Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andraž Bajt

Prevajalnik iz Haskell Core v JavaScript

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matjaž Kukar

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Kandidat naj zasnuje in implementira prevajalnik za Haskellovo visokonivojsko vmesno kodo imenovano Haskell Core. Tarča prevajanja naj bo JavaScript (kot ga definira ECMAScript 5 specifikacija). Opredeli naj motivacijo za delo in se pri opisu dela osredotoči na posebnosti Haskell Core (in posredno Haskella) kot funkcijskega jezika. Ustrezno naj opiše implementacijo in jo primerja z zasnovo klasičnih prevajalnikov za imperativne jezike.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andraž Bajt, z vpisno številko **63110186**, sem avtor diplomskega dela z naslovom:

Prevajalnik iz Haskell Core v JavaScript

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matjaža Kukarja,
- 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 15. septembra 2014 Podpis avtorja:

Mateji

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GHC	Glasgow Haskell Compiler	glasgowski Haskell prevajalnik
JIT	just-in-time compilation	prevajanje v zadnjem trenutku
CPS	continuation passing style	slog s podajanjem kontinuacij
AJAX	Asynchronous JavaScript + XML	Asinhron JavaScript in XML

Povzetek

Zasnovali in implementirali smo prevajalnik iz Haskellove vmesne kode Haskell Core v JavaScript. Cilj prevajalnika je možnost uporabe visokonivojskih zmogljivosti Haskella za razvoj JavaScript aplikacij. V uvodu spoznamo prednosti in slabosti obeh jezikov ter razkorak med njima. Predstavili smo posebnosti in semantiko jezika Haskell Core ter podrobneje samo implementacijo prevajanja posameznih konstruktov. Razložili smo ne-striktnost in njeno simuliranje v striktnem jeziku, definirali smo sistem in predstavitev algebraičnih podatkovnih tipov ter primerjavo vzorcev. Zasnovali smo lastno vmesno kodo kot korak med ne-striktnim funkcijskim jezikom in striktnim imperativnim. Razvili smo tudi potreben sistem izvajanja za JavaScript in delovanje prevedenih programov preverili v več spletnih brskalnikih.

Ključne besede: prevajalnik, Haskell, Haskell Core, JavaScript, funkcijsko programiranje.

Abstract

We designed and implemented a compiler for Haskell's immediate representation Haskell Core to JavaScript. Purpose of this compiler is the ability to use high level Haskell to develop JavaScript applications. We introduce ups and downs of both languages as well as the gap between them. We present features and semantics of the language Haskell Core and implementation of compilation of separate constructs in more detail. We explain non-strictness and its simulation in a strict language, define a system and representation for algebraic datatypes as well as pattern matching. We designed immediate representation to serve as a step between a non-strict functional language and a strict imperative one. We also developed the required runtime system for JavaScript and tested the compiled programs in several web browsers.

Keywords: compiler, Haskell, Haskell Core, JavaScript, functional programming.

Poglavje 1

Uvod

Namen diplomskega dela je implementacija prevajalnika za jezik Haskell Core[40]. Core je jezik, katerega primarni namen je uporaba kot vmesna koda znotraj prevajalnika GHC[4] za Haskell. Je precej manjši jezik kot Haskell, implementira pa podobno filozofijo. Tarča prevajanja pa je JavaScript, ki je trenutno eden najbolj razširjenih jezikov na planetu; teče v domala vsakem spletnem brskalniku in na velikem številu strežnikov. Izbrali smo ga, ker je pri veliko modernih projektnih nujna platforma, ni pa posebej primeren jezik za razvijanje velikih projektov. To rešujemo s prevajanjem močnejšega jezika (Haskell Core) v JavaScript.

1.1 Haskell

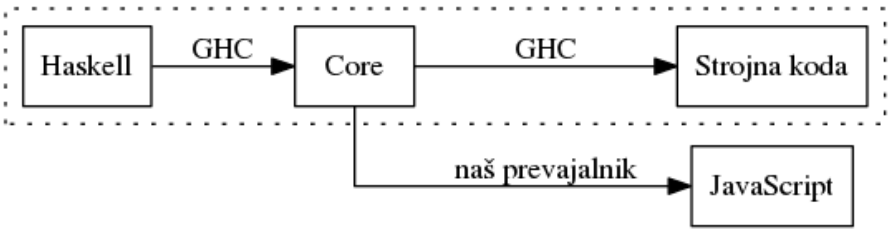
Haskell[3] ima ekspresiven in fleksibilen statičen sistem tipov, ki skupaj z nespremenljivostjo vrednosti odstrani velik razred defektov že v času prevajanja. Ima tudi hierarhičen sistem modulov in enostavno sintakso z malo vgrajenimi posebnimi konstrukti. To nam omogoča, da gradimo lastne abstrakcije in po potrebi razširjamo jezik. To so koristne lastnosti pri gradnji velikih projektov, predvsem pa pri njihovem vzdrževanju. Močno statično tipiziranje daje programerju veliko več samozavesti pri prestrukturiranju kode, saj po uspešnem prevajanju ve, da ni pozabil na noben del kode.

Žal pa popularni Haskell prevajalniki generirajo samo strojno kodo, zaradi česar Haskell ni direktno uporaben za razvijanje JavaScript aplikacij.

1.2 Core

Posredni cilj diplomskega dela je omogočanje prevajanja Haskella v JavaScript, Core pa je eden lažjih načinov za dosego tega cilja, saj nam omogoča, da uporabimo prednji del obstoječega prevajalnika za Haskell npr. GHC. Core ima specificirano eksterno gramatiko in nam tako omogoča interakcijo po dobro definiranem standardu brez zanašanja na programski vmesnik drugega prevajalnika. Tako lahko za generiranje Core kode uporabimo katerokoli orodje ali jo celo napišemo ročno.

Slika 1.1 prikazuje primer integracije z GHC. Ta prevaja Haskell v strojno kodo, s primernimi ukazi pa shrani še vmesno kodo Core, ki jo naš prevajalnik prevede še v JavaScript.



Slika 1.1: Integracija prevajanja v JavaScript z GHC

1.3 JavaScript

JavaScript[6] teče v večini spletnih brskalnikov, kar pomeni, da JavaScript kodo dnevno uporabljajo domala vsi uporabniki računalnikov in z razmahom pametnih mobilnikov tudi uporabniki teh. Povečane zmogljivosti osebnih računalnikov in internetnih povezav so povzročile nastanek t.i. "debelih odjemalcev" (angl. fat client) – JavaScript aplikacij, ki tečejo v brskalniku in komunicirajo s strežnikom prek programskega vmesnika. To pomeni, da je vse več programske kode potrebno pisati in vzdrževati v JavaScriptu, ki pa ni posebej primeren jezik za programiranje velikih sistemov. Njegove glavne pomanjkljivosti so pomanjkanje vgrajenega sistema modulov, šibko tipiziranje, gostobesedna sintaksa, pozno vezanje in dinamično tipiziranje z avtomatskim pretvarjanjem. Nekatere od teh točk so sicer

stvar osebnega okusa, a njihova odprava zagotovo pripomore k pravilnosti in enostavnosti vzdrževanja pri velikih sistemih.

1.4 Sorodni projekti

Kljub temu pa zaradi potencialnih prednosti že obstaja nekaj prevajalnikov iz Haskella v JavaScript. Pristopi in orodja so si precej različni, trenutno pa ni nobene dovolj enostavne rešitve, ki bi se integrirala v obstoječi ekosistem in bila dovolj stabilna za produkcijsko rabo.

GHCJS

Projekt GHCJS[9] razvija zadnji del prevajalnika GHC, ki generira JavaScript kodo, in ima enak namen kot to delo - prevajanje Haskella v JavaScript. Je tudi edini prevajalnik, ki zna v JavaScript prevajati celoten jezik Haskell. Poleg tega, da trenutno še nima stabilne različice, ima dve poglavitni težavi. Uporablja močno spremenjen prevajalnik GHC, kar pomeni da ni združljiv s trenutno različico GHCja in ga mora uporabnik namestiti ločeno. Druga težava je, da ima za JavaScript precej velik sistem izvajanja in proizvaja precej veliko kodo. Diplomsko delo se osredotoča predvsem na prvi problem in se poskuša integrirati z obstoječim prevajalnikom.

Haste

Haste[13] je dialekt Haskella, ki ima zadnji del na voljo tudi za JavaScript. Kljub temu, da je jezik zelo soroden Haskellu, ima drugačen nabor knjižnic in z njim ni povsem združljiv.

Fay

Fay[12] je podjezik Haskella. Programi v njem so veljavni programi v Haskellu, obratno. Za preverjanje tipov uporablja GHC, nato pa generira lastno JavaScript kodo. Integracija GHC je precej površinska, zato ne podpira nekaterih od tipov odvisnih zmogljivosti; najbolj opazna razlika je odsotnost razredov tipov.

Sorodni jeziki

Obstaja precej Haskellu podobnih jezikov z možnostjo prevajanja v JavaScript, npr. Idris[10], PureScript[14] in Elm[11]. Vendar so ti jeziki precej obskurni in imajo majhne skupnosti. Edini projekt, ki zares prevaja Haskell v JavaScript, je trenutno GHCJS.

Naš pristop

Za razliko od večine drugih projektov v diplomskem delu stremimo k prevajanju polnega Haskella z minimalnimi spremembami za uporabnika. Tako ne potrebujemo posebne različice prevajalnika kot npr. GHCJC, ampak se priklopimo na obstoječo namestitev GHC, ki ga uporabimo za prevajanje v vmesno kodo Core.

Poglavje 2

Programski jezik Haskell

Haskell spada v družino funkcijskih programskih jezikov, tako kot recimo jezika ML[34] in OCaml[5], s posebnostma ne-striktnega vrednotenja in čistih funkcij. Nastal je za akademsko rabo, a se v zadnjih letih njegova skupnost širi in počasi se začenja uporabljati tudi v industriji.

2.1 Funkcijski jeziki

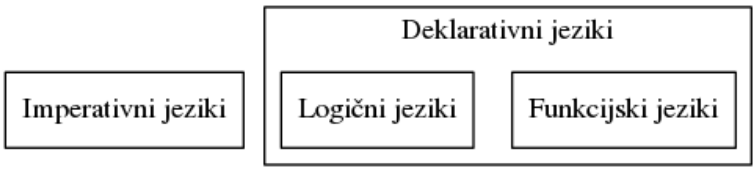
To so jeziki, ki sledijo paradigmi funkcijskega programiranja. V tej paradigmi se izračunov ne modelira z avtomati in pomnilnikom, temveč kot vrednotenje matematičnih funkcij, ki sprejmejo argumente in *izračunajo* nek rezultat. Zato se izogiba stanju in spremenljivim podatkom, osrednjega pomena so izrazi in vrednosti, svoje korenine ima v lambda računu, ki je nastal kot možna osnova matematike[20] (več v poglavju 3.1).

Na drugi strani imamo imperativne jezike, v katerih so programi kombinacija stanja in ukazov, ki to stanje manipulirajo. Tudi ti jeziki poznajo funkcije, ampak ne v matematičnem smislu, temveč kot procedure – poimenovano zaporedje ukazov, ki zopet manipulira isto stanje programa.

Objektno orientirani jeziki, ki so trenutno popularni, k temu modelu dodajo še objekte. Objekti služijo za logično združevanje spremenljivk in procedur, osnovni model izvajanja ostaja spremenljivo stanje (sedaj razporejeno v objektih) in sledenje zaporedju ukazov (sedaj prav tako v objektih).

V drugi skrajnosti poznamo še logične programske jezike. V takem jeziku

programer zgolj definira logične relacije, nato pa od računalnika zahteva nek cilj. Reševalnik bo sam izbral red izvajanja in poskusil sproducirati dokaz. Ta način programiranja je bližje funkcijskemu. Logični in funkcijski jeziki, za razliko od imperativnih, spadajo v skupino deklarativnih jezikov (glej sliko 2.1), v katerih programer ne definira zaporedja ukazov, temveč opiše rešitev. Pozitivna posledica je berljivost in enostavnost kode, negativna pa težja optimizacija.



Slika 2.1: Taksonomija jezikov

2.1.1 Kratka zgodovina funkcijskih jezikov

Ta razdelek je pretežno povzet po [25].

LISP je nastal leta 1958 [32] in je začetnik paradigme funkcijskih progamskih jezikov. V poznih 70-ih letih je Robin Milner spisal jezik ML [33] kot implementacijo svojega polimorfičnega sistema tipov. Ta sistem je pred tem odkril že Roger Hindley [24] in ga zato poimenujemo po obeh avtorjih: Hindley-Milner. Temelji na sistemu F [35] z dodatnimi omejitvami, ki omogočajo avtomatsko sklepanje tipov.

V tem obdobju in kasneje v zgodnjih 80-ih se je pojavil interes za leno vrednostenje programov. Pojavile so se ideje o klicu po imenu in o klicu po potrebi (ne-striktno vrednotenje). Začelo se je z več neodvisnimi članki, ki so raziskovali podobne koncepte, sledila pa je prava eksplozija jezikov, ki so implementirali te ideje. Implementacije so bile precej raznolike, od posebnih procesorjev, posebnih načinov vrednotenja (redukcija grafov), do bolj konvencionalnih prevajalnikov npr. Miranda[42], LazyML[17], Orwell[44], Id[16] in Clean[2]. Z izjemo Mirande so bili to jeziki, ki jih je razvijala ena oseba in niso nikoli dosegli kritične mase - služili so zgolj kot orodje za akademske raziskave. Med seboj pa so si bili zelo podobni (z izjemo sintakse) in pojavile so se ideje o združenju v en standarden jezik.

Leta 1987 so Hudak, Peyton Jones in Wadler osnovali komite (takrat še neura-

dno), ki naj bi standardiziral ta jezik. Prvi kandidat je bila Turnerjeva Miranda, ki je bila v tem času daleč najbolj razširjena. Turner pa je zavoljo svojih komercialnih interesov ponudbo zavrnil.

Januarja 1988 je imel komite svoje prvo srečanje na univerzi Yale. Tu so dorekli osnovno filozofijo jezika in se odločilli za ime Haskell - v čast matematiku in logiku Haskellu Curryju, predvsem zaradi njegovega dela na kombinatorični logiki, ki je močno vplivalo na zasnovo funkcijskih jezikov, pa tudi Curry-Howard korespondence, ki povezuje progame z dokazi. Nekaj srečanj in veliko izmenjanih elektronskih sporočil kasneje je izšlo prvo Haskell poročilo - leta 1990 v uredništvu Wadlerja in Hudaka [26]. V istem letu je izšel tudi prvi prevajalnik - *hbc*, leto kasneje pa še prevajalnik *Yale Haskell*. Leta 1992 se je pojavil še Glasgowski prevajalnik *GHC*, ki je danes daleč najbolj uporabljan Haskell prevajalnik.

Skozi leta so izhajale posodobljene različice poročila, s pomembnejšo izdajo (1.3) v letu 1996 je bila prvič specificirana standardna knjižnica in monadični vhod/izhod. Leta 1998 pa je izšlo poročilo Haskell 98, ki definira stabilno različico jezika, ki jo večina prevajalnikov podpira še danes.

Trud komiteja se je izplačal, danes je Haskell de facto standard ne-striktnih funkcijskih jezikov.

2.2 Glavne značilnosti

Daleč najpomembnejša lastnost Haskella je ne-striktno vrednotenje izrazov[45], kar pa ni enako lenemu vrednotenju, ki je običajno omenjeno v zvezi s Haskellom. Pri lenem vrednotenju se izraza ne ovrednoti, ampak se samo ustvari objekt, ki predstavlja ta izraz v neovrednoteni obliki. Ob prvi potrebi po vrednosti tega izraza se izraz ovrednoti, objekt pa prepiše z rezultatom. Ne-striktno vrednotenje je širši pojem, predpisuje samo, da se izraz vrednoti od zunaj navznoter in ne obratno. Izraz $(a*b)+c$ bi se na primer v striktnem jeziku ovrednotil tako, da bi se najprej izračunal produkt in nato vsota. V ne-striktnem jeziku se najprej poskuša ovrednotiti najbolj zunanji konstrukt, ki je v tem primeru vsota. Ker pa vsota potrebuje vrednost produkta, to povzroči, da se med vrednotenjem vsote ovrednoti tudi produkt. Na tem primeru lahko vidimo, da je možno ne-striktno vrednotenje implementirati z lenim, dopušča pa nam tudi prostor za dodatne optimizacije.

Napreden prevajalnik bi lahko sam zaznal neodvisne podizraze in jih ovrednotil paralelno.

Takšna strategija vrednotenja vsili čisto funkcijsko programiranje. Čistoča se tu nanaša na odsotnost stranskih učinkov, kot je na primer delo z datotekami, pisanje v terminal ali spreminjanje stanja (npr. vrednosti spremenljivke). Zaradi ne-striktnega vrednotenja namreč ne moremo sklepati o vrstnem redu izvajanja programa in posledično o vrstnem redu stranskih učinkov, zato je nujno, da so stranski učinki prepovedani.

Ker pa je cilj večine programov, da dosežejo neke učinke, Haskell ponuja vódene učinke, ki so združljivi s konceptom čistega funkcijskega programiranja - monade [28]. Haskell je populariziral funktorje, monade, monoide in druge koncepte iz teorije kategorij[46], njihove implementacije so namreč del standardne knjižice.

Haskell je statično tipiziran, preverjanje tipov se vrši zgolj ob prevajanju in ima zelo močan sistem tipov. Ta temelji na sistemu tipov Hindley-Milner [24, 33] s precej dodatki. To je parametrični sistem tipov z algoritmom, ki zna vsakemu izrazu sam prirediti najbolj splošen tip. Parametričnost tu pomeni, da imajo tipi vrednosti lahko "luknje", v katere lahko vtaknemo poljubne tipe. Parametričnost torej pove nekaj o izrazu, ki naseljuje ta tip. "Luknje" predstavljajo univerzalno kvantifikacijo in elementov tega tipa progam ne more posebej obravnavati, saj so zgolj netransparentni kazalci. Tako lahko iz tipa funkcij pridelamo zastonj izreke[1] (angl. free theorems)[43]. Med pomembnejšimi dodatki glede na sistem Hindley-Milner so razredi tipov (omogočajo omejeno parametričnost), eksistencialna kvantifikacija in omejena implementacija odvisnih tipov.

Pomembna posledica čistoče je referenčna transparentnost funkcij: vsak klic funkcije lahko nadomestimo z memoiziranim[2] rezultatom in vsako vrednost lahko nadomestimo s klicem funkcije, ki to vrednost izračuna. Ker stranskih učinkov ni, vemo, da obnašanja in rezultata programa ne bomo spremenili, vplivamo le na časovno (in prostorsko) kompleksnost.

[1]Zastonj izrek je izrek, katerega dokaz je neposredna posledica parametričnosti tipa. Neomejene spremenljivke na nivoju tipov preprečijo programu, da bi posebej obraval posebne primere in zato omejijo število možnih implementacij tega tipa

[2]Memoizacija je postopek shranjevanja rezultata klica funkcije z določenim naborom parametrov v neko podatkovno strukturo, tako da ob ponovnem klicu z enakimi parametri vrnemo shranjeni rezultat in se izognemo ponovnemu klicu drage funkcije.

Referenčna transparentnost sama po sebi je zanimiva lastnost, vendar ni pretirano uporabna. Omogoči pa nam bolj zanimivo tehniko, t.i. ekvivalenčno sklepanje. To je tehnika sklepanja o programu. Dokazovanje pravilnosti imperativnih programov je običajno precej kompleksna naloga, predvsem zaradi spreminjanja stanja in stranskih učinkov, zato se je programerji običajno izogibajo. V čistem funkcijskem jeziku pa lahko izkoristimo referenčno transparentnost in uporabimo definicije funkcij in vrednosti kot enakosti, ki nam omogočajo prepisovanje. To je tehnika ekvivalenčnega sklepanja. Skupaj z zastonj izreki nam omogoča neformalno dokazovanje veliko želenih lastnosti programov z zelo malo truda.

Kot enostaven primer takega neformalnega dokaza si poglejmo pravilnost instance funktorja za podatkovni tip *Maybe*[3] [15].

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

Najprej definiramo tip kot algebraično vsoto *samo vrednosti* ali *ničesar*. Instanca funktorja aplicira funkcijo na vrednost, v njeni odsotnosti pa vrne nazaj prazno vrednost.

```
instance Functor Maybe where
  fmap f = \m -> case m of {Nothing -> Nothing;
                            Just a  -> Just (f a)}
```

Specializiran tip za *fmap* je torej

```
(a -> b) -> Maybe a -> Maybe b
```

Da je funktor pravilen, mora ohranjati identiteto in kompozicijo funkcij

```
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

Najprej poglejmo identiteto. Iz tipa lahko razberemo, da *fmap* vse tipe vrednosti obravnava enako. Torej ima natanko dva načina, da ustvari izhodno vrednost tipa *Maybe b* – lahko na vrednosti uporabi podano funkcijo ali pa vrne prazno vrednost. Ker vanjo podajamo funkcijo identitete, lahko zgolj vrne to vrednost ali pa prazno vrednost, pokazati želimo samo še, da ohranja konstruktor, kar pa je jasno razvidno iz definicije.

[3]tip *Maybe* predstavlja možnost odsotnosti, podobno kot ničelni kazalci, vendar na način, ki programerja prisili k pravilni obravnavi ničelnih vrednosti.

Za pravilnost kompozicije pa si poglejmo vsako stran posebej. Najprej leva

```
fmap (p . q) = = \m -> case m of {Nothing -> Nothing;
                                  Just ((p . q) a)}
```

Tukaj smo samo vrinili definicijo *fmap*. Desna stran je nekoliko bolj zapletena. Začnimo z eta razširitvijo[4].

```
(fmap p) . (fmap q) = \m -> ((fmap p) . (fmap q)) m
                    = \m -> (fmap p) (fmap q m)
```

Nato vrinemo definicijo

```
= \m -> (fmap p) (case m of {Nothing -> Nothing;
                             Just a  -> Just (q a)})
```

in vrinemo še drugo definicijo

```
= \m -> case (
     case m of {Nothing -> Nothing;
                Just a  -> Just (q a)}
  ) of {Nothing -> Nothing;
        Just a  -> Just (p a)}
```

Ker je vrednost, ki jo analiziramo, znana, lahko rezultate vzorcev ponovno vrinemo

```
= \m -> case m of {Nothing -> Nothing; Just (p (q a))}
= \m -> case m of {Nothing -> Nothing; Just ((p . q) a)}
```

In prispemo do enakega rezultata kot na levi strani, naša instanca funktorja je torej pravilna.

Vsemu akademskemu ozadju navkljub pa se Haskell trudi biti pragmatičen jezik, zato ni totalen - omogoča neterminacijo. To pomeni, da lahko spišemo tudi kodo, ki se bo ob izvajanju ujela v neskončno rekurzijo in ne bo nikoli izračunala nobenega rezultata. Obstajajo namreč tudi jeziki (npr. Idris), ki omogočajo preverjanje totalnosti in takšne programe zavrnejo[5].

[4]Eta(η) razširitev je pretvorba v lambda računu, ki funkcijsko vrednost f nadomesti z abstrakcijo, ki pokliče to vrednost s svojim argumentom: $\lambda x.f\ x$.

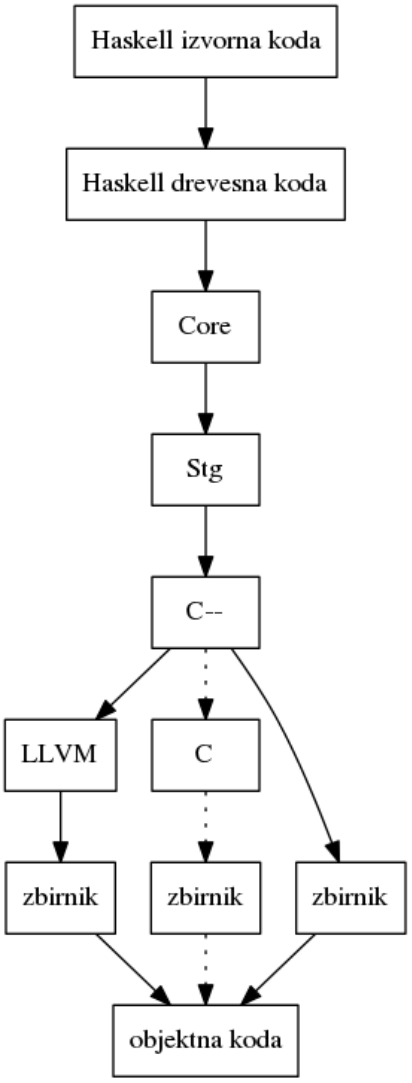
[5]Ti prevajalniki seveda ne rešujejo problema ustavljanja, ampak postavijo dodatne omejitve glede veljavnih programov. To sicer pomeni, da zavrnejo vse progame, ki se ne ustavijo, zavrnejo pa tudi nekaj takih, ki se.

2.3 GHC

The Great Glassgow Haskell Compilation System oz. na kratko GHC[4] je de facto standard Haskella v industriji in daleč najbolj uporabljan prevajalnik. Služil nam bo za integracijo prevajalnika za Core v samo prevajanje Haskella kot zunanji zadnji del, zato si bomo tukaj okvirno pogledali njegovo arhitekturo.

Pomemba lastnost GHC-ja je, da je sam napisan v Haskellu - jeziku, ki ga prevaja. To povzroča nekaj težav (za delo na samem prevajalniku), vendar je tudi velik bonus, saj tako razvijalci prevajalnika redno prevajajo velik in kompleksen program (sam GHC) in tako redno testirajo pravilnost in performančnost prevajalnika.

To in Haskellovo akademsko ozadje sta povzročila, da je GHC tudi laboratorij za nove funkcije jezika/prevajalnika. GHC tako implementira veliko razširitev jezika, kot je definiran v poročilu, ki jih lahko selektivno vklopimo in izklopimo.



Slika 2.2: Predstavitve programa znotraj GHC

Da lahko tako hitro prototipiranje funkcij omogočimo, želimo potrebne spremembe v implementaciji prevajalnika omejiti na majhen del - želimo, da je prevajalnik zelo modularen. In GHC to zahtevo izpolnjuje. Cevovod prevajanja se začne s precej klasičnim prednjim delom, ki mu sledi napredno preverjanje tipov in generiranje vmesne kode. Velika večina novih funkcij prevajalnika zahteva spremembe samo v teh delih. Osrednja vmesna koda Haskell Core je od nastanka GHC doživela samo eno spremembo - uvedbo dobro definiranih pretvorb (casts).

Nato sledi prevajanje v vmesno kodo STG (Spineless Tagless G-machine [27]),

ki odstrani tipe in poenostavi preostale konstrukte iz prejšnje vmesne kode. To je tudi zadnja funkcijsko usmerjena vmesna koda, v naslednjem koraku se program prevede v C--[29], ki je jezik podoben Cju, vendar je bil osnovan kot prenosljiv zbirnik namenjen za pisanje prevajalnikov. Ima sistem tipov, ki bolj ustrezajo registrom, in sistem izvajanja, ki med drugim omogoča implementacijo upravljanega pomnilnika in izjem.

Od tu naprej se vodi več poti, kot je prikazano na sliki 2.2. Prvotna implementacija je generirala kodo v jeziku C in nato uporabila zunanji pravajalnik za prevajanje v zbirnik. C ni posebej primeren kot tarča prevajanja in zato so imeli pisci prevajalnika precej težav. Posledično so kasneje spisali svoj generator zbirnika, ki je trenutno še vedno privzeti zadnji del. A s širitvijo na več platform je ta začel prinašati veliko vzdrževalnega dela in kompleksnosti. Zato se je pojavil eksperimentalni zadnji del, ki generira LLVM vmesno kodo [39] in nato uporabi LLVM za generiranje zbirnika za specifično platformo. Ta zadnji del je precej manj kompleksen in za računsko zahtevne programe tudi hitrejši, izkorišča pa tudi trud ekipe projekta LLVM za podpiranje novih platform in novih optimizacij, ki pomenijo še hitrejšo kodo brez sprememb v samem GHC.

Poglavje 3

Programski jezik Haskell Core

Haskell Core je imenu primeren jezik, ki zaobjame jedro Haskella in nič več. Nastal je kot drevesna vmesna koda znotraj prevajalnika GHC in temelji na sistemu F[35] z nekaj dodatki - to je lambda račun s parametričnimi tipi, z dodatnimi algebraičnimi podatkovnimi tipi in eksistencialno kvantifikacijo ter klicem po potrebi (ne-striktnim vrednotenjem). Za primere prevodov iz Haskella v Core glej poglavje 6.

Pomemben je, ker je predstavil idejo tipizirane vmesne kode. To sicer predstavlja več dela za pisca prevajalnika, saj mora prevajalnik sedaj generirati tudi ustrezne tipe, vendar ravno ti tipi zagotavljajo večjo pravilnost generirane kode. Po generiranju vmesne kode lahko namreč preverimo njene tipe. Za običajno prevajanje to sicer izpustimo (zaradi hitrosti), je pa skrajno uporabno pri testiranju prevajalnika. Tipi pridejo posebej do izraza ob agresivnih transformacijah. Po vsakem koraku transformacije vmesne kode lahko preverimo konsistentnost tipov generirane kode. To sicer ne zagotavlja popolne pravilnosti, je pa velik korak v to smer. V praksi se je namreč pokazalo, da se večina napak pri generiranju in transformiranju vmesne kode izraža tudi v obliki napak tipov v vmesni kodi [28].

Te transformacije so namreč lahko zelo agresivne, tako kot Haskell je namreč tudi Core čist funkcijski jezik z referenčno transparentnostjo (glej razdelek 2.2). To dopušča prosto vrinjanje in prepisovanje izrazov. Generirana koda je lahko zelo daleč od kode, ki jo je na začetku napisal programer, zato je preverjanje konsistentnosti ključnega pomena.

Core je zanimiv tudi, ker se je od svojega zametka v letu 1990 spremenil samo

enkrat in sicer z dodatkom dokazanih prelivov (angl. cast) za implementacijo naprednih razširitev jezika Haskell, ki pa jih to delo ne obsega. Zaradi svoje stabilnosti predstavlja primerno tarčo za pisanje prevajalnikov, žal pa ni posebej dobro izkoriščen zaradi močne prepletenosti z GHC.

3.1 Lambda račun

Osnova jezika Core je lambda račun, zato je opazno drugačen od popularnih programskih jezikov. Lambda račun je nastal kot možna osnova matematike v zgodnjih 1930 kot plod dela Alonza Churcha [20]. Church je lambda račun uporabljal kot sintakso za logiko, izkazalo pa se je, da lahko samo v lambda računu brez dodatkov definira vse izračunljive izraze. V tem času je bilo vprašanje, kaj je izračunljivo, še odprto [23], danes lahko rečemo, da je lambda račun po Turingu poln. Predstavlja zanimivo in enostavno (ter uporabno) alternativo Turingovim strojem.

3.1.1 Osnovni konstrukti

Enostavni netipizirani lambda račun lahko definiramo s samo tremi konstrukti

- abstrakcija: definicija izraza z neznanimi vrednostmi za določena imena oz. anonimne funkcije
- referenca: sklicevanje na vrednost, definirano v oklepajoči abstrakciji
- aplikacija: uporaba funkcijske vrednosti z enim argumentom

Lambda račun ne potrebuje posebnih primitivnih števil, logičnih vrednosti ali operacij, vse lahko definiramo z zgornjimi tremi konstrukti.

Abstrakcija

Rezultat abstrakcije je funkcijska vrednost. V splošnem abstrakcija poimenuje en argument in definira telo. Notacija za zapis abstrakcije, ki z argumentom x izračuna y, je

$$\lambda x.y \tag{3.1}$$

Identiteto lahko zapišemo kot

$$\lambda x.x \tag{3.2}$$

Kot že omenjeno, imajo vse abstrakcije številnost ena. Kako torej definiramo funkcijo, ki sprejme dva argumenta in ju sešteje? [1] Odgovor so funkcije v Curryjevi obliki: funkcijo dveh argumentov lahko zapišemo tudi kot funkcijo, ki sprejme en argument in vrne novo funkcijo, ki sprejme še drugi argument. Tu je očitno, da je notranja funkcijska vrednost v resnici zaprtje, saj nosi s seboj še vrednost prvega argumenta, ki je v njej sicer prosta spremenljivka.

$$\lambda x.\lambda y.x + y \tag{3.3}$$

Referenca

Referenca je najpreprostejši konstrukt. Omogoča nam, da se sklicujemo na ime, definirano v sintaktično ovijajoči abstrakciji. V abstrakciji $\lambda x.y$ lahko izraz y vsebuje ime x in to ime se ob vrednotenju ovrednoti v vrednost argumenta x.

Aplikacija

Abstrakcija nam omogoča, da skonstruiramo novo funkcijsko vrednost, aplikacija pa porabi to vrednost in nam da rezultat. Drugače povedano je aplikacija klic funkcije. Notacija za aplikacijo je

$$f\ x \tag{3.4}$$

namesto običajne $f(x)$. Church je zaradi pogoste rabe aplikacije namreč izpustil oklepaje, kjer ti niso potrebni za enolično določanje pomena. Kaj pa funkcije, ki vračajo nove funkcije? Takrat preprosto naštejemo agrumente s presledki.

$$f\ x\ y\ z \tag{3.5}$$

To interpretiramo kot $((f(x))(y))(z)$, aplikacija je namreč levo asociativna. Seveda nam nič ne preprečuje, da bi takšni funkciji podali samo dva argumenta. V tem primeru bi bil nov rezultat funkcija, ki bi sprejela še z, temu pravimo *parcialna aplikacija*.

[1] Zaradi enostavnosti privzamemo, da imamo tudi števila in osnovno aritmetiko s standardno notacijo.

Na tem mestu se poraja vprašanje vrstnega reda vrednotenja argumentov. Za denotacijsko semantiko to ni pomembno, je pa ključnega pomena, če hočemo definirati operacijsko semantiko - implementirati jezik. Lambda račun ima tradicionalno dve strategiji vrednotenja: klic po vrednosti in klic po imenu, obstajajo pa tudi druge. Klic po vrednosti je analogen implementaciji klicev funkcij v imperativnih programskih jezikih; najprej ovrednotimo argument in nato pokličemo funkcijo. V lambda računu to pomeni, da najprej ovrednotimo prvi argument, ga porabimo, nato ovrednotimo drugi argument in tako naprej.

Druga strategija je klic po imenu. V tem primeru argumenta ne vrednotimo, ampak vse pojavitve argumenta v telesu nadomestimo z izrazom argumenta - ob vsaki uporabi na novo ovrednotimo argument. V nekaterih primerih to sicer pomeni eksponentno upočasnitev v primerjavi s klicem po imenu, ampak prinaša svoje prednosti. Z uporabo klica po imenu lahko definiramo izraze, ki jih ni mogoče definirati s klicem po imenu. Na primer pogojno funkcijo

$$if = \lambda p.\lambda t.\lambda f.p\ t\ f \tag{3.6}$$

Za predikat je tu uporabljeno Churchevo kodiranje Boolovih vrednosti

$$true = \lambda t.\lambda f.t \tag{3.7}$$

$$false = \lambda t.\lambda f.f \tag{3.8}$$

$$alsoFalse = if\ true\ false\ true \tag{3.9}$$

Če takšno funkcijo implementiramo s klicem po vrednosti, bo vedno ovrednotila oba argumenta. To sicer ni problem, ker nimamo stranskih učinkov, funkcija bi zgolj opravila nepotrebno delo. Težava se pojavi, če pogoj uporabimo v rekurzivni funkciji z namenom, da se odločimo med robnim pogojem in rekurzivnim korakom. Implementacija s klicem po imenu bi se vedno ujela v neskončno rekurzijo, saj bi vedno ovrednotila tudi rekurzivni korak.

3.2 Sistem tipov

Churchev sistem tipov je služil predvsem za osnove logike, vendar se je kmalu pokazalo, da ima veliko pomanjkljivost - možno je uporabiti krožno sklepanje in tako dokazati poljubne izjave. Enostaven primer takšnega krožnega sklepanja je fiksna

točka funkcije identitete, ki v principu izračuna karkoli, vendar se ob vrednotenju ujame v neskončno rekurzijo - "vrne" spodnjo vrednost $\perp$.

$$Y\ (\lambda x.x) \tag{3.10}$$

Y tu stoji za *Y-kombinator*, operacijo fiksne točke, zaprt izraz, ki predstavlja abstrakcijo čez splošno rekurzijo, z njim lahko definiramo rekurzivno anonimno funkcijo. Odkril pa ga je ravno Haskell Curry[22].

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \tag{3.11}$$

Alonzo Church je zato leta 1940 definiral enostavno teorijo tipov [21] - enostavno tipizirani lambda račun. V tej različici lambda računa ima vsak veljaven izraz tudi tip. Tipi τ pa so definirani z rekurzivno definicijo,

$$\tau ::= \tau \rightarrow \tau \mid T \in \mathcal{B} \tag{3.12}$$

kjer je $\mathcal{B}$ bazna množica tipov. Izkaže se, da *Y-kombinator* v tem sistemu nima tipa, saj bi morali skonstruirati neskončno velik tip, takšne strukture pa ne moremo (v končnem času) primerjati z ničemer. V takšnem sistemu lahko definiramo samo izraze brez splošne rekurzije; dosegli smo totalnost. To je uporabno za preverjanje odsotnosti krožnega sklepanja v dokazih, kot stranski produkt pa lahko ob prevajanju programa s takšnim sistemom tipov preverimo odsotnost velikega razreda napak.

Za to pa smo morali žrtvovati splošno rekurzijo, ki je nujna, če hočemo, da je jezik po Turingu poln. Na srečo lahko tudi ob prisotnosti enostavnih tipov pridobimo rekurzijo, če dovolimo po imenu rekurzivne definicije vrednosti.

$$Y = \lambda f.f\ (Y\ f) \tag{3.13}$$

Na ta način smo žrtvovali totalnost, ostalo pa je preverjanje konsistentnosti.

Enostavno tipiziran lambda račun je enakovreden predikatni logiki prvega reda. Če v sistem tipov dodamo še univerzalno kvantifikacijo, dobimo *Sistem F*[35] - lambda račun drugega reda. Ko primerjamo tipe, sta dva tipa enaka, če sta strukturalno enaka ali če lahko postaneta enaka tako, da univerzalno kvantificirano spremenljivko v enem opredelimo v sprecifično vrednost.

Na Sistemu F temelji sistem tipov za jezik Core, doda pa še eksistencialno kvantifikacijo in algebraične tipe (glej poglavje 3.3). Vsi tipi so obvezni in eksplicitni, za razliko od nekaterih implementacij Sistema F, ki izvajajo inferenco tipov.

3.3 Algebraični podatkovni tipi

Algebraični podatkovni tipi so uporabniško definirani tipi, ki zajemajo vsote in produkte. Ti dve operaciji se tukaj nanašata na moč množice vseh elementov tipa. Prazen tip *Void* je ničla, tip *Unit* z enim praznim konstruktorjem pa enica tega sistema. Vsota je podatkovni tip z dvema konstruktorjema. Elementi tega tipa so lahko ali elementi levega konstruktorja ali elementi desnega konstruktorja, skupaj jih je torej ravno za vsoto obeh konstruktorjev. Produkt pa je konstruktor z dvema poljema. Ob konstrukciji lahko kombiniramo kateri koli element prvega polja s katerim koli elementom drugega, skupaj imamo torej ravno produkt moči obeh tipov. Z dodano rekurzijo lahko tako definiramo poljubne tipe. Vsoto več kot dveh konstruktorjev lahko prevedemo na gnezdene vsote in produkt več kot dveh polj na gnezdene produkte.

data One = O1

Tip *One* ima natanko eno vrednost, to je *O1*. *O1* je konstruktor, ki pa ne sprejme nobenega argumenta in je zato že kar vrednost tipa *One*.

data Two = T1 | T2

Tip *Two* definira dva alternativna konstruktorja (oba brez argumentov), ki sta tudi obe vrednosti.

data Five = F1 Two Two | F2 One

Tip *Five* ima prav tako dva alternativna konstruktorja, ki pa sta sestavljena. Konstruktor *F2* zavije vrednosti tipa *One*, zato ga lahko skonstruiramo na toliko načinov, kolikor je vrednosti tipa *One* – 1. Konstruktor *F1* pa je produkt dveh vrednosti tipa *Two*, zato ga lahko sestavimo na $2 * 2$ načina. Vseh vrednosti tipa *Five* je torej $5 = 2 * 2 + 1$

F1 T1 T1

F1 T1 T2
F1 T2 T1
F1 T2 F2
F2 O1

Vsi ti primeri so imeli končno mnogo vrednosti, z uvedbo proste rekurzije pa pridelamo neskončno vrednosti. Enostaven primer so Peanova naravna števila *Nat*, ki so definirana z ničlo in naslednikom. To sta tudi oba potrebna konstruktorja; ničla brez argumentov in naslednik, ki kot argument prejme drugo naravno število. Število n v takem zapisu predstavimo z gnezdenjem n naslednikov in ničlo na najglobljem nivoju.

```
data Nat = Zero | Succ Nat
```

Poseben konstrukt v jeziku Core je še definicija *newtype*, ki vedno definira podatkovni tip z enim konstruktorjem z enim poljem.

```
newtype Example = Example Nat
```

Omogoči nam definicije tipov, ki so ob prevajanju različne in tako zagotavljajo varnost, ob izvajanju pa se dodatni konstruktorji izpustijo. Predstavitev je v tem primeru enaka *Nat*, zato pridobimo na hitrosti.

3.4 Ne-striktnost

Klic po imenu lahko implementira določene funkcije, ki jih klic po vrednosti ne more, a trpi za nepotrebnim (v odsotnosti učinkov) večkratnim vrednotenjem, ki v patološkem primeru lahko privede do eksponentne upočasnitve. Lahko pa rezultat prvega vrednotenja izraza memoiziramo in tako na račun višje porabe pomnilnika izraz ovrednotimo kvečjemu enkrat. Na ta način je hitrost izvajanja primerljiva s klicem po potrebi, semantika pa je (ob odsotnosti stranskih učinkov) enaka kot pri klicu po imenu. Ta strategija vrednotenja se imenuje klic po potrebi in je strategija, ki jo uporablja Core.

Ob prisotnosti stranskih učinkov lahko sklepamo pod predpostavko, da se bodo učinki izvršili kvečjemu enkrat, vendar je takšno sklepanje še vedno težavno. V netrivialnih programih je namreč pogosto težko sklepati, kdaj in pod kakšnimi pogoji se bo vrednost zares ovrednotila.

3.4.1 Izrazi *let*

Ne-striktno vrednotenje lahko izkoristimo za pohitritev programov z izločanjem enakih podizrazov. Vzamemo enostaven izraz

```
f (a+b) (a+b)
```

Vidimo, da se vsota izračuna dvakrat. Lahko pa izraz zavijemo v anonimno funkcijo, ki ji kot parameter podamo vsoto

```
(\x :: Int -> f x x)(a + b)
```

Izraza $a + b$ tu še ne ovrednotimo, ovrednoti se še vedno samo po potrebi znotraj funkcije f, vendar samo prvič - ime x bo memoiziralo rezultat.

Zaradi berljivosti obstaja posebna sintaksa, imenovana *let*

```
%let { x :: Int = a + b }
%in f x x
```

Programer lahko tako bolje nakaže svoj namen. Ampak izrazi *let* nam omogočajo več, definiramo lahko tudi med seboj (rekurzivno) odvisne vrednosti

```
%let %rec
      {a :: List Int = prepend 1 b;
       b :: List Int = prepend 2 a}
%in a
```

Vrstni red ni pomemben, ob definiciji se noben izraz ne ovrednoti. Ob vrednotenju enega izraza se bodo po potrebi ovrednotili še potrebni izrazi, od katerih je trenutni izraz odvisen.

3.5 Primerjava vzorcev

Primerjava vzorcev je način razločevanja med posameznimi konstruktorji algebraičnih podatkovnih tipov. Core nima gnezdenja vzorcev, zato je primerjanje analogno večsmernim pogojnim stavkom - izrazom *switch*. Izraz *case* veže izraz na ime tako, da je njegova vrednost dostopna v lokalnem področju, in nato z njim primerja naštete vzorce. V primeru primitivne vrednosti so to literali, v primeru algebraičnega tipa pa konstruktorji in nova imena za polja konstruktorja. Na voljo pa so tudi privzete alternative, analogne *else* delu pogojnih stavkov.

Izvajanje ovrednoti primerjani izraz in po vrsti primerja alternative. Ko se prva alternativa ujame, se izbere in vrne ustrezna desna stran.

Primer funkcije negacije s primerjanjem vzorcev nad algebraičnim tipom Boolovih vrednosti.

```
negate :: Bool -> Bool =  \(p::Bool) ->
    %case Bool p %of (wild::Bool) {
        True -> False;
        False -> True
    };
```

Funkcija *negate* slika iz logične vrednosti v logično vrednost, to nam pove že njen tip. Implementacija pa je abstrakcija, ki sprejme argument p in nato nad njim izvede primerjanje vzorcev. V njem definiramo, da se konstruktor *True* preslika v neresnično vrednost *False*, konstruktor *False* pa v resnično vrednost *True*. Tukaj je pomembno, da simbola *True* in *False* na levi strani služita kot vzorec, s katerim se primerja, na desni pa kot konstruktor brez argumentov, ki ustvari novo vrednost.

3.6 Stranski učinki

Čista funkcija preslika svoje argumente in proste spremenljivke v rezultat. Kakršnakoli drugačna operacija je učinek; torej dostop do pomnilnika, delo z omrežjem ali vhodno/izhodnimi napravami ipd. Jezik sam ne ponuja konstruktov za takšne operacije, ne prepoveduje pa definicije primitivnih funkcij (npr. v standardni knjižnici), ki bi imele učinke. Semantika jezika pa naredi uporabo nečistih funkcij precej zoprno.

Prva težava je sklepanje o vrstnem redu učinkov ob ne-striktnem vrednotenju izrazov. Tudi če se skozi to prebijemo, se bodo prevajalniki za Core zanašali na referenčno transparentnost in prepisovali izraze ter jih poenostavljali. Kot primer lahko vzamemo eta razširitev

$$g = \lambda x.f\ x \tag{3.14}$$

Takšna definicija funkcije g bo ob vsakem izvajanju ovrednotila izraz f in s tem izvedla tudi vse učinke fa. Prevajalnik (ali programer) lahko opazi tak izraz in ga

z eta redukcijo spremeni v bolj učinkovit

$$g = f \tag{3.15}$$

V odsotnosti učinkov je ta izraz ekvivalenten, vendar lahko memoizira vrednost izraza f in je zato lahko učinkovitejši. Ob prisotnosti učinkov pa smo spremenili semantiko, $f$ovi učinki se zdaj izvedejo samo enkrat.

Zaradi boljših možnosti optimizacije prevajalnik predpostavi, da so vsi izrazi čisti, kar pa oteži delo z učinki. Obstaja več rešitev tega problema, konsenzus v skupnosti (in implementacija v GHC) pa je uporaba monade za sekvenčenje učinkov.

3.6.1 IO monada

Najprej definiramo tip *RealWorld*, katerega vrednost predstavlja trenutno stanje sveta. Seveda vrednosti, ki bi opisala cel svet, ne moremo skonstruirati, lahko pa ustvarimo preprosto vrednost, ki služi kot žeton. Lastnik te vrednosti ima pravico, da na mestu spreminja svet, torej opravlja stranske učinke. Seveda mora na koncu žeton tudi vrniti. Funkcije, ki uporabljajo tak žeton, imajo torej tip oblike

```
RealWorld -> (a, RealWorld)
```

kjer je a tip rezultata funkcij. Paziti moramo le, da vse funkcije, ki uporabljajo žeton, tega ne duplicirajo ali uničijo. Tako zagotovimo, da vedno obstaja natanko ena instanca. To dosežemo tako, da direktno z žetonom dela samo peščica primitivnih funkcij, ki so implementirane izven jezika, tako konstruktorji kot tudi tip *RealWorld* sam pa so skriti, izvozimo le netransparentni *newtype* ovitek.

```
newtype IO a = IO (RealWorld -> (a, RealWorld))
```

Primitivne funkcije torej sprejemajo nekaj parametrov in vračajo vrednosti tipa *IO a*, kjer je a funkciji sprecifičen tip rezultata.

Ker se za *IO* skriva funkcija, s tem še nismo izvršili primitivne funkcije in s tem stranskih učinkov. *IO* objekt je torej samo opis akcije, še vedno pa je čista funkcijska vrednost in ne potrebuje nobene posebne obravnave v prevajalniku. Lahko izvajamo poljubno agresivne optimizacije, vrsti red učinkov se bo ohranil, saj smo uvedli *podatkovno odvisnost* med klici primitivnih funkcij, katerih deklaracije prevajalniku niso vidne.

Ker pa je tip *IO* netransparenten, progamer ne more sekvenčiti akcij. Zato potrebuje vmesnik, ki omogoča kompozicijo *IO* akcij. Izkaže se, da je *IO* samo specializacija stanja in ima instanco monade. To definiramo z dvema funkcijama

```
return :: %forall a     . a -> IO a
bind   :: %forall a b . IO a -> (a -> IO b) -> IO b
```

Funkcija *return* dvigne poljubno vrednost tako, da poleg nje naprej posreduje žeton, *bind* pa zaporedno zveže akcije. Opravlja nalogo, podobno kompoziciji funkcij, vendar poleg pravilno nosi žeton. Implementacija v Core je nekoliko dolga, zato prilagamo raje ekvivalentno in precej bolj koncizno implementacijo v Haskellu

```
module IO (IO, return, (>>=)) where

data RealWorld = RealWorld
newtype IO a = IO { unIO :: RealWorld -> (a, RealWorld)}

return :: a -> IO a
return a = IO (\w -> (a, w))

bind :: IO a -> (a -> IO b) -> IO b
bind (IO f) g =
    IO (\w -> let (a, w') = f w in (unIO (g a)) w)
```

Tako smo zagotovili, da je programiranje vedno varno, smo pa tudi povsem onemogočili opravljanje učinkov, saj programer ne more nikjer dobiti vrednosti *RealWorld* in *IO* akcije pognati. Naš program potrebuje vstopno točko, običajno poimenovano *main* tipa *IO ()*, ki jo pokliče sistem izvajanja in ji poda vrednost tipa *RealWorld*. Nato zahteva nazaj končno vrednost žetona in tako posredno, preko podatkovnih odvisnosti, zahteva izvajanje celotnega programa in z njim vseh učinkov.

Na tej točki opazimo, da te vrednosti nismo nikjer potrebovali in je služila le za definiranje eksplicitnih odvisnosti, zato lahko podamo arbitrarno vrednost.

3.7 Konkretna sintaksa

Sedaj, ko smo seznanjeni s semantiko in konstrukti jezika Core, si poglejmo še gramatiko konkretne sintakse[40] za tekstovno predstavitev.

V predstavitvi sintakse oglati oklepaji pomenijo opcijsko pojavitev, zaviti pa nič ali več pojavitev. Znak + za zavitimi oklepaji pomeni vsaj eno pojavitev. Vse ključne besede se začnejo z znakom %, ki je tudi del same sintakse.

```
modul          module   ->  %module mident { tdef ; }
                                          { vdefg ; }
```

Modul definiramo z ključno besedo *%module*, imenom modula in definicijami tipov ter vrednosti.

```
def. tipa      tdef     ->  %data qtycon { tbind } =
                              { [ cdef {; cdef } ] }
                        |   %newtype qtycon qtycon
                              { tbind } = ty
```

Definicija tipa je lahko definicija algebraičnega tipa *%data* s seznamom spremenljivk tipov in konstruktorjev ali pa definicija *%newtype*, ki definira samo en konstruktor.

```
def. konstr.   cdef     ->  qdcon { @ tbind }{ aty }+
```

Definicija konstruktorja je sestavljena iz imena konstruktorja in njegovih polj.

```
def. vrednosti vdefg    ->  %rec { vdef { ; vdef } }
                        |   vdef
               vdef     ->  qvar :: ty = exp
```

Definicija vrednosti je lahko definicija enega imena, njegovega tipa in izraza, v katerega se to ime ovrednosti. Lahko pa je seznam definicij, ki so med seboj rekurzivne.

```
atomarni izraz aexp     ->  qvar
                        |   qdcon
                        |   lit
                        |   ( exp )
```

Atomarni izraz je samo okrajšava za nekaj alternativ, izraz pa je bistvo jezika. Izraz je lahko

```
izraz          exp      ->  aexp
```

atomarni izraz,

```
                        |   aexp { arg }+
```

aplikacija argumentov na atomarni izraz,

```
                        |   \ { binder }+ -> exp
```

abstrakcija,

```
                        |   %let vdefg %in exp
```

izraz *let*, ki definira vrednosti iz *vdefg* tako, da so vidne znotraj izraza *exp*

```
                        |   %case ( aty ) exp
                              %of vbind
                              { alt { ; alt } }
```

ali primerjava vzorcev. V primerjavi vzorcev definiramo atomarni tip in novo ime *vbind* za primerjani izraz, saj ta ni nujno atomaren in se želimo v telesu sklicevati nanj. Sledi zaporedje vzorcev.

```
                        |   %cast exp aty
                        |   %note " { char } " exp
```

%cast definira prelive, *%note* pa opombe na izrazih. Te so namenjene izključno za programerju oz. razhroščevalniku.

```
argument       arg      ->  @ aty
                        |   aexp
```

Argumenti so lahko podani funkcijam na nivoju vrednosti ali funkcijam na nivoju tipov, zato so lahko vrednosti ali tipi.

```
alt. vzorci    alt      ->  qdcon { @ tbind }
                                  { vbind } -> exp
                        |   lit -> exp
                        |   %_ -> exp
```

Posamezne alternative pri primerjanju vzorcev so definirane na tri načine. Alternativa je lahko ujemanje konstruktorja; v tem primeru podamo seznam imen, na katera se vežejo polja konstruktorja. Lahko primerjamo tudi z literali ali podamo privzeto alternativo %_. Vse tri definicije pa imajo tudi izraz na desni strani, ki je ovrednoten, če se vzorec ujema.

```
vezava imena     binder   ->  @ tbind
                          |   vbind
vezava tipa      tbind    ->  tyvar
                          |   ( tyvar :: kind )
vez. vrednosti   vbind    ->  ( var :: ty )
```

Različne vezave so samo različne sintakse podajanja novih imen za vrednosti za različne namene.

```
literal          lit      ->  ( [-] { digit }+ :: ty )
                          |   ( [-] { digit }+ %
                              { digit }+ :: ty )
                          |   ( ' char ' :: ty )
                          |   ( " { char } " :: ty )
```

Podprti literali so števila (poljubne velikosti), znaki v enojnih narekovajih in nizi v dvojnih narekovajih.

```
znak             char     ->  ASCII znak in v razponu
                              0x20-0x7E razen
                              0x22,0x27,0x5c
                          |   \x hex hex
                  hex     ->  0..9 |a..f
ime              mident   ->  pname : uname
                 tycon    ->  uname   type
                 qtycon   ->  mident . tycon
                 tyvar    ->  lname
                 dcon     ->  uname
                 qdcon    ->  mident . dcon
                 var      ->  lname
                 qvar     ->  [ mident . ] var
```

```
lname     -> lower { namechar }
uname     -> upper { namechar }
pname     -> { namechar }+
namechar  -> lower | upper | digit
lower     -> a | b | .. | z | _
upper     -> A | B | .. | Z
digit     -> 0 | 1 | .. | 9
```

Imena z različnimi pravili se uporabljajo v različne namene. Tu je definiranih nekaj sinonimov za večjo preglednost.

Sledijo še definicije tipov in vrst, s katerimi pa se to delo ne ukvarja. Definirani so v članku [38].

```
atomarni tip     aty      -> tyvar
                          |  qtycon
                          |  ( ty )
osnovni tip      bty      -> aty
                          |  bty aty
                          |  %trans aty aty
                          |  %sym aty
                          |  %unsafe aty aty
                          |  %left aty
                          |  %right aty
                          |  %inst aty aty
tip              ty       -> bty basic type
                          |  %forall { tbind }+ . ty
                          |  bty -> ty
atomarna vrsta   akind    -> *
                          |  #
                          |  ?
                          |  bty :=: bty
                          |  ( kind )
vrsta            kind     -> akind
                          |  akind -> kind
```

Poglavje 4

Programski jezik JavaScript

JavaScript je interpretiran in dinamično tipiziran programski jezik. Združuje skriptno, imperativno in funkcijsko paradigmo s posebnim pristopom k objektom - prototipnim dedovanjem. Poudarek ima na enostavni sintaksi, večina zmogljivosti je implementirana v sistemu izvajanja. Pretežno se uporablja v spletnih brskalnikih.

4.1 Kratka zgodovina

Ta razdelek je povzet pretežno po [36].

V letu 1992 je svetovni splet s predstavitvijo brskalnika Mosaic postal podprt na vseh večjih platformah: Windows, Macintosh in Unix. Prinesel je možnost prikazovanja slik med tekstom. Pojavili so se zametki vizije, da bo svetovni splet postal platforma za razvoj od operacijskega sistema neodvisnih aplikacij. A HTML sam po sebi je omogočal samo opis statičnih dokumentov in zato ni bil dovolj zmogljiv, splet je potreboval prenosljiv programski jezik.

Sunova Java je bila dober kandidat za ta jezik; sorodna popularnemu jeziku C++ in s prenosljivim sistemom izvajanja. Imela pa je svoje težave. Java je bila v tem času (pred implementacijo JIT tehnologije) precej počasna, še hujša pa je bila latenca zagona virtualnega stroja. Ker je vsak program tekel v svojem, je to pomenilo, da je bilo potrebno čakati na zagon ob vsakem programu znova.

Vedoč, da je Java kompleksen prevajan jezik, je podjetje Netscape želelo razviti enostaven interpretiran jezik, ki ga bo enostavno umestiti v spletne strani in bo privlačen amaterskim programerjem kot komplement Javi. Zaposlili so Brendana

Eicha in mu dali 10 dni, da razvije in implementira prototip tega jezika. Čez 10 dni je bilo namreč napovedano lansiranje beta različice novega brskalnika podjetja Netscape. JavaScript se je prijel in do danes praktično izrinil Javo iz brskalnika. K temu so pripomogli tudi ECMA standardi, ki so definirali točno specifikacijo jezika. Tako se je začela t.i. *Web 2.0* revolucija, v kateri so spletne strani počasi začele nadomeščati spletne aplikacije, ki jih poganja JavaScript.

4.2 Glavne značilnosti

JavaScript je šibko in dinamično tipiziran. Faze prevajanja nima, ob nalaganju kode v interpreter se preveri samo sintaksa. Sistem tipov ima več posebnosti. Zanimiv je, ker ima vsa števila implementirana z enako predstavitvijo - 64-bitnimi števili v plavajoči vejici. Podpira objektno orientacijo, vendar ne s klasičnimi razredi in dedovanjem, ampak uporablja prototipno dedovanje. To pomeni, da objekt podeduje polja od svojega prototipa, ki je zgolj drug objekt. To daje programerju več fleksibilnosti na račun daljše kode v nekaterih primerih.

Ker je bil namenjen enostavnemu pisanju skript nima koncepta poimenovanih področij in modulov, vse vrednosti so vidne v globalnem področju. To od programerja zahteva pazljivost, da deklarira lokalne spremenljivke in ne ponesreči uporablja globalnih spremenljivk in tako pokvari stanja.

Funkcije so v jeziku prvorazredne vrednosti, lahko jih shranimo v spremenljivko, podamo funkciji in jih iz funkcije vrnemo. Lahko definiramo tudi anonimne funkcije z uporabo funkcijskih literalov. Funkcija se v svojem telesu lahko nanaša na posebno ime *this*, ki vedno predstavlja objekt, katerega del je ta funkcija. Posebnost tega imena je, da se veže pozno - šele ob klicanju funkcije. Tako lahko ista funkcijska vrednost, shranjena v različnih objektih, vidi drugačno vrednost za ime *this*.

4.3 Node.js

Ker želimo zaradi lažjega testiranja JavaScript kodo poganjati iz ukazne vrstice in ne samo iz brskalnika, potrebujemo samostoječi pogon. Odločili smo se za Node.js[8], ki je najbolj razširjena implementacija in ovija Googlov pogon V8[7],

pri tem pa ponuja tudi ustrezne vmesnike za interakcijo z operacijskim sistemom in delovanje kot strežnik.

JavaScript je bil sprva namenjen za rabo v brskalnikih in se je dolgo časa uporabljal samo tam, sprva le za razne bonbončke, nato pa tudi za implementacijo ključnih zmogljivosti. Ključnega pomena je bil pojav programskega vmesnika za asinhrono nalaganje (AJAX). Ta programerju omogoča opravljanje HTTP zahtevkov tudi po tem, ko je stran naložena in tako omogoča, da spletna stran komunicira z oddaljenimi programskimi vmesniki.

Porast uporabe je spodbudila razvoj bolj učinkovitih pogonov za interpretacijo v brskalniku. Hitrost izvajanja JavaScripta je za nekaj časa postala ena od prednosti brskalnika in vsi večji brskalniki so se podali v tekmovanje, najbolj pa je pridobil končni uporabnik s sedaj boljšo izkušnjo.

Leta 2009 je Ryan Dahl izdal prvo različico strežniške platforme *Node.js*. To je JavaScript pogon V8 iz podjetja Google, ki poganja njihov brskalnik Chrome z nekaj dodatki za pisanje strežniške kode. Prvi večji dodatek je sistem modulov. V eno datoteko programer zapiše en modul, nato pa lahko iz drugih modulov zahteva ta modul in dobi objekt z vmesnikom tega modula. Moduli si med seboj ne delijo globalnega imenskega področja in je programiranje zato nekoliko bolj varno.

Drugi večji dodatek pa so razširitve napisane v jeziku C in C++. Te razširitve primarno implementirajo vhodno/izhodne operacije. Kot razširitev je lahko napisana tudi procesorsko intenzivna koda, obstajajo pa celo celotni gonilniki za uporabo zunanjih podatkovnih baz, ki so napisani v C++. Taka razširitev izpostavi vmesnik, ki ga programer lahko kliče iz JavaScripta.

Node.js ohranja model hkratnega izvajanja iz JavaScripta v brskalniku, vse teče v eni niti. Iz tega sledi, da si ne moremo privoščiti blokiranja niti zaradi sistemskega klica, saj bi tako popolnoma ustavili vse izvajanje v naši aplikaciji. *Node.js* to rešuje z asinhronimi vhodno-izhodnimi operacijami. Vsaki operaciji podamo še povratni klic - funkcijo, ki se pokliče z rezultatom operacije. Vsi ti klici se izvajajo na isti niti po principu dogodkovne zanke.

Na prvi pogled je tak pristop k V/I operacijam precej samosvoj, vendar je konceptualno zelo združljiv z monadičnimi operacijami (poglavje 3.6.1). Kot povratni klic podamo neke vrste delimitirano kontinuacijo, podobno kot funkcija, ki jo podamo v *bind*.

4.4 Standardizacija

Kot omenjeno (v razdelku 4.1), je prvo različico JavaScripta razvilo podjetje Netscape. Drugi v tistem času popularni spletni brskalnik je bil Internet Explorer podjetja Microsoft. To je želelo prav tako podpreti skripte na interaktivnih straneh in je razvilo svoj bolj ali manj združljiv dialekt JScript.

Da bi zagotovili združljivo okolje v vseh brskalnikih, je podjetje Microsoft leta 1996 dostavilo na JScript specifikacijo mednarodni standardizacijski organizaciji ECMA[1]. Ta je naslednje leto izdala prvo specifikacijo jezika ECMAScript, ime JavaScript pa se je prijelo in se še vedno uporablja.

Kasneje so se pojavili novi JavaScript pogoni s svojimi dialekti. Prinašali so nove zmogljivosti jezika, predvsem pa so bile razlike v vmesniku standardne knjižnice. Zato so se razvijale tudi ECMAScript specifikacije. V času pisanja diplomske naloge je aktivna specifikacija ECMASCript 5, večina popularnih pogonov pa je precej blizu standarda, dodaja le nekaj svojih razširitev.

[1]Evropsko združenje proizvajalcev računalnikov (angl.European Computer Manufacturers Association)

Poglavje 5

Implementacija prevajanja

Prevajalnik je sam napisan v programskem jeziku Haskell in je zastavljen kot cevovod več pretvorb in predstavitev kode (slika 5.1).

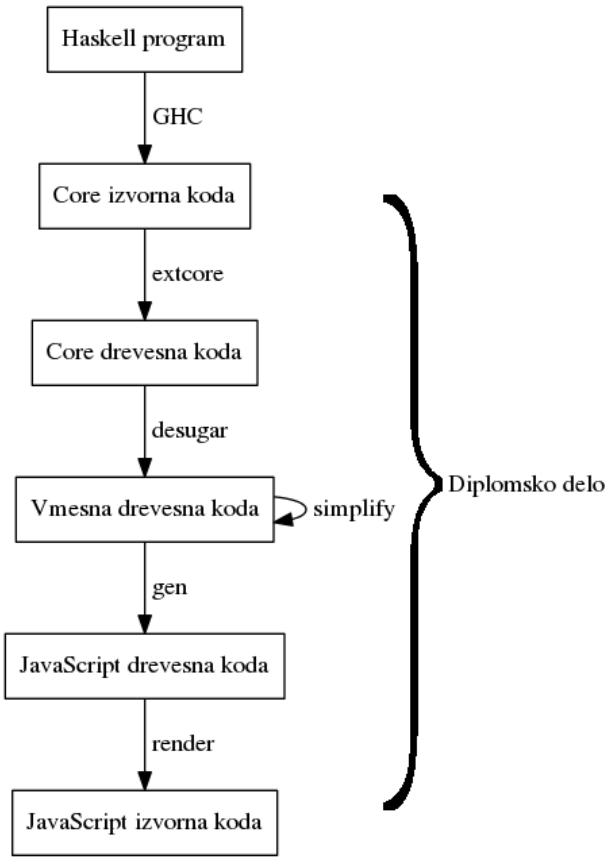
Začetno tekstovno kodo naloži in razpozna v drevesno kodo s pomočjo paketa *extcore*[41]. Nato naš modul *Desugar* odstrani tipe in doda informacijo o stanju ovrednotenosti ter tako spremeni program v vmesno drevesno kodo. Nad njo teče poenostavljalnik *Simplify*, ki odstrani nepotrebna zaprtja in vrne enostavnejšo drevesno vmesno kodo. Naslednji korak je generiranje drevesne kode JavaScripta, kar opravi naš modul *Gen* in drevo preda modulu *ECMAScript*[19], ki pripravi končno tekstovno obliko.

5.1 Vhod

Jezik Core je nastal primarno kot vmesna koda znotraj prevajalnika GHC. Za potrebe razhroščevanja so avtorji definirali tudi tekstovno predstavitev programov v tem jeziku. V [40] je razvojna ekipa GHC definirala predstavitev tako v abstraktni sintaksi kot tudi konkretno gramatiko za izpis in razpoznavanje programov v jeziku Core.

Ta konkretna gramatika je nekakšen psevdo-Haskell. Osnovna sintaksa je sicer podobna, a manjka veliko sladkorčkov. Vsi tipi so sedaj obvezni, prav tako so obvezna nekatera ločila, ki jih lahko v Haskellu izpustimo na račun belih presledkov.

Ker naš prevajalnik za vhod sprejema konkretno sintakso (tekstovne datoteke), je za njegovo implementacijo potrebna tudi konkretna implementacija za razpo-



Slika 5.1: Koraki prevajanja

znavalnika. Odločili smo se, da uporabimo obstoječo knjižnico *extcore* [41], ki so jo objavili avtorji izvirnega članka, kot del prestrukturiranja izvorne kode prevajalnika GHC.

Ta knjižnica implementira abstraktno sintakso za Core po [40] in razpoznavalnik ter izpisovalnik te sintakse.

5.2 Sistem modulov

JavaScript v trenutni specifikaciji (ECMAScript 5) ne definira sistema modulov. To bi pomenilo, da bi prevajalnik, ki generira JavaScript kodo, moral implementirati tudi fazo povezovanja in sproducirati nestrukturiran izhod. Na srečo pa obstaja nekaj tehnologij, ki implementirajo sisteme modulov v "uporabniškem

prostoru" JavaScripta. Glavna dilema pri izbiri tehnologije je sinhrono/asinhrono nalaganje. Asinhrono nalaganje je ključnega pomena za JavaScript v brskalniku, ker omogoča hkratno nalaganje modulov in tako skriva latenco, sinhrono nalaganje pa poenostavi program, a je primerno le za kodo, ki teče lokalno.

Obstaja več kompromisov, odločili smo se, da uporabimo Node.js sistem nalaganja modulov, ki je sinhron, in orodje Browserify[1] za povezovanje in izgradnjo izhodne datoteke s celotno izvorno kodo. Browserify je program, ki simulira sistem nalaganja modulov v brskalniku ob pomoči faze predprocesiranja. Pregleda izvorne datoteke in najde vse potrebne module, nato pa njihovo kodo združi v eno datoteko tako, da vsak modul definira vrednost s svojim imenom. Implementacija nalaganja modulov samo vrne ustrezno vrednost, ki je že naložena.

5.2.1 Uvozi

Pri prevajanju jezika Core v binarne dvojiške datoteke z GHC se izvede tudi korak povezovanja tako, da so vsi simboli vidni. Pri prevajanju v JavaScript pa želimo, da koda ostane razdeljena po modulih kot izvirna[1], zato se pojavi potreba po mehanizmu uvoza (import) simbolov, kot ga ima na primer Haskell. Kot že omenjeno, smo za to nalogo uporabili sistem modulov iz *Node.js*, ostaja nam še problem določanja uvozov, v Core so namreč vsi simboli povsod vidni.

Na srečo so vse reference na vrednosti absolutne; iz imena je razvidno, ali gre za lokalno spremenljivko ali kličemo v drug modul. Potrebujemo le seznam vseh modulov, ki jih referenciramo. To pa enostavno rešimo z enim prehodom čez abstraktno sintaksno, med katerim si beležimo uporabljene module. Za vsak modul vstavimo na vrh generiranega JavaScripta en stavek *require*, ki definira ime modula v trenutnem področju tako, da uvozi ustrezno datoteko.

5.2.2 Ciklične reference in vidnost

Haskell je po svojih področjih in vidnosti precej različen od JavaScripta. Zaradi ne-striktnega vrednotenja je smiselno, da vrstni red definicij ni pomemben in se torej znotraj modula definicije poljubno referencirajo.

[1]Povezovanje opravljamo v kasnejši fazi zaradi zmanjšanja kompleksnosti prevajalnika.

V JavaScriptu pa so področja dinamična in je vrstni red zelo pomemben, referenciramo lahko samo vrednosti, ki so že nastavljene. Področja so sicer leksikalno ločena, vendar ne na bloke, kot je popularno v drugih jezikih, temveč je eno področje telo ene funkcije. Ker so definicije funkcij lahko gnezdene, so lahko gnezdena tudi področja.

JavaScript ima še eno posebnost, ki sicer nima formalnega imena, pogovorno pa se imenuje dvigovanje. Dvigujejo se namreč deklaracije spremenljivk znotraj področij. Če se držimo pravil lepega programiranja v JavaScriptu, svoje spremenljivke vedno deklariramo (v JavaScriptu to ni nujno potrebno!), običajno na mestu inicializacije, lahko pa že prej (ne pa kasneje, vrstni red je pomemben). Kjerkoli pa spremenljivko deklariramo, se njena deklaracija zgodi, kot bi bila deklarirana na vrhu trenutnega področja. Najbolj je to razvidno na primeru. Sledeči program

```
function sampleScope() {
  helper();
  var a = 1;
  var b = 2;
}
```

se interpretira kot

```
function sampleScope() {
  var a, b;
  helper();
  a = 1;
  b = 2;
}
```

Seveda podobno velja tudi za funkcije. Ta program

```
function sampleScope() {
  helper();
  function local() {
    return 42;
  }
  return local();
}
```

se interpretira kot

```
function sampleScope() {
  var local;
  helper();
  local = function () {
    return 42;
  };
  return local();
}
```

Ker se telo funkcije izvaja kasneje kot njena definicija, lahko to izrabimo za referenciranje naprej! Ob času definicije funkcije smo že deklarirali tudi imena funkcij, ki jih bomo definirali kasneje, zato jih lahko referenciramo. Pogoj za to je, da se prvo izvajanje funkcije zgodi šele po vseh definicijah vrednosti, ki jih referenciramo, v nasprotnem primeru te ne kažejo na nobeno vrednost. Temu zlahka zadostimo z upoštevanjem enega preprostega pravila: med definicije funkcij ne vstavljamo prostih izrazov. Če najprej definiramo vse funkcije in nato ovrednotimo izraz, ki te funkcije uporablja, lahko znotraj funkcij prosto referenciramo naprej in nazaj, ob prvem izvajanju funkcije bodo vsa imena že kazala na definicije. Še primer takega programa:

```
function foo() {
  return bar();
}

function bar() {
  return "hello world";
}

console.log(foo());
```

Program se interpretira kot

```
var foo, bar;

foo = function () {
```

```
  return bar();
};

bar = function () {
  return "hello world";
};

console.log(foo());
```

Ne-striktne vrednosti (vse vrednosti v Core) so implementirane z funkcijami (glej poglavje 5.3) tako, da lahko prosto uporabljamo reference naprej brez posebnih sprememb. Takšne definicije uporabimo za lokalne vrednosti, za vrednosti, ki jih modul izvozi, pa je stanje nekoliko bolj enostavno. Sistem modulov nam v vsaki datoteki definira poseben objekt *exports*, ki definira zunanji vmesnik modula. Na njem definiramo vse vrednosti, ki jih izvozimo. Ker gre za objekt, se selekcija posameznih polj zgodi šele ob uporabi. Pravila so podobna kot pri spremenljivkah, le da so deklaracije tu implicitne. Potrebujemo le še eno preslikavo, da omogočamo referenciranje lastnih izvoženih definicij znotraj modula po polnem imenu.

Na vrhu modula dodamo

```
var module_name = exports;
```

kjer je *module_name* enako imenu modula. Tako so definicije na polnem imenu modula avtomatsko izvožene, kot tudi direktno uporabne znotraj trenutnega modula. Primer interakcije med lokalnimi in izvoženimi vrednostmi:

```
var module = exports;

var local_const1 = function() {
  return module.exported();
}

module.export = function() {
  return "hello";
}
```

Stavki v programu se izvedejo po vrsti. Najprej definiramo modul kot sinonim za

izvožene vrednosti. Nato deklariramo lokalno spremenljivo in ji priredimo funkcijsko vrednost. Pomembno je, da na tej točki interpreter ne poskša razrešiti imen znotraj telesa literala funkcije. Na koncu v modulu definiramo še funkcijo, ki vrne konstanten niz. Znotraj modula lahko pokličemo obe funkciji. Klic izvožene funkcije je trivialen, klic lokalne pa bo šele sedaj razrešil ime izvožene funkcije in uspešno vrnil njen rezultat.

5.3 Implementacija ne-striktnosti

V večini striktnih jezikov je programerju omogočeno implementirati ne-striktno vrednotenje s pomočjo funkcij brez parametrov[27] (t.i. thunki). Tako lahko v jeziku s klicem po vrednosti implementiramo klice po imenu. Dodatne težave se pojavijo, če želimo semantiko klica po potrebi – torej klic po imenu, ampak z memoizacijo.

V ne-striktnem jeziku je dobra implementacija ne-striktnega vrednotenja ključnega pomena. Osredotočili se bomo ravno na funkcije brez parametrov z memoizacijo. Poznamo dva osnovna modela implementacije memoizacije.

5.3.1 Celični model

Klasično memoizacijo implementira t.i. celični model. Namesto slovarja, ki ga običajno potrebujemo za implementacijo memoizacije, tukaj potrebujemo samo eno celico, saj imamo natanko en možen seznam argumentov funkciji, prazen seznam. Po tej celici je model dobil tudi svoje ime.

Za memoizacijo funcije brez parametrov zgradimo objekt, ki ga opremimo s poljem, ki hrani funkcijo za klic po imenu, poljem, ki memoizira vrednost in poljem za hranjenje zastavice o opravljeni memoizaciji. Ta je ključnega pomena saj so vse možne “prazne vrednosti” dovoljene kot rezultat funkcije in jih moramo prav tako memoizirati, drugače bi po nepotrebnem podvajali delo.

Težava se pojavi, ko želimo dostopati do vrednosti. Ob vsakem dostopu je potrebno najprej preveriti zastavico in po potrebi ovrednotiti funkcijo in shraniti njen rezultat, šele nato lahko dostopamo do polja z vrednostjo. Izkaže se, da moderni JavaScript pogoni dovolj dobro opravljajo vrinjanje konstantnih funkcij,

da lahko spišemo svojo abstrakcijo za dostop do vrednosti in jo zapakiramo v objekt. Končni rezultat je naslednji konstruktor

```
var cell1 = function(f) {
  return {
    memod: false,
    value: null,
    func: f,
    apply: function() {
      if (!this.memod) {
        this.value = this.func();
        this.memod = true;
      }
      return this.value;
    }
  };
};
```

Uporaba tega konstruktorja pa izgleda tako

```
var byName = function(){
  return 1 + 2
};
var byNeed = cell1(byName);
var value = byNeed.apply();
```

Ozko grlo tu predstavlja dostop do polj objekta, objekti so v JavaScriptu namreč predstavljeni s slovarji z nizi za ključ vsakega polja. Temu se lahko ognemo tako, da uporabimo JavaScriptu lastna zaprtja čez funkcije namesto objektov. Na ta način se ognemo indeksiranju v objekt in gradnji nizov za ključe. V času izvajanja tako še vedno skonstruiramo objekt, vendar sta njegova velikost in seznam članov statična in tako podvržena optimizaciji.

```
var cell2 = function(f) {
  var memod = false;
  var value = null;
  return function() {
```

```
    if (!memod) {
      value = f();
      memod = true;
    }
    return value;
  };
};
```

Poleg tega nam takšna implementacija ponudi tudi lepši vmesnik; vrnemo namreč novo funkcijo, ki ima spremenljivo stanje v svojem zaprtju. Primer uporabe

```
var byName = function(){
  return 1 + 2
};
var byNeed = cell2(byName);
var value = byNeed();
```

5.3.2 Samoposodobljajoči model

Alternativa je, da celico nadomestimo z navzven čisto vrednostjo (podobno kot *cell2*), ki interno skrbi za mutiranje vrednosti in tako implementira memoizacijo.

Takšna funkcija brez parametrov ob prvem izvajanju izvede klic po imenu in se nadomesti z novo funkcijo, ki preprosto vrne novo vrednost. Prednost take implementacije je, da ne vsebuje pogojnih stavkov in tako odpira vrata novim optimizacijam.

Žal pa moramo tako funkcijo zaviti v neke vrste celico, ki bo enkapsulirala spremenljivo stanje, za uporabo namreč potrebujemo stabilno ime za to spremenljivo stanje.

```
var self1 = function(f) {
  return {
    apply: function() {
      var r = f();
      this.apply = function() {
        return r;
      };
```

```
      return r;
    }
  };
};
```

Uporaba je enaka kot pri *cell1*.

Hitro opazimo, da imamo podobne težave z ustvarjanjem objektov in indeksiranjem vanje, vendar pa objekt tu potrebujemo zaradi stabilnosti imena. Če bi ga nadomestili z zaprtjem, bi morali vrniti vrednost, ki pa je kasneje ne bi mogli več spreminjati.

Na srečo si lahko sposodimo trik iz lambda računa: eta razširitev. V lambda računu lahko dano funkcijo f razširimo v izraz $\lambda x.fx$. V JavaScriptu lahko tako razširitev implementiramo s funkcijo

```
var expand = function(f) {
  return function (x) {
    return f(x);
  };
};
```

V čistem lambda računu eta razširitev ohranja semantiko, kar pa ne drži v prisotnosti stranskih učinkov. V prisotnosti leksikalnih področij (npr. JavaScript) lahko f v področju definicije spreminja vrednost. Eta razširitev lahko tako uporabimo za vračanje stabilnega imena za spremenljivo vrednost, kar je točno funkcionalnost, ki jo potrebujemo.

```
var self2 = function(f) {
  var apply = function() {
    var r = f();
    apply = function() {
      return r;
    };
    return r;
  };
  return function() {
    return apply();
```

```
    };
};
```

Najprej definiramo spremenljivko *apply* v trenutnem zaprtju in nato vrnemo eta razširjeno referenco. Eta razširitev je tu ročno vrinjena. Ključ se skriva v definiciji spremenljivke *apply*. Definirana je kot funkcija, ki najprej ovrednoti f, nato povozi svojo implementacijo in na koncu vrne rezultat. Nova implementacija pa takoj vrne rezultat, ki ga hrani v svojem zaprtju.

5.3.3 Primerjava

Za različne implementacije smo primerjali čas izvajanja, skupaj z klicem po imenu (implementirano kot anonimna funkcija brez parametrov) za referenco. Testi so se izvajali na platformi *Node.js 0.10* na procesorju *Intel Ivy Bridge i7*.

število branj	1	2	10	1000
klic po imenu	25	18	19	11
cell1	76	48	26	13
cell2	57	40	23	13
self1	450	230	70	13
self2	86	64	39	13

Tabela 5.1: Primerjava časov izvajanja različnih implementacij ne-striktnosti za različno število branj na eno pisanje

Za vsako tehniko smo merili povprečen čas enega branja vrednosti, pri čemer smo testirali različne obremenitve branja in pisanja. Samo ustvarjanje izraza ima namreč drugačno ceno od branja njegove vrednosti, zato smo testirali scenarije, kjer je izraz ustvarjen in nato branj 1, 2, 10 in 1000 krat. Vsak scenarij je tekel v zanki s skupno 10^6 branji, tako da je imel pogon možnost optimizirati in prevesti kodo. Računan izraz je preprosto $1 + 1$, zato se za samo vrednotenje porabi minimalno časa. Meritve v nanosekundah na branja so v tabeli 5.1.

Klic po imenu je brez dvoma najhitrejši v vseh pogojih, kar je tudi pričakovano, saj opravi najmanj dela - ostale implementacije potrebujejo dodatne mehanizme za memoizacijo.

Po počasnosti izstopa *self1*; bremenita ga dodatna cena objektov in dinamično spreminjanje funkcij. Kot omenjeno, so moderni JavaScript pogoni zelo sposobni pri vrinjanju in optimizaciji *statičnih* funkcij, težave pa imajo, če kličemo funkcijo prek imena, ki ne kaže vedno na isto implementacijo.

Implementaciji s celičnim modelom teh težav nimata, dodatni pogojni stavek pa ne stane veliko, levji delež časa je potreben za dodaten nivo indirekcije. To je lepo vidno iz rezultatov. Pri eno- in dvokratem vrednotenju je čas približno dvakrat daljši, ravno cena gradnje dvojne količine zaprtij, pri desetkratnem pa se cena hitro približuje klicu po imenu, saj je vrednotenje nadaljnjih klicev od klica po imenu dražje le za en pogojni stavek.

Samoposodobljujajoči model pa ima prednost, ki ni razvidna iz meritev, to je razširljivost. Če želimo dodati logiko za zaznavanje ciklov v vrednotenju ali podporo hkratnemu izvajanju, lahko preprosto dodamo nove implementacije za funkcijo *apply* in tako zelo enostavno modeliramo končni avtomat brez nobene dodatne cene. Celični model pa za isti učinek potrebuje nova polja z zastavicami in več pogojnih stavkov; njegova cena izvajanja raste z novo funkcionalnostjo. Na srečo pa imata obe implementaciji navzven združljiv vmesnik in ju lahko kadarkoli zamenjamo, zato smo se za začetno implementacijo odločili za hitrejši celični model *cell2*.

5.3.4 Konsistentnost

Kdaj izraze oviti z ne-striktnim konstruktorjem in kdaj ne ter kdaj jih ovrednotiti, da bo program interno konsistenten? Odgovor je zelo enostaven: neovrednotene izraze ustvarjamo v izrazih *let* in jih vrednotimo v izrazih *case*, pomembno pa je, da tudi parametre aplikacije funkcij interpretiramo kot implicitne izraze *let* in samo funkcijo kot vrednost v izrazu *case*.

Pomembna opazka je, da imajo vsi izrazi v jeziku Core tip, torej imajo vrsto ★ (beri: tip). Ta vrsta predstavlja ne-striktne izraze. Izjema so le razpakirane vrednosti, na primer primitivna števila, ki imajo vrsto #.

Sprehodimo se skozi možne konstruktorje abstraktne sintakse in opazujmo obnašanje vrednotenja izrazov..

Reference na vrednosti

Ime vrednosti vrne natanko to, kar vsebuje vrednost. Torej v generirani kodi samo uporabimo ime vrednosti, ki jo referenciramo brez posebnih sprememb.

Aplikacija

Za aplikacijo fx lahko izberemo več taktik, med drugim bolj popularni *push-enter* in *eval/apply* [31](glej tudi za primerjavo). Uporabili smo *eval/apply*, ki je tradicionalno bliže imperativnim jezikom, ampak s poenostavitvijo, vse generirane funkcije so že v Curryjevi obliki in zato je številnost njihovih argumentov vedno statično znana: 1.

Za vrednotenje aplikacije torej postopamo po imenu taktike, najprej ovrednotimo funkcijo in jo nato apliciramo na argument, ki ga še nismo vrednotili. Končni rezultat je že ovrednoten - nalogo odlašanja prenesemo na klicatelja. Tako omogočimo primitivne funkcije, ki vračajo razpakirane vrednosti in si s tem pustimo odprta vrata za visoko performančne algoritme.

Abstrakcija

Abstrakcija je običajen izraz, torej mora imeti vrsto ★. Kaj pa se dogaja znotraj same abstrakcije? Najpomembneje je, da se semantika abstrakcije sklada s semantiko aplikacije, ta dva konstrukta sta si namreč dualna in kakršnokoli neskladje bi privedlo do nepravilnega delovanja. Ob aplikaciji uporabimo še nevrednotene argumente, zato abstakcija sprejme takšen argument, pričakujemo pa vrednotene, zato mora biti telo abstrakcije ovrednoten izraz. To preprosto dosežemo tako, da je telo običajen izraz, ki ga naknadno ovrednotimo.

Definicija vrednosti

Ta konstrukt po definiciji ustvarja nove neovrednotene izraze. Tako izrazi, ki jih pripnemo na imena, kot končni izraz so še neovrednoteni, prav tako pa je v neovrednotenem stanju celoten izraz *let*.

Primerjava vzorcev

Primerjava vzorcev po definiciji vrednoti izraze. Tako je edino smiselno, da sprejme neovrednoteni izraz in ga ovrednoti. Ni pa takoj jasno ali naj bo rezultat primerjave ovrednoten ali ne. Izkaže se, da imamo v vsakem alternativem stavku po en neovrednoteni izraz, tako da lahko preprosto izberemo pravega in ga vrnemo - celoten izraz *case* je tako neovrednoten, čeprav je izvedel vrednotenje nekega izraza.

5.3.5 Senčenje

Senčenje imena spremenljivke je definicija lokalne spremenljivke z enakim imenom, kot ga ima spremenljivka, ki je vidna in definirana v oklepajočem področju. Na žalost specifikacija jezika Haskell Core tega ekspilictno ne prepoveduje [40]. To pomeni, da lahko gnezdenje izrazov *let* definira isto ime z več vrednostmi, ki so vidna v različnih področjih. Primer takšnega izraza (zaradi berljivosti v Haskellu, kjer obstaja isti problem)

```
let a = 1
    b = a
 in let a = 2
     in a + b
```

Vrednost tega izraza mora biti 3. Vrednost b namreč vidi za a-jem skrito vrednost 1, v notranjem izrazu pa je vidna vrednost 2, čeprav je vrednost 1 še vedno posredno dostopna preko b.

To pomeni, da bi imeli v JavaScriptu z naivnim zaporednim stavljenjem definicij za gnezdene izraze težave. Druga definicija a-ja bi povozila prvo definicijo tako, da bi ob vrednotenju b-ja a že imel vrednost 2, kar pa ni zaželeno. To lahko rešimo z uvedbo novih področij v JavaScriptu. To ustreza takoj poklicani anonimni funkciji - edini način za definicijo novih področij je namreč definicija nove funkcije. Nihče pa nam ne brani, da ne bi te funkcije tudi kar takoj poklicali. Vsak izraz *let* ovijemo v anonimno funkcijo, tako bo a v zunanjem imel svojo vrednost, notranji let pa bo v svoji funkciji in bo imel zato svoje področje v katerem bo vrednost a-ja 2.

5.4 Algebraični podatkovni tipi

Algebraični tipi (glej razdelek 5.4) igrajo zelo pomembno vlogo, saj so eden od treh konstruktov, ki obstajajo v času izvajanja - ostala dva sta še neovrednoteni izrazi in primitivni tipi. Z učinkovitostjo neovrednotenih izrazov smo se že ubadali (glej razdelek 5.3), primitivni tipi pa so preprosto objekti iz JavaScripta. Na algebraičnih tipih pa leži breme implementacije vseh uporabniških tipov.

Odločili smo smo se za zelo lahkokategorno implementacijo; vrednost algebraičnega tipa je samo tabela njegovih polj in oznake konstruktorja. Tabelo smo uporabili, ker lahko z indeksiranjem omogočimo hiter dostop do elementov brez uporabe dragih ključev, ki jih imajo objekti. Prav tako statično vemo, da se velikost tabele ne bo nikoli spremenila. Vsaka tabela ima na indeksu 0 najprej oznako konstruktorja. Oznake so naravna števila. Statično vemo, kakšnega tipa je vrednost, in tako vemo, da ne moremo mešati konstruktorjev različnih tipov. Tako lahko preprosto vzamemo konstruktorje po vrsti deklaracije in jim priredimo naraščajoče oznake. Ta števila enolično določajo konstruktor. Naslednji elementi tabele so polja tipa po vrsti deklaracije.

Za ujemanje vzorcev (glej razdelek 5.5) potrebujemo tudi dekonstruktorje podatkovnih tipov. Dekonstruktor je funkcija, ki sprejme vrednost algebraičnega podatkovnega tipa in jo razstavi nazaj v posamezne argumente. Da se izognemo ponovnemu pakiranju argumentov, lahko v dekonstruktor podamo še funkcijo, ki iz argumentov izračuna vrednost - povratni klic. Ta funkcija se kliče z z argumenti konstruktorja, če je objekt bil konstruiran z ustreznim konstruktorjem, in njen rezultat je vrnjen, v nasprotnem primeru dekonstruktor ne vrne ničesar. Ostane pa problem lokacije dekonstruktorja. Za sam konstruktor je smiselno, da se nahaja v funkciji z enakim imenom, za dekonstruktor pa bi potrebovali novo sveže ime. Na srečo so v JavaScriptu tudi funkcije objekti in jim lahko dodamo polja. Dekonstruktor tako lahko živi v smiselnem polju *ImeKonstruktorja.unapply*.

Ta pristop izkorišča izbris tipov v prevajanju - vsa polja konstruktorja obravnavamo enako, to pa lepo sodeluje z dinamičnim tipiziranjem v JavaScriptu. Od algebraičnih *tipov* tako ostanejo samo konstruktorji, ki navzven izgledajo kot običajne funkcije, vračajo pa zgoraj definirano predstavitev s tabelo.

5.5 Implementacija primerjave vzorcev

Primerjava vzorcev je na nek način dualna definicijam - dekonstruira namreč konstruirane vrednosti. Drugače pogledano pa primerjavo vzorcev potrebujemo zgolj zaradi razločevanja alternativ v vsoti iz algebraičnih tipov. Čisti lambda račun, ki algebraičnih tipov nima, namreč ne potrebuje takšnega konstrukta.

Primarna naloga izraza *case* je odločanje med več izbirami. Naravna implementacija v JavaScriptu bi uporabila gnezdene *if* stavke za odločanje. Vendar se hitro pojavi vprašanje, kaj storiti v telesu tega stavka. Rezultata ne moremo vrniti, ker nismo v funkciji, zato moramo izbrano vrednost nekam shraniti: potrebujemo nove lokalne spremenljivke. To pa prinese težave, iskati moramo prosta imena, ki ne bodo trčila z imeni spremenljivk kode, ki jo prevajamo, paziti moramo na tok izvajanja in na polnost preverjanja.

Želeli bi zgraditi izraz, ki bo izbral med več alternativami, in se ovrednotil v pravo desno stran. Potrebujemo mehanizem za izbiro med alternativami in mehanizem za preverjanje pogojev znotraj izraza.

JavaScript ima, kot večina popularnih jezikov, kratkostične booleanove operatorje *ali* ter *in*. To pomeni, da bo vrednost izraza $a||b$ enaka a, kadar je a resničen (v tem primeru se b ne ovrednoti), drugače pa bo b. Ta konstrukt lahko izkoristimo za izbiro med alternativami.

Konstrukt za pogojno vrednotenje je blizu *if izrazom* iz funkcijskega sveta. Za razliko od *if* stavka ima *if* izraz vrednost. To pomeni, da ima vedno obe veji, ki se obe ovrednotita v vrednost, nato pa na podlagi predikata izbere pravo in ovrednoti samo tisto. Ta konstrukt je t.i. ternarni operator ? :, tako poimenovan, ker je edini operator, ki sprejme tri argumente.

Za vsako alternativo generiramo izraz, ki se ovrednoti v resnično vrednost, če je pogoj izpolnjen, v nasprotnem primeru pa v neresnično. Za podatkovne konstruktorje uporabimo njihove ustrezne dekonstruktorje, v katere podamo anonimne funkcije, ki vežejo imena in vrednotijo desno stran v svojem telesu. Za ujemanje literalov pa uporabimo ternarne izraze. Privzete alternative ne potrebujejo dodatnega ovijanja in se lahko takoj ovrednotijo. Na tem mestu izkoristimo dejstvo, da so desne strani neovrednoteni izrazi, torej s stališča JavaScripta funkcije, te pa so enakovredne resničnim vrednostim. Alternative zvežemo skupaj z logičnim operatorjem ali. Tako zagotovimo pravilen vrstni red preverjanja alternativ in kratki

stik ob prvi pravilni alternativi.

5.6 Vmesna koda

Na tej točni je razkorak med jezikoma Core in JavaScript že precej jasno viden. Daleč največja razlika je v striktnosti vrednotenja. JavaScript je inherentno strikten in je v njem ne-striktno vrednotenje potrebno simulirati. To je prisotno v vseh konstruktih in prinaša več robnih pogojev. Generiranje kode lahko poenostavimo tako, da ga razbijemo v dve fazi. V prvi fazi iz Core generiramo vmesno kodo, v drugi pa iz vmesne kode JavaScript.

Lastnosti vmesne kode so tu ključnega pomena, določajo namreč kompleksnost implementacije in do neke mere lahko tudi zagotavljajo pravilnost. Odločili smo se za jezik, ki je še precej bilzu Core, vendar ni tipiziran in je nekoliko poenostavljen. Vsak izraz pa ima dodano lastnost stanja ovrednotenosti, ki je lahko *Deferred* (odloženo ovrednoten) ali *Forced* (prisilno ovrednoten). V jeziku pa sta eksplicitna konstrukta *Force* in *Defer* za prehajanje med obema stanjema. V času izvajanja ta dva konstrukta ustrezata vrednotenju izrazov in ustvarjanju neovrednotenih izrazov. Tu pa sta vidna in preverjena dela jezika, kar zagotavlja večjo preglednost same kode prevajalnika in nam zagotavlja neko mero pravilnosti - ne moremo mešati izrazov v različnih stanjih.

Sledi komentirana predstavitev drevesne predstavitve vmesnega jezika.

```
type Name = String
```

Najprej definiramo tip imen. Imena so predstavljena kar z nizi in nimajo posebne strukture, zato lahko imena poenostavljamo že ob prevajanju iz Core v vmesno kodo, in se drugi fazi ni potrebno ukvarjati z različnimi oblikami imen. Nov tip pa je definiran zaradi berljivosti kode, uporaba preimenovanih tipov namreč omogoča pisanje kode, ki je sama sebi komentar.

```
data Literal = LiteralInteger Integer
             | LiteralRational Rational
             | LiteralChar Char
             | LiteralString String
```

Nato definiramo literale. Še vedno imamo enak nabor literalov kot v Core. Literal

je lahko celo število, racionalno število, znak ali niz.

```
data Evaluation = Deferred | Forced
```

Tip *Evaluation* ni namenjen rabi znotraj drevesa, ampak samo za čas prevajanja. Za to smo izkoristili razširitev jezika Haskell, poimenovano *DataKinds*, ki omogoča programerju, da definira svojo vrsto tipa s končno mnogo tipi te vrste. *Evaluation* je torej vrsta z dvema tipoma. Ta dva tipa pa nista naseljena in sta namenjena samo fantomski rabi - kot parameter, ki statično izraža lastnosti neke vrednosti, vrednosti tega tipa (ki jih tu celo ni) pa se v času izvajanja ne pojavljajo.

```
data Expression :: Evaluation -> * where
  Lit :: Literal -> Expression Forced
  Var :: Name -> Expression Deferred
  Lam :: Name -> (Expression Forced)
                -> Expression Forced
  App :: Expression Forced  -> Expression Deferred
           -> Expression Forced
  Let :: [Vdef] -> Expression Deferred
           -> Expression Deferred
  Case :: (Expression Deferred) -> Name -> [Alt]
            -> Expression Deferred
  Force :: Expression Deferred -> Expression Forced
  Defer :: Expression Forced -> Expression Deferred
```

Tip *Expression* je največji in daleč najbolj kompleksen, predstavlja pa srce jezika. Za njegovo definicijo potrebujemo dve razširitvi jezika *KindSignatures* (eksplicitni podpisi vrst) in *GADTs*(generalizirani algebraični podatkovni tipi).

V glavi definiramo *Expression* kot konstruktor tipov, ki sprejme en parameter, ki določa stanje tega izraza - naš tip *Evaluation*, nato pa vklopimo GADTs sintakso. Nadaljnje vrstice definiranjo konstruktorje z njihovimi podpisi, obravnavajo jih kot funkcije.

Tu lahko opazimo, da noben konstruktor ni nikjer polimorfičen v stanju vrednotenja. Povsod, kjer izraz gradimo iz podizrazov, eksplicitno zahtevamo, da je ta podizraz bodisi ovrednoten bodisi ne. Tudi vsi rezultati konstruktorjev imajo dobro definirano stanje vrednotenja. Ko staknemo ti dve lastnosti, dobimo skupek

omejitev pri gradnji izrazov. Če bomo uporabili neprimerno kombinacijo ovrednotenih in neovrednotenih izrazov, se bo to pokazalo kot napaka pri preverjanju tipov pri prevajanju samega prevajalnika. Tako lahko eliminiramo cel razred potencialnih semantičnih napak pri prevajanju z relativno malo dodatne sintakse, prva faza prevajalnika pa bo dokazano pravilna glede vrednotenja izrazov. Zastonj pa dobimo tudi dodatne omejitve. Lahko na primer napišemo funkcijo nad izrazi, ki je glede stanja vrednotenja agnostična, ampak ga ohranja. Takšna funkcija ima tip *Expression a* $\rightarrow$ *Expression a*, želeno lastnost pa pridelamo kot zastonj izrek [43].

Sedaj pa še o posameznih konstruktorjih. *Lit* preprosto ovije literal v ovrednoten izraz, *Var* pa ime (referenco) v neovrednoteni izraz, poimenovani izrazi v področju so namreč vedno neovrednoteni. *Lam* definira abstrakcijo (lambda izraz) tako, da sprejme ime in neovrednoteni izraz (telo) ter vrne ovrednoten izraz - literal funkcije. Aplikacija *App* operira nad takšnimi ovrednotenimi izrazi in poleg njih zapakira še en agrument, ki še ni ovrednoten; vse funkcije so v Curryjevi obliki. Naslednja konstrukta sta *Let* in *Case*. *Let* vzame seznam definicij vrednosti *Vdef* in končni izraz ter vse skupaj predstavi kot neovrednoteni izraz (znotraj enega *Let* so definicije lahko vzajemno rekurzivne), *Case* pa vzame neovrednoteni izraz(*Case* opravi vrednotenje), ime za ta izraz v svojem področju, spisek alternativ *Alt* in vrne neovrednoteni izraz.

Do tukaj je vmesni jezik precej analogen Core, le da doda eksplicitno stanje vrednotenja. Naslednja dva konstrukta pa izstopata. *Force* vzame neovrednoteni izraz in vrne ovrednoten izraz, je torej abstrakcija čez koncept vrednotenja izrazov, *Defer* pa je ravno dualen konstrukt.

```
data Alt = AltDefault (Expression Deferred)
         | AltLit Literal (Expression Deferred)
         | AltCon Name [Name] (Expression Deferred)
```

Tip *Alt* definira možne alternativne stavke za ujemanje vzorcev. *AltDefault* je privzeta alternativa, ki se izvede brez posebnih pogojev. *AltLit* predstavlja alternativo, ki preveri, ali je pregledovana vrednost enaka podanemu literalu, *AltCon* pa je alternativa za preverjanje algebraičnih tipov in predstavlja izbrani konstruktor. To je prvi paramter *Name*, sledeči seznam imen so imena na katera naj se vežejo posamezni argumenti iz konstruktorja tega tipa, in so lahko uporabljeni na desni

strani. Zadnji parameter je vseh alternativah neovrednoten izraz, ki predstavlja desno stran alternative.

```
data Vdef = Vdef Name (Expression Deferred)
```

Konstruktor *Vdef* uporabljamo za definicijo vrednosti v izrazih *Let*. Skupaj zapakira ime in neovrednoten izraz - desno stran, definiran pa je zgolj zaradi berljivosti kode in ločevanja nalog.

```
type Arity = Int
data Constructor = Constructor Name Arity
```

Vse potrebno znanje o enem konstruktorju algebraičnega podatkovnega tipa sta njegovo ime in številnost. Na podlagi tega lahko implementiramo njegov konstruktor in dekonstruktor (glej poglavje 5.4).

```
data Tdef = Data [Constructor]
          | Newtype Name
```

Definicija tipa *Tdef* je lahko definicija algebraičnega podatkovnega tipa, v tem primeru je to seznam nejgovih konstruktorjev, lahko pa je *newtype* - netransparentno ovijanje tipa. V slednjem primeru potrebujemo samo ime novega tipa, njegova številnost je vedno 1.

```
data Dependency = Dependency Name (Name, [Name], Name)
```

Preden lahko definiramo modul, potrebujemo še seznam njegovih odvisnosti. V Core so odvisnosti namreč implicitne - vrednosti preprosto uporabljamo s polnim imenom, v JavaScriptu pa bomo morali vsako odvisnost najprej uvoziti (poglavje 5.2). Odvisnosti najdemo preprosto s sprehodom čez drevo in beleženjem omenjenih polnih imen, potrebujemo pa konstrukt za predstavitev teh odvisnosti. To je tip *Dependency*, ki zabeleži interno ime odvisnosti ter podatke o lokaciji module: ime paketa, pot starševskih modulov in končno ime modula. Lokacija je razčlenjena, da lahko druga faza prevajanja nadzira razpostavitev datotek in tega ni potrebno početi že v prvi fazi.

```
data Module = Module Name [Dependency] [Tdef] [Vdef]
```

Zadnja definicija je *Module*, ki poveže vse skupaj in zgradi opis celotnega modula. Modul vsebuje svoje ime, seznam odvisnosti, seznam definicij tipov in seznam definicij vrednosti.

5.7 Pretvorba v vmesno kodo

Pretvorbo iz drevesne kode Core v drevesno vmesno kodo definirano v razdelku 5.6 implementira modul *Desugar*. Pretvorbo začne z modulom in se rekurzivno kliče na sestavnih delih. Za vsak dosegljiv tip vozlišča v drevesu definira funkcijo, ki nad tem vozliščem primerja vzorce.

Iz ustreznega vzorca izlušči potrebne informacije (odstrani tipe) in doda potrebne informacije o stanju ovrednotenosti (glej razdelek 5.3). Potrebna poddrevesa se prav tako pretvorijo v ustrezno obliko z rekurzivnim klicem in nato zapakirajo v ustrezen konstruktor. Tukaj nam statični tipi in dobro specificirana vmesna koda zagotavljajo, da smo ustrezno pretvorili vse potrebne sestavne dele. To je posebej pomembno pri izrazih, saj so ti lahko sestavljeni iz podizrazov. Posebnost so izrazi prelivanja, saj nimajo ustreznice v vmesni kodi. Ker pa naša vmesna koda nima tipov, lahko prelive odstranimo in jih nadomestimo kar z izrazi v njih[2].

Še zadnja ovira so imena. Core uporablja kvalificirana imena in podaja polno strukturo poti do modulov. V JavaScriptu pa se želimo temu izogniti in implementirati plitvo gnezdenje. Na srečo GHC uporablja "enkodiranje Z", ki pretvori vse veljavne posebne znake v alfanumerike. Tako lahko sestavljene poti združimo z podčrtaji, ki so veljavni znaki v imenih v JavaScriptu. Kvalificirana imena predstavimo kot *polno_ime_modula.vrednost*, pri čemer vsem modulom dodamo predpono __, da se izognemo trkom z lokalnimi spremenljivkami.

5.8 Poenostavljanje

Dodatna faza poenostavljanja vmesne kode lahko pomeni drastično hitrejše izvajanje prevedenega programa, v tem primeru pa prinese tudi precej elegantnejšo implementacijo.

Eksplicitna konstrukta v vmesni kodi za gradnjo neovrednotenih izrazov in prisiljeno vrednotenje poenostavita generiranje vmesne kode. Namesto njiju bi lahko implementirali funkcijo, ki generira vmesno kodo, a način pri katerem to kodo generiramo šele ob prevajanju v ciljni jezik, se izkaže za bolj učinkovitega.

V primeru naivnega generiranja vmesne kode bomo ustvarili veliko odvečnih

[2]Preliv je sestavljen iz izraza in tipa v katerega se preliva ter dokaza, da bo preliv uspel.

parov neovrednotenih izrazov in vrednotenja, na primer pri klicu funkcij z več parametri. V primeru eksplicitnih konstruktov je te kombinacije konstruktov trivialno odkrivati s uporabo ujemanja vzorcev. Nato pa najdene pare iz vmesne kode odstranimo in tako pohitrimo program. Če te faze ne bi imeli, bi bilo potrebno za dosego istega cilja bistveno več kompleksnosti v generiranju vmesne kode, saj bi morali beležiti kontekst v katerem smo in primerno preklapljati med predstavitvama izrazov.

Naš poenostavljalnik implementira dve konkretni preslikavi, ki odstranita nepotrebne pare.

```
Force (Defer a) -> a
Defer (Force a) -> a
```

Ostale elemente drevesne kode pa ustrezno obravnava tako, da se rekurzivno sprehodi po celotnem drevesu abstraktne sintakse in najde vse nepotrebne pare.

5.9 Sistem izvajanja (runtime)

Določenih funkcionalnosti ni možno implementirati v samem jeziku, drugih pa ne želimo zaradi velikosti generirane kode, zato potrebujemo sistem izvajanja, ki ga povežemo s prevedeno kodo.

5.9.1 mkthunk

Najbolj uporabljana funkcija v sistemu izvajanja je *mkthunk*, ki spremeni "klic po imenu" v "klic po potrebi". Ta funkcija je v sistemu izvajanja, ker bi zaradi pogoste uporabe drastično povečala velikost generirane kode, če bi jo prevajalnik vedno vrinjal. Za podrobnosti implementacije glej razdelek 5.3.

5.9.2 Aritmetika

Jezik Core je enostaven tudi zato, ker ne definira aritmetike. Zahteva samo možnost definicije številskih literalov v poljubni natančnosti. Aritmetika v JavaScriptu pa trpi za začetnimi odločitvami o zasnovi jezika. JavaScript ima samo en številski tip, 64-bitna števila v plavajoči vejici po standardu IEE754. Cela

števila so implementirana z uporabo mantise, in tako omogočajo hranjenje samo do 52-bitnih vrednosti. Zato se je pojavilo več knjižnic za delo z aritmetiko v poljubni natančnosti. Uporabili smo knjižnico *big.js*[30], ki implementira aritmetiko, za same literale pa uporabili predstavitev z nizi.

Še vedno lahko dodamo aritmetiko v fiksni natančnosti. V JavaScriptu moramo implementirati pretvorbo iz literala v želen tip in nato definirati funkcije, ki izpostavijo operacije nad tem tipom, in njihove tipe uvoziti v modul, ki želi uporabljati takšno aritmetiko.

5.9.3 Vhodno-izhodne operacije

Vhodno izhodne operacije so skrite za tipom *IO*, ki predstavlja opise akcij. Primitivne operacije s stranskimi učinki implementiramo neposredno v JavaScriptu in pri tem samo pazimo, da bodo ustrezale tipu *IO* (glej razdelek 3.6.1). Za ostalo lahko poskrbimo v jeziku.

Definiramo lahko pomožno funkcijo, ki nam JavaScript funkcije brez argumentov adaptira na ustrezen tip

```
var primitiveIO = function(f) {
  return function (world) {
      return [0, world, f()];
  };
};
```

Njena oblika je enaka, kot jo pričakuje prevedena koda iz tipa IO, zaradi enostavnosti pa smo vrinili tudi kreacijo para stanja sveta in izhodne vrednosti. Par je namreč algebraični tip (glej razdelek 5.4), ki je produkt dveh vrednosti, zato je njegova predstavitev v času izvajanja tabela z ničlo na indeksu 0 (ker je to prvi in edini konstruktor) in vrednostma na naslednjih dveh indeksih.

5.9.4 Interakcija z JavaScriptom

Za uporabo v JavaScript okolju je potrebna integracija v obstoječo kodo, to pa pomeni interakcijo z JavaScriptom. Klicanje prevedene kode iz JavaScripta samo po sebi ne predstavlja nobenega problema. Struktura modulov se ohranja, nekoliko težavno je le pravilno vrednotenje izrazov (dodatni klici funkcij na pravih mestih)

in za JavaScript neidiomatska raba funkcij v Curryjevi obliki, ki so sicer dobro podprte.

Nekoliko bolj problematično je klicanje obstoječe JavaScript kode iz prevedene kode. Prva težava je neobstoj statičnih tipov za JavaScript, druga pa morebitna prisotnost stranskih učinkov. JavaScript uporablja klic po vrednosti in drugačne tipe. Zato potrebujemo vmesni korak. Najbolj enostaven in fleksibilen način je, da v Core izpostavimo zmogljivosti *eval* funkcije. JavaScript namreč podpira dinamično generiranje in interpretiranje kode iz nizov. Morebitne učinke pa naredimo varne tako, da rezultat v vsakem primeru ovijemo v *IO*. Odprto vprašanje je le še predstavitev rezultata. Rezultate bi lahko serializirali v JSON in jih podajali nazaj prek nizov, ampak na ta račun izgubimo veliko fleksibilnosti; ne moremo namreč serializirati zaprtij in rekurzivnih struktur. Ker pa tipov ob interakciji tako ali tako ne moremo preverjati, lahko brez žrtvovanja varnosti naredimo tip rezultata polimorfičen. Tako nam bo prevajalnik omogočil poljubno rabo, naloga programerja pa je, da rezultat pravilno uporabi.

Prevajalnik se mora zavedati definicije tipa

```
callJs :: forall a . String -> IO a
```

Na strani sistema izvajanja pa je implementacija trivialna.

```
exports.callJs = exports.mkthunk(function() {
  return function(js) {
    return primitiveIO(function(){
      return eval(js);
    });
  };
});
```

Poglavje 6

Rezultati

V nadaljevanju predstavljamo praktično delovanje implementiranega prevajalnika – izvorno kodo in njen prevod, kot je ustvarjen s prevajalnikom. Vsak primer ima najprej izvorno kodo v idiomatskem JavaScriptu in njen ekvivalent v Haskellu, nato njen prevod v Core z GHC in na koncu še JavaScript, generiran z našim prevajalnikom. Ta je direktno izvršljiv na Node.js, z uporabo orodja Browserify pa lahko module združimo v eno datoteko za uporabo v spletnih brskalnikih. Predpostavljamo, da v idiomatskem JavaScriptu ne uporabljamo ne-striktnega vrednotenja in funkcij v Curryjevi obliki. Za predstavitev algebraičnih tipov pa smo uporabili tabelo, čeprav bi človek ob pisanju JavaScripta najbrž uporabil objekt. Takšna predstavitev ohranja vsaj neko mero združljivosti z generirano kodo.

6.1 Definicija in aplikacija funkcij

Najbolj enostaven primer implementira enojno in dvojno aplikacijo funkcije na parameter. Najprej ročno spisana JavaScript koda

```
exports.once = function(f, x) {
  return f(x);
};
exports.twice = function(f, x) {
  return f(f(x));
};
```

in njen ekvivalent v Haskellu

```
module Apply where
once f x = f x
twice f x = f (f x)
```

ki ga z prevajalnikom GHC prevedemo v Core,

```
%module main:Twice
  main:Twice.once :: %forall taHT t1aHU .
    (t1aHU -> taHT) -> t1aHU -> taHT =
    \ @ taHW @ t1aHX
    (fapx::t1aHX -> taHW) (xapy::t1aHX) -> fapx xapy;
  main:Twice.twice ::
    %forall taHF . (taHF -> taHF) -> taHF -> taHF =
    \ @ taHH (fapzz::taHH -> taHH) (xapA::taHH) ->
      fapzz (fapzz xapA);
```

ki je nato z našim prevajalnikom preveden v JavaScript.

```
var __main__Twice = exports;
var ___runtime = require("../runtime/runtime.js");
__main__Twice.once = ___runtime.mkthunk(function () {
   return function (fapx) {
      return function (xapy) {
         return fapx()(xapy);
      };
   };
});
__main__Twice.twice = ___runtime.mkthunk(function () {
   return function (fapzz) {
      return function (xapA) {
          return fapzz()(___runtime.mkthunk(
            function () {
              return fapzz()(xapA);
            }
          ));
```

```
        };
    };
});
```

Ta prevod je precej blizu ročno pisane kode, doda le ne-striktno vrednotenje in funkcije pretvori v Curryjevo obliko.

6.2 Podatkovni tipi in primerjanje vzorcev

Drugi primer prikazuje uporabo algebraičnih tipov vključno s konstrukcijo in primerjanjem vzorcev. V ročno pisanem JavaScriptu v primeru tipa vsote običajno razlikujemo med alternativami šele na mestu uporabe, zato dekonstruktorjev nismo definirali.

```
exports.Foo = function(a1, n, a2) {
  return [0, a1, n, a2];
};
exports.Bar = function(f) {
  return [1, f];
};
exports.Baz = function(foo) {
  return foo;
};
exports.Baz.unapply = function(foo) {
  return foo;
};
exports.Tr = [0];
exports.Fl = [1];
exports.foo2bool = function(foo) {
  if (foo[0]) {
    return exports.Tr;
  } else {
    return exports.Fl;
  }
}
```

V Haskellu dekonstruktorje dobimo avtomatsko ob definiciji.

```
module Foo where

data Foo a = Foo a Int a | Bar Float
newtype Baz = Baz (Foo String)

data Bl = Tr | Fl

foo2bool (Foo _ _ _) = Tr
foo2bool (Bar _) = Fl
```

Z GHC prevedemo ta Haskell v Core

```
%module main:Foo
  %data main:Foo.Foo aapE =
    {main:Foo.Foo aapE ghczmprim:GHCziTypes.Int aapE;
     main:Foo.Bar ghczmprim:GHCziTypes.Float};
  %newtype main:Foo.Bazz main:Foo.NTCoZCBazz
    = main:Foo.Foo (ghczmprim:GHCziTypes.ZMZN
                    ghczmprim:GHCziTypes.Char);
  %data main:Foo.Bl =
    {main:Foo.Tr;
     main:Foo.Fl};
  main:Foo.foo2bool :: %forall taIn .
    (main:Foo.Foo taIn) -> main:Foo.Bl =
    \ @ taIp (dsdIy::main:Foo.Foo taIp) ->
        %case main:Foo.Bl dsdIy %of
        (wildX4::main:Foo.Foo taIp)
          {main:Foo.Foo
           (ds1dIzz::taIp)
           (ds2dIA::ghczmprim:GHCziTypes.Int)
           (ds3dIB::taIp) ->
             main:Foo.Tr;
           main:Foo.Bar
             (ds1dIC::ghczmprim:GHCziTypes.Float) ->
```

```
                main:Foo.F1};
```

in nato z našim prevajalnikom v JavaScript.

```
var __main__Foo = exports;
var ___runtime = require("../runtime/runtime.js");
var __main__Foo = require("../main/Foo.js");
__main__Foo.Foo = ___runtime.mkthunk(function () {
   return function (arg3) {
      return function (arg2) {
         return function (arg1) {
            return [0,arg3,arg2,arg1];
         };
      };
   };
});
__main__Foo.Foo.unapply = function (data,
f) {
   data = data();
   if (data[0] == 0)
   return f.apply(undefined,
      data.slice(1));
};
__main__Foo.Bar = ___runtime.mkthunk(function () {
   return function (arg1) {
      return [1,arg1];
   };
});
__main__Foo.Bar.unapply = function (data,
f) {
   data = data();
   if (data[0] == 1)
   return f.apply(undefined,
      data.slice(1));
};
```

```
__main__Foo.Bazz = ___runtime.mkthunk(function () {
   return function (x) {
      return x;
   };
});
__main__Foo.Bazz.unapply = ___runtime.mkthunk(
   function () {
     return function (x,f) {
        return f(x);
     };
});
__main__Foo.Tr = ___runtime.mkthunk(function () {
   return [0];
});
__main__Foo.Tr.unapply = function (data,
f) {
   data = data();
   if (data[0] == 0)
   return f.apply(undefined,
      data.slice(1));
};
__main__Foo.Fl = ___runtime.mkthunk(function () {
   return [1];
});
__main__Foo.Fl.unapply = function (data,
f) {
   data = data();
   if (data[0] == 1)
   return f.apply(undefined,
      data.slice(1));
};
__main__Foo.foo2bool = ___runtime.mkthunk(function () {
   return function (dsdIy) {
```

```
        return function () {
            var wildX4 = dsdIy;
            return __main__Foo.Foo.unapply(wildX4,
            function (ds1dIzz,
            ds2dIA,
            ds3dIB) {
                return __main__Foo.Tr;
            }) || (__main__Foo.Bar.unapply(wildX4,
            function (ds1dIC) {
                return __main__Foo.Fl;
            }) || undefined);
        }()();
    };
});
```

Konstruktorji so še vedno blizu ročno pisane kode, dekonstruktorji pa so nov element, ki naj bi poenostavil pisanje primerjanja vzorcev. V funkciji *foo2bool* pa vidimo, da temu ni tako. Implementacija te funkcije je precej zapletena in ima več nivojev kot ročno napisana alternativa. Edina prednost prevedene kode je, da loči podatke o predstavitvi od imena tipa.

6.3 V/I operacije

Kot zelo enostaven primer, ki izvaja vhodno-izhodne operacije si poglejmo variacijo na klasičen program "Pozdravljen svet". Uporabili smo izpis posameznih znakov, da lahko demonstriramo zaporedno vezavo operacij.

V čistem JavaScriptu je to zaporedje stavkov.

```
exports.main = function() {
  putChar('h');
  putChar('a');
  putChar('i');
};
```

V Haskellu prav tako, ampak znotraj bloka *do*.

```
module Hai where
main = do
  putChar 'h'
  putChar 'a'
  putChar 'i'
```

Ta program se ob prevodu v Core precej razpihne, operacije vezave so namreč postale eksplicitne (*GHCziBase.zgzg*), prav tako kot izbira implementacije monade (*zdfMonadIO*).

```
%module main:Hai
 main:Hai.main ::
  ghczmprim:GHCziTypes.IO ghczmprim:GHCziTuple.Z0T =
  base:GHCziBase.zgzg @ ghczmprim:GHCziTypes.IO
  base:GHCziBase.zdfMonadIO @ ghczmprim:GHCziTuple.Z0T
 @ ghczmprim:GHCziTuple.Z0T
  (base:SystemziIO.putChar
   (ghczmprim:GHCziTypes.Czh
    ('h'::ghczmprim:GHCziPrim.Charzh)))
  (base:GHCziBase.zgzg @ ghczmprim:GHCziTypes.IO
   base:GHCziBase.zdfMonadIO @ ghczmprim:GHCziTuple.Z0T
  @ ghczmprim:GHCziTuple.Z0T
   (base:SystemziIO.putChar
    (ghczmprim:GHCziTypes.Czh
     ('a'::ghczmprim:GHCziPrim.Charzh)))
   (base:SystemziIO.putChar
    (ghczmprim:GHCziTypes.Czh
     ('i'::ghczmprim:GHCziPrim.Charzh))));
```

Prevod z našim prevajalnikom v JavaScript da sledeči rezultat. Odstranili smo tipe, a zaradi eksplicitnega ustvarjanja in vrednotenja izrazov je program še nekoliko daljši. Še vedno pa ohranja okvirno obliko in semantiko.

```
var __main__Hai = exports;
var ___runtime = require("../runtime/runtime.js");
var __base__GHC_Base = require("../base/GHC_Base.js");
```

```
var __base__System_IO =
 require("../base/System_IO.js");
var __ghczmprim__GHC_Types =
 require("../ghczmprim/GHC_Types.js");
__main__Hai.main = ___runtime.mkthunk(function () {
 return __base__GHC_Base.zgzg()
        (__base__GHC_Base.zdfMonadIO)
        (___runtime.mkthunk(function () {
  return __base__System_IO.putChar()(
   ___runtime.mkthunk(function () {
    return __ghczmprim__GHC_Types.Czh()(
      ___runtime.mkthunk(function () {
        return "h";
      })
    );
  }));
 }))(___runtime.mkthunk(function () {
  return __base__GHC_Base.zgzg()
         (__base__GHC_Base.zdfMonadIO)
         (___runtime.mkthunk(function () {
    return __base__System_IO.putChar()(
     ___runtime.mkthunk(function () {
      return __ghczmprim__GHC_Types.Czh()
        (___runtime.mkthunk(function () {
          return "a";
      }));
     }));
  }))(___runtime.mkthunk(function () {
    return __base__System_IO.putChar()
     (___runtime.mkthunk(function () {
      return __ghczmprim__GHC_Types.Czh()
           (___runtime.mkthunk(function () {
              return "i";
```

```
                }));
        }));
    }));
  }));
});
```

V tem primeru je prevod očitno slabši od ročno napisane kode. Težava je v eksplicitni zaporedni vezavi stavkov, ki jo v JavaScriptu lahko dosežemo tudi implicitno z zaporednim zapisom. Prevajanje kode, ki uporablja monado *IO*, bi potrebovalo specifične optimizacije, da bi se rezultat lahko kosal z ročno napisano kodo.

Poglavje 7

Sklepne ugotovitve

Prevajalnik, ki je nastal iz implementacije, opisane v prejšnjem poglavju, ima sicer minimalen tekstovni uporabniški vmesnik, a je končna velikost izvedljive datoteke (samega prevajalnika) zato zgolj 800kb, odvisen pa je le od nekaj standardnih deljenih knjižnic. Hitrost prevajanja je sorazmerno dobra: približno 15000 vrstic vhodnega programa na sekundo. Samo prevajanje prevajanje pa je dovolj blizu standardu, da uspešno prevede nemodificirano izvorno kodo iz standardne knjižnice GHC.

7.1 Nadaljnje delo

Prevajalnik sam je sicer zaključen, a pušča veliko odprtih priložnosti za izboljšave in dodatne zmogljivosti. V trenutnem stanju žal še ni uporaben za resno delo in razvijanje uporabnih projektov, temveč zgolj za eksperimentiranje.

7.1.1 Optimizacija repnih klicev

Optimizacija repnih klicev je nujno potrebna v vsakem jeziku brez zank, saj programer nima drugega mehanizma za izvajanje ponovljivega izračuna poljubne globine. Primer repnega klica najdemo recimo v rekurzivni funkciji za izračun fakultete z eksplicitnim akumulatorjem

```
factAcc acc 1 = acc
factAcc acc n = factAcc (n*acc) (n-1)
```

Rekurzivni korak je tukaj repni klic, vse kar naredi je, da vrne rezultat klica druge funkcije. Naivna implementacija bo za nov klic uporabila nov okvir na skladu [1], počakala na vračilo funkcije in nato sama vrnila isti rezultat. A to prinaša omejitve z globino rekurzije, česar pa si ne želimo, saj nimamo zank in jih moramo implementirati z rekurzijo. Želimo si generirati kodo, analogno sledečemu Cju.

```
Int factAcc(int acc, int n) {
    LOOP:
    if (n == 1)
        return acc;
    acc *= n--;
    goto LOOP;
}
```

Znotraj GHC-ja to nalogo opravlja vsak zadnji del posebej. Jezik C--, ki služi kot imperativna vmesna koda, zagotavlja optimizacijo repnih klicev, implementacija tega pa leži je del prevajanja v LLVM vmesno kodo oziroma prevajanja C-- naravnost v zbirnik.

Žal (ali na srečo) pa JavaScript ne podpira stavkov GOTO. Obstajajo sicer projekti, ki jih implementirajo [37], vendar s precej omejitvami in dodatnim procesiranjem izvorne kode. Rekurzivne repne klice lahko sicer implementiramo z zankami, a to se ne generalizira na repne klice drugih funkcij, npr. pri vzajemni rekurziji. Za splošno implementacijo potrebujemo čiste brezpogojne skoke; najprej moramo sprostiti svoj okvir, nato pa skočiti in pustiti vrnitveni naslov enak. Ta zelo nizkonivojska operacija pa ni možna v JavaScriptu. Funkcijsko programiranje pa nam ponuja rešitev kako to simulirati s konstrukti na višjem nivoju: s trampolini [18].

Trampolin je zanka, znotraj katere izvajamo vse funkcijske klice. Alocira ekspliciten sklad in pokliče začetno funkcijo. V primeru repnega klica ta funkcija potisne argumente za sklad in *vrne kazalec na novo funkcijo*. Trampolin spremeni trenutno funkcijo v vrnjeno funkcijo in jo pokliče - naredi nov obhod zanke. Ko pridemo do končnega rezultata, ki ni klic funkcije, rezultat preprosto potisnemo

[1] V tem primeru ne-striktnega vrednotenja izrazov se samo ustvari neovrednoten izraz, a problem se pojavi pri vrednotenju izraza, ki je sedaj rekurziven. Zaradi enostavnosti ta korak izpustimo in govorimo kar o okvirjih.

na sklad in vrnemo ničelni kazalec. Ker tega kazalca trampolin ne more poklicati, to vzame za signal konca in vrne rezultat. Kaj pa nerepni klici? Opravimo lahko CPS transformacijo in se takšnih klicev znebimo ali pa za nerepne inicializiramo nov trampolin. Ta bo uporabljal isti sklad, uporabil pa bo tudi okvir sistemskega sklada. To vprašanje je pri našem prevajalniku še odprto.

7.1.2 Integracija s Cabal/GHC

Prevajalnik v trenutni obliki zna prevajati en modul (eno datoteko naenkrat). V resnih projektih želimo prevajati celoten sistem (več modulov) naenkrat, prav tako želimo definirati zunanje odvisnosti in jih integrirati v prevajanje.

Ker je, kot že omenjeno, dolgoročni cilj projekta prevajanje Haskella v JavaScript, je naš namen integracija s prevajalnikom GHC, ki lahko ob prevajanju izstavi tudi vmesni Core. Integracijo lahko opravimo brez posegov v GHC. Implementirati je potrebno le vtičnik za prevajalno ogrodje Cabal, ki je de facto standard za večje Haskell projekte. Ta vtičnik GHCju poda ustrezne zastavice za generiranje datotek s Core kodo in se registrira v cevovod prevajanja za obdelavo modulov tako, da prevede ustrezni Core. Cabal ponuja tudi definicije zunanjih odvisnosti, ki jih namesti iz izvorne kode. Naš vtičnik se lahko vključi tudi v ta proces in tako prevede zunanje odvisnosti. Dodamo lahko še dodatni korak, ki nastalo JavaScript kodo ustrezno zapakira, in tako k obstoječemu prevajalniku dodamo novo tarčo.

7.1.3 Sistem izvajanja

Za uporabno integracijo z GHCjem pa je potreben še sistem izvajanja. GHC namreč precej zmogljivosti implementira izven jezika, v knjižnici napisani v Cju, ki je povezana z vsakim prevedenim projektom. Minimalen sistem izvajanja smo sicer razvili (poglavje 5.9), a GHC implementira še veliko drugih primitivnih operacij. Za prevajanje obstoječe Haskell kode je potrebno implementirati vsaj dovolj veliko podmnožico tega programskega vmesnika, da se prevedejo bolj popularni paketi.

Vsi mehanizmi so na mestu, pripravili smo tudi nekaj primitivnih funkcij, ta del je še v nastajanju le zaradi obsega dela in pomanjkanja časa.

7.2 Zaključek

V diplomskem delu smo razvili prevajalnik iz jezika Haskell Core v JavaScript. Na začetku smo podrobneje opisali tako Core kot tudi Haskell, saj sta jezika močno povezana. Ogledali smo si njune posebnosti in semantiko ter pri jeziku Core podrobneje tudi nekatere konstrukte. V nadaljevanju smo opisali JavaScript in njegovo semantiko kot kontrast jeziku, ki ga prevajamo.

Opisali smo strukturo samega prevajalnika, nato pa podrobneje posamezne konstrukte, izzive pri njihovem prevajanju in rešitve. Veliko smo se posvetili pravilni obravnavi ne-striktnosti in sami pravilnosti prevajanja. Na koncu smo predstavili še rezultate prevajanja z razvitim prevajalnikom.

Uspeli smo implementirati polno delujoč prevajalnik in minimalen sistem izvajanja kot zastavljeno. Naša implementacija pušča veliko odprtega prostora za nadgradnje in izboljšave: nekaj sprememb v samem prevajalniku, predvsem pa v polnosti implementacije sistema izvajanja in standardne knjižnice, ki je ključnega pomena za prevajanje večjih, resnih programov. Prav tako so še odprte možnosti testiranja in izboljševanja hitrosti izvajanja prevedenih programov.

Izvajanje JavaScripta v spletnem brskalniku skupaj z uporabo našega prevajalnika odpira vrata celotnemu skladu razvoja (full-stack development) spletnih aplikacij v Haskellu. To programerju prinaša močna orodja in znane koncepte, ki jih lahko sedaj uporabi tudi na klientu. To je mogoče tudi z nekaterimi drugimi prevajalniki/jeziki, z našim pa lahko tudi deli kodo med strežniškim delom in klientom (pod pogojem, da sta oba napisana v Haskellu). To zmanjša obremenitev, saj odstrani nepotrebne dvojne definicije podatkov, prav tako pa razširit preverjanje tipov, da zajema celotno aplikacijo. Takšen pristop prinaša hiter razvoj aplikacij, z manj izvorne kode in manj napakami, vse tri točke pa imajo neposredne ugodne ekonomske prednosti.

Literatura

[1] Browserify. http://browserify.org/, year = 2010, note =.

[2] Clean. http://clean.cs.ru.nl/Clean, 1986. [dostopano 14.9.2014].

[3] Haskell. http://www.haskell.org/, 1991. [dostopano 14.9.2014].

[4] The glasgow haskell compiler. http://www.haskell.org/ghc/, 1992. [dostopano 14.9.2014].

[5] OCaml. http://ocaml.org/, 1996. [dostopano 14.9.2014].

[6] JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript, 2005. [dostopano 14.9.2014].

[7] Google V8. https://code.google.com/p/v8/, 2008. [dostopano 15.9.2014].

[8] Node.js. http://www.nodejs.org/, 2009. [dostopano 15.9.2014].

[9] GHCJS. https://github.com/ghcjs/ghcjs, 2010. [dostopano 14.9.2014].

[10] Idris. http://www.idris-lang.org/, 2011. [dostopano 14.9.2014].

[11] Elm. http://elm-lang.org/, 2012. [dostopano 14.9.2014].

[12] Fay. https://github.com/faylang/fay, 2012. [dostopano 14.9.2014].

[13] Haste. http://haste-lang.org/, 2012. [dostopano 12.9.2014].

[14] PureScript. http://www.purescript.org/, 2012. [dostopano 14.9.2014].

[15] Implementacija tipa Maybe. http://hackage.haskell.org/package/base-4.7.0.1/docs/src/Data-Maybe.html, 2014. [dostopano 15.9.2014].

[16] K Gostelow Arvind and Wil Plouffe. The id report: An asynchronous programming language and computing machine. *Department of Information and Computer Science, University of California, Irvine, California*, 1978.

[17] Lennart Augustsson. A compiler for lazy ml. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227. ACM, 1984.

[18] Henry G Baker. Cons should not cons its arguments, part ii: Cheney on the mta. *ACM Sigplan Notices*, 30(9):17–20, 1995.

[19] Andrey Chudnov, Arjun Guha, Spiridon Aristides Eliopoulos, Joe Gibbs Politz, and Claudiu Saftoiu. language-ecmascript. `http://hackage.haskell.org/package/language-ecmascript`, 2012. [dostopano 16.9.2014].

[20] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, pages 345–363, 1936.

[21] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940.

[22] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[23] David Hilbert and Wilhelm Ackermann. Grundzüge der theoretischen logik. *Berlin, Heidelberg*, 1928.

[24] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, pages 29–60, 1969.

[25] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[26] Paul Hudak, Philip Wadler, Simon Peyton Jones, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.0. 1990.

[27] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of functional programming*, 2(02):127–202, 1992.

[28] Simon L Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In *Programming Languages and Systems—ESOP'96*, pages 18–44. Springer, 1996.

[29] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C-—: A portable assembly language that supports garbage collection. In *Principles and Practice of Declarative Programming*, pages 1–28. Springer, 1999.

[30] Michael M. big.js. `https://github.com/MikeMcl/big.js/`, 2012. [dostopano 20.8.2014].

[31] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *ACM SIGPLAN Notices*, volume 39, pages 4–15. ACM, 2004.

[32] John McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978.

[33] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[34] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[35] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[36] Charles Severance. Java script: Designing a language in 10 days. *Computer*, 45(2):0007–8, 2012.

[37] Alex Sexton. Summer of GOTO. `http://summerofgoto.com/`, 2009. [dostopano 25.8.2014].

[38] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[39] David A Terei and Manuel MT Chakravarty. An llvm backend for ghc. In *ACM Sigplan Notices*, volume 45, pages 109–120. ACM, 2010.

[40] Andrew Tolmach, Tim Chevalier, GHC Team, et al. An external representation for the ghc core language. *URL citeseer. ist. psu. edu/tolmach01external. html*, 2001.

[41] Andrew Tolmach, Tim Chevalier, GHC Team, et al. extcore. `http://hackage.haskell.org/package/extcore`, 2013. [dostopano 28.7.2014].

[42] David A Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional programming languages and computer architecture*, pages 1–16. Springer, 1985.

[43] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.

[44] Philip Wadler, Quentin Miller, and M Raskovsky. An introduction to orwell. *Computing Laboratory, University of Oxford, Oxford*, 1985.

[45] Stuart C Wray and Jon Fairbairn. Non-strict languages–programming and implementation. *The computer journal*, 32(2):142–151, 1989.

[46] Brent Yorgey. The typeclassopedia. *The Monad. Reader Issue 13*, page 17, 2009.