      {
        tmp := A[i];
        A[i] := A[i+1];
        A[i+1] := tmp;
        swapped := true;
      }
      else skip;
    }
  }

  # Return result #
  bubbleSort := A;
}
```

E.2 Udregning af primtal

```
function main () : int
uses
{
  firstPrimes : int list(100);
}
does
{
  firstPrimes := findPrimes(1000);
}

function findPrimes( startCandidate : int ) : int list(100)
uses
{
  output : int list(100);
  trialDivisor : int;
  candidate : int;
  whileLoopIter : int;
  maxPrimes : int;
  primesFound : int;
  prime : int;
}
does
{
  maxPrimes := 100;
  candidate := startCandidate;
  primesFound := 0;

  while (primesFound < maxPrimes) do
  {
    trialDivisor := 2;
    whileLoopIter := trialDivisor * trialDivisor;
    prime := 1;
    while ( whileLoopIter < (candidate + 1) ) do
    {
      if( (candidate mod trialDivisor) = 0 ) then
      {
        prime := 0;
        whileLoopIter := candidate + 1;
      }
      else
        skip;

      if (whileLoopIter < (candidate + 1)) then
      {
        trialDivisor := trialDivisor + 1;
        whileLoopIter := trialDivisor * trialDivisor;
      }
      else
        skip;
    }

    if( prime > 0 ) then
    {
      output[primesFound] := candidate;
      primesFound := primesFound + 1;
    }
    else
      skip;

    candidate := candidate + 1;
  }
  findPrimes := output;
}
```

E.3 Dårligt eksempel på parallelisering

```
function main () : int
uses
{
  i : int;
  a : int;
  b : int;
}
does
{
  for i from 0 to 20000 do
  {
    a := i;
    b := i;
  }
}
}
```

LITTERATUR

[ACH+07] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr. og Sam Tobin-Hochstadt. The fortress language specification. *sunsource.net*, 1.0 β, March 6, 2007. `http://research.sun.com/projects/plrg/fortress.pdf`.

[CDRV98] Pierre-Yves Calland, Alain Darte, Yves Robert og Frederic Vivien. On the removal of anti- and output-dependences. *International Journal of Parallel Programming*, 26(3), 1998.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis og Vipin Kumar. *Introduction to Parallel Computing.* Pearson, Second Edition udgave 2003.

[Gor07] Pam Frost Gorder. Multicore processors for science and engineering. *Computing in Science and Engineering*, 9(2):3–7, 2007.

[Hüt07] Hans Hüttel. *Pilen ved træets rod.* Aalborg Universitet, 2007.

[KAM82] A. J. Kfoury, Michael A. Arbib og Robert N. Moll. *Programming Approach to Computability.* Springer-Verlag New York, Inc., 1982.

[Kri01] S. Krishnaprasad. Uses and abuses of amdahl's law. *J. Comput. Small Coll.*, 17(2):288–293, 2001.

[Man04] Indrek Mandre. *Alternative output for SableCC 3*, November 14, 2004. `http://www.mare.ee/indrek/sablecc/`.

[PW86] David A. Padua og Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), 1986.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, Second Edition International Edition udgave 2006.

[Tho07] Bent Thomsen. *Slides fra SPO*, 2007. Slide nr. 47 på `http://www.cs.aau.dk/~bt/SPOF07/SPOF07-4.pdf`.

[WB00] David A. Watt og Deryck F. Brown. *Programming Language Processors in Java – Compilers and Interpreters.* Prentice Hall, 2000.