Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Sven Cerk

Prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Izdelajte prevajalnik za prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo. Za funkcijske ščepce definirajte ustrezni na Haskellu temelječ programski jezik, grafna vmesna koda pa naj bo zgrajena na osnovnih gradnikih podatkovo vodenih grafov, ki jih določa Maxelerjeva podatkovno-pretokovna arhitektura. Natančno opišite postopek in omejitve prevajanja.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Sven Cerk sem avtor diplomskega dela z naslovom:

Prevajanje funkcijskih ščepcev v grafno vmesno kodo za Maxelerjevo arhitekturo

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 11. septembra 2015 Podpis avtorja:

Moji Petri.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DFE	dataflow engine	podatkovno-pretokovna enota
SLiC API	Simple Live CPU API	vmesnik za komunikacijo z DFE
WHNF	weak head normal form	šibka korenska normalna oblika

Povzetek

V diplomskem delu je predstavljen nov način programiranja Maxelerjevih enot za podatovno-pretokovno računanje. Obstoječa orodja omogočajo programiranje v prilagojeni različici Jave. Verjamemo, da so za to nalogo bolj primeri funkcijski programski jeziki.

V ta named definiramo MaxHs, nov programski jezik za programiranje ščepcev, ki se izvajajo na podatkovno-pretokovnih enotah. MaxHs temelji na idejah programskega jezika Haskell. Predstavimo tudi možnosti uporabe MaxHs v povezavi s Haskellom. Na vzorčnih programih primerjamo našo rešitev z obstoječimi orodji za programiranje.

V zadnjem delu predstavimo način prevajanja našega jezika v grafno vmesno kodo. Elementi grafne vmesne kode odražajo vozlišča, ki jih ponuja Maxelerjeva arhitektura. Poleg prevajanja v grafno vmesno kodo predstavimo tudi, kako omogočimo povezavo s Haskellom.

Ključne besede: podatkovno-pretokovno računanje, funkcijski programski jeziki, prevajalnik.

Abstract

A new method for programming Maxeler's dataflow engines is presented. The existing tools allow programmers to program dataflow engines in Java. We belive that functional programming languages might provide a more suitable alternative.

We define a new programming language called MaxHs for programming dataflow engine kernels. In addition, we present a way of interfacing MaxHs programs with programs written in Haskell. With the help of simple example programs we compare our solution with existing tools.

In the last part we present a method for compiling MaxHs into a graph-based intermediate code. The elements of the intermediate code are based on the nodes supported by Maxeler's dataflow engines. We also outline how the support for interfacing with Haskell is achieved.

Keywords: dataflow computing, functional programming languages, compiler.

Poglavje 1

Uvod

Zaradi tehnoloških omejitev pri proizvodnji procesorjev je za hitrost računalniških programov vedno bolj pomembno, da izkoriščajno različne možnosti vzporednega računanja. V običajnih računalniških sistemih že prevladujejo večjedrni procesorji, za katere lahko programerji pišejo vzporedne aplikacije. Kljub temu so zahteve po hitrosti izvajanja še mnogo večje, zato snovalci računalniških sistemov iščejo nove možnosti za pohitritev aplikacij. Najočitnejša možnost je dodajanje različnih enot za računanje, izmed katerih je vsaka namenjena različnim opravilom, ki jih lahko izvrši veliko hitreje kot običajni procesorji.

Primer takšnih posebnih enot so denimo grafične kartice, ki so namenjene izrisovanju grafičnih elementov in zato primerne tudi za reševanje računsko intenzivnih problemov. V običajnem računalniškem sistemu tako najdemo centralno procesno enoto, ki je prilagojena izvajanju zaporednih ukazov in primerna za večino osnovnih nalog računalnika, poleg nje pa še grafične kartice, ki izvedejo operacije, v katerih so učinkovitejše od centralne procesne enote.

Računalniškim sistemom, ki za računanje izkoriščajo več različnih enot, pravimo tudi *heterogeni računalniški sistemi*. Najpogosteje so sestavljeni iz centralne procesne enote, ki usmerja delovanje, in posebnih enot, ki učinkovito izvajajo določene vrste operacij. Pogosto so prilagojene vzporednemu

računanju na veliki količini podatkov, saj so tu običajni procesorji najbolj prikrajšani.

Za programerje, ki želijo izkoristiti prednosti heterogenih sistemov, pogosto predstavlja veliko težavo način programiranja, ki je potreben za izkoriščanje posebnih enot, kajti lahko se zelo razlikuje od klasičnega programiranja.

V tem diplomskem delu se bomo ukvarjali s programiranjem posebnih enot za vzporedno računanje, ki jih proizvaja podjetje Maxeler Technologies. Predstavili bomo alternativo obstoječemu načinu programiranja, ki bo temeljila na idejah iz funkcijskega programskega jezika Haskell.

Poglavje 2

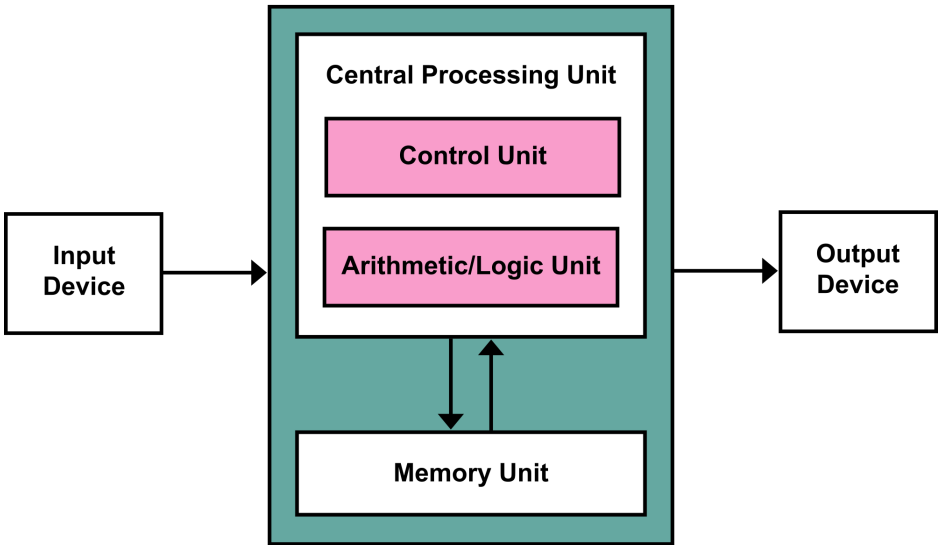
Podatkovno-pretokovno računanje

Večina običajnih računalniških sistemov je zasnovana po Von Neumannovi arhitekturi. V svojem delu na računalniku *EDVAC* je Von Neumann opisal splošno zasnovo, kateri bi morali slediti računalniški sistemi.

Von Neumannovo arhitekturo sestavljajo centralna procesna enota, spominska enota in vhodno-izhodne enote (slika 2.1). Računanje na takšni arhitekturi poteka z zaporednim izvajanjem ukazov, ki se nahajajo na določenem mestu v pomnilniku računalnika. Centralna procesna enota vsebuje kontrolno enoto, ki skrbi za pridobivanje ukazov iz pomnilnika in njihovo interpretacijo. Poseben register v kontrolni enoti – programski števec – določa položaj naslednjega ukaza, ki ga mora računalnik izvesti. Ob izvedbi vsakega ukaza se programski števec poveča in s tem povzroči izvajanje naslednjega ukaza.

Ker je izvajanje na takšnih računalnikih vodeno z zaporedjem ukazov, jim pravimo tudi ukazno-pretokovni računalniki. Dogajanje v računalniku je določeno s tokom ukazov, ki jih računalnik izvaja enega za drugim.

Takšen način delovanja je primeren za veliko število problemov, vendar pa obstajajo tudi problemi, pri katerih odpove. Osnovna lastnost Von Neumannove arhitekture je, da izvaja ukaze enega za drugim ne glede na to, ali



Slika 2.1: Shema Von Neumannove arhitekture [1].

je to zares potrebno. Določene probleme bi lahko brez težav razdelili na podprobleme, ki se jih da rešiti neodvisno. Za učinkovito implementacijo bi bilo dobro, da med seboj neodvisne podprobleme rešujemo sočasno. Ker pa Von Neumannova arhitektura zahteva, da izvajanje ukazov sledi programskemu števcu, moramo ukaze vseeno razporediti linearno. Na ta način v program vnesemo odvečno strukturo, saj vrstni red ukazov ni tako natančno določen, kot je zapisano v programu. Zaradi tega je izvajanje počasnejše.

Poleg zgoraj opisanega problema se pri Von Neumannovi arhitekturi pojavi tudi težava v komunikaciji med centralno procesno enoto in pomnilnikom. Ukazi računalniku opisujejo spremembo stanja, ki se mora zgoditi ob njihovem izvajanju. Večina sprememb se nanaša na spremembe v pomnilniku, saj se tam nahajajo podatki. Tako mora centralna procesna enota ob izvajanju ukaza pridobiti podatke iz pomnilniške enote, na njih izvesti določeno operacijo in jih zapisati nazaj v pomnilnik [2].

Opisane omejitve razvijalci računalniških sistemov rešujejo na različne načine. Ena možnost so majhne izboljšave klasičnih procesorjev, kot so cevovodi in prerazporejanje ukazov ali večjedrni procesorji. Druga možnost, ki je tema tega diplomskega dela, je podatkovno-pretokovno računanje.

2.1 Osnovne lastnosti podatkovno-pretokovnih arhitektur

Za razliko od Von Neumannove arhitekture podatkovno-pretokovni računalniki ne vsebujejo programskega števca, ki bi določal vrstni red ukazov, temveč je izvajanje odvisno od toka podatkov. Cilj takšnih računalnikov je, da v program za reševanje problema ne vnašamo nepotrebne strukture. Izvajanje ukaza se mora zgoditi takoj, ko so na voljo vsi njegovi vhodni podatki, in lahko se izvaja več ukazov hkrati. Takšna arhitektura dobro podpira vzporedno računanje.

Podatkovno-pretokoven program ni opisan s seznamom ukazov temveč s potjo, ki jo opravijo vhodni podatki skozi različne operacije. Takšne programe lahko predstavimo kot graf, kjer v vozliščih nastopajo operacije, povezave pa povezujejo rezultat ene operacije z vhodom druge.

Skozi čas se je razvilo več različnih podatkovno-pretokovnih arhitektur, vendar danes še nobena ne more nadomestiti Von Neumannove arhitekture v klasičnih računalnikih. Pri vsaki aplikaciji, kjer je hitrost velikega pomena in izkazuje visoko mero vzporednosti, pa lahko izkoristimo prednosti podatkovno-pretokovnih arhitektur tako, da kose programa izvedemo na posebnih enotah za podatkovno-pretokovno računanje.

Navadno so deli programa, primerni za izvajanje na podatkovno-pretokovni arhitekturi, majhni in morajo sodelovati s programom, ki teče na centralni procesni enoti. Tu se pokaže prednost heterogenega sistema, kjer lahko večino aplikacije izvajamo na centralni procesni enoti, računsko zahtevne dele pa prepustimo posebnim enotam, namenjenim podatkovno-pretokovnemu računanju.

Arhitektura računalniškega sistema seveda tudi močno vpliva na način programiranja, ki ga zahteva. Imperativno programiranje je primerno prav za programe namenjene Von Neumannovi arhitekturi, saj vsak imperativen program natančno določa zaporedje ukazov, ki se bodo izvedli na centralni procesni enoti. Če bi želeli v imperativnem programskem jeziku napisati pro-

gram za podatkovno-pretokovno arhitekturo, bi naleteli na velike težave. V podatkovno-pretokovni arhitekturi namreč koncepta spremenljivk in spreminjanja vrednosti v pomnilniku nimata pravega pomena. Opisati je potrebno, kako se rezultat neke operacije povezuje z drugimi in ne kam ga želimo shraniti [3].

V tem diplomskem delu bomo zasnovali programski jezik, namenjen programiranju podatkovno-pretokovne arhitekture, ki jo v svojih produktih uporablja podjetje Maxeler Technologies. Pred tem si bomo ogledali njene osnovne lastnosti in obstoječi način programiranja.

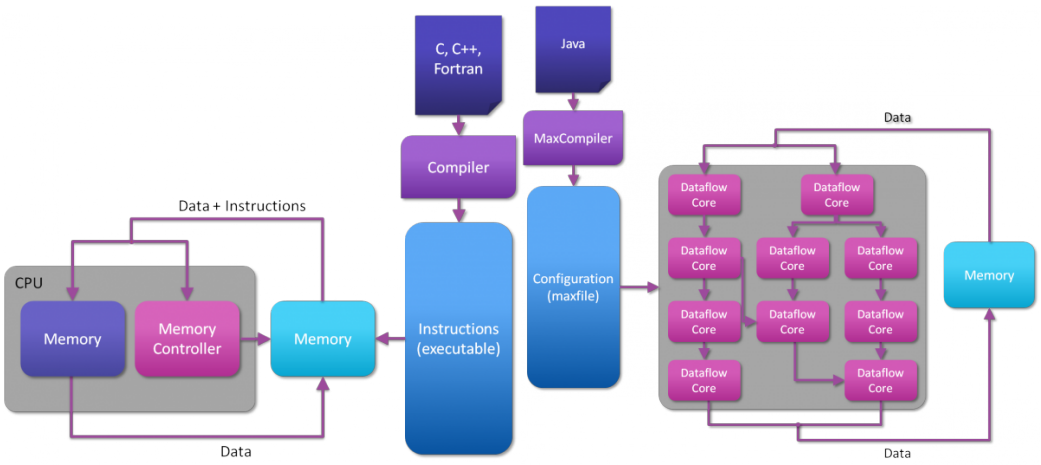
2.2 Podatkovno-pretokovne enote podjetja Maxeler Technologies

Podjetje *Maxeler Technologies* ponuja rešitve za visoko zmogljivo računanje (angl. *high-performance computing*), ki v veliki meri vključujejo podatkovno-pretokovno računalniško tehnologijo. Razvili so več različnih enot za podatkovno-pretokovno računanje, ki so namenjene različnim aplikacijam, vendar si delijo skupne lastnosti in način programiranja. Opis delovanja teh enot presega obseg tega diplomskega dela, natančneje si bomo ogledali le okolje za programiranje aplikacij, ki izkoriščajo Maxelerjeve tehnologije.

2.2.1 Osnovni gradniki aplikacije za Maxelerjevo podatkovno-pretokovno enoto

Sistem, ki izkorišča Maxelerjevo tehnologijo za podatkovno-pretokovno računanje, je sestavljen iz klasičenga računalniškega sistema, ki izvaja večino operacij po Von Neumannovem modelu in za računsko zahtevne dele izkoristi dodatne enote, ki se imenujejo *podatkovno-pretokovne enote* (angl. *dataflow engine*, DFE).

Programiranje aplikacije za Maxelerjev DFE sestoji iz pisanja programa, ki bo tekel na CPE, ter opisa ščepca za podatkovno-pretokovno enoto in



Slika 2.2: Primerjava klasične računalniške arhitekture (levo) z Maxelerjevo (desno) [4].

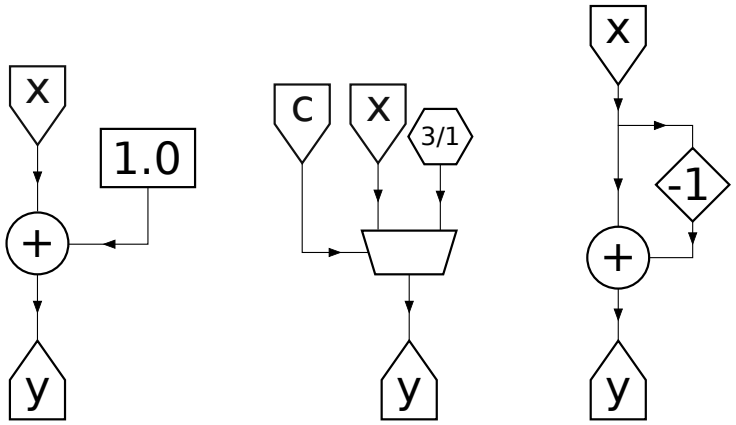
načina prenosa podatkov znotraj nje. *Ščepec* (angl. *kernel*) predstavlja implementacijo računskega postopka in je opisan s podatkovno-pretokovnim grafom.

Program, ki teče na CPE, pripravi potrebne podatke in kliče posebno funkcijo, ki povzroči prenos podatkov na DFE. Znotraj DFE-ja podatki tečejo iz spomina skozi vrsto artimetičnih enot in na koncu nazaj v spominsko enoto. Iz spominske enote se podatki prenesejo na računalnik, kjer program na CPE prevzame nadzor in nadaljuje svoje delo.

2.2.2 Opis DFE ščepca

Kot smo že omenili, razporeditev enot znotraj podatkovno-pretokovne enote opišemo z grafom. Na voljo imamo več različnih tipov vozlišč, ki jih lahko pri tem uporabimo:

- **računska vozlišča** opisujejo aritmetične ali logične operacije nad tokom podatkov – ◯;
- **vrednostna vozlišča** vsebujejo določeno vrednost, ki je lahko konstantna ali pa določena s strani aplikacije, ki teče na CPE – ▭;



Slika 2.3: Preprosti primeri grafov za Maxelerjeve podatkovno-pretokovne enote.

- **vozlišča za odmik** predstavljajo zamik toka podatkov za določeno število mest – ◇;
- **multiplekserji** predstavljajo izbor med seznamov vhodnih tokov podatkov na osnovi kontrolnega toka – ▱;
- **števci** predstavljajo tok podatkov, ki se začne z neko vrednostjo in se ob vsakem prehodu ure spreminja – ⬡;
- **vhodno-izhodna volišča** predstavljajo tokove podatkov, ki prihajajo v DFE ali odhajajo iz njega – ▽ / △.

Vozlišča lahko med seboj povezujemo in tako sestavimo želeni graf. Preprosti primeri grafov so prikazani na sliki 2.3.

2.2.3 Programiranje aplikacij za Maxelerjevo arhitekturo

Za uporabo Maxelerjevih podatkovno-pretokovnih enot je potrebno napisati program, ki bo tekel na centralni procesni enoti, in sestaviti opis grafa za podatkovno-pretokovno enoto. Programiranje navadno poteka v spremenjeni

različici razvojnega okolja Eclipse, ki podpira vse Maxelerjeve razširitve, imenovani *MaxIDE*.

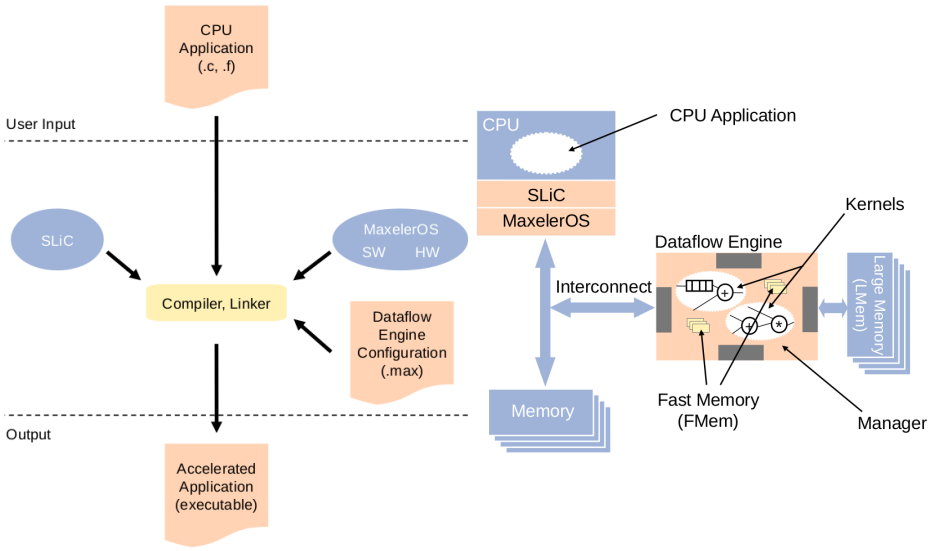
Program, ki teče na CPE, je lahko napisan v različnih programskih jezikih. Najbolje je podprt programski jezik C, ostale možnosti pa so še Matlab, R in Python. Ogledali si bomo primer v C-ju.

Za izvajanje operacij na DFE-ju moramo v C programu klicati funkcije, ki sestavljajo Maxelerjev *Simple Live CPU* (SLiC) API. Klic teh funkcij povzroči, da se podatki prenesejo na podatkovno-pretokovno enoto in se tam izvede računski postopek. Pred tem moramo seveda tudi sestaviti opis grafa za podatkovno-pretokovno enoto — to si bomo ogledali malo kasneje. Rezultat prevajanja opisa grafa je konfiguracijska datoteka (končnica .max), ki pravzaprav vsebuje C izvorno kodo, v kateri so definirane funkcije SLiC API-ja. Te funkcije poskrbijo za konfiguracijo podatkovno-pretokovne enote in prenos podatkov nanjo. Poleg .max datoteke se nahaja še C zaglavna datoteka (končnica .h), v kateri so funkcije SLiC API-ja deklarirane, tako da jih lahko vključimo v naš C program. V glavnem programu na mestu, kjer želimo uporabiti DFE, samo pripravimo vhodne podatke in prostor za izhodne ter s klicem prave funkcije SLiC API-ja izvajanje prenesemo na DFE, ki nam izhodne podatke vrne na podano mesto.

2.2.4 Programiranje grafa za podatkovno-pretokovno enoto

Kot smo omenili v prejšnjem razdelku, moramo za uporabo podatkovno-pretokovne enote pripraviti datoteko z opisom grafa. Ta vsebuje konfiguracijo podatkovno-pretokovne enote in je rezultat prevajanja opisa grafa, ki ga zapišemo v programskem jeziku Java z uporabo Maxelerjevih knjižnic. Za lažje programiranje uporablja okolje MaxIDE namesto standardne Jave razširjeno različico *MaxJ*, ki podpira večpomenske (angl. *overloaded*) operatorje. Prevajalnik v Maxelerjevem razvojnem okolju se imenuje *MaxCompiler*.

Opis grafa zapišemo v izvorni kodi MaxJ tako, da iz osnovnih konstruk-



Slika 2.4: Interakcija med programskimi komponentami (levo) in arhitektura sistema s podatkovno-pretokovno enoto (desno) [5].

tov, ki jih ponuja Maxelerjeva knjižnica, sestavimo želeni graf. V knjižnici najdemo različne vrste objektov, ki predstavljajo vozlišča opisana v razdelku 2.2.2.

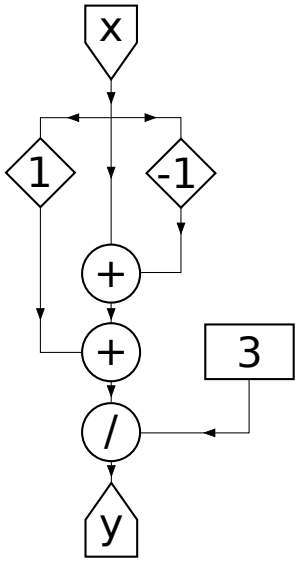
Osnovni podatkovni tip, ki ga uporabljamo pri programiranju ščepca, je `DFEVar`. Vrednosti tega tipa predstavljajo tokove podatkov, operacije nad njimi pa sestavljanje večjih delov grafa. Za vsako od osnovnih vozlišč obstaja v Maxelerjevi knjižnici metoda, s katero ga lahko ustvarimo. Natančen opis Maxelerjeve knjižnice za to diplomsko delo ni zelo pomemben, saj se bomo kasneje ukvarjali le s prevajanjem novega jezika za opis grafov v vmesno kodo, ki bo temeljila na konstruktih iz razdelka 2.2.2.

V naslednjem razdelku si bomo ogledali primer programiranja preprostega ščepca za podatkovno-pretokovno enoto in C programa, ki jo uporablja.

2.2.5 Primer aplikacije

Preprost primer aplikacije, ki uporablja podatkovno-pretokovno enoto, je izračun tekočega povprečja zaporedja števil. Ta primer je povzet po [5].

Tekoče povprečje zaporedja števil definiramo kot zaporedje povprečji sosednjih treh vrednosti v zaporedju. Na primer za zaporedje $1, 2, 1, 3, 1$ dobimo rezultat $1, \frac{4}{3}, 2, \frac{5}{3}, \frac{5}{3}$. Definicija je jasna za člene zaporedja, ki niso na začetku ali koncu. Tako smo dobili vrednosti $\frac{1+2+1}{3} = \frac{4}{3}$, $\frac{2+1+3}{3} = 2$ in $\frac{1+3+1}{3} = \frac{5}{3}$. Za robne člene zaporedja pa določimo posebni pravili: pred prvim členom zaporedja nastopa vrednost 0, zadnjemu členu zaporedja pa sledi število enako zadnjemu. Tako dobimo še $\frac{0+1+2}{3} = 1$ in $\frac{3+1+1}{3} = \frac{5}{3}$. Takšno definicijo smo izbrali, ker je na ta način implementacija v MaxJ najkrajša (in je zato ta primer dovolj enostaven in kratek). Graf, ki ga želimo opisati, je prikazan na sliki 2.5.



Slika 2.5: Graf ščepca za izračun tekočega povprečja.

Za implementacijo tega ščepca v MaxJ najprej napišemo razred `MovingAverageKernel` (program 2.1 na strani 13).

V konstruktorju razreda najprej definiramo spremenljivko `x`, ki predstavlja vhodni tok podatkov. Z metodo `stream.offset` nato ustvarimo še dve novi spremenljivki `prev` in `next`. Vrednost teh spremenljivk je ob vsakem prehodu ure ravno eno nazaj ali naprej od spremenljivke `x`. V naslednji vrstici izračunamo vsoto prejšnjih treh spremenljivk, jo delimo s 3 in rezultat povežemo na izhod ščepca.

Vsota dveh spremenljivh tipa `DFEVar` v resnici predstavlja le ustvarjanje novega objekta razreda `DFEVar`, ki predstavlja vozlišče za računanje vsote z vhodoma, povezanima na operanda vsote.

Poleg implementacije ščepca moramo poskrbeti tudi za upravitelja (angl. *manager*), ki skrbi za prenos podatkov, vendar je ta v enostavnih primerih vedno enak, zato ga tu ne bomo izpostavljali.

Drugi pomembni del naše aplikacije je program, ki bo tekel na CPE (program 2.2). Ta program napišemo v programskem jeziku C in v njem upora-

bimo funkcije, ki jih ponuja SLiC API za naš ščepec. Kot rezultat prevajanja ščepca (program 2.1) z MaxCompilerjem dobimo konfiguracijsko datoteko (.max) in zaglavno datoteko (.h) z deklaracijami funkcij SLiC API-ja, ki jih lahko uporabimo v programu za CPE. V našem primeru sta to datoteki MovingAverage.max in MovingAverage.h.

Znotraj teh datotek najdemo različne funkcije za povezovanje s podatkovno-pretokovno enoto. Mi bomo uporabili najpreprostejšo. Tej kot argumente podamo dolžino vhodnega toka, kazalec na polje vhodnih podatkov in kazalec na prostor pripravljen za izhodne podatke.

Funkcija `MovingAverage` je definirana v datoteki MovingAverage.max, ki jo je za nas pripravil prevajalnik MaxCompiler iz opisa ščepca v MovingAverageKernel.maxj.

Prevajanje celotnega projekta s prevajalnikom MaxCompiler poteka na sledeči način:

- prevajanje MaxJ datotek z razširjenim Java prevajalnikom,
- izvajanje razrednih datotek, ki ustvarijo konfiguracijsko datoteko in pripadajočo zaglavno datoteko,
- prevajanje C izvorne kode, ki uporablja funkcije definirane v konfiguraciji.

Rezultat je izvedljiva datoteka, ki z uporabo SLiC API-ja komunucira s podatkovno-pretokovno enoto.

Način programiranja ščepcev je torej grajenje grafa iz osnovnih objektov v Maxelerjevi Java knjižnici, ki se ob izvajanju izpiše v konfiguracijsko C datoteko. Funkcije iz konfiguracijske datoteke lahko nato uporabljamo v svojih programih, saj so pripravljene tako, da ob klicu podane vhodne podatke prenesejo na podatkovno-pretokovno enoto in začnejo njeno izvajanje. Ob vrnitvi iz funkcije pa prenesejo rezultate iz podatkovno-pretokovne enote nazaj na podano mesto.

Program 2.1 MovingAverageKernel.maxj

```
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
class MovingAverageKernel extends Kernel {
  MovingAverageKernel(KernelParameters parameters) {
    super(parameters);
    DFEVar x = io.input("x", dfeFloat(8, 24));
    DFEVar prev = stream.offset(x, -1);
    DFEVar next = stream.offset(x, 1);
    DFEVar sum = prev + x + next;
    DFEVar result = sum / 3;
    io.output("y", result, dfeFloat(8, 24));
  }
}
```

Program 2.2 MovingAverage.c

```
#include "MovingAverage.h"
#include <MaxSLiCInterface.h>
float dataIn[5] = { 1, 2, 1, 3, 1 };
float dataOut[5];
int main()
{
  MovingAverage(5, dataIn, dataOut);
  for (int i = 0; i < 5; i++)
    printf("dataOut[%d] = %f\n", i, dataOut[i]);
  return 0;
}
```

2.2.6 Možne izboljšave

Programiranje Maxelerjevih podatkovno-pretokovnih enot od programerja zahteva grajenje grafa v prilagojeni različici Jave in nato uporabo dobljenih funkcij v programu, ki bo tekel na CPE.

Na drugih podatkovno-pretokovnih arhitekturah se za opisovanje podatkovno-pretokovnih grafov pogosto uporablja funkcijske programske jezike. Funkcije brez stranskih učinkov lahko namreč zelo dobro opisujejo zgradbo podatkovno-pretokovnega grafa [6].

V nadaljevanju si bomo ogledali osnovne lastnosti funkcijskih jezikov in predstavili nov način programiranja za Maxelerjevo arhitekturo, ki bo temeljil na funkcijskem programskem jeziku Haskell. Takšen način lahko programerju olajša delo, saj bo moral obvladati samo en način programiranja, ki bo zadostoval za programe na centralni procesni enoti in za opis grafa ščepca.

Poleg tega bo opisani način v veliki meri zmanjšal potrebo programerja po razreševanju podrobnosti na nivoju arhitekture in mu s tem omogočili koncentracijo na reševanje problema.

Poglavje 3

Funkcijski programski jeziki

Funkcijsko programiranje je paradigma programiranja, v kateri se delovanje programa opiše s čistimi funkcijami (brez stranskih učinkov). Odsotnost stranskih učinkov pomeni, da v funkcijskih jezikih ne opisujemo sprememb stanja, ki bi nas vodile do rezultata, temveč je program sestavljen le iz izrazov, ki jih ob izvajanju izračunamo [7].

Danes funkcijski programski jeziki postajajo čedalje pogosteje uporabljeni, saj programerjem omogočajo lažje sklepanje o delovanju programov. Ker je vsak kos programa le funkcija brez stranskih učinkov, je testiranje posameznih enot zelo olajšano. Pogosto so funkcijski jeziki tudi močno tipizirani in na ta način omogočajo hitrejše odkrivanje napak.

Eden prvih funkcijskih programskih jezikov – *LISP* se je pojavil že v 50-ih letih prejšnjega stoletja. Osnovna podatkovna struktura v LISP-u je seznam, saj je delo s seznami v funkcijskih jezikih pogosto zelo preprosto. Zaradi tega je tudi v modernejših funkcijskih jezikih seznam ena izmed glavnih podatkovnih struktur.

Kasneje bomo videli, da je delo s seznami ključnega pomena tudi za naš programski jezik, kajti Maxelerjeve podatkovno-pretokovne enote so namenjene ravno izvajanju operacij na toku (seznamu) velike količine podatkov.

Pred tem si bomo ogledali še *lambda račun*, ki je služil kot teoretična osnova za razvoj skoraj vseh funkcijskih programskih jezikov.

3.1 Lambda račun

Lambda račun je formalen sistem, ki ga je v tridesetih letih prejšnjega stoletja razvil matematik Alonzo Church, da bi lahko formalno opisal način izračunavanja matematičnih funkcij. Sistem je precej uporaben kot model računanja, saj je ekvivalenten Turingovemu stroju.

Natančnejši uvod v lambda račun se nahaja v [8, 9].

3.1.1 Izrazi lambda računa

Lambda račun definira jezik sestavljen iz λ-*izrazov* in množico pravil za njihovo preoblikovanje. Najprej definirajmo veljavne λ-izraze.

Definicija 3.1 (Lambda izrazi)**.** Za neskončno množico spremenljivk $V = \{v, v', v'', \ldots\}$ je množica vseh veljavnih λ-izrazov Λ definirana induktivno s pravili:

$$x \in V \implies x \in \Lambda \tag{3.1}$$

$$M, N \in \Lambda \implies (MN) \in \Lambda \tag{3.2}$$

$$M \in \Lambda, x \in V \implies (\lambda x.M) \in \Lambda \tag{3.3}$$

Pravilo 3.2 se imenuje *aplikacija*, pravilo 3.3 pa *abstrakcija*. Primeri veljavnih λ-izrazov so $x, (\lambda x.x), (\lambda x.(\lambda y.x)), (\lambda x.z), ((\lambda x.y)z)$.

Spremenljivke, ki nastopajo v λ-izrazih, so lahko *proste* ali *vezane*.

Definicija 3.2 (Proste spremenljivke)**.** V λ-izrazu $M \in \Lambda$ je množica *prostih* spremenljih $\mathrm{FV}(M)$ definirana kot

$$\mathrm{FV}(M) = \begin{cases} x & ;\text{če } M = x,\ x \in V \\ \mathrm{FV}(N) \cup \mathrm{FV}(O) & ;\text{če } M = (NO),\ N, O \in \Lambda \\ \mathrm{FV}(N) \setminus \{x\} & ;\text{če } M = (\lambda x.N),\ N \in \Lambda,\ x \in V \end{cases}$$

Če spremenljivka v izrazu ni prosta, je *vezana*.

Proste spremenljivke v λ-izrazih lahko nadomestimo z drugimi λ-izrazi.

Definicija 3.3 (Substitucija)**.** Rezultat zamenjave proste spremenljivke x z izrazom N v izrazu M je nov λ-izraz, ki ga označimo z $M[x := N]$, in je definiran s pravili:

$$x[x := N] = N \tag{3.4}$$

$$y[x := N] = y \text{ ; če } y \neq x \tag{3.5}$$

$$(M_1 M_2)[x := N] = (M_1[x := N] M_2[x := N]) \tag{3.6}$$

$$(\lambda y.M)[x := N] = \lambda y.(M[x := N]) \tag{3.7}$$

V vseh kontekstih, kjer se pojavi več λ-izrazov $M_1, \ldots, M_n$, privzamemo, da so njihove proste spremenljivke drugačne od vezanih. Brez te predpostavke bi morali pri pravilu (3.7) upoštevati še dodaten pogoj $y \neq x$ in $y \notin \mathrm{FV}(N)$ ter v primeru, da ni izpolnjen, ustrezno preimenovati spremenljivke.

Zdaj lahko zapišemo aksiome lambda računa in s tem zaključimo njegovo definicijo.

Definicija 3.4 (Shema aksiomov lambda računa)**.**

$$\lambda x.M = \lambda y.M[x := y] \tag{α}$$

$$(\lambda x.M)N = M[x := N] \tag{β}$$

$$\lambda x.(Mx) = M \tag{η}$$

Prvo pravilo imenujemo α-preimenovanje, saj pravi, da pomen λ-abstrakcije ostane enak, če v njej zamenjamo spremenljivko, ki jo veže. Drugo pravilo določa pomen aplikacije in mu pravimo β-redukcija. Izraz, ki ga dobimo z β-redukcijo je telo λ-abstrakcije, kjer so vse pojavitve spremenllјivke, ki jo veže, zamenjane z izrazom na desni. Tretje pravilo odpravi odvečne λ-abstrakcije.

Pri definicijah smo upoštevali predpostavko, ki zagotavlja, da se imena spremenljivk ne prekrivajo.

3.1.2 Redukcija

Na lambda račun lahko gledamo tudi kot na model izračunavanja funkcij. Vsaka λ-abstrakcija predstavlja anonimno funkcijo z enim argumentom, β-redukcija pa klic funkcije. Na podlagi tega lahko definiramo relacije nad λ-izrazi, ki povedo, ali je nek izraz možno poenostaviti v drugega.

Definicija 3.5. Relacijo $\rightarrow_\beta$ definiramo induktivno:

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$
$$M \rightarrow_\beta N \implies (ZM) \rightarrow_\beta (ZN), (MZ) \rightarrow_\beta (NZ), (\lambda y.M) \rightarrow_\beta (\lambda y.N)$$

Relacija $\twoheadrightarrow_\beta$ je tranzitivna ovojnica $\rightarrow_\beta$.

Podobno naredimo tudi za η-konverzijo.

Definicija 3.6. Relacija $\rightarrow_\eta$ je definirana s pravili:

$$\lambda x.(Mx) \rightarrow_\eta M$$
$$M \rightarrow_\eta N \implies (ZM) \rightarrow_\eta (ZN), (MZ) \rightarrow_\eta (NZ), (\lambda y.M) \rightarrow_\eta (\lambda y.N)$$

Njena tranzitivna ovojnica je $\twoheadrightarrow_\eta$.

Definicija 3.7. Za izraza M in N velja $M \rightarrow N$, če $M \rightarrow_\beta N$ ali $M \rightarrow_\eta N$. Tranzitivna ovojnica te relacije je $\twoheadrightarrow$.

Definicija 3.8. (i) *β-redeks* je izraz oblike $(\lambda x.M)N$.
η-redeks je izraz oblike $\lambda x.(Mx)$.
Izraz je *redeks*, če je β-redeks ali η-redeks.

(ii) Izraz je v *normalni obliki*, če ne vsebuje nobenega redeksa.

(iii) Izraz M *ima* normalno obliko, če obstaja takšen izraz N, da velja $M \twoheadrightarrow N$ in je N v normalni obliki.

V kontekstu programskih jezikov so lambda računu pogosto dodane še konstante (vrednosti in vgrajene funkcije), ki so podrejene svojim pravilom

redukcije. Takšna pravila lahko obravnavamo kot dodatne aksiome, katerim rečemo δ-konverzije.

Redukcija do normalne oblike v programskih jezikih pogosto ni potrebna ali učinkovita, zato definiramo še eno obliko izrazov [10].

Definicija 3.9. Izraz je v *šibki korenski normalni obliki* (angl. *weak head normal form*, WHNF), če je oblike

$$(((MN_1)N_2)\cdots N_n)$$

in je M spremenljivka, vrednost ali vgrajena funkcija, za katero velja

$$\forall m \leq n : (((MN_1)N_2)\cdots N_m) \text{ ni } \beta \text{ ali } \delta\text{-redeks}$$

Opomba. Šibka korenska normalna oblika (WHNF) pomeni, da je popolnoma reduciran samo izraz na najvišjem nivoju. Na primer izraz $\lambda x.((\lambda y.y)x)$ je v WHNF.

Vsak izraz v normalni obliki je tudi v šibki korenski normalni obliki. Obratno ne velja.

3.1.3 Vrstni red izračuna izrazov

λ-izraze izračunamo tako, da nad njimi opravimo vse možne redukcije. Na ta način lahko pridemo do izraza v normalni obliki, ki je (če obstaja) enolično določen.

Izrek 3.1 (Church-Rosser)**.** *Če velja $M \twoheadrightarrow N_1$ in $M \twoheadrightarrow N_2$, potem obstaja takšen izraz N_3, da $N_1 \twoheadrightarrow N_3$ in $N_2 \twoheadrightarrow N_3$.*

Vendar izrazi lahko vsebujejo tudi več redeksov. Vrstni red njihovega izračunavanja je pomemben, saj ne vodi vsaka stragtegija do normalne oblike.

Za primer si oglejmo izraz $(\lambda x.(xx))(\lambda x.(xx))$. Opravimo lahko le β-redukcijo in dobimo $(\lambda x.(xx))(\lambda x.(xx)) \rightarrow_\beta (\lambda x.(xx))(\lambda x.(xx)) \rightarrow_\beta \cdots$. Ta izraz torej nima normalne oblike. Označimo ga z M.

Uporabimo M v izrazu $(\lambda x.y)M$. Na voljo imamo več poti redukcije. Lahko najprej poskusimo reducirati M, vendar smo ravno prej videli, da tako ne pridemo nikamor. Če pa izvedemo β-redukcijo

$$(\lambda x.y)M \rightarrow_\beta (\lambda x.y)[x := M] = y$$

, dobimo izraz v normalni obliki.

Izrek 3.2. *Če izraz M ima normalno obliko, potem zaporedna redukcija najbolj levega redeksa v izrazu pripelje do izraza v normalni obliki.*

Opomba. Za strategijo, ki vsak izraz z normalno obliko pripelje do normalne oblike, pravimo, da izraze *normalizira*. V zgornjem izreku nastopa primer takšne strategije.

V implementacijah lambda računa najdemo več različnih strategij redukcije.

1. **Normalna redukcija** - najprej reduciramo redeks, ki se nahaja najbolj zunaj najbolj levo v izrazu. $((\lambda x.y)((\lambda x.x)z) \rightarrow y)$

2. **Call-by-name redukcija** - podobno kot normalna, vendar ne reduciramo redeksov, ki se nahajajo znotraj λ-abstrakcij.

3. **Call-by-need redukcija** - kot call-by-name, vendar v implementaciji argument pri β-redukciji ovrednotimo samo enkrat in ga v telesu abstrakcije (za razliko od call-by-name) ne podvajamo.

4. **Aplikativna redukcija** - najprej reduciramo redeks, ki se nahaja najbolj znotraj najbolj levo v izrazu. $((\lambda x.y)((\lambda x.x)z) \rightarrow (\lambda x.y)z)$

5. **Call-by-value redukcija** - najprej ovrednotimo najbolj zunanji redeks, vendar pred β-redukcijo najprej popolnoma poenostavimo argument.

Izmed zgornjih normalizira le normalna redukcija. Call-by-name in call-by-need pripeljeta do WHNF. Aplikativna in call-by-value redukcija ne reducirata vseh izrazov do WHNF ali normalne oblike, kajti lahko generirata

neskončen niz redukcij (v kontekstu računalništva to pomeni, da se program nikoli ne konča).

Glede na način izračunavanja funkcij ločimo programske jezike na *striktne* in *ne-striktne*. V striktnih jeziki je možno definirati le funkcije, katerih argumenti bodo izračunani pred izračunom funkcije (call-by-value). Ne-striktni dovoljujejo tudi definicijo funkcij, katerih argumenti niso izračunani pred vstopom v funkcijo (call-by-need).

3.2 Haskell

Programski jezik Haskell [11] je eden najpopularnejših funkcijskih jezikov. Osnovan je na tipizirani različici lambda računa z več dodatki. Odlikujejo ga zelo lepe teoretične lastnosti, kot sta čistost funkcij in ne-striktna semantika. Čistost funkcij pomeni, da nobena funkcija nima stranskih učinkov in njen rezultat ni odvisen od globalnega stanja programa.

Sintaksa Haskella je precej podobna sintaksi lambda računa. Vključuje več konstruktov, s katerimi olajša programiranje in poveča berljivost programov.

Sintaksa funkcij v Haskellu izgleda tako:

```
f :: Int -> Int  -- deklaracija tipa
f x = x + 1      -- definicija funkcije
f 3              -- klic funkcije
(\x -> x + 1)    -- lambda abstrakcija
```

V prvi vrstici določimo tip funkcije, v drugi pa njeno telo. Deklaracija tipa ni obvezna, saj lahko Haskell prevajalnik o tipih sklepa sam. Sintaksa klica funkcije je takšna kot sintaksa aplikacije v lambda računu. Kot v lambda računu lahko funkcijo zapišemo tudi z λ-abstrakcijo.

Funkcije z več argumenti zapišemo v *curryirani* (angl. *curried*) obliki.

```
g :: Int -> Int -> Int
g x y = x^2 + y
g x = \y -> x^2 + y    -- ekvivalentna definicija
g = (\x y -> x^2 + y)  -- ekvivalentna definicija
```

```
g 3 5                   -- klic
(g 3) 5                 -- ekvivalenten klic
```

Nekaj dodatnih sintaksnih konstruktov:

- **Let**: `let v = B in E` predstavlja povezavo imena z vrednostjo. Povsod znotraj izraza `E`, kjer se pojavi spremenljivka `v`, bo uporabljena ista vrednost, ki se izračuna ob prvi uporabi (call-by-need). Znotraj enega izraza 'let' lahko definiramo tudi več spremenljivk.
- **Where**: `E where v = B` ima podoben pomen kot `let`, vendar ni izraz, temveč je vezan na oklepajoči sintaksni konstrukt (npr. deklaracijo funkcije).
- **If**: `if E then A else B` je izraz, katerega vrednost je `A`, če `E == True`, in `B`, če `E == False`.
- **Case**: Haskell podpira primerjanje vzorcev. Vzorec je lahko spremenljivka, konstanta ali izraz sestavljen iz konstruktorjev in spremenljivk. Izraz 'case' je

  ```
  case E of
    P1 -> E1
    P2 -> E2
    ...
  ```

  , kjer so `P1`,`P2`, ... vzorci, `E`, `E1`, `E2`, ... pa izrazi.

  Če se vrednost izraza `E` ujema z vzorcem `Pi` (in nobenim pred njim), potem je vrednost izraza 'case' `Ei`.

Haskell uporablja call-by-need strategijo izračunavanja izrazov. Implementaciji takšne strategije se pogosto reče *leno ovrednotenje* (angl. *lazy evaluation*). Zato lahko v Haskellu implementiramo tudi neskončne sezname:

```
ones = 1 : ones
```

3.3 Opis ščepca v funkcijskem jeziku

V prejšnjem poglavju smo si ogledali osnovne tipe vozlišč, ki lahko nastopajo v grafih Maxelerjevih ščepcev. Omenili smo tudi, da bi bili funkcijski jeziki lahko zelo primerni za opise takšnih grafov. Tu bomo predstavili idejo, kako s čistimi funkcijami opisati vse tipe vozlišč. V naslednjem poglavju na podlagi te ideje definiramo nov programski jezik *MaxHs*, ki služi programiranju ščepcev za Maxelerjeve podatkovno-pretokovne enote.

Tok podatkov

Temeljni koncept pri programiranju podatkovno-pretokovnih enot je tok podatkov. V jeziku MaxJ ga predstavljajo objekti razreda `DFEVar`.

Za opis tokov podatkov v funkcijskem jeziku uporabimo sezname, katerih elemente omejimo na vrednosti atomarnih tipov (tipov, s katerimi lahko operira DFE).

Omejitev na atomarne tipe je potrebna, ker lahko Maxelerjeva arhitektura izvaja operacije le nad vrednostmi atomarnih tipov. Za predstavitev toka podatkov zato uporabimo poseben tip, ki se obnaša kot seznam, vendar so njegovi elementi zagotovo atomarne vrednosti.

Kompozicija funkcij

V funkcijskem programiranju je kompozicija eden glavnih konstruktov. V podatkovno-pretokovnem grafu se kompozicija funkcij, ki delujeta nad tokovi podatkov, prevede v povezavo med dvema vozliščema. Izhod končnega vozlišča prve funkcije je ravno vhod v začetno vozlišče druge.

Računska vozlišča

Računske operacije, ki jih podpirajo Maxelerjeve podatkovno-pretokovne enote, lahko v našem funkcijskem jeziku predstavimo z nekaj vgrajenimi operatorji, katerih argumenti so vrednosti atomarnih tipov.

Računska vozlišča nad dvema vhodnima tokoma izvedejo določeno računsko operacijo, katere rezultat je njihov izhod. Operacije nad tokovi, ki jih podpira Maxelerjeva arhitektura, lahko v funkcijskem jeziku predstavimo s funkcijama za procesiranje tokov:

```
map op stream
zipWith op stream1 stream2
```

Njun pomen ja takšen, kot za podobni funkciji v Haskellu. Funkcija `map` aplicira podano funcijo na vsaki komponenti vhodnega toka. Funkcija `zipWith` pa operacijo izvede na vsakem paru komponent dveh vhodnih tokov.

Če kot operacijo funkciji `zipWith` podamo enega izmed osnovnih operatorjev, to odraža natanko del podatkovno-pretokovnega grafa, kjer dva tokova tečeta v računsko vozlišče.

Vrednostna vozlišča

Vrednostna vozlišča predstavljajo ravno konstante uporabljene v funkcijskem opisu grafa.

Vozlišča za odmik

Zamik toka podatkov lahko predstavimo kot funkcijo `offset off stream`. Rezultat te funkcije je nov tok, kjer so vrednosti zamaknjene za podano število mest. V Maxelerjevi arhitekturi to predstavlja zakasnitev toka podatkov v času.

Multiplekserji

Tudi multiplekser je lahko predstavljen kot funkcija nad tokovi podatkov. Njeni argumenti so kontrolni tok ter več vhodnih tokov, rezultat pa nov tok, v katerem so vhodni tokovi združeni na podlagi kontrolnega.

Števci

Števci predstavljajo naraščajoče zaporedje celih števil.

```
counter max step
```

Vsak števec se začne z vrednostjo 0 in narašča do vrednosti prvega argumenta, po tem se vrne na 0. Korak naraščanja je določen z drugim argumentom.

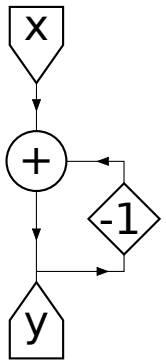
Zanke v grafih ščepcev

Nekateri programi, ki jih želimo izvajati na podatkovno-pretokovnih enotah, zahtevajo uporabo zank. Zanka se v podatkovno-pretokovnem grafu pojavi, kadar za izračun elementov nekega toka podatkov potrebujemo prejšnje elemente istega toka.

V ne-striktnem funkcijskem programskem jeziku lahko takšno zanko predstavimo z rekurzivno definicijo vrednosti, na primer

```
ys = zipWith (+) xs (offset (-1) ys)
```

V zgornji definiciji se spremenljivka `ys` pojavi v telesu svoje definicije. To odraža tok podatkov, ki v podatkovno-pretokovnem grafu teče v prejšnjo fazo lastnega izračuna.



Slika 3.1: Podatkovno-pretokovni graf z zanko.

Poglavje 4

Programski jezik MaxHs

V tem poglavju definiramo nov programski jezik, ki je namenjen programiranju Maxelerjevih podatkovno-pretokovnih enot. Sintaksa in semantika sta podobni sintaksi in semantiki Haskella, vendar podpira zelo majhen del njegovih zmogljivosti. Temelji na idejah opisanih v razdelku 3.3.

Rezultat prevajanja programa v MaxHs je Haskellov modul, ki ga lahko uporabimo za klicanje Maxelerjevih podatkovno-pretokovnih enot iz Haskella.

Poleg Haskellovega modula je rezultat še konfiguracijska datoteka (.max), s pomočjo katere lahko glavni program napišemo v drugih jezikih.

4.1 Struktura programa

Ščepec za Maxelerjevo podatkovno-pretokovno enoto predstavlja neko operacijo nad tokovi podatkov. Vhodnih in izhodnih tokov je lahko več. V funkcijskem jeziku je ščepec za podatkovno-pretokovno enoto funkcija, katere argumenti so tokovi podatkov, rezultat pa terka (angl. *tuple*) tokov:

```
kernel ::  Stream a -> Stream b -> ...
       -> (Stream c ,  Stream d ,  ...)
```

V jeziku MaxHs je vsak ščepec opisan v svoji datoteki. Program se začne z opisom modula, iz katerega izvozimo funkcijo, ki predstavlja ščepec:

```
module Kernel(kernel) where { ... }
```

V tem primeru je ime modula `Kernel`, ime funkcije pa `kernel`.

V telesu modula se mora pojaviti definicija funkcije, ki je izvožena, in njen tip mora biti primeren za Maxelerjevo podatkovno-pretokovno enoto, tj. `Stream a -> Stream b -> ... -> (Stream c, Stream d, ...)`.

4.2 Sintaksa in semantika

Sintaksa jezika MaxHs je skladna s Haskellovo, vendar ne vključuje njegovih zapletenejših konstruktov, ki za programiranje podatkovno-pretokovnih enot niso tako pomembni.

Spodaj bomo opisali sintakso jezika MaxHs. *Vmesni* simboli so zapisani kurzivno, `končni` pa v pisavi pisalnega stroja. Celotna sintaksa je opisana v dodatku A.

Vsak program je zapisan v svojem modulu.

$$\textit{module} \;\rightarrow\; \texttt{module}\ \textit{modid}\ \textit{exports}\ \texttt{where}\ \textit{body}$$

Sintaksa sicer dovoljuje več izvoženih funkcij, vendar jezik dovoljuje natanko eno – funkcijo ščepca.

$$\begin{aligned} \textit{exports} \;&\rightarrow\; \epsilon \\ &\mid\; \texttt{(}\ \textit{vars}\ \texttt{)} \end{aligned}$$

Telo modula je sestavljeno iz deklaracij.

$$\begin{aligned} \textit{body} \;&\rightarrow\; \{\ \textit{decls}\ \} \\ \textit{decls} \;&\rightarrow\; \textit{decl} \\ &\mid\; \textit{decl}\ \texttt{;}\ \textit{decls} \end{aligned}$$

4.2.1 Deklaracije

Deklaracije lahko opisujejo tip spremenljivk, funkcijo ali spremenljivko.

$$
\begin{array}{lcl}
\textit{decl} & \rightarrow & \textit{gendecl} \\
 & | & \textit{funlhs rhs} \\
 & | & \textit{var rhs} \\
\textit{gendecl} & \rightarrow & \textit{vars} \texttt{ :: } \textit{type} \\
\textit{funlhs} & \rightarrow & \textit{var args} \\
\textit{rhs} & \rightarrow & \texttt{=} \textit{ exp} \\
 & | & \texttt{=} \textit{ exp} \texttt{ where \{ } \textit{decls} \texttt{ \}}
\end{array}
$$

Za vsako deklaracijo spremenljivke ali funkcije mora obstajati pripadajoča deklaracija tipa.

Simbol *rhs* predstavlja desno stran deklaracije, v kateri nastopa izraz, ki določa vrednost spremenljivke ali funkcije. S konstruktom 'where' v program vpeljemo lokalne definicije, ki so veljavne le znotraj sintaksnega konstrukta, ki ga predstavlja simbol *rhs*. Trenutno je 'where' praktično enakovreden izrazu 'let', definiranem na sledečih straneh.

Jezik MaxHs dovoljuje definicijo ne-striktnih funkcij. Vrstni red deklaracij ni pomemben in definicije so lahko vzajemno rekurzivne.

Definicije simbolov *vars*, *type* in *args* so zapisane v dodatku A, kar velja tudi za vse izpuščene definicije od tod dalje.

4.2.2 Izrazi

Izrazi so lahko sestavljeni iz vgrajenih operatorjev, katerih moč vezave je določena v gramatiki. Najšibkeje veže operator za bitni ali (`|`), sledi mu bitni ekskluzivni ali (`^`) in tako dalje. Pri vsakem izrazu lahko zapišemo tudi njegov tip.

$$
\begin{array}{lcl}
\textit{exp} & \rightarrow & \textit{bit\_or\_exp} \texttt{ :: } \textit{type} \\
 & | & \textit{bit\_or\_exp} \\
\textit{bit\_or\_exp} & \rightarrow & \textit{bit\_or\_exp} \texttt{ | } \textit{bit\_xor\_exp} \\
 & | & \textit{bit\_xor\_exp} \\
 & \vdots &
\end{array}
$$

Vgrajeni operatorji v jeziku MaxHs so `|`, `^`, `&`, `===`, `!==`, `<`, `>`, `<=`, `>=`, `<<`, `>>`, `+`, `-`, `*`, `/` in unarni `-`. Našteti so po naraščajoči moči vezave, kjer so si med seboj enakovredni primerjalni operatorji ter pari operatorjev (`<<`, `>>`), (`+`, `-`) in (`*`, `/`).

Močneje kot vsi operatorji vežejo izraz 'let', izraz 'if' in klic funkcije.

$$
\begin{array}{lcl}
 & \vdots & \\
\textit{pre\_exp} & \rightarrow & \texttt{-}\ \textit{epx10} \\
 & | & \textit{exp10} \\
\textit{exp10} & \rightarrow & \texttt{let}\ \{\ \textit{decls}\ \}\ \texttt{in}\ \textit{exp} \\
 & | & \texttt{if}\ \textit{exp}\ \texttt{then}\ \textit{exp}\ \texttt{else}\ \textit{exp} \\
 & | & \textit{fexp}
\end{array}
$$

Izraz 'let' predstavlja izraz, znotraj katerega lahko definiramo lokalne spremenljivke in funkcije. Uporabimo ga lahko povsod, kjer bi uporabili konstrukt 'where', in tudi kjerkoli znotraj izrazov.

Izraz 'if' v jeziku MaxHs ni preveč uporaben, saj bomo kasneje zaradi boljšega prevoda na Maxelerjevo arhitekturo njegov tip omejili na izključno atomarne tipe.

Simbol *fexp* lahko predstavlja neko spremenljivko, vrednost ali klic funkcije, kjer so njeni argumenti ločeni s presledki.

$$
\begin{array}{lcl}
\textit{fexp} & \rightarrow & \textit{aexp} \\
 & | & \textit{fexp}\ \textit{aexp} \\
\textit{aexp} & \rightarrow & \textit{var} \\
 & | & \textit{literal} \\
 & | & \texttt{(}\ \textit{exps}\ \texttt{)} \\
 & | & \texttt{[}\ \textit{exps}\ \texttt{]} \\
\textit{var} & \rightarrow & \textit{varid}
\end{array}
$$

Leksikalna pravila so sledeča: vsa imena so sestavljena iz znakov `a`–`z`, `A`–`Z`, `0`–`9` ali `_`. Imena spremenljivk (*varid*) se začnejo z malo začetnico ali `_`, imena tipov in ime modula pa z veliko.

Literali so lahko cela števila zapisana v desetiškem sistemu, decimalna realna števila ter konstanti `True` in `False`.

4.3 Sistem tipov

Ena najpomembnejših lastnosti Haskella je njegov sistem tipov. Pri programiranju nam pomaga odkriti velik delež napak, ki bi jih sicer našli šele pri testiranju.

Sistem tipov jezika MaxHs je veliko preprostejši od Haskellovega in večinoma odraža tipe, s katerimi lahko računajo Maxelerjeve podatkovno-pretokovne enote.

4.3.1 Atomarni tipi

V jeziku MaxHs smo za atomarne tipe izbrali 32-bitna cela števila, števila s plavajočo vejico v dvojni natančnosti in logični tip. V Maxelerjevi arhitekturi sicer lahko definiramo tudi druge velikosti števil s fiksno ali plavajočo vejico, vendar smo se zaradi enostavnosti omejili na ta dva tipa. Kasneje bo brez večjih težav možno vpeljati tudi druge.

32-bitna števila predstavimo z vrednostmi tipa `Int`, števila s plavajočo vejico z `Double`. Logični tip se imenuje `Bool`.

4.3.2 Tip toka podatkov

Tok podatkov predstavimo z vrednostjo tipa `Stream a`, kjer je `a` eden izmed atomarnih tipov.

Možni tokovi so torej `Stream Int`, `Stream Double` in `Stream Bool`. Zaradi omejitev Maxelerjeve arhitekture lahko vrednosti tipa `Stream Bool` uporabljamo le znotraj implementacije ščepca, argumenti in rezultati pa morajo biti tipa `Stream Int` ali `Stream Double`.

4.3.3 Tipi funkcij

Funkcije imajo tip `a -> b`, kjer sta `a` in `b` lahko poljubna tipa (ne nujno atomarna). V MaxHs-ju lahko definiramo tudi funkcije višjega reda.

Ena izmed trenutnih omejitev jezika MaxHs je odsotnost polimorfnih funkcij. Znotraj jezika ni mogoče definirati polimorfnih funkcij, vseeno pa so polimorfne nekatere vgrajene funkcije, ki jih bomo opisali v naslednjem razdelku.

V kasnejših razširitvah jezika bo možno brez večjih sprememb dodati tudi podporo polimorfnim funkcijam.

4.3.4 Terke in seznami

V Haskellu so terke in seznami uporabljeni zelo pogosto. V MaxHs-ju vlogo seznamov večinoma nadomestijo tokovi podatkov (`Stream`), pravi seznami pa so potrebni za uporabo nekaterih vgrajenih funkcij. Te so tudi edine, ki lahko s seznami operirajo. Običajne funkcije za delo s seznami iz Haskella v MaxHs-ju ne obstajajo.

Terke so uporabljene le za rezultat izvožene funkcije – funkcije ščepca. Ker MaxHs ne podpira primerjanja vzorcev in ne vsebuje funkcij za delo s terkami, jih znotraj programa ni mogoče uporabljati.

4.4 Vgrajene funkcije

Za opis ščepcev potrebujemo v jeziku MaxHs še nekaj vgrajenih funkcij, ki predstavljajo funkcionalnosti, ki jih ponuja Maxelerjeva arhitektura. Takšne funkcije smo predstavili že v razdelku 3.3. Tu bomo podali njihovo natančnejšo definicijo in še nekaj dodatnih funkcij.

Naslednje definicije med drugim vsebujejo tudi tipe vgrajenih funkcij. Nekatere izmed njih so polimorfne, pri takšnih v definicji njihovega tipa uporabimo spremenljivke `a`, `b`, ..., ki vedno predstavljajo enega izmed atomarnih tipov.

Ker jezik MaxHs ne podpira uporabniško definiranih polimorfnih funkcij, takšne definicije znotraj programov niso veljavne.

4.4.1 Map in zipWith

Funkciji `map` in `zipWith` predstavljata izvajanje določene operacije nad vhodnimi tokovi podatkov.

```
map :: (a -> b) -> Stream a -> Stream b
```

Prvi argument `map` je funkcija, ki predstavlja operacijo nad vrednostjo atomarnega tipa. Klic `map` povzroči, da se operacija izvede nad vsemi elementi podanega toka.

```
zipWith :: (a -> b -> c) -> Stream a -> Stream b -> Stream c
```

Podobno kot pri `map` je prvi argument operacija, s katero v tem primeru združimo dva toka podatkov.

4.4.2 Offset

```
offset :: Int -> Stream a -> Stream a
```

Rezultat te funkcije je tok podatkov, zamaknjen za število mest, podano v prvem argumentu. Če je vrednost prvega argumenta negativna se tok zamakne v desno, sicer levo. Vrednosti, ki se ob zamiku dodajo, so nedefinirane. Omejitev je, da mora biti vrednost zamika znana že ob času prevajanja, ker vozlišče za odmik v Maxelerjevi arhitekturi predstavlja statičen odmik.

4.4.3 Mux

```
mux :: Stream Int -> [Stream a] -> Stream a
```

Funkcija `mux` predstavlja multiplekser. Prvi argument je kontrolni tok števil, drugi pa seznam vhodnih tokov. Števila v kontrolnem toku določajo, element katerega izmed vhodnih tokov se preslika na izhod.

Na ta način s funkcijo `mux` združimo vhodne tokove na podlagi kontrolnega toka.

4.4.4 IntToDouble in doubleToInt

```
intToDouble :: Int -> Double
doubleToInt :: Double -> Int
```

Ti dve funkciji služita pretvarjanju tipov. Če želimo pretvoriti tok podatkov, lahko seveda uporabimo `map intToDouble` ali `map doubleToInt`.

4.4.5 Counter in counterChain

Števec je neskončen tok celih števil, ki se začnejo pri 0 in naraščajo po določenem koraku do največje vrednosti, pri kateri se vrnejo na 0.

```
counter :: Int -> Int -> Stream Int
```

Prvi argument je najvišja vrednost, drugi pa korak naraščanja.

Števce lahko v Maxelerjevi arhitekturi združujemo v verige. To pomeni, da števec, ki nastopa pred drugim števcem v verigi, narašča počasneje. Njegova vrednost se spremeni šele, ko števec za njim opravi cel krog.

```
counterChain :: [(Int, Int)] -> [Stream Int]
```

Ta funkcija predstavlja ustvarjanje verige števcev. Argument je seznam najvišjih vrednosti in korakov za vsak števec. Rezultat je seznam števcev, v katerem prvi narašča najpočasneje, zadnji najhitreje.

Za dostop do števcev iz verige imamo na voljo funkcijo , ki iz seznama vrne želeni tok: `at :: Int -> [Stream a] -> Stream a`.

Verigo števcev lahko uporabimo na sledeči način:

```
let
{ cc = counterChain [(4,2), (3,1)]
; c1 = at 0 cc
; c2 = at 1 cc
}
in
  zipWith add c1 c2
```

Vrednost tega izraza je tok števil

```
    0, 1, 2, 2, 3, 4, 0, 1, 2, 2, 3, 4, ...
```

, saj sta vrednosti `c1` in `c2`

```
c1 = 0, 0, 0, 2, 2, 2, 0, 0, 0, 2, 2, 2, ...
c2 = 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, ...
```

4.5 Povezava s Haskellom

Ščepec opisan v programskem jeziku MaxHs se prevede v Haskellov modul, iz katerega je izvožena funkcija za uporabo Maxelerjeve podatkovno-pretokovne enote.

V MaxHs-ju ima izvožena funkcija tip

```
Stream a -> Stream b -> ... -> (Stream c, Stream d, ...)
```

, kjer `a`, `b`, `c` in `d` predstavljajo `Int` ali `Double`.

V Haskellu predstavimo tip `Stream` z `Data.Vector.Storable.Vector`, saj nam to omogoča hiter prenos podatkov na podatkovno-pretokovno enoto. Tip `Int` v Haskellu ustreza tipu `Int32`, `Double` pa ima enak pomen. Modul `Data.Vector.Storable` bomo od zdaj naprej označevali s `SV`.

Zgornji tip je v Haskellu torej

```
    SV.Vector a -> SV.Vector b -> ...
-> (SV.Vector c ,  SV.Vector d ,  ...)
```

, kjer so `a`, `b`, `c` in `d` lahko `Int32` ali `Double`.

Tip `Data.Vector.Storable.Vector` v Haskellu omogoča preprost prenos polj v C funkcije. Za delo z vrednostmi tega tipa imamo na voljo funkcije za pretvorbo v C kazalce in iz njih. Te operacije so hitre in nam zato omogočajo učinkovit prenos polja vrednosti v C funkcijo, kar je v našem primeru pomembno. Za komunikacijo s podatkovno-pretokovno enoto imamo namreč na voljo le C funkcije.

4.6 Omejitve jezika MaxHs

Definicija jezika MaxHs močno temelji na konstruktih, ki nam jih ponuja Maxelerjeva arhitektura. Ker ima vsak od teh konstruktov svoje omejitve, jih moramo upoštevati tudi v definiciji našega jezika.

V Maxelerjevi arhitekturi večino operacij izvajamo nad tokovi atomarnih vrednosti. Zato smo definirali tip `Stream a` drugače kot tip seznama. Seznam

lahko vsebuje elemente poljubnega tipa, `Stream` pa le elemente atomarnih tipov.

Sami seznami v jeziku MaxHs nimajo pomembne vloge, saj pri programiranju ščepca vedno opisujemo operacije nad tokovi podatkov (`Stream`). Vseeno smo vključili tudi tip seznam, ker nam je pomagal pri definicije nekaterih vgrajenih funkcij. V Maxelerjevi arhitekturi splošnega seznama ne moremo učinkovito predstaviti, zato je podpora operacijam nad seznami zelo omejena. Nad njimi delujejo le vgrajene funkcije `mux`, `counterChain` in `at`.

Podobno kot seznamov tudi terk ne moremo učinkovito predstaviti v podatkovno-pretokovnem grafu. V definiciji jezika jih vseeno potrebujemo pri tipu glavne funkcije ščepca in pri funkciji `counterChain`. V končnem podatkovno-pretokovnem grafu ne seznami ne terke niso eksplicitno predstavljivi, zato jih funkcije, ki nad njimi delujejo, že v času prevajanja razstavijo na posamezne komponente.

Programski jezik MaxHs smo nekoliko omejili tudi z izbiro osnovnih vozlišč. Maxelerjeva arhitektura jih namreč ponuja več. Primer vozlišča, ki ni vključeno v naš izbor, je vozlišče za dinamičen odmik, pri katerem je zamik določen s tokom števil in ne konstanto. Konstanta pri vgrajeni funkciji `offset` mora biti zaradi naše izbire znana že v času prevajanja, saj `offset` predstavlja vozlišče s statičnim odmikom.

Način prevajanja, ki je predstavljen v tem diplomskem delu, je neodvisen od izbire osnovnih vozlišč. Tako bo v prihodnosti možno dodati tudi preostale tipe vozlišč.

Poglavje 5

Programiranje z MaxHs

V 2. poglavju smo si ogledali primer programiranja aplikacije za Maxelerjeve podatkovno-pretokovne enote z uporabo obstoječih orodji. Zdaj bomo predstavili implementacijo istega primera v jeziku MaxHs.

Program, tako kot prej, sestoji iz dveh delov. Prvi del je opis grafa ščepca za podatkovno-pretokovno enoto, ki bo zdaj zapisan v MaxHs. Drugi del pa je program za CPE, katerega bomo napisali v Haskellu.

5.1 Programiranje grafa ščepca

Implementirati želimo ščepec, ki izračuna tekoče povprečje treh členov v zaporedju. Glavna funkcija ščepca je

```
movingAverage :: Stream Double -> Stream Double
movingAverage xs = ...
```

Problema se lotimo tako, da najprej izračunamo zaporedje vsot vsake trojice členov in na koncu elemente tega zaporedja delimo s 3. Za izračun zaporedja vsot moramo po komponentah sešteti vhodni tok s seboj, zamaknjenim za eno mesto v desno, ter nato še s seboj, zamaknjenim v levo. Za zamik toka uporabimo funkcijo `offset`, za seštevanje po komponentah pa `zipWith` z operacijo seštevanja `add`.

Prva vsota se torej glasi:

```
s1 :: Stream Double
s1 = zipWith add xs (offset (-1) xs) xs
```

in druga:

```
s2 :: Stream Double
s2 = zipWith add s1 (offset 1 xs)
```

Spremenljivka `s2` predstavlja zaporedje vsot, katerega elemente moramo deliti s 3:

```
map div3 s2
```

Funkcija `div3` predstavlja deljenje s 3.

Celotna izvorna koda je zapisana spodaj v programu 5.1.

Program 5.1 MovingAverage.maxhs

```
module MovingAverage(movingAverage) where
{ movingAverage :: Stream Double -> Stream Double
; movingAverage xs =
    let
    { s1 :: Stream Double
    ; s1 = zipWith add (offset (-1) xs) xs
    ; s2 :: Stream Double
    ; s2 = zipWith add s1 (offset 1 xs)
    }
    in
      map div3 s2

; add :: Double -> Double -> Double
; add x y = x + y
; div3 :: Double -> Double
; div3 x = x / 3.0
}
```

5.2 Programiranje glavne CPE aplikacije v Haskellu

Rezultat prevajanja programa 5.1 je Haskellov modul, iz katerega je izvožena funkcija

```
movingAverage :: Data.Vector.Storable.Vector Double
              -> Data.Vector.Storable.Vector Double
```

Ta funkcija uporabi Haskellov sistem za klicanje tujih funkcij (angl. *Foreign Function Interface*, FFI), s katerim pokliče C funkcjo, ki jo pripravi zadnji del prevajalnika za MaxHS (ta ni del diplomskega dela).

V glavnem programu (5.2) jo lahko uporabimo kot katerokoli drugo funkcijo, čeprav v resnici povzroči prenos izvajanja na DFE.

Program 5.2 MovingAverageMain.hs

```
module MovingAverageMain where
import MovingAverage
import qualified Data.Vector.Storable as SV
main =
  let
    xs = SV.fromList [1.0,2.0,1.0,3.0,1.0]
    out = movingAverage xs
  in
    print out
```

5.3 Izračun tekoče vsote

Prednost uporabe jezika MaxHs za programiranje DFE-jev se v prejšnjem primeru pokaže v tem, da lahko programer CPE aplikacijo in opis ščepca za DFE sprogramira v podobnih jezikih. Z uporabo funkcijskega jezika za opis ščepcev pa ne pridobimo samo tega. MaxHs nam ponuja tudi konstrukte, ki jih je v Javi (ali MaxJ) težko realizirati. V sledečem primeru si bomo

ogledali izračun tekoče vsote zaporedja, ki jo definiramo kot zaporedje delnih vsot vhodnega zaporedja.

Najprej bomo predstavili način programiranja, ki bi ga lahko uporabili tudi v programskem jeziku MaxJ, kasneje pa še drugačno rešitev, ki je bolj prilagojena funkcijskemu programskemu jeziku in bi jo v MaxJ veliko težje realizirali.

5.3.1 Osnoven pristop

Glavna funkcija našega ščepca je

```
rollingSum :: Stream Double -> Stream Double
```

Tekočo vsoto zaporedja lahko izračunamo tako, da vsakemu členu zaporedja prištejemo delno vsoto členov, ki se pojavijo pred njim. V MaxHs lahko to zapišemo tako:

```
rollingSum :: Stream Double -> Stream Double
rollingSum xs = ys
  where
  { ys :: Stream Double
  ; ys = zipWith add xs (offset (-1) ys)
  }
```

V definiciji spremenljivke `ys` se nanjo tudi sklicujemo, to lahko naredimo, ker ima MaxHs ne-striktno semantiko. Tok `ys` je tok podatkov, ki ga dobimo s seštevanjem komponent tokov `xs` in `offset (-1) ys`, to pomeni, da vsaki komponenti vhodnega toka prištejemo prejšnjo delno vsoto v zaporedju.

Na podoben način bi lahko ta ščepec opisali tudi v MaxJ, čeprav bi imeli z opisom zanke veliko več dela. V MaxHs pa lahko to rešitev z uporabo funkcij višjega reda še posplošimo.

5.3.2 Boljši pristop

V funkcijski programskih jezikih so zelo pogosto uporabljene funkcije višjega reda. To so funkcije, v katerih kot argument ali rezultat prav tako nastopa funkcija.

V definiciji jezika MaxHs smo že srečali dve funkciji višjega reda: `map` in `zipWith`. Vendar MaxHs dovoljuje tudi uporabniško definirane funkcije višjega reda. To lahko izkoristimo in posplošimo primer tekoče vsote.

Zgoraj smo tekočo vsoto izračunali z definicijo rekurzivne vrednosti

```
ys = zipWith add xs (offset (-1) ys)
```

, kjer smo kot argument `zipWith` podali operacijo seštevanja (`add`). Namesto tega lahko definiramo novo funkcijo, ki ji lahko podamo poljubno operacijo in nam vrne zaporedje delnih rezultatov, dobljenih z uporabo podane operacije na elementih vhoda. Takšno funkcijo poimenujemo `scan` in seveda je višjega reda, saj je njen argument neka operacija.

```
scan :: (Double -> Double -> Double) -> Stream Double
     -> Stream Double
scan op xs = ys
  where
  { ys = zipWith op xs (offset (-1) ys)
  }
```

Zdaj lahko to funkcijo uporabimo za različne namene. V primeru izračuna tekoče vsote izgleda tako:

```
rollingSum :: Stream Double -> Stream Double
rollingSum xs = scan add xs
```

To definicijo lahko še polepšamo, saj lahko izkoristimo MaxHs-jevo podporo delno apliciranim funkcijam. Pomen izraza `scan add xs` je zaradi leve asociativnosti aplikacije pravzaprav `(scan add) xs`. Tu opazimo, da bi lahko definicijo `rollingSum` zapisali brez argumenta (tako, da opravimo η-redukcijo). Končna definicija je

```
rollingSum :: Stream Double -> Stream Double
rollingSum = scan add
```

Vsi programi so v celoti zapisani v dodatku B.

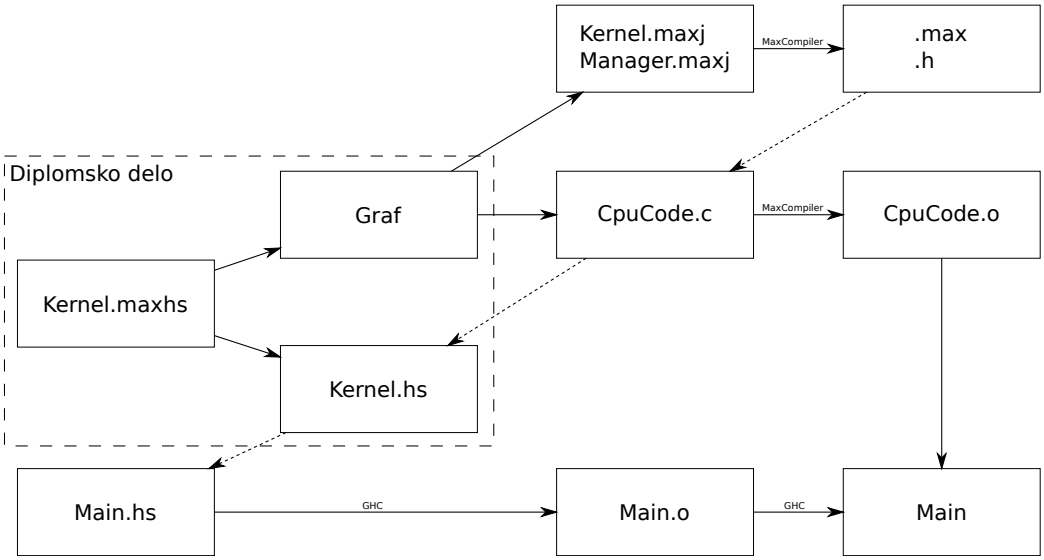
Poglavje 6

Prevajanje v grafno vmesno kodo

Celoten prevajalnik za programski jezik MaxHs deluje na podoben način kot Maxelerjev MaxCompiler. Vhodni program je sestavljen iz glavnega programa v Haskellu ter opisa ščepca v MaxHs. Prevajanje poteka tako, da iz opisa ščepca najprej zgradimo podatkovno-pretokovni graf na osnovi vozlišč, ki jih podpira Maxelerjeva arhitektura. Zadnji del prevajalnika na podlagi grafne vmesne kode generira kodo ščepca v jeziku MaxJ, ki jo potem prevede z Maxelerjevim prevajalnikom.

Poleg prevoda v grafno vmesno kodo moramo v prednjem delu prevajalnika poskrbeti še za povezavo s Haskellom, kar dosežemo z generiranjem Haskellovega modula, ki s pomočjo Haskellove razširitve za podporo klicem tujih funkcij (angl. *Foreign Function Interface*) kliče C funkcijo, ki jo pripravi zadnji del prevajalnika. Na koncu še prevedemo glavni Haskell program ter ga povežemo s C programom za uporabo DFE-ja. Postopek prevajanja je prikazan na sliki 6.1.

Zadnji del prevajalnika je obravnavan v diplomskem delu Mihe Eleršiča "Prevajanje grafne vmesne kode za Maxelerjeve podatkovno-pretokovne enote" [12]. Tu se bomo osredotočili na prevod iz jezika MaxHs v grafno vmesno kodo ter na pripravo Haskell modula.



Slika 6.1: Postopek prevajanja.

Prevajalnik je implementiran v programskem jeziku Java. Izvorno kodo najprej pretvori v zaporedje besed, nato sintaksni analizator iz zaporedja besed zgradi abstraktno sintaksno drevo. Implementacija prvih faz prevajanja sledi opisu v knjigi [13].

Nad abstraktnim sintaksnim drevesom opravimo analizo tipov in nato v zadnji, najpomembnejši fazi na podlagi abstraktnega sintaksnega drevesa zgradimo podatkovno-pretokovni graf, ki je vhod v zadnji del prevajalnika.

6.1 Sintaksna analiza

V fazi sintaksne analize na podlagi zaporedja besed zgradimo abstraktno sintaksno drevo. Kasnejše faze prevajanja izvajajo operacije samo še nad abstraktnim sintaksnim drevesom.

6.1.1 Abstraktno sintaksno drevo

Abstraktno sintaksno drevo sestavljajo različna vozlišča, ki predstavljajo konstrukte uporabljene v izvornem programu. Spodaj so predstavljeni vsi možni

tipi vozlišč tako, kot so implementirani v našem prevajalniku. Vsak razred je zapisan v obliki `Razred (atribut1: tip1, ...)`. Zamik vrstice pomeni, da je razred v tisti vrstici izpeljan iz razreda nad njim z manjšim zamikom. Vsi razredi so izpeljani iz abstraktnega razreda `ASNode`. Povsod je izpuščena predpona `AS`.

```
Module (modid: String, exports: Exports, body: Body)
Body (decls: Decls)
Exports (exports: List<VarExp>)
Decls (decls: List<Decl>)
Decl ()
  ADecl (varid: String)
    VarDecl (exp: Exp)
    ArgDecl ()
    FunDecl (args: List<ArgDecl>, exp: Exp)
    StdLibDecl ()
  TypeDecl (vars: List<String>, type: Type)
Type ()
  AType ()
    AtomType (type: AtomType.Type)
    TupleType (elements: List<Type>)
    ListType (type: Type)
    StreamType (type: Type)
  FunType (arg: Type, result: Type)
Exp ()
  FExp ()
    AExp ()
      VarExp (varid: String)
      LiteralExp (type: LiteralExp.Type, literal: String)
      TupleExp (elements: List<Exp>)
      ListExp (elements: List<Exp>)
    FunExp (fun: Exp, arg: Exp)
  BinExp (operator: BinExp.Operator, exp1: Exp, exp2: Exp)
  IfExp (condExp: Exp, thenExp: Exp, elseExp: Exp)
  LetExp (decls: Decls, exp: Exp)
  WhereExp (decls: Decls, exp: Exp)
  ExpWithType (exp: Exp, type: Type)
```

Celoten modul je predstavljen z objektom razreda `ASModule`, ki vsebuje svoje ime in povezavi na vozlišči razredov `ASExports` ter `ASBody`. `ASExports` vsebuje seznam imen izvoženih funkcij, `ASBody` pa predstavlja telo modula in vsebuje seznam deklaracij.

Deklaracije so predstavljene z vozlišči tipov `ASVarDecl`, `ASFunDecl` ter `ASTypeDecl`. `ASVarDecl` in `ASFunDecl` predstavljata deklaracijo neke spremenljivke, kjer `ASVarDecl` nima argumentov, `ASFunDecl` pa vsebuje še povezave na deklaracije argumentov `ASArgDecl`. Deklaracija tipa vsebuje povezave na imena spremenljivk ter na vozlišče, ki opisuje tip teh spremenljivk. Poleg deklaracij, ki se lahko pojavijo v izvorni datoteki, imamo še deklaracije vgrajenih funkcij, ki jih predstavimo z razredom `ASStdLibDecl`. Takšne deklaracije se v izvorni datoteki seveda ne pojavijo.

Deklaracije spremenljivk ali funkcij vsebujejo še povezavo na izraz, ki določa njihovo vrednost. Izrazi so predstavljeni z vozlišči razreda `ASExp`. Najpreprostejši primeri so `ASVarExp` (spremenljivka), `ASLiteralExp` (literal), `ASTupleExp` (terka) in `ASListExp` (seznam).

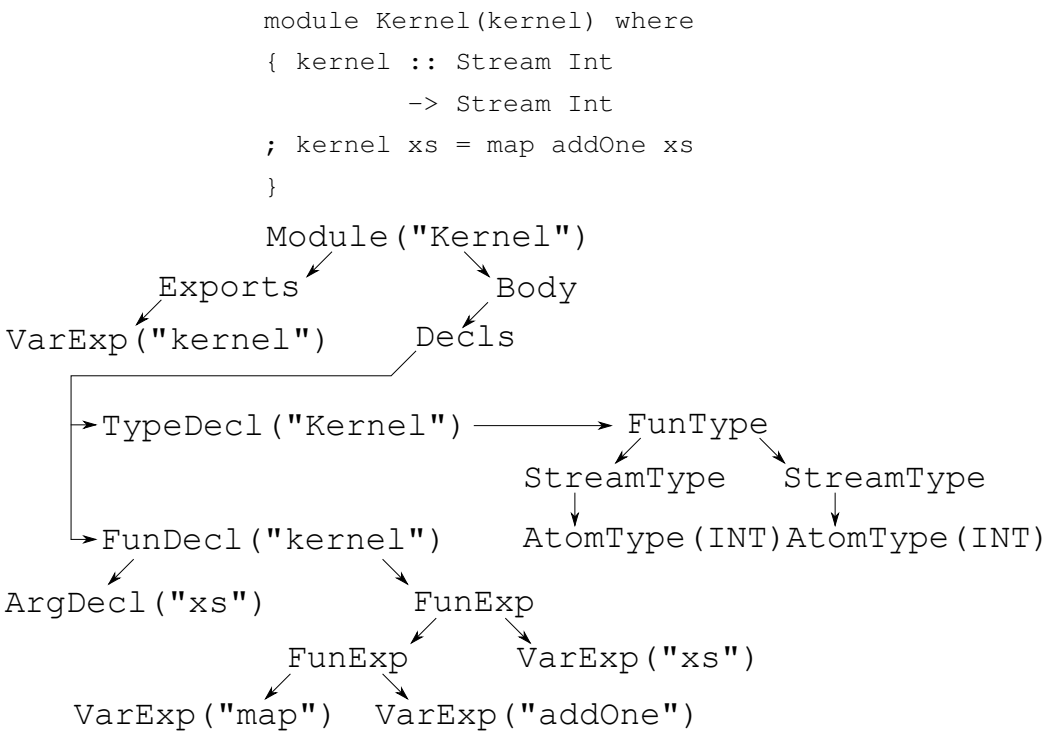
`ASFunExp` predstavlja aplikacijo funkcije. Njegov levi otrok je funkcija, desni pa argument.

Ostala vozlišča izrazov so `ASBinExp`, ki predstavlja izraz z vgrajenim binarnim operatorjem, ter `ASIfExp`, `ASLetExp` in `ASWhereExp`, ki predstavljajo istoimenske konstrukte jezika MaxHs. Izrazi lahko vsebujejo tudi eksplicitno določen tip, za kar uporabimo vozlišče razreda `ASExpWithType`.

6.2 Razreševanje imen in analiza tipov

Pred grajenjem podatkovno-pretokovnega grafa moramo poskrbeti še za vse povezave med uporabami imen (`ASVarExp`) in njihovimi definicijami, ki so lahko `ASVarDecl`, `ASFunDecl`, `ASArgDecl` ali `ASStdLibDecl`.

Da zagotovimo veljavnost določenih operacij v naslednji fazi ter do neke mere preverimo pravilnost programa, pred nadaljevanjem preverimo še skladnost tipov.



Slika 6.2: Primer prevoda programa v abstraktno sintaksno drevo.

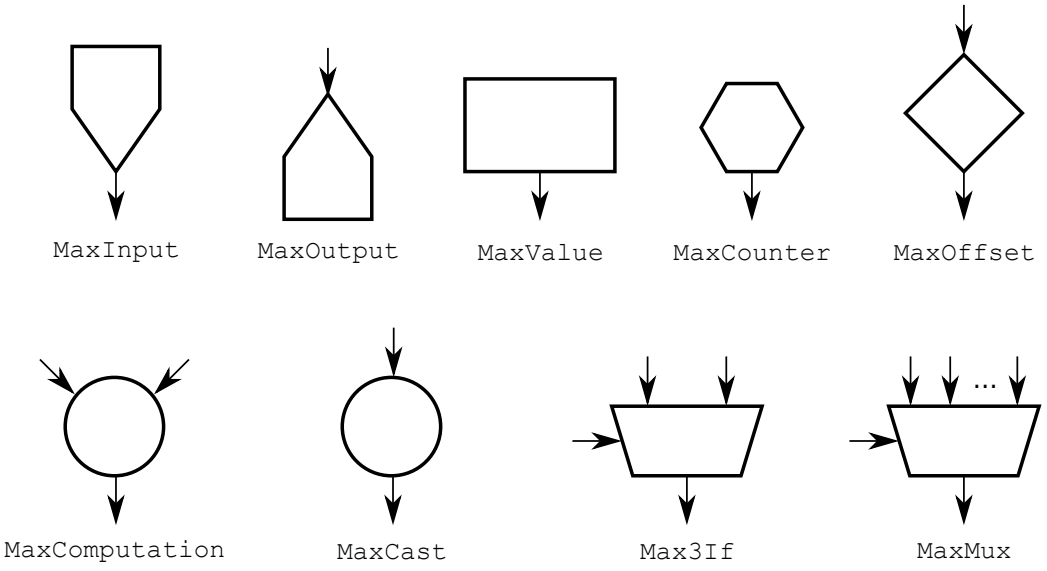
Rezultat faze razreševanja imen je tabela povezav med uporabami imen ter njihovimi definicijami, ki jo potrebujemo v fazi grajenja grafa. Pri analizi tipov pa zgradimo tabelo, ki vsakemu izrazu priredi tip.

6.3 Grajenje grafa

Vhod v to fazo prevajanja je abstraktno sintaksno drevo skupaj s tabelo razrešenih imen in tabelo tipov. Izhod je podatkovno-pretokovni graf, sestavljen iz osnovnih gradnikov, ki jih določa Maxelerjeva arhitektura.

Osnovne gradnike smo opisali že v razdelku 2.2.2. Znotraj prevajalnika so predstavljeni z razredi prikazanimi na sliki 6.3. Vsi so izpeljani iz abstraktnega razreda `MaxNode`.

Tu nastopata še dva gradnika, ki ju do zdaj nismo opisali. `MaxCast` služi pretvarjanju tipov, v MaxHs ga predstavljata funkciji `intToDouble` in



Slika 6.3: Razredi vmesne kode.

`doubleToInt`. `Max3If` je podoben multiplekserju, vendar ima natanko dva vhodna toka in kontrolni tok sestavljen iz logičnih vrednosti.

Atributi predstavljenih razredov so:

- `MaxInput`
  - `name : String` ime vhodnega toka
- `MaxOutput`
  - `input : MaxNode` vozlišče iz katerega prihaja vhodni tok podatkov
  - `name : String` ime izhodnega toka
- `MaxValue`
  - `value : String` vrednost konstante
- `MaxCounter`
  - `max : int` največja vrednost števca
  - `increment : int` korak števca
- `MaxOffset`
  - `offset : int` število mest zamika

  - `input : MaxNode` vozlišče, iz katerega prihaja tok podatkov, ki ga zamika

- `MaxComputation`
  - `operator : Operator` eden izmed vgrajenih binarnih operatorjev
  - `input1 : MaxNode` vozlišče prvega vhodnega toka
  - `input2 : MaxNode` vozlišče drugega vhodnega toka
- `MaxCast`
  - `type : MaxType` tip v katerega spreminja (`INT` ali `DOUBLE`)
  - `input1 : MaxNode` vozlišče vhodnega toka
- `Max3If`
  - `control : MaxNode` vozlišče kontrolnega toka
  - `input1 : MaxNode` vozlišče prvega vhodnega toka
  - `input2 : MaxNode` vozlišče drugega vhodnega toka
- `MaxMux`
  - `control : MaxNode` vozlišče kontrolnega toka
  - `inputs : List<MaxNode>` seznam vhodnih tokov

Vsako izmed vozlišč vsebuje tudi podatek o tipu, kajti ta je potreben v zadnjem delu prevajalnika. Možni tipi so `INT`, `DOUBLE` ali `BOOLEAN`.

6.3.1 Funkcija ščepca

Podatkovno-pretokovni graf je znotraj MaxHs izvorne datoteke opisan s funkcijo, ki je izvožena (v primeru `module Kernel(kernel) where {...}` je to funkcija `kernel`). V fazi preverjanja tipov prevajalnik preveri, ali tip te funkcije ustreza nekemu podatkovno-pretokovnemu grafu. To pomeni, da je tip izvožene funkcije oblike

```
Stream a -> Stream b -> ... -> (Stream c, Stream d, ...)
```

Vsak izmed argumentov ustreza nekemu `MaxInput` vozlišču, tokovi v rezultatu pa `MaxOutput` vozliščem. Naša naloga zdaj je, da na podlagi telesa funkcije ščepca zgradimo pravilen graf ter ga povežemo z vhodnimi in izhodnimi vozlišči. V nadaljevanju s `kernel` vedno označimo funkcijo, ki predstavlja ščepec (je izvožena iz modula).

6.3.2 Redukcija izrazov

Izgradnja grafa poteka na podoben način kot redukcija λ-izrazov. Iz dela abstraktnega sintaksnega drevesa, ki predstavlja telo funkcije `kernel`, zgradimo podatkovno-pretokovni graf, sestavljen iz zgoraj opisanih vozlišč.

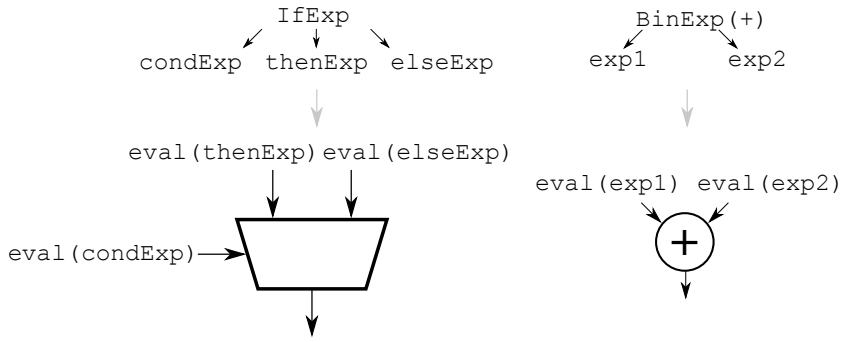
Abstraktno sintaksno drevo sestavljajo vozlišča tipa `ASNode`. Preko redukcije na njihovi podlagi zgradimo graf iz vozlišč `MaxNode`. Redukcija je določena z množico pravil, ki povedo, kako se vsak izraz v abstraktnem sintaksnem drevesu prevede v kos podatkovno-pretokovnega grafa.

Funkcijo, ki izrazu v abstraktnem sintaksnem drevesu (`ASNode`) priredi kos grafa (`MaxNode`), poimenujemo `eval`. Pravila za redukcijo so podana kot `eval(ASNode node) = MaxNode(...)`.

Preprosti izrazi

Zdaj že lahko zapišemo nekaj preprostih pravil.

```
eval(ASLiteralExp litExp) = MaxValue(litExp.literal)
eval(ASBinExp binExp) = MaxComputation( binExp.operator
                                      , eval(binExp.exp1)
                                      , eval(binExp.exp2)
                                      )
eval(ASWhereExp where) = eval(where.exp)
eval(ASLetExp let) = eval(let.exp)
eval(ASExpWithType exp) = eval(exp.exp)
eval(ASIfExp if) = Max3If( eval(if.condExp)
                         , eval(if.thenExp)
                         , eval(if.elseExp)
                         )
```



Slika 6.4: Preprosta pravila redukcije.

Spremenljivke

Kadar se v izrazu pojavi spremenljivka moramo poiskati njeno definicijo. Ker pa imamo na voljo tebelo razrešenih imen iz prejšnje faze, to ni težka naloga. Naj funkcija `declOf` predstavlja preslikavo med imeni in deklaracijami. Tako lahko definiramo redukcijo za izraze `ASVarExp`:

```
eval(ASVarExp varExp) = eval(declOf(varExp))
```

Deklaracija spremennljivke je lahko `ASVarDecl`, `ASFunDecl`, `ASArgDecl` ali `ASStdLibDecl`.

Pred pravili za redukcijo deklaracij si oglejmo še redukcijo klica funkcije.

Klic funkcije

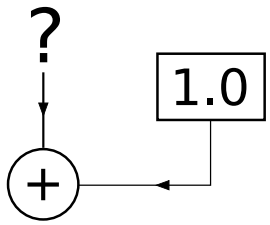
Klic funkcije je v abstraktnem sintaksnem drevesu predstavljen z vozliščem `ASFunExp`. Vsak klic ima samo en argument, funkcije z več argumenti so zapisane kot več zaporednih klicev.

Argumenti so lahko poljubnega tipa — lahko so tudi funkcije. Tu se pojavi problem, saj samih funkcij v grafni vmesni kodi ne moremo predstaviti. Funkcija namreč predstavlja nepopoln kos grafa, ker lahko vsebuje povezave brez začetnih vozlišč (slika 6.5) ali pa celo nima nikakršne predstavitve.

Ker pa vemo, da je funkcija `kernel` tipa

```
Stream a -> Stream b -> ... -> (Stream c, Stream d, ...)
```

```
f :: Double -> Double
f x = x + 1.0
```



Slika 6.5: Funkcija v podatkovno-pretokovnem grafu.

in imajo njeni argumenti pomen vhodnih vozlišč `MaxInput`, smo lahko prepričani, da končni graf ne bo vseboval povezav brez začetnih vozlišč. Za pravilen prevod v graf moramo samo reducirati izraze v takšnem vrstnem redu, da neapliciranih funkcij nikoli ne poskušamo prevesti v graf. Takšen način redukcije je natanko *call-by-name*, saj pri njem argumente v telo funkcije vedno vstavimo nereducirane in jih reduciramo po potrebi.

V implementaciji prevajalnika za prenašanje argumentov poskrbimo z globalnim skladom, na katerega se med prevajanjem ob vsakem klicu naloži argument (predstavljen kot nek `ASExp`), ob vstopu v telo vsake funkcije pa se s sklada odstrani in poveže s pravim argumentom (`ASArgDecl`).

Sklad argumentov se torej med prevajanjem spreminja in pomen redukcije je od njega odvisen. Spremembe stanja na skladu ob klicu funkcije `eval` označimo z oglatimi oklepaji.

```
eval(ASNode node) = eval(node.c1) [ stack.push(node.c2) ]
```

Zapis zgoraj pomeni, da pred redukcijo v `eval(node.c1)` na sklad naložimo `node.c2`.

Zdaj lahko določimo pravilo za klic funkcije:

```
eval(ASFunExp funExp) = eval(funExp.fun)
    [ stack.push(funExp.arg) ]
```

Pri klicu funkcije se skrajno levo vedno pojavi vozlišče tipa `ASVarExp` (`VarExp("f")` na sliki 6.6), ki predstavlja ime klicane funkcije. Z zaporedno redukcijo izrazov `ASFunExp` pridemo do tega vozlišča, kjer uporabimo že znano pravilo `eval(ASVarExp varExp) = eval(declOf(varExp))`

Telo funkcije

Vstop v telo funkcije se zgodi, ko naletimo na vozlišče tipa `ASFunDecl`. Pred redukcijo izraza v telesu moramo še pravilno nastaviti argumente za funkcijo, ki se trenutno nahajajo na skladu.

```
eval(ASFunDecl funDecl) = eval(funDecl.exp)
    [ funDecl.arg1 <- stack.pop()
    , funDecl.arg2 <- stack.pop()
    , ...
    ]
```

Prvi argument se nahaja na vrhu sklada, ker smo ga pri redukciji klicev dosegli zadnjega.

Ob nastavljanju argumentov moramo še shraniti njihove prejšnje vrednosti, saj se lahko klic neke funkcije pojavi večkrat znotraj redukcije enega izraza. Ob vrnitvi argumentom povrnemo stare vrednosti.

V primeru na sliki 6.6 na strani 54 je prikazana reukcija klica funkcije `f 3 5`, kjer je `f x y = x + y`. V prvih korakih na sklad naložimo argumente in jih povežemo z lokalnimi imeni v telesu funkcije. Z redukcijo telesa funkcije nato zgradimo del podatkovno-pretokovnega grafa, ki ga predstavlja klic funkcije.

Deklaracije spremenljivk in argumentov

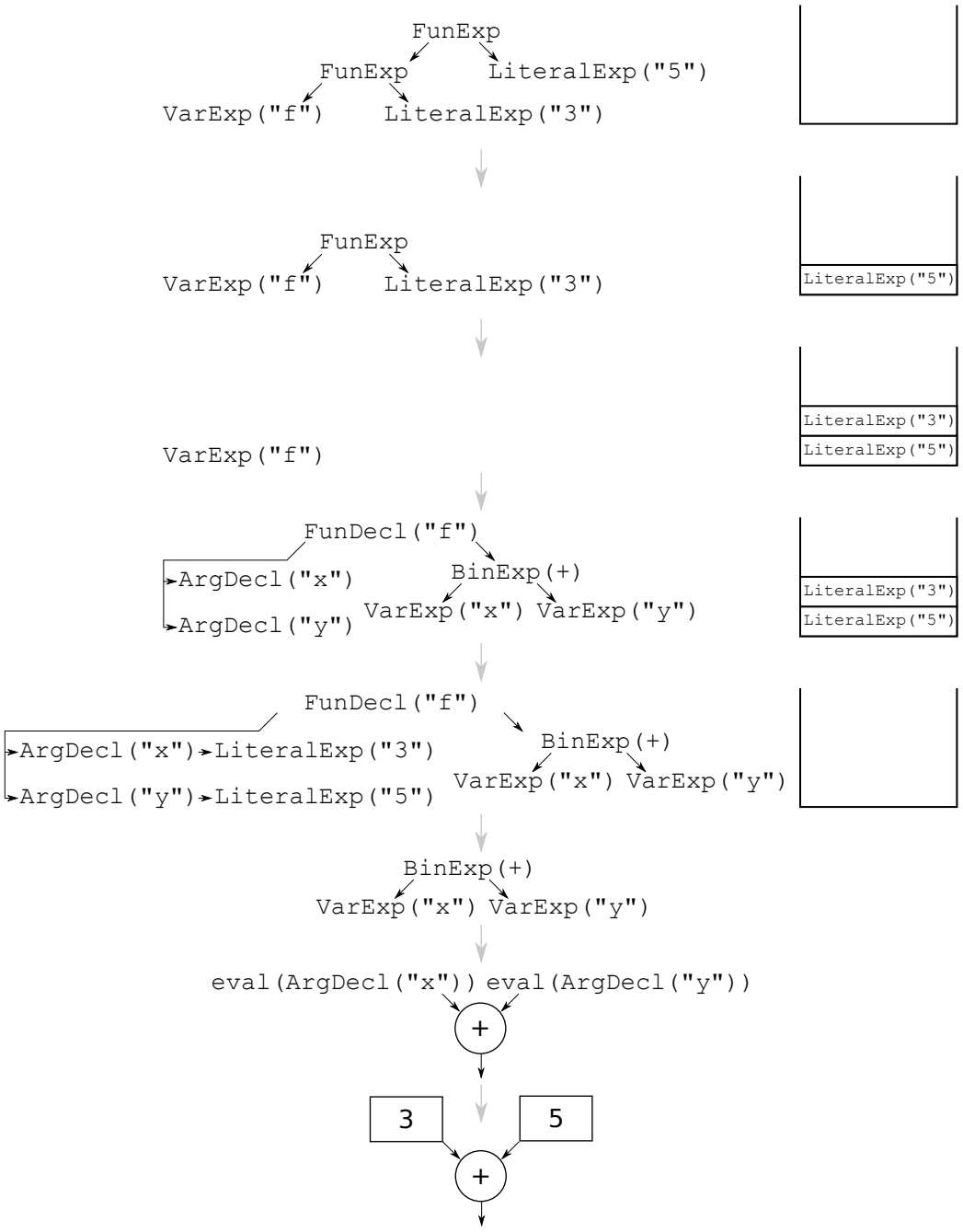
Pri deklaraciji spremenljivke `ASVarDecl` moramo samo reducirati njeno telo. Poseben primer je le, če je spremenljivka tipa `Stream a`, v tem primeru moramo poskrbeti še, da se lahko na vrednost te spremenljivke sklicujemo tudi v njenem telesu. Na ta način ustvarimo zanko v podatkovno-pretokovnem grafu.

Če tip spremenljivke ni `Stream a`, velja pravilo:

```
eval(ASVarDecl varDecl) = eval(varDecl.exp)
```

sicer:

```
eval(ASVarDecl varDecl) = MaxConnect(eval(varDecl.exp))
```



Slika 6.6: Redukcija klica funkcije `f 3 5`, kjer je `f x y = x + y`.

Vozlišče `MaxConnect` predstavlja samo začasno vrednost, ki nam omogoča, da jo v telesu definicije uporabimo. V implementaciji zato vedno ob vstopu v vozlišče `ASVarDecl` preverimo, če smo mu že dodelili začasno vrednost in jo v primeru, da je to res, vrnemo brez ponovnega reduciranja telesa.

Redukcija argumenta (`ASArgDecl`) povzroči redukcijo vozlišča, ki je bilo nanj povezano ob vstopu v funkcijo. Preslikava med argumentom in vozliščem, ki je nanj povezano, je označena z `nodeOf`.

```
eval(ASArgDecl argDecl) = eval(nodeOf(argDecl))
```

Vgrajene funkcije

Za vgrajene funkcije veljajo posebna pravila redukcije. Vstop v vgrajeno funkcijo se zgodi, ko naletimo na vozlišče tipa `ASStdLibDecl`. Za vsako vgrajeno funkcijo podamo svoje pravilo.

```
eval(ASStdLibDecl map) = eval(f) [ f <- stack.pop() ]
eval(ASStdLibDecl zipWith) = eval(f) [ f <- stack.pop() ]
```

Ti dve funkciji predstavljata izvajanje ene operacije na vse komponentah ali parih komponent nekih tokov. Ker Maxelerjeva arhitektura ne pozna razlike med tokom podatkov in atomarno vrednostjo, povzročita le redukcijo podane funkcije z argumenti, ki se seveda že nahajajo na skladu.

```
eval(ASStdLibDecl offset) = MaxOffset(off, stream)
    [ off <- eval(stack.pop()).value
    , stream <- eval(stack.pop())
    ]
```

Funkcija `offset` ustvari `MaxOffset` vozlišče tako, da reducira oba argumenta ter vrednost zamika vzame iz reducirane vrednosti prvega (, za katero vemo, da je tipa `MaxValue`).

```
eval(ASStdLibDecl mux) = MaxMux(control, s1, s2, ...)
    [ control <- eval(stack.pop())
    , { s1, s2, ... } <- eval(stack.pop())
    ]
```

```
eval(MaxStdLibDecl intToDouble) = MaxCast(DOUBLE, eval(s))
    [ s <- stack.pop() ]
eval(MaxStdLibDecl doubleToInt) = MaxCast(INT, eval(s))
    [ s <- stack.pop() ]

eval(ASStdLibDecl counter) = MaxCounter(max, increment)
    [ max <- eval(stack.pop()).value
    , increment <- eval(stack.pop()).value
    ]

eval(ASStdLibDecl counterChain) =
    { MaxCounter(max1, inc1)
    , MaxCounter(max2, inc2)
    , ...
    }
    [ {(max1, inc1), (max2, inc2), ...} <- eval(stack.pop()) ]
```

Funkcija `counter` ustvari vozlišče `MaxCounter` tako, da iz vrednosti `MaxValue`, dobljenih pri redukciji argumentov, prebere konstanto.

Pri pravilih za redukcijo funkcij `mux` in `counterChain` smo uporabili dva pomožna konstrukta: seznam `{}` in terko `()`, ki ju ob redukciji razstavimo na posamezne komponente.

Vsako verigo števcev shranimo še v dodaten seznam, ki podatek o verigi prenese v zadnji del prevajalnika.

Seznami in terke

Seznami in terke so v prevajalniku predstavljene z začasnimi objekti razredov `MaxList` in `MaxTouple`. Takšna vozlišča se nikoli ne pojavijo v končnem grafu, saj jih prevajalnik vedno razstavi na posamezne komponente, kot smo to zapisali pri pravilih za redukcijo funkcij `mux` in `counterChain`.

Za dostop do elementov v seznamu imamo na voljo vgrajeno funkcijo `at`.

```
eval(ASStdLibDecl at) = el [ idx <- eval(stack.pop()).value
                           , el <- eval(stack.pop()).get(idx)
                           ]
```

Za delo s terkami nimamo na voljo nobene funkcije (razen `counterChain`) in preverjanje tipov nam zagotovi, da takšnih vrednosti tudi nikjer napačno ne uporabimo.

Terka se lahko pojavi še kot rezultat funkcije `kernel`. V takšnem primeru jo po končani redukciji razpakiramo in vsak element povežemo na svoje izhodno vozlišče `MaxOutput`.

Pravili za redukcijo izrazov s seznami in terkami sta

```
eval(ASListExp listExp) = { eval(listExp.el1)
                          , eval(listExp.el2)
                          , ...
                          }
```

in

```
eval(ASTupleExp tupleExp) = ( eval(tupleExp.el1)
                            , eval(tupleExp.el2)
                            , ...
                            )
```

Združevanje v celoto

Prevajanje abstraktnega sintaksnega drevesa se, kot smo že omenili, začne z redukcijo izvožene funkcije (`kernel`). Pred njeno redukcijo na sklad argumentov naložimo pravo število vhodnih vozlišč `MaxInput`, ki jih prevajalnik ob redukciji vključi v graf. Število argumentov razberemo iz tipa funkcije `kernel`.

Po končani redukciji glavne funkcije dobimo rezultat, ki je lahko en sam tok podatkov ali pa jih je več združenih v terko. Vsak izhodni tok povežemo na eno izhodno vozlišče `MaxOutput` in končamo gradnjo podatkovno-pretokovnega grafa.

6.3.3 Call-by-need redukcija

V prejšnjem razdelku smo opisali postopek za redukcijo izrazov abstraktnega sintaksnega drevesa v grafno vmesno kodo. Uporabili smo call-by-name stra-

tegijo redukcije, ki je za ta problem zelo primerena, vendar ustvari tudi eno težavo.

Osnovna lastnost call-by-name redukcije je, da se argumenti funkcij v njihovem telesu podvajajo in izračunajo večkrat. V lambda računu se na primer izraz $(\lambda x.((yx)x))((\lambda z.z)w)$ po strategiji call-by-name reduira tako:

$$\begin{aligned}&(\lambda x.((yx)x))((\lambda z.z)w)\\ \rightarrow&(y((\lambda z.z)w))((\lambda z.z)w)\\ \rightarrow&(yw)((\lambda z.z)w)\\ \rightarrow&(yw)w\end{aligned}$$

Z uporabo strategije call-by-value, bi se reduciral tako:

$$\begin{aligned}&(\lambda x.((yx)x))((\lambda z.z)w)\\ \rightarrow&(\lambda x.((yx)x))w\\ \rightarrow&(yw)w\end{aligned}$$

Podoben problem se pojavi pri prevajanju jezika MaxHs, vendar tu zaradi težav pri implementaciji funkcij višjega reda ne želimo uporabiti strategije call-by-value.

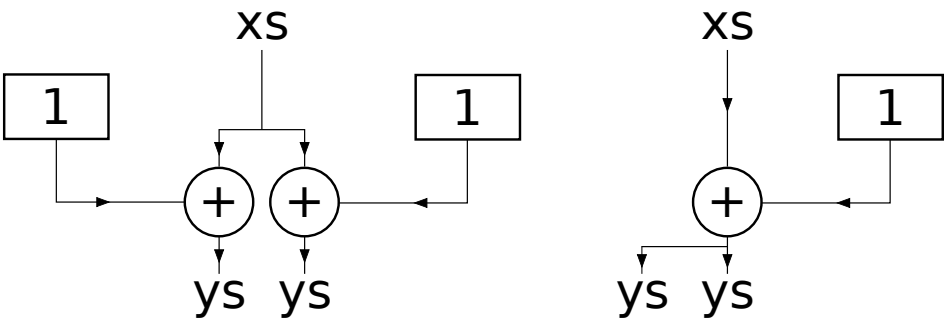
Poglejmo si izraz

```
let { ys = map addOne xs } in (ys, ys)
```

, kjer je `addOne x = x + 1`.

Redukcijo začnemo pri paru `(ys, ys)`. Najprej reduciramo prvi element in se podamo v deklaracijo spremenljivke `ys`, v kateri sestavimo prvi kos grafa. Po vrnitvi iz deklaracije `ys` začnemo reducirati drugi element para, ki je sicer enak prvemu, vendar reducirane vrednosti prvega argumenta nimamo nikjer shranjene in zato spet nadaljujemo v deklaraciji `ys` in zgradimo še en enak kos grafa. Rezultat je prikazan na sliki 6.7 (levo).

Podvajanje lahko odpravimo tako, da namesto strategije call-by-name uporabimo strategijo call-by-need. V tem primeru si reducirane izraze shranimo in pri vseh naslednjih pojavitvah uporabimo že izračunano vrednost.



Slika 6.7: Primerjava med strategijo call-by-name (levo) in call-by-need (desno).

Shranimo lahko vse izraze, ki jih je mogoče predstaviti v grafni vmesni kodi. Funkcije moramo ponovno reducirati ob vsaki uporabi, saj ob različnih argumentih predstavljajo različne dele grafa.

Pozorni moramo biti še na območja vidnosti, ki jih določimo v fazi razreševanja imen, in ob prehodih ponastaviti vrednosti izrazov, ki niso več veljavni.

S takšno strategijo dobimo graf prikazan na sliki 6.7 (desno).

6.4 Generiranje Haskell modula

Na podlagi grafne vmesne kode zadnji del pravajalnika pripravi C funkcijo, s katero lahko izvedemo računski postopek na podatkovno pretokovni enoti. Glavna funkcija v opisu ščepca je: (`a`, `b`, `c`, `d` so `Int` ali `Double`)

```
kernel ::  Stream a -> Stream b -> ...
       -> (Stream c ,  Stream d ,  ...)
```

Zadnji del prevajalnika na podlagi opisa grafa pripravi C funkcijo, ki ustreza tipu funkcije `kernel`. To funkcijo lahko uporabimo za prenos podatkov na DFE.

Prototip C funkcije je: (`a`, `b`, `c`, `d` so `int32_t` ali `double`)

```
void kernel( int size_in0, int size_in1, ...
           , a *in0, b *in1, ...
```

```
, int *p_size_out0, int *p_size_out1, ...
, c **p_out0, d **p_out1, ...
)
```

Argumenti `size_in` predstavljajo velikosti vhodnih tokov `in`. Vhodni tokovi `in` so kazalci na prve elemente polj, v katerih so shranjeni vhodni tokovi.

Rezultat funkcija vrne v argumentih `p_out` tako, da alocira dovolj veliko polje in vanj zapiše izhodni tok DFE-ja. Velikosti zapiše na naslove podane v argumentih `p_size_out`.[1]

Če želimo takšno funkcijo klicati iz Haskella moramo uporabiti razširitev jezika za klice tujih funkcij (angl. *Foreign Function Interface*, FFI) [11]. Oblika modula, ki ga zgeneriramo, je predstavljena v programu 6.1 na strani 62.

Uvoz C funkicje je zapisan na vrsticah 5–10. V 5. vrstici Haskell prevajalniku povemo, kje se nahaja prototip tuje funkcije in kašno konvencijo klicanja naj uporabi; `ccal` pomeni, da uporabi konvencijo C prevajalnika na sistemu, kjer modul prevajamo. Vrstice 6–10 opisujejo, kako tujo funkcijo uporabimo znotraj Haskella. Podamo ji novo ime (`c_kernel`) in tip. Ker ima funkcija stranske učinke – alocira kos pomnilnika, je njen rezultat tipa `IO ()`.

V 11. vrstici se začne definicija funkcije, ki jo izvozimo in predstavlja glavno funkcijo za prenos izvjanja iz Haskella na DFE. Njen tip je

```
    SV.Vector a -> SV.Vector b -> ...
-> (SV.Vector c ,  SV.Vector d ,  ...)
```

Telo funkcije `kernel` se začne z izrazom ‘let’, kjer določimo dolžine vhodnih tokov. Telo izraza ‘let’ je klic funkcije `unsafePerformIO`. Ta funkcija nam omogoča, da iz `IO` operacije vrnemo rezultat, ne da bi ga ovili v `IO` monado. Potrebna je zato, ker klic funkcije `c_kernel` vrne `IO` operacijo — saj alocira pomnilnik. Kljub temu želimo, da funkcija `kernel` ne vrne `IO` operacije, saj so izhodni tokovi vedno enaki, ne glede na to, kam v pomnilniku se

[1] Trenutno prevajalnik podpira le vhodne in izhodne tokove enakih dolžin. Argumentov za dolžino tokov je več zaradi možnosti razširitev v prihodnosti.

shranijo.

V 20. vrstici s pomočjo funkcije `SV.unsafeWith` vhodni tok pretvorimo v predstavitev, primerno za klicanje C funkcije. V nadaljnih vrsticah še alociramo prostor za kazalce, ki predstavljajo rezultat.

Vrstice 28-41 zajemajo klic C funkcije in pretvorbo rezultatov v primerno obliko. Funkcija `peek` prebere vrednost kazalca.

Rezultate `out` ustvarimo s funkcijo `SV.unsafeFromForeignPtr0`, ki iz C polja, na katerega kaže kazalec `fp_out`, ustvari `SV.Vector`. Pri tem uporabimo kazalec tipa `ForeignPtr`, saj je bil prostor, kamor kaže, alociran v tuji funkciji. S klicem `newForeignPtr finalizerFree p_out0` ga ustvarimo iz navadnega kazalca in povemo, da naj ob čiščenju pomnilnika Haskell na njem kliče funkcijo `free()`.

Generiranje modula poteka tako, da prevajalnik najprej iz tipa funkcije, izvožene iz MaxHs modula, razbere število in tipe vhodnih ter izhodnih tokov. Nato zgradi Haskell modul v opisani obliki, ki ustreza prebranemu tipu. Modul in izvoženo funkcijo poimenuje tako, kot sta poimenovani v izvorni MaxHs datoteki.

S tem zaključimo delo v prednjem delu prevajalnika.

Program 6.1 Oblika Haskell modula, ki ga ustvari prevajalnik.

```
1  module Kernel(kernel) where
2  import Foreign
3  import System.IO.Unsafe (unsafePerformIO)
4  import qualified Data.Vector.Storable as SV
5  foreign import ccall unsafe "../CPUCode/CpuStreamCpuCode.c kernel"
6      c_kernel :: Int         -> Int         -> ...
7               -> Ptr a       -> Ptr b       -> ...
8               -> Ptr Int     -> Ptr Int     -> ...
9               -> Ptr (Ptr c) -> Ptr (Ptr d) -> ...
10              -> IO ()
11 kernel ::  SV.Vector a -> SV.Vector b -> ...
12        -> (SV.Vector c ,  SV.Vector d ,  ...)
13 kernel in0 in1 ... =
14   let
15     size_in0 = SV.length in0
16     size_in1 = SV.length in1
17     ...
18   in
19     unsafePerformIO $
20     SV.unsafeWith in0 $ \p_in0 ->
21     SV.unsafeWith in1 $ \p_in1 ->
22     ...
23     alloca $ \p_size_out0 ->
24     alloca $ \pp_out0 ->
25     alloca $ \p_size_out1 ->
26     alloca $ \pp_out1 ->
27     ...
28       do
29         c_kernel size_in0 size_in1 ...
30                  p_in0 p_in1 ...
31                  p_size_out0 p_size_out1 ...
32                  pp_out0 pp_out1 ...
33         size_out0 <- peek p_size_out0
34         p_out0 <- peek pp_out0
35         fp_out0 <- newForeignPtr finalizerFree p_out0
36         let out0 = SV.unsafeFromForeignPtr0 fp_out0 size_out0
37         size_out1 <- peek p_size_out1
38         p_out1 <- peek pp_out1
39         fp_out1 <- newForeignPtr finalizerFree p_out1
40         let out1 = SV.unsafeFromForeignPtr0 fp_out1 size_out1
41         ...
42         return (out0, out1, ...)
```

Poglavje 7

Sklepne ugotovitve

V diplomskem delu smo predstavili nov način programiranja Maxelerjevih podatkovno-pretokovnih enot. Definirali smo na Haskellu temelječ programski jezik MaxHs.

MaxHs je funkcijski jezik, namenjen opisovanju podatkovno-pretokovnih grafov. V njem smo definirali osnovne konstrukte, s katerimi lahko zgradimo poljubne konfiguracije vozlišč podatkovno-pretokovnih grafov. Množico osnovnih vozlišč smo določili na podlagi Maxelerjeve arhitekture.

V našem jeziku smo implementirali preproste primere ščepcev za Maxelerjeve podatkovno-pretokovne enote. V primerjavi z obstoječim načinom programiranja smo pri njih lahko dosegli večjo mero abstrakcije, saj smo uporabljali konstrukte, ki nam jih ponuja funkcijski jezik. Primer takšne abstrakcije so funkcije višjega reda, ki bi jih v obstoječem načinu programiranja veliko težje predstavili.

Za jezik MaxHs smo implementirali prevajalnik v grafno vmesno kodo. Prevajanje temelji na redukciji izrazov v abstraktnem sintaksnem drevesu. Na osnovi pravil za redukcijo izrazov lambda računa smo za prevajanje jezika MaxHs določili pravila za redukcijo izrazov v abstraktnem sintaksnem drevesu v dele podatkovno-pretokovnih grafov. Grafna vmesna koda predstavlja most med prednjim in zadnjim delom prevajalnika, opisanim v diplomskem delu Mihe Eleršiča “Prevajanje grafne vmesne kode za Maxelerjeve

podatkovno-pretokovne enote" [12].

Največje možnosti za izboljšavo jezika so v razširitvah standardne knjižnice. Vgrajene funkcije v jeziku MaxHs večinoma predstavljajo vozlišča, ki jih ponuja Maxelerjeva arhitektura. Ta pa vsebuje še veliko drugačnih tipov vozlišč, ki bi jih lahko z dodajanjem vgrajenih funkcij podprli tudi v jeziku MaxHs. Predstavili smo splošen način prevajanja MaxHs v grafno vmesno kodo, ki bi deloval tudi na dodanih vgrajenih funkcijah.

Na predstavljenih primerih smo prikazali, kako jezik MaxHs olajša delo programerja. Omogoča programiranje Maxelerjevih podatkovno-pretokovnih enot, ne da bi moral programer poznati podrobnosti ciljne arhitekture. V prihodnosti bi lahko programiranje še olajšali s podporo polimorfnim funkcijam in avtomatskemu sklepanju o tipih, kakor to omogoča programski jezik Haskell.

Dodatek A

Gramatika jezika MaxHs

module	→	`module` *modid exports* `where` *body*
exports	→	ϵ
	|	`(` *vars* `)`
body	→	{ *decls*}
decls	→	*decl*
	|	*decl* `;` *decls*
decl	→	*gendecl*
	|	*funlhs rhs*
	|	*var rhs*
gendecl	→	*vars* `::` *type*
vars	→	*var*
	|	*var* `,` *vars*
type	→	*btype*
	|	*btype* `->` *type*
btype	→	*atype*
	|	*btype atype*
atype	→	*conid*
	|	(*types*)
	|	`[` *type* `]`
types	→	*type*
	|	*type* `,` *types*

funlhs	→	*var args*
args	→	*var*
	|	*var args*
rhs	→	= *exp*
	|	= *exp* `where` { *decls* }
exp	→	*bit_or_exp* `::` *type*
	|	*bit_or_exp*
bit_or_exp	→	*bit_or_exp* `|` *bit_xor_exp*
	|	*bit_xor_exp*
bit_xor_exp	→	*bit_xor_exp* `^` *bit_and_exp*
	|	*bit_and_exp*
bit_and_exp	→	*bit_and_exp* `&` *comp_exp*
	|	*comp_exp*
comp_exp	→	*shift_exp* === *shift_exp*
	|	*shift_exp* !== *shift_exp*
	|	*shift_exp* < *shift_exp*
	|	*shift_exp* > *shift_exp*
	|	*shift_exp* <= *shift_exp*
	|	*shift_exp* >= *shift_exp*
	|	*shift_exp*
shift_exp	→	*shift_exp* << *add_exp*
	|	*shift_exp* >> *add_exp*

add_exp	→	*add_exp* + *mul_exp*
	|	*add_exp* − *mul_exp*
	|	*mul_exp*
mul_exp	→	*mul_exp* ⋆ *pre_exp*
	|	*mul_exp* / *pre_exp*
	|	*pre_exp*
pre_exp	→	− *epx10*
	|	*exp10*
exp10	→	`let` { *decls* } `in` *exp*
	|	`if` *exp* `then` *exp* `else` *exp*
	|	*fexp*
fexp	→	*aexp*
	|	*fexp aexp*
aexp	→	*var*
	|	*literal*
	|	`(` *exps* `)`
	|	`[` *exps* `]`
var	→	*varid*
exps	→	*exp*
	|	*exp* `,` *exps*

Dodatek B

Primeri programov v MaxHs

Program B.1 MovingAverage.maxhs

```
module MovingAverage(movingAverage) where
{ movingAverage :: Stream Double -> Stream Double
; movingAverage xs =
    let
    { s1 :: Stream Double
    ; s1 = zipWith add (offset (-1) xs) xs
    ; s2 :: Stream Double
    ; s2 = zipWith add s1 (offset 1 xs)
    }
    in
      map div3 s2

; add :: Double -> Double -> Double
; add x y = x + y
; div3 :: Double -> Double
; div3 x = x / 3.0
}
```

Program B.2 RollingSum.maxhs (osnovna različica)

```
module RollingSum (rollingSum) where
{ rollingSum :: Stream Double -> Stream Double
; rollingSum xs = ys
    where
      { ys :: Stream Double
      ; ys = zipWith add xs (offset (-1) ys)
      }
; add :: Double -> Double -> Double
; add x y = x + y
}
```

Program B.3 RollingSum.maxhs (izboljšana različica)

```
module RollingSum (rollingSum) where
{ rollingSum :: Stream Double -> Stream Double
; rollingSum = scan add

; scan :: (Double -> Double -> Double) -> Stream Double
       -> Stream Double
; scan f xs = ys
    where
    { ys :: Stream Double
    ; ys = zipWith f xs (offset (-1) ys)
    }

; add :: Double -> Double -> Double
; add x y = x + y
}
```

Literatura

[1] Wikipedia, *Von Neumann architecture — Wikipedia, The Free Encyclopedia* [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Von_Neumann_architecture. [Dostopano 17. 8. 2015].

[2] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, št. 8, str. 613–641, 1978.

[3] A. R. Hurson in M. K. Krishna, "Dataflow computers: Their history and future," v *Wiley Encyclopedia of Computer Science and Engineering*, Hoboken: John Wiley & Sons, Inc., 2007.

[4] Maxeler Technologies, *Dataflow computing* [Online]. Dosegljivo: https://www.maxeler.com/technology/dataflow-computing/. [Dostopano: 17. 8. 2015].

[5] *Multiscale Dataflow Programming*, Maxeler Technologies: London, UK, 2014.

[6] W. M. Johnston, J. R. Hanna in R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, št. 1, str. 1–34, 2004.

[7] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, št. 3, str. 359–411, 1989.

[8] H. Barendregt in E. Barendsen, "Introduction to lambda calculus," *Nieuw archief voor wisenkunde*, vol. 4, št. 2, str. 337–372, 1984.

[9] H. Barendregt in E. Barendsen. Introduction to lambda calculus: Revised edition. (2005). *The types project: Tutorials and advanced lectures* [Online]. Dosegljivo: http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf. [Dostopano: 19. 8. 2015].

[10] S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall international series in computer science. Hemel Hempstead: Prentice-Hall, Inc., 1987.

[11] S. Marlow et. al. (2011). *Haskell 2010 language report* [Online]. S. Marlow, ur. Dosegljivo: https://www.haskell.org/definition/haskell2010.pdf. [Dostopano 23. 8. 2015].

[12] M. Eleršič, "Prevajanje grafne vmesne kode za Maxelerjeve podatkovno-pretokovne enote," diplomsko delo, Fakluteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, Slovenija, 2015.

[13] A. W. Appel, *Modern Compiler Implementation in Java*, 2. izd. Cambridge: Cambridge University Press, 2002.