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

Timotej Lazar

Programiranje z algebrajskimi učinki

diplomsko delo
na interdisciplinarnem univerzitetnem študiju

Mentor: izred. prof. dr. Andrej Bauer

Ljubljana, 2012

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00036/2012
Datum: 03.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **TIMOTEJ LAZAR**

Naslov: **PROGRAMIRANJE Z ALGEBRAJSKIMI UČINKI**
PROGRAMMING WITH ALGEBRAIC EFFECTS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Računske učinke v teoriji programskih jezikov obravnavamo na različne načine. Najnovejši od njih so t.i. algebrajski učinki, v katerih so računski učinki operacije v algebri za dano signaturo. Homomorfizmi med algebrami ustrezajo splošnim prestreznikom, kot so prestrezniki za izjeme, preusmerjanje vhoda in izhoda in razne oblike sestopanja.
Andrej Bauer in Matija Pretnar sta razvila programski jezik Eff, ki je zasnovan na ideji algebrajskih učinkov in prestreznikov. V diplomi predstavite Eff, nato pa njegovo implementacijo nadgradite s preverjanjem izčrpnosti v prirejanju in obravnavi primerov.

Mentor:
prof. dr. Andrej Bauer

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:
prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:
akad. prof. dr. Franc Forstnarič

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 6

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Timotej Lazar, z vpisno številko **63070015**, sem avtor diplomskega dela z naslovom:

Programiranje z algebrajskimi učinki

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izred. prof. dr. Andreja Bauerja,
- 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 v zbirki "Dela FRI".

V Ljubljani, dne 3. 7. 2011 Podpis avtorja:

Kazalo

Povzetek

Teorija programskih jezikov se med drugim ukvarja tudi z računskimi učinki, kot so vhod/izhod, pomnilnik, izjeme in nedeterminizem. Te lahko obravnavamo na različne načine, nov pristop pa uporablja programski jezik eff, ki za njihovo predstavitev uporablja algebrajske teorije. Učinki so v njem prvorazredni objekti, kar omogoča nove načine reševanja raznih problemov. V prvem delu te diplomske naloge najprej predstavimo osnove effa, nato pa skozi primere razložimo delovanje učinkov in prestreznikov ter prikažemo nekaj možnosti, ki nam jih ponuja programiranje z njimi.

Eff si številne značilnosti deli z OCamlom in ostalimi jeziki družine ML. Ena izmed njih je uporaba vzorcev, s katerimi lahko vrednosti razstavljamo glede na njihovo strukturo. Obravnava primerov je nadzorna struktura, ki na podlagi seznama vzorcev in podane vrednosti določi, kako naj se izvajanje programa nadaljuje. Pri tem poišče prvi vzorec v seznamu, s katerim se ta vrednost ujema; pravimo, da je obravnava primerov izčrpna, če se vsaka možna vrednost ujema z vsaj enim vzorcem. V nasprotnem primeru pride ob neujemajoči se vrednosti do napake med izvajanjem. Praktični cilj tega dela je bil implementirati algoritem za preverjanje izčrpnosti. Ta programerja opozori, če je izpustil kakšno možnost, poleg tega pa izpiše tudi primer nepokrite vrednosti. V drugem delu predstavimo uporabljen algoritem in njegovo implementacijo, opišemo pa tudi njegove omejitve in nekaj možnosti za izboljšave.

Ključne besede

računski učinki, eff, prirejanje vzorcev

Abstract

One of the concepts that programming language theory deals with are computational effects such as input/output, state, exceptions and nondeterminism. They can be represented in various ways. A novel approach is used by the programming language eff where they are modeled using algebraic theories. Effects are first-class objects in eff which gives us new ways of solving various problems. In the first part of this thesis the basic language is presented, followed by a number of examples demonstrating effects and handlers, and the various possibilities offered by this new approach.

Eff is in many ways similar to OCaml and other languages in the ML family. One example is the support for patterns, which are used to match and decompose values. Pattern matching is used in case analysis, which is a control structure that uses a list of patterns to determine a branch of program execution according to the input value. The branch corresponding to the first pattern matching that value is selected; the analysis is said to be exhaustive if every possible value is matched by at least one pattern. If a given input value does not match any pattern, a runtime error occurs. The practical goal of this thesis was to implement an algorithm to check case analyses for exhaustiveness. The idea is to warn programmers about possible inexhaustive matchings and also produce an example of an unmatched value. The algorithm and its implementation are described in the second part of the thesis. Finally, the limitations and some possible improvements of our work are stated.

Keywords

computational effects, eff, pattern matching

Poglavje 1

Uvod

Teorija programskih jezikov se med drugim ukvarja tudi z računskimi učinki. To so konstrukti v programskih jezikih, ki lahko povzročijo, da dva klica funkcije z enakimi parametri vrneta različni vrednosti. Primeri učinkov so komunikacija z uporabnikom in drugimi programi, pomnilnik, izjeme in nedeterministične operacije. Imperativni jeziki temeljijo na uporabi učinkov, zato so v takšni ali drugačni obliki prisotni v skoraj vseh delih programa. Nasprotno funkcijski jeziki stremijo k temu, da je učinkov malo in da so jasno ločeni od čistega dela programa, ki ne vsebuje učinkov.

Obstaja več načinov, na katere lahko v programskem jeziku obravnavamo učinke. Dobro uveljavljen primer so monade, kot jih uporablja Haskell. Novejši pristop uporablja t.i. algebrajske učinke, v katerih so računski učinki predstavljeni z operacijami primerne algebrajske teorije. Andrej Bauer in Matija Pretnar sta razvila programski jezik eff [2], ki je zasnovan na ideji algebrajskih učinkov. V njem lahko definiramo nove učinke, ki opisujejo določene operacije, in jih obravnavamo s prestrezniki (angl. *handlers*), s katerimi definiramo obnašanje teh operacij.

Ta diplomska naloga ima dva poglavitna cilja. Prvi je opisati programski jezik eff in programiranje z algebrajskimi učinki na pristopen način. V poglavju 2 opišemo poglavitne značilnosti effa s poudarkom na njegovi predstavitvi računskih učinkov. V poglavju 3 skozi vrsto primerov pokažemo nekaj možnosti, kako lahko razne probleme rešujemo z uporabo učinkov in prestreznikov kot prvorazrednih vrednosti.

Drugi cilj je razširitev prototipne implementacije effa z opozorili za prirejanja vzorcev v obravnavi primerov. Enako kot pri jezikih družine ML je tudi v effu prirejanje vzorcev pomembno orodje pri definicijah rekurzivnih funkcij. Ideja je, da izvajanje funkcije usmerimo glede na strukturo vhodnih parame-

trov; ločimo lahko recimo med praznim in nepraznim seznamom. V obravnavi primerov naštejemo vzorce za vse možne vhodne parametre. Če kakšno možnost izpustimo, lahko med izvajanjem pride do napake, zato želimo, da nas v tem primeru prevajalnik opozori in nam tako olajša pisanje pravilnih programov. Poglavje 4 opisuje pristop, ki smo ga ubrali za implementacijo teh opozoril.

V zadnjem poglavju na kratko povzamemo diplomsko delo in podamo zaključek. Poleg tega opišemo omejitve naše implementacije in podamo nekaj predlogov za njeno razširitev ter možnosti za nadaljnji razvoj.

Poglavje 2

Programski jezik eff

V tem poglavju predstavimo osnove programskega jezika eff. Ker je tukaj naš glavni namen vzpostaviti ogrodje, znotraj katerega bomo v nadaljevanju opisovali algebrajske učinke in prestreznike, obravnavamo le poenostavljeno inačico jezika. Poleg konstruktov, ki implementirajo učinke, v njej ohranimo le najnujnejše elemente. Razlaga je namenjena programerjem, zato se poskušamo držati praktičnih vidikov in največ pozornosti posvetimo novim konceptom.

Osnovni jezik si številne značilnosti deli z OCamlom. Najbolj očitna je sintaksa, ki jo opišemo v prvem razdelku. Sledi glavni del poglavja, v katerem opišemo sintakso in obnašanje računskih učinkov in prestreznikov. Podamo tudi pravila za izpeljavo tipov, na koncu pa orišemo ključne lastnosti, po katerih se dejanska implementacija razlikuje od opisanega jezika [2].

2.1 Sintaksa

Eff vsebuje osnovne tipe za cela števila in boolove vrednosti, poleg tega pa še tip enote `unit` z edino vrednostjo `()` in prazen tip `empty`, ki nima nobene vrednosti in ga potrebujemo za opis izjem. Osnovne tipe lahko kombiniramo s pomočjo urejenih parov oziroma produktov in vsot. Dodatno ima eff tipe učinkov E in prestreznikov $A \Rightarrow B$. Notacija $(\cdots)_i$ spodaj in drugje pomeni, da lahko $\cdots$ ponovimo poljubnokrat.

$$\begin{aligned} \text{Tip } A, B, C ::= {} & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{empty} \mid \\ & A \times B \mid A + B \mid A \to B \mid \\ & E \mid A \Rightarrow B \\ \text{Tip učinka } E ::= {} & \texttt{effect}\ (\texttt{operation}\ \texttt{op}_i : A_i \to B_i)_i\ \texttt{end} \end{aligned}$$

Tipi učinkov opisujejo množico operacijskih simbolov $\texttt{op}_i$. Kadar tip učinka E vsebuje operacijo `op` s parametrom tipa A in rezultatom tipa B, pišemo tudi $\texttt{op} : A \to B \in E$. Če določena operacija ne sprejema ali vrača vrednosti, za vhodni oziroma izhodni tip uporabimo enoto. Za primer si oglejmo naslednji tip učinka, ki predstavlja komunikacijski kanal z operacijama branje in pisanje.

```
type channel = effect
  operation read: unit -> string
  operation write: string -> unit
end
```

Operacija branja ne sprejme nobenega parametra, prebere niz iz kanala in ga vrne. Pisanje je nasprotna operacija, ki sprejme niz, ga izpiše in ne vrne ničesar.

Eff loči med funkcijsko čistimi *izrazi* (angl. *expression*), ki ne morejo povzročiti računskih učinkov, med katere spada tudi divergenca, in *izračuni* (angl. *computation*), ki lahko. Med osnovne izraze spadajo spremenljivke x, celoštevilske konstante n, boolovi vrednosti in ostale vgrajene konstante c. Izraz `()` predstavlja vrednost s tipom enote. Produkte vpeljemo z uporabo izrazov oblike (e_1, e_2), vsote pa z `Left` e in `Right` e. Izraz $\texttt{fun}\, x : A \mapsto c$ definira funkcijo s parametrom x tipa A, ki ob aplikaciji na izraz e vrne izračun c, v katerem so vse pojavitve spremenljivke x zamenjane z e. Operacije $e\,\texttt{\#op}$ in prestreznike h razložimo v naslednjem razdelku. V pravilih za izraze in izračune spodaj uporabljamo visok $\mid$ za ločevanje med možnimi oblikami pravila, nizek `|` pa ločuje posamezne primere v stavkih `match` in `handler`.

$$\begin{aligned}
\text{Izraz } e ::= {} & x \mid n \mid c \mid \texttt{true} \mid \texttt{false} \mid \\
& \texttt{()} \mid (e_1, e_2) \mid \texttt{Left}\, e \mid \texttt{Right}\, e \mid \texttt{fun}\, x : A \mapsto c \mid \\
& e\,\texttt{\#op} \mid h \\
\text{Prestreznik } h ::= {} & \texttt{handler}\ (e_i\,\texttt{\#op}\, x\, k \mapsto c_i)_i\ \texttt{|}\ \texttt{val}\, x \mapsto c_v\ \texttt{|}\ \texttt{finally}\, x \mapsto c_f
\end{aligned}$$

Poljuben izraz lahko povzdignemo v izračun, ki ne povzroči učinkov, z uporabo operatorja `val`. Dejanska implementacija effa to pretvorbo naredi avtomatično, kjer je potrebna. V stavku $\texttt{let}\ x = c_1\ \texttt{in}\ c_2$ se najprej izračuna vrednost c_1, nato pa se izvede c_2, v katerem so pojavitve spremenljivke x zamenjane s to vrednostjo. Stavek $\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2$ v izračunu c_2 definira rekurzivno funkcijo f s parametrom x in telesom c_1. Tri oblike stavkov `match` omogočajo razgradnjo praznega tipa, produktov in vsot, pogojni stavek `if` pa naredi enako za boolove vrednosti. Zadnje tri oblike pravila za izračune se uporabljajo za ustvarjanje novih instanc učinkov in prestrezanje operacij ter

so opisane v naslednjem razdelku.

$$\begin{aligned}\text{Izračun } c ::= {} & \texttt{val}\ e \mid \texttt{let}\ x = c_1\ \texttt{in}\ c_2 \mid \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 \mid \\ & \texttt{match}\ e\ \texttt{with} \mid \texttt{match}\ e\ \texttt{with}\ (x, y) \mapsto c \mid \\ & \texttt{match}\ e\ \texttt{with Left}\ x \mapsto c_1\ \texttt{|}\ \texttt{Right}\ y \mapsto c_2 \mid \\ & \texttt{if}\ x\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid e_1\ e_2 \mid \\ & \texttt{new}\ E \mid \texttt{new}\ E\ \texttt{@}\ e\ \texttt{with}\ (\texttt{operation op}_i\ x\ \texttt{@}\ y \mapsto c_i)_i\ \texttt{end} \mid \\ & \texttt{with}\ e\ \texttt{handle}\ c\end{aligned}$$

Aritmetični operatorji so predstavljeni z vgrajenimi konstantami, tako da na izračun oblike $e_1 + e_2$ gledamo kot na dvojno aplikacijo. Vsi aritmetični izrazi torej štejejo kot izračuni, ne glede na to, ali lahko povzročijo računske učinke (deljenje z nič) ali ne. To nam omogoča, da jih lahko vse obravnavamo enakovredno.

2.2 Učinki v effu

V tem razdelku predstavimo konstrukte, s katerimi eff obravnava računske učinke. Princip delovanja prestreznikov učinkov je površinsko podoben mehanizmu za obravnavo izjem v mnogih popularnih programskih jezikih. S pomočjo izjem lahko programer obravnavo napak med izvajanjem loči od funkcionalnega dela programa. Eff to idejo posploši na druge vrste računskih učinkov in nam omogoča, da njihovo obnašanje prav tako ločimo od preostalega programa. Za začetek si oglejmo, kako izjeme obravnava Python. Kot primer vzemimo naslednji izsek kode, ki ponazarja osnovne koncepte.

```
try:
    print(0)
    raise Exception
    print(1)
except Exception:
    print("izjema")
finally:
    print(2)
```

Ko ta program poženemo, po vrsti izvaja stavke v bloku `try`, dokler ne naleti na `raise Exception`. Ta povzroči izjemo, zaradi česar izvajalno okolje preskoči preostali del tega bloka in nadzor da bloku `except`. Teh je lahko več in vsak obravnava svojo izjemo. Nadzor dobi tisti blok, ki obravnava tip izjeme, ki se

je dejansko zgodila. Stavki v bloku `finally` se na koncu zmeraj izvedejo, ne glede na to, ali je med izvajanjem prišlo do izjeme ali ne. Program tako torej izpiše naslednje vrstice:

```
0
izjema
2.
```

Preden se spustimo v podrobnejši opis prestreznikov, podamo kratek primer, da pokažemo, kateri konstrukti imajo intuitivno podobne vloge kot `try`, `except` in `finally` zgoraj. Tukaj je `std` instanca učinka tipa `channel`, ki je vgrajena v eff in predstavlja standardni vhod/izhod programa. Privzeto obnašanje operacije `std#read` je, da prebere vrstico s tipkovnice in jo vrne.

```
with
  handler
    | std#read () k -> k "foo"
    | finally x -> (x, x)
handle
  std#read ()
```

Konstrukt `with` h `handle` c izvaja izračun c, dokler ne naleti na kakšno operacijo. Izračun c torej ustreza prvemu delu stavka `try`, medtem ko operacije sprožijo računske učinke, podobno kot z `raise` sprožimo izjeme. S prestrezni-kom h definiramo obnašanje posameznih operacij, prav tako pa lahko določimo končno transformacijo, ki se zgodi, ko se c evaluira v vrednost. Posamezni primeri, ki jih definira h, torej v grobem ustrezajo blokom `except` in `finally` zgoraj. Podobno kot pri izjemah lahko izračun c ovijemo v več nivojev prestreznikov. Ko pride do operacije, prevzame nadzor najbližji prestreznik, ki jo obravnava.

Prestrezniki učinkov se od izjem bistveno razlikujejo v tem, da so bolj splošni. Določimo lahko, ali in na kakšen način naj se izvajanje izračuna c nadaljuje, ko naleti na operacijo, medtem ko se pri izjemah izvajanje zmeraj prekine. V zgornjem primeru sestavlja izračun c le operacija `std#read ()`. Uporabili smo prestreznik, ki povozi (angl. *override*) privzeto obnašanje in zmeraj vrne niz `"foo"`. Člen `finally` pa naredi urejen par iz dveh kopij končnega rezultata izračuna c. Ta program torej vrne

```
string * string = ("foo", "foo").
```

V nadaljevanju tega razdelka najprej predstavimo pojma instance učinka in operacije. Nato opišemo postopek za prestrezanje operacij, na koncu pa še,

kako lahko s pomočjo virov ohranjamo stanje (angl. *state*) med posameznimi pojavitvami nekega učinka in določimo privzeto obnašanje operacij.

2.2.1 Instance in operacije

V razdelku 2.1 smo opisali tipe učinkov, ki določajo množice sorodnih operacij. Vsak učinek lahko ima več *instanc*; za tip učinka `channel` vsaka instanca predstavlja drug komunikacijski kanal, kot so standardni vhod/izhod (vgrajena instanca `std`), datoteka ali omrežna povezava. Novo instanco učinka tipa E ustvarimo z izračunom `new` E. Razširjena oblika operatorja `new` nam omogoča, da za instanco definiramo tudi pripadajoč vir.

Za vsako instanco e učinka tipa E in operacijski simbol `op` $\in E$ obstaja *operacija* e `#op`. Grobo rečeno služijo operacije kot oznake izvajalnemu okolju, da je v nekem izračunu prišlo do računskega učinka. Nadaljevanje izračuna je potem odvisno od prestreznikov, ki operacijo ovijajo, in morebitnega vira, ki pripada instanci e.

2.2.2 Prestrezniki

V uvodu tega razdelka smo videli primer prestrezanja računskega učinka. Prestrezniki so v splošnem oblike

$$h = \texttt{handler}\ (e_i\ \texttt{\#op}\ x\ k \mapsto c_i)_i\ \texttt{|}\ \texttt{val}\ x \mapsto c_v\ \texttt{|}\ \texttt{finally}\ x \mapsto c_f.$$

Če katerega od členov `val` ali `finally` izpustimo, se privzame identiteta oziroma funkcija $x \mapsto \texttt{val}\ x$. Prestreznik h uporabimo na izračunu c s pomočjo konstrukta

$$\texttt{with}\ h\ \texttt{handle}\ c. \tag{1}$$

Računski učinki – z izjemo divergence – se v effu lahko zgodijo le med operacijami. Povedano drugače, če izračun c ne divergira, se njegova evaluacija lahko konča le, ko doseže vrednost, ali pa naleti na aplikacijo operacije na parameter. Začasno zanemarimo člen `finally` v prestreznikih in si oglejmo, kako se obnaša (1) v obeh primerih.

1. Če se c evaluira v vrednost `val` e, se izračun (1) evaluira v c_v s spremenljivko x vezano na izraz e. To se zgodi takrat, ko c ne vsebuje nobene operacije.

2. Če izvajalno okolje med evaluacijo izračuna c naleti na operacijo e_i `#op`$_i$ e, se izračun (1) evaluira v c_i s spremenljivko x vezano na izraz e in k

vezana na *kontinuacijo* operacije e_i `#` $\texttt{op}_i$ e. Kontinuacija predstavlja tisti del izračuna (1), ki se še mora evaluirati po operaciji, in je v splošnem funkcija[1], ki sprejme rezultat operacije. Denimo, da imamo izračun

```
with h handle (std#read ())^"bar",
```

kjer operator `^` združi dva niza. Ko operacijo `std#read` v tem izračunu prestrežemo, predstavlja neevaluiran del kontinuacija, ki sprejme rezultat operacije kot parameter:

```
k x = with h handle x^"bar".
```

Če uporabimo prestreznik iz začetka tega razdelka, ki preprosto pokliče kontinuacijo s parametrom `"foo"`, je naslednji korak v evaluaciji

```
with h handle "foobar".
```

Seveda so v praksi prestrezniki običajno bolj kompleksni in manj neuporabni. Kot katerokoli funkcijo lahko tudi kontinuacijo pokličemo večkrat z različnimi parametri in na ta način razdelimo izvajanje programa na več vej. Tako lahko implementiramo razne mehanizme za nadzor toka programa, kot so izjeme in iskanje s sestopanjem, ki so opisani v naslednjem poglavju.

Člen `finally` opisuje transformacijo, ki se zgodi po tem, ko je (1) do konca evaluiran po teh dveh točkah. Namesto (1) bi lahko torej ekvivalentno pisali tudi

```
let x = (with h' handle c) in c_f,
```

kjer je h' enak h brez člena `finally`.

Če izvajalno okolje med evaluacijo izračuna c naleti na operacijo e `#` `op`, ki je ni v h, prevzame nadzor prestreznik na višjem nivoju. Operacija tako napreduje na zmeraj višji nivo, dokler je kateri prestreznik ne obravnava. Če pride do vrha, jo prestreže izvajalno okolje in je rezultat odvisen od vira, ki pripada instanci učinka e.

2.2.3 Viri

S pomočjo *virov* lahko uporabnik za vsako instanco nekega učinka definira privzeto obnašanje operacij. To pride v poštev takrat, ko operacije ne prestreže uporabnik in jo obravnava izvajalno okolje. Vir ohranja tudi stanje med

[1] To je poenostavljena predstavitev kontinuacij, ki zadošča za naše potrebe.

posameznimi klici operacij te instance. Novo instanco n tipa učinka E s pripadajočim virom ustvarimo z

$$\texttt{let}\ n = \texttt{new}\ E\ \texttt{@}\ e\ \texttt{with}\ (\texttt{operation}\ \texttt{op}_i\ x\ \texttt{@}\ y \mapsto c_i)_i\ \texttt{end}.$$

Začetno stanje vira za n je enako e. Ko izvajalno okolje naleti na neprestreženo operacijo $n\,\texttt{\#}\,\texttt{op}_i\,e'$, evaluira izračun c_i s spremenljivko x vezano na e' in y vezano na trenutno stanje vira instance n. Rezultat izračuna mora biti par vrednosti, v katerem prva komponenta predstavlja parameter, s katerim se pokliče kontinuacija, in druga novo stanje vira. Izračuni c_i se morajo evaluirati v par vrednosti in ne v operacijo, saj je izvajalno okolje ne bi moglo obravnavati.

Eff uporablja vgrajene vire za implementacijo določenih računskih učinkov. Primera sta komunikacija z okoljem prek instance `std` tipa učinka `channel` in izbiranje naključnih naravnih števil z operacijo `rnd#int` instance tipa učinka `random`.

2.3 Preverjanje tipov

Eff ima statičen sistem tipov, kar pomeni, da se tipi izrazov in izračunov izpeljejo in preverijo pred izvajanjem. Če tolmač odkrije kakšno napako pri uporabi tipov, zavrne program kot neveljaven. Primer programa, ki vsebuje takšno napako, je `3 + true`; operator `+` namreč pričakuje dva parametra tipa `int`. Prednost tega pristopa je, da program, ki se prevede, zagotovo ne vsebuje napačne uporabe tipov. Zaradi tega je izvajalno okolje preprostejše, saj informacije o tipih med izvajanjem niso več potrebne.

Tipe programov določajo pravila sklepanja, ki jih predstavimo v tem razdelku. Vsako pravilo je sestavljeno iz seznama predpostavk $P_1, \ldots, P_n$ nad črto in zaključka Q pod črto ter pomeni $P_1 \wedge \cdots \wedge P_2 \Rightarrow Q$. Če je seznam predpostavk prazen, pravilo zmeraj drži in mu pravimo *aksiom*. Tromestna relacija $\Gamma \vdash_{\texttt{e}} e : A$ pomeni, da ima izraz e v kontekstu Γ tip A, relacija $\vdash_{\texttt{c}}$ pa pomeni enako za izračune. Kontekst Γ je seznam spremenljivk, ki se lahko pojavijo proste v e, in njihovih tipov. Sama spremenljivka je izraz, katerega tip določimo neposredno iz konteksta s pravilom

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{\texttt{e}} x : A}.$$

Konstante imajo zmeraj isti tip, ne glede na kontekst. Njihove tipe zato določajo naslednji aksiomi. Manjkajo pravila za preostale vgrajene konstante,

kot so aritmetični operatorji.

$$\frac{}{\Gamma \vdash_{\mathtt{e}} n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash_{\mathtt{e}} \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash_{\mathtt{e}} \mathtt{false} : \mathtt{bool}}$$

Preostala standardna pravila opisujejo tipe izrazov, s katerimi vpeljemo enoto, produkte, vsote in funkcije.

$$\frac{}{\Gamma \vdash_{\mathtt{e}} () : \mathtt{unit}} \qquad \frac{\Gamma \vdash_{\mathtt{e}} e_1 : A \qquad \Gamma \vdash_{\mathtt{e}} e_2 : B}{\Gamma \vdash_{\mathtt{e}} (e_1, e_2) : A \times B} \qquad \frac{\Gamma \vdash_{\mathtt{e}} e : A}{\Gamma \vdash_{\mathtt{e}} \mathtt{Left}\ e : A + B}$$

$$\frac{\Gamma \vdash_{\mathtt{e}} e : B}{\Gamma \vdash_{\mathtt{e}} \mathtt{Right}\ e : A + B} \qquad \frac{\Gamma, x : A \vdash_{\mathtt{c}} c : B}{\Gamma \vdash_{\mathtt{e}} (\mathtt{fun}\ x : A \mapsto c) : A \to B}$$

Eff poleg teh izrazov vsebuje še definicije operacij in prestreznikov. Operacija $e\,\mathtt{\#}\,\mathtt{op}$, kjer je e instanca nekega učinka, ima tip funkcije:

$$\frac{\Gamma \vdash_{\mathtt{e}} e : E \qquad \mathtt{op} : A \to B \in E}{\Gamma \vdash_{\mathtt{e}} e\,\mathtt{\#}\,\mathtt{op} : A \to B}.$$

Pravilo za tip prestreznikov preveri, da za vsako operacijo $e_i\,\mathtt{\#}\,\mathtt{op}_i$ tip instance učinka e_i vsebuje operacijski simbol $\mathtt{op}_i$. Prav tako morajo rezultati izračunov za vse operacije in člen `val` biti istega tipa B, da lahko na njih člen `finally` na koncu izvede izračun c_f. Celotno pravilo pomeni, da prestreznik operacije na izračunih tipa A transformira v izračun tipa C.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{e}} e_i : E_i \quad \mathtt{op}_i : A_i \to B_i \in E_i \\ \Gamma, x : A, k : B_i \to B \vdash_{\mathtt{c}} c_i : B\end{array} \qquad \Gamma, x : A \vdash_{\mathtt{c}} c_v : B \qquad \Gamma, x : B \vdash_{\mathtt{c}} c_f : C}{\Gamma \vdash_{\mathtt{e}} \mathtt{handler}\ (e_i\,\mathtt{\#}\,\mathtt{op}_i\ x\ k \mapsto c_i)_i\ \mathtt{|}\ \mathtt{val}\ x \mapsto c_v\ \mathtt{|}\ \mathtt{finally}\ x \mapsto c_f : A \Rightarrow C}$$

Pravila za izračune so podobno preprosta in večinoma samoumevna. Dvig izraza v izračun z operatorjem `val` ohrani tip, kar povemo s pravilom

$$\frac{\Gamma \vdash_{\mathtt{e}} e : A}{\Gamma \vdash_{\mathtt{c}} \mathtt{val}\ e : A}.$$

Tipi stavkov `let` in `let rec` so določeni s praviloma spodaj.

$$\frac{\Gamma \vdash_{\mathtt{c}} c_1 : A \qquad \Gamma, x : A \vdash_{\mathtt{c}} c_2 : B}{\Gamma \vdash_{\mathtt{c}} \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : B}$$

$$\frac{\Gamma, f : A \to B, x : A \vdash_{\mathtt{c}} c_1 : B \qquad \Gamma, f : A \to B \vdash_{\mathtt{c}} c_2 : C}{\Gamma \vdash_{\mathtt{c}} \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 : C}$$

Boolove vrednosti eliminiramo s pogojnim stavkom, kjer preverimo, da sta obe veji istega tipa:

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \texttt{bool} \qquad \Gamma \vdash_{\mathsf{c}} c_1 : A \qquad \Gamma \vdash_{\mathsf{c}} c_2 : A}{\Gamma \vdash_{\mathsf{c}} \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : A}.$$

Prva oblika stavka `match` eliminira izraze praznega tipa in lahko dobi poljuben tip. Drugi dve obliki se uporabljata za razgradnjo produktov in vsot. Tu preverimo, ali se tipa dobljenih komponent x in y ujemata z njuno uporabo v izračunih c_i na desni strani puščice.

$$\frac{\Gamma \vdash_{\mathsf{e}} e : \texttt{empty}}{\Gamma \vdash_{\mathsf{c}} \texttt{match}\ e\ \texttt{with} : A} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e : A \times B \qquad \Gamma, x : A, y : B \vdash_{\mathsf{c}} c : C}{\Gamma \vdash_{\mathsf{c}} \texttt{match}\ e\ \texttt{with}\ (x,y) \mapsto c : C}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A + B \qquad \Gamma, x : A \vdash_{\mathsf{c}} c_1 : C \qquad \Gamma, y : B \vdash_{\mathsf{c}} c_2 : C}{\Gamma \vdash_{\mathsf{c}} \texttt{match}\ e\ \texttt{with}\ \texttt{Left}\ x \mapsto c_1\ \texttt{|}\ \texttt{Right}\ y \mapsto c_2 : C}$$

Za aplikacije preverimo, ali ima vhodni izraz e_2 tip, ki ga funkcija e_1 pričakuje:

$$\frac{\Gamma \vdash_{\mathsf{e}} e_1 : A \to B \qquad \Gamma \vdash_{\mathsf{e}} e_2 : A}{\Gamma \vdash_{\mathsf{c}} e_1\ e_2 : B}.$$

Osnovna oblika pravila za tipe instanc učinkov, ki jih ustvarimo z operatorjem `new`, je aksiom – instanca dobi isti tip kot učinek. Razširjena oblika ustvari še pripadajoč vir, ki predstavlja stanje in opisuje privzeto obnašanje operacij, ki jih vsebuje tip učinka E. Naj bo izraz e, ki predstavlja začetno stanje, tipa C. Potem moramo preveriti, da za vsako operacijo $\mathtt{op}_i : A_i \to B_i$ pripadajoč izračun c_i vrne par, v katerem prva komponenta predstavlja rezultat operacije, s tipom B_i, in druga novo stanje vira, tipa C.

$$\frac{}{\Gamma \vdash_{\mathsf{c}} \texttt{new}\ E : E}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} e : C \qquad \mathtt{op}_i : A_i \to B_i \in E \qquad \Gamma, x : A_i, y : C \vdash_{\mathsf{c}} c_i : B_i \times C}{\Gamma \vdash_{\mathsf{c}} \texttt{new}\ E\ \texttt{@}\ e\ \texttt{with}\ (\texttt{operation}\ \mathtt{op}_i\ x\ \texttt{@}\ y \mapsto c_i)_i\ \texttt{end} : E}$$

Preostane še pravilo za prestrezanje, ki je analogno pravilu za tip aplikacije funkcije:

$$\frac{\Gamma \vdash_{\mathsf{e}} e : A \Rightarrow B \qquad \Gamma \vdash_{\mathsf{c}} c : A}{\Gamma \vdash_{\mathsf{c}} \texttt{with}\ e\ \texttt{handle}\ c : B}.$$

2.4 Implementacija

Prototipna implementacija effa[2] poleg konstruktov, opisanih v tem poglavju, vsebuje še nekaj razširitev za lažje pisanje večjih programov. Med njih spadajo nekatere imperativne lastnosti, kot sta zanki `for` in `while`. Reference so s pomočjo učinkov implementirane kot prvorazredni objekti, kar je opisano v razdelku 3.3. Za lažje razstavljanje vrednosti jezik vključuje vzorce, ki jih uporablja pri analizi primerov. To je podrobno opisano v poglavju 4. Eff podpira tudi parametrični polimorfizem za izraze. Vse naštete lastnosti lahko v praktično enaki obliki najdemo v OCamlu [4].

V razdelku o sintaksi smo videli, da so izrazi in izračuni obravnavani ločeno. V konkretni sintaksi jih lahko poljubno mešamo, eff pa pred izvajanjem program pretvori v pravilno obliko. Kadar naleti na izraz na mestu, kjer bi moral biti izračun, vstavi `val`. V nasprotnem primeru spremeni izračun v izraz z uporabo stavka `let`; tako recimo `(f x, g y)` postane

```
let a = f x and b = g x in (a, b),
```

kjer vrstni red izvajanja izračunov `f x` in `g y` ni določen. To je pomembno, kadar lahko več izračunov povzroči računske učinke. Eff nas lahko v takih primerih opozori na morebitno nepredvideno obnašanje.

Izpostavimo še eno sintaktično bližnjico, ki jo uporabljamo v nekaterih primerih v naslednjem poglavju. Večkrat se zgodi, da kakšen prestreznik uporabimo le enkrat. Takrat lahko namesto

```
with handler ... handle c
```

pišemo

```
handle c with ....
```

To je med drugim priročno za obravnavo izjem in takrat, kadar želimo kakšno operacijo v določenem izračunu obravnavati na poseben način.

[2]Dostopna na `https://github.com/matijapretnar/eff`.

Poglavje 3

Uporaba računskih učinkov

V tem poglavju si ogledamo, kako lahko s pomočjo računskih učinkov in njihovih prestreznikov rešujemo razne probleme. Ena glavnih prednosti prestreznikov je, da lahko z njimi neposredno nadzorujemo tok programa. Tako lahko le z zamenjavo prestreznika popolnoma spremenimo obnašanje izračunov. Druga pomembna lastnost je, da jih lahko gnezdimo in na ta način povozimo ali prilagodimo obnašanje operacij v določenih kontekstih.

V prvih razdelkih tega poglavja opišemo delovanje prestreznikov na vrsti tipičnih primerov, kot so V/I, izjeme in stanje. V drugem delu pa se posvetimo učinkom in prestreznikom, s katerimi lahko pišemo nedeterministične izračune.

3.1 Vhod/izhod

Skozi prejšnje poglavje smo za predstavitev osnovnih konceptov uporabljali učinek za komunikacijski kanal, zato začnemo z njim. V effu je ta učinek predstavljen s tipom

```
type channel = effect
  operation read: unit -> string
  operation write: string -> unit
end,
```

standardni vhod/izhod programa pa predstavlja vgrajena instanca `std`. Dejansko komunikacijo z operacijskim sistemom implementira vir za to instanco, ki definira privzeto obnašanje operacij za branje in pisanje.

Kot katerokoli drugo instanco lahko tudi `std` obravnavamo z lastnim prestreznikom, ki spremeni to obnašanje. Videli smo že, kako dosežemo, da vsaka operacija branja vrne isti niz; podobno lahko s prestreznikom

```
handler std#write _ k -> k ()
```

izbrišemo ves izhod programa. Namesto tega lahko nize, ki jih program izpisuje, shranimo v seznam s prestreznikom

```
let accumulate = handler
  | std#write x k -> let (v, xs) = k () in (v, x :: xs)
  | val v -> (v, []).
```

Ko ga uporabimo na izračunu, vrne vrednost, v katero se je račun evaluiral, in seznam vseh nizov, ki jih je med tem poslal na standardni izhod. Tako izračun

```
with accumulate handle
  std#write "hello"; std#write "world"; 6 * 9
```

ne izpiše ničesar in vrne `(54, ["hello"; "world"])`. Če se izračun, ki ga obravnavamo, evaluira v vrednost – torej ne vsebuje nobene operacije in ne izpiše ničesar, jo člen `val` vrne skupaj s praznim seznamom. Kadar pa obravnavan izračun naleti na operacijo `std#write str`, se izvede prvi člen prestreznika, ki nadaljuje izračun s klicem kontinuacije `k ()`. Dobljeno vrednost in seznam nizov shrani v `(v, xs)` ter vrne isto, le da na začetek seznama nizov doda še `str`. Prestreznik `accumulate` torej zgornji izračun razvije v obliko, ki izgleda približno tako[1]:

```
let (v, xs) =
  let (v, xs) =
    (6 * 9, [])
  in (v, "world" :: xs)
in (v, "hello" :: xs).
```

Prav tako lahko prestrežemo operacije branja in nize namesto iz standardnega vhoda vračamo iz podanega seznama. Za to definiramo malo bolj zapleten prestreznik

```
let read_from_list lst = handler
  | val x -> (fun _ -> x)
  | std#read () k -> (fun (s::lst') -> k s lst')
  | finally f -> f lst.
```

Glavna ideja tukaj je, da člena za operacijo in vrednost izračun pretvorita v funkcijo, ki sprejme seznam nizov, člen `finally` pa jo aplicira na vhodni

[1]Bolj natančno bi morali na konec prvih dveh vrstic dodati `();`. Kontinuacija izračuna `std#write (); x` je namreč `(); x` in ne le `x`. Ti dve možnosti sta sicer enakovredni, saj $c_1;c_2$ vrne le rezultat evaluacije izračuna na desni.

seznam. Tak pristop bomo srečali tudi pozneje, zato pobliže oglejmo, kako deluje na naslednjem preprostem primeru.

```
with read_from_list ["bar"] handle
  "foo" ^ std#read ()
```

Za lažje razumevanje evaluacije tega izračuna člen `finally` najprej razvijemo v stavek `let`, da dobimo ekvivalentni izračun

```
let f = with h handle
  "foo" ^ std#read ()
in f ["bar"].
```

Tu je `h` enak originalnemu prestrezniku brez zadnjega člena in parametra `lst`; kot rezultat namreč vrne funkcijo, ki ji podamo vhodni seznam. Ko evaluacija doseže operacijo `std#read`, dobi prestreznik kontinuacijo

```
k x = with h handle "foo" ^ x
```

in izračun pretvori v funkcijo

```
let f = fun (s::lst) ->
  (with h handle "foo" ^ s) lst
in f ["bar"].
```

Funkcijo `f` lahko zdaj apliciramo na vhodni seznam, da dobimo naslednjo stopnjo v evaluaciji

```
(with h handle "foobar") [].
```

Ko se obravnavan izračun evaluira v vrednost, kot parameter preostane prazen seznam. Da se ga znebimo, člen `val` iz vrednosti naredi funkcijo, ki zavrže vhodni parameter:

```
(fun _ -> "foobar") [].
```

Končni rezultat evaluacije je tako `"foobar"`. Prestreznik `read_from_list` torej izračun ovije v novo funkcijo za vsako operacijo branja, ki se v njem pojavi, na koncu pa najbolj zunanjo funkcijo aplicira na vhodni seznam. Na vsakem nivoju se porabi prvi element iz seznama, preostanek pa dobi funkcija na enem nivoju niže. Na podoben način lahko implementiramo reference, ki jih opišemo v razdelku 3.3.

3.2 Izjeme

Izjeme so mehanizem za obravnavanje napak med izračuni. V effu jih predstavlja učinek `exception` z operacijo `raise`, ki izjemo vrže:

```
type 'a exception = effect
  operation raise: 'a -> empty
end.
```

Operacija `raise` sprejme parameter poljubnega tipa, v katerem lahko podamo dodatne informacije o izjemi, rezultat pa ima prazen tip, ki ne vsebuje nobene vrednosti. Posledica tega je, da prestreznik za to operacijo ne more poklicati kontinuacije in tako nadaljevati izračuna, ki je izjemo sprožil. To obnašanje je konsistentno s prestrezanjem izjem v drugih programskih jezikih. Primer iz začetka razdelka 2.2 lahko zdaj prepišemo v eff.

```
let e = new exception in
handle
  print 0;
  e#raise ();
  print 1
with
  | e#raise _ _ -> print "izjema"
  | finally _ -> print 2
```

Za prestrezanje izjem običajno uporabljamo enkratne prestreznike kot v zgornjem primeru. Seveda lahko definiramo tudi splošne prestreznike. Eff recimo vsebuje naslednjega, ki v poljubnem izračunu prestreže izjemo `e` in vrne rezultat kot opcijski tip.

```
let option_catch e = handler
  | e#raise _ _ -> None
  | val x -> Some x
```

V jezikih družine ML lahko `raise` uporabimo kjerkoli, ker ima polimorfni tip. Nasprotno ima `e#raise e'` prazen tip, zato ga na primer ne moremo uporabiti kot v izrazu

```
fun a b -> if b = 0 then e#raise "divisionByZero" else a / b,
```

ker se tip leve veja pogojnega stavka ne ujema z desno. Ta problem lahko razrešimo z uporabo pomožne funkcije

```
let raise e x = match (e#raise x) with.
```

Stavek `match` eliminira prazen tip in je polimorfen, torej ima funkcija `raise` tip α `exception` $\to \alpha \to \beta$. Prejšnji izraz lahko s pomočjo te funkcije torej prepišemo v

```
fun a b -> if b = 0 then raise e "divisionByZero" else a / b.
```

3.3 Reference

V stavkih oblike `let` x `=` e `in ...` je identifikator x le oznaka za izraz e, zato njegove vrednosti ne moremo spremeniti. Prave spremenljivke oziroma *reference*, kot jih poznamo iz drugih programskih jezikov, predstavljajo računski učinek, saj je lahko zaradi njih rezultat neke funkcije odvisen od stopnje izvajanja programa. Reference predstavimo s tipom učinka, ki vsebuje operaciji za branje in pisanje njihovih vrednosti.

```
type 'a ref = effect
  operation lookup: unit -> 'a
  operation update: 'a -> unit
end
```

Prestreznik, ki za referenco `r` implementira pričakovano obnašanje teh operacij, uporablja podoben pristop kot `read_from_list` iz razdelka 3.1. V obravnavanem izračunu vsako operacijo spremeni v funkcijo, ki sprejme trenutno stanje, člen `finally` pa na koncu tako dobljeno funkcijo aplicira na začetno stanje `x`.

```
let state r x = handler
  | val y -> (fun s -> y)
  | r#lookup () k -> (fun s -> (k s) s)
  | r#update s' k -> (fun s -> (k ()) s')
  | finally f -> f x
```

Če izračun ne vsebuje več nobene operacije, ga člen `val` pretvori v funkcijo, ki enostavne zavrže parameter `s` (stanje). Tudi operacijska člena vrneta funkciji, ki sprejmeta stanje reference. Kontinuaciji za branje vrnemo trenutno stanje kot rezultat operacije in jo obravnava isti prestreznik. Zato se spet evaluira v funkcijo `(k s)`, ki sprejme stanje. Ker se vrednost reference med branjem ne spremeni, tudi novi funkciji podamo `s`. Kontinuaciji za pisanje vrnemo enoto in se prav tako evaluira v novo funkcijo `(k ())`, ki pa ji podamo posodobljeno vrednost reference `s'`. Kot prej imamo torej prestreznik, ki za vsako operacijo izračun ovije v funkcijo, na koncu pa izvede funkcijo na najbolj zunanjem nivoju z začetnim stanjem. Ta evaluira del izračuna do naslednje operacije in pokliče funkcijo na nivoju niže z novim stanjem, kar se ponavlja, dokler se celoten izračun ne evaluira v vrednost.

Ta prestreznik lepo ponazarja možen pristop k obravnavanju računskih učinkov, vendar ima dve pomembni pomanjkljivosti. Prva je ta, da moramo vsak izračun c, v katerem želimo uporabljati referenco, oviti v stavka `let` in `handle`:

```
let r = new ref in
  with state r x handle
    c.
```

Prav tako se nam lahko zgodi, da nam referenca „uide“ iz bloka, ki ga obravnava prestreznik `state`. V tem primeru vsaka operacija na referenci sproži napako med izvajanjem. Trivialen primer za to je, da zgoraj uporabimo

$$c = \texttt{fun \_ -> r\#lookup ()},$$

ki ne sproži nobene operacije, dokler funkcije ne apliciramo na parameter. V splošnem je težko ali nemogoče povedati, kdaj se bo to zgodilo. Oba problema lahko rešimo z uporabo virov, kar se zdi naravno, saj tako kot reference opisujejo stanje. Vpeljemo naslednjo funkcijo, ki vrne novo referenco.

```
let ref x =
  new ref @ x with
    operation lookup () @ s -> (s, s)
    operation update s' @ _ -> ((), s')
  end
```

Ta definicija je precej enostavnejša, saj uporablja podporo za stanje, ki je že vgrajena v eff. Spomnimo se, da operacije v virih vrnejo par vrednosti, kjer prva predstavlja rezultat operacije in druga novo stanje vira. Za referenco to stanje predstavlja kar njeno trenutno vrednost. Takšna implementacija referenc je že prisotna v effu, poleg tega pa sta definirana še operatorja

```
let (!) r = r#lookup ()
let (:=) r v = r#update v,
```

s pomočjo katerih lahko reference uporabljamo na enak način kot v programskem jeziku OCaml [4].

3.3.1 Transakcije

Kot za vse ostale učinke lahko tudi obnašanje vgrajenih referenc povozimo z uporabo lastnih prestreznikov. Za primer vzemimo prestreznik, ki implementira *transakcije*. Transakcija je izračun, ki uporablja reference, njihovo vrednost pa spremeni le, če se uspešno konča. Če med izvajanjem recimo pride do izjeme, se reference ponastavijo na vrednosti, ki so jih imele ob začetku izračuna. Za takšno obnašanje poskrbi naslednji prestreznik.

```
let transaction r = handler
```

```
  | r#lookup () k -> (fun s -> (k s) s)
  | r#update s' k -> (fun s -> (k ()) s')
  | val x -> (fun s -> r := s; x)
  | finally f -> f !r
```

Deluje na podoben način kot prvi poskus implementacije referenc z začetka tega razdelka. Med izvajanjem obravnavanega izračuna uporablja in spreminja kopijo začetne vrednosti reference `r`, končno vrednost pa zapiše nazaj le, če se izračun evaluira v vrednost. Tako izračun

```
with transaction r handle
  r := 23;
  raise e (3 * !r);
  r := 34
```

sproži izjemo `e` s parametrom `69`, ne spremeni pa vrednosti reference `r`.

3.4 Izbira

Do zdaj smo se v primerih osredotočali predvsem na to, kako lahko s prestrezniki spreminjamo in prilagajamo podatke v izračunih. V naslednjih razdelkih pa si ogledamo, kako lahko definiramo prestreznike, ki usmerjajo tok programa na različne načine. Začnemo s tipom učinka, ki predstavlja odločitveno točko v programu.

```
type choice = effect
  operation decide: unit -> bool
end
```

Operacijo `decide` uporabimo v izračunu, kadar želimo izbiro med dvema možnostima prepustiti prestrezniku. V spodnjem primeru in pozneje `c`, `c1` in `c2` označujejo instance tega učinka.

```
let x = (if c#decide () then 2 else 4) in
let y = (if c#decide () then 0 else 1) in
  x + y
```

Ta izračun pomeni, da dobi `x` vrednost `2` ali `4`, `y` pa `0` ali `1`. Rezultat torej lahko zavzame vrednosti med 2 in 5, odvisno od obnašanja operacije `c#decide` v obeh primerih. Najpreprostejši prestreznik se zmeraj odloči za isto možnost:

```
handler c#decide () k -> k true.
```

Če ga uporabimo za zgornji izračun, dobimo rezultat `2`. Tak prestreznik je zelo osnoven in bi namesto njega lahko uporabili tudi vir s privzeto operacijo. Za primer vzamemo

```
let c = new choice @ true with
  operation decide () @ s -> (s, not s)
end,
```

kjer je začetno stanje `true`, vsaka operacija pa vrne trenutno vrednost stanja `s` in novo stanje ¬`s`. Vsaka odločitev je torej nasprotna prejšnji. Zgornji izračun za to instanco vrne `3` ali `4`, odvisno od števila operacij `c#decide`, ki so se pojavile pred začetkom njegove evaluacije.

Oglejmo si zdaj naslednji prestreznik, ki pri vsaki odločitvi upošteva *obe* možnosti. Ker bomo pozneje imeli primere z večimi instancami učinka `choice`, mu kot parameter `c` podamo instanco, katere operacije želimo obravnavati.

```
let choose_all c = handler
  | c#decide () k -> k true @ k false
  | val x -> [x]
```

Če izračun ne vsebuje nobene operacije in se evaluira v vrednost, jo člen `val` pretvori v seznam z enim elementom. Kadar pa izračun naleti na odločitev, operacijski člen pokliče kontinuacijo dvakrat z različnima parametroma. Vsako kontinuacijo spet obravnava isti prestreznik, zato bo njun končni rezultat prav tako seznam. Dobljena seznama združimo z operatorjem `@`. Program tako na vsaki odločitveni točki nadaljuje izvajanje v obeh vejah, rezultate posameznih vej pa shranjuje v seznam. Če poženemo izračun

```
with choose_all c handle
  let x = (if c#decide () then 2 else 4) in
  let y = (if c#decide () then 0 else 1) in
    x + y
```

z začetka tega razdelka, je rezultat seznam `[2;3;4;5]`. Na prvi odločitveni točki program najprej izbere `x = 2`, kontinuacija `k true` pa vrne seznam rezultatov za vse možne kombinacije odločitev v nadaljevanju izračuna. V tem primeru preostane le še odločitev za `y`, zato kontinuacija vrne `[2;3]`. Podobno se zgodi tudi za možnost `x = 4`, na koncu pa prestreznik seznama združi.

Obnašanje programa lahko spremenimo tako, da za vsako odločitveno točko uporabimo svoj prestreznik.

```
with choose_all c1 handle
with choose_all c2 handle
```

```
let x = (if c1#decide () then 2 else 4) in
let y = (if c2#decide () then 0 else 1) in
  x + y
```

Tukaj je bistveno, da kontinuacijo zmeraj obravnavata oba prestreznika. Glede na prvo odločitev notranji prestreznik ustvari seznama `[2;3]` in `[4;5]`. Zunanji prestreznik ju kot vrednosti najprej pretvori v seznama z enim elementom, nato pa združi v `[[2;3];[4;5]]`. Z eno instanco učinka smo dobili enostaven seznam vseh možnosti, ta pristop pa vrne seznam s strukturo dvojiškega drevesa.

3.5 Sestopanje

Primer iz prejšnjega razdelka lahko razširimo tako, da se na vsaki točki lahko odloča med večimi možnostmi. Prav tako lahko imamo v obravnavanem izračunu pogoj, ki določa, ali je za določene odločitve izvajanje uspelo. Na ta način implementiramo evaluator `amb` [1], ki izbere eno od množic odločitev, za katere izvajanje uspe. Najprej definiramo tip učinka z operacijo `select`, ki predstavlja izbiro elementa iz poljubnega seznama.

```
type 'a selection = effect
  operation select: 'a list -> 'a
end
```

Prestreznik, ki to operacijo implementira na opisan način, je prikazan spodaj. Člen `select` pokliče kontinuacijo za vsak element iz seznama možnosti po vrsti, dokler izračun ne uspe. Če so vse možnosti neuspešne, vrne `None`.

```
let amb s = handler
  | s#select lst k ->
      let rec try = function
        | [] -> None
        | x::xs ->
            (match k x with
               | None -> try xs
               | Some result -> Some result)
      in
      try lst
```

Uporabo tega prestreznika si ogledamo na problemu osmih dam. Ta sprašuje po taki postavitvi osmih dam na šahovnico, da se nobeni dve med sabo ne napadata. Najprej definiramo predikat `no_attack`, ki za dami na koordinatah

`(x,y)` in `(x',y')` preveri, ali sta v neveljavnem položaju, torej v isti vrstici, stolpcu ali diagonali.

```
let no_attack (x,y) (x',y') =
  x <> x' && y <> y' && abs (x - x') <> abs (y - y')
```

Dame postavljamo na šahovnico po stolpcih oziroma po naraščajoči koordinati `x` in za vsako shranimo pripadajoč `y`. Seznam postavljenih dam `[4;6;8]` tako pomeni, da smo za prve tri izbrali koordinate `(1,4)`, `(2,6)` in `(3,8)`. Funkcija `available` za podan seznam vrne veljavne možnosti za koordinato `y` dame v naslednjem stolpcu.

```
let available x qs =
  filter (fun y -> forall (no_attack (x,y)) qs)
         [1;2;3;4;5;6;7;8]
```

Z uporabo prestreznika `amb` je izračun, ki reši ta problem, dokaj preprost. Pomožna funkcija `place` sprejme seznam `qs` dam, ki so že na šahovnici, in poskusi postaviti naslednjo v stolpec `x`. Ko mu uspe postaviti vseh osem, kot rešitev vrne seznam koordinat. Izračun začnemo v prvem stolpcu s praznim seznamom.

```
let s = new selection in
  with amb s handle
    let rec place x qs =
      if x = 9 then Some qs else
        let y = s#select (available x qs) in
        place (x+1) ((x,y) :: qs)
    in place 1 []
```

Z uporabo takih nedeterminističnih izračunov lahko enostavno napišemo rešitve za vrsto podobnih problemov, kot so sudoku in križanke. Zmeraj lahko uporabimo isti prestreznik, ki vsebuje celotni algoritem za preiskovanje prostora rešitev. Po drugi strani pa nam to omogoča, da za te izračune obravnavamo z drugačnim iskalnim algoritmom. Prestreznik `amb` uporablja iskanje v globino; za iskanje v širino lahko uporabimo prestreznik

```
let bfs s =
  let q = ref [] in
  handler
    | s#select lst k ->
        (q := !q @ (map (fun x -> (k,x)) lst) ;
          match !q with
```

```
        | [] -> None
        | (k,x) :: lst -> q := lst ; k x).
```

Tukaj `q` predstavlja fronto iskanja v širino, to je vrsto odločitvenih točk, ki jih še moramo preveriti. Operacija `select` v vrsto najprej doda možnosti za naslednji korak kot pare `(k,x)`, kjer je `k` kontinuacija in `x` argument, s katerim se naj pokliče. Nato vzame prvi element iz vrste in ga izvede.

3.5.1 Sestopanje s sledenjem

Prestreznika iz prejšnjega razdelka poiščeta rešitev – če ta obstaja – in jo vrneta. Včasih pa nas poleg rešitve zanimajo tudi odločitve, ki so do nje pripeljale. V tem primeru najprej operaciji odločitve dodamo parameter, s katerim lahko označimo posamezne odločitve.

```
type ('a,'b) selection = effect
  operation select: 'a * 'b list -> 'b
end
```

Želimo, da si prestreznik zapomni odločitve za posamezne oznake; če imata dve odločitveni točki v izračunu enako oznako, naj obakrat izbere isto možnost. Poleg tega mu kot parameter podamo vrednost `v`, ki jo mora doseči uspešna evaluacija. Prestreznik generira možne nabore odločitev in sprejme prvega, pri katerem se rezultat izračuna ujema s podano vrednostjo.

```
let select s v = handler
  | s#select (tag,ys) k -> (fun cs ->
      (match assoc tag cs with
         | Some y -> k y cs
         | None ->
             let rec try = function
               | [] -> None
               | y::ys ->
                   (match k y ((tag,y)::cs) with
                      | Some lst -> Some lst
                      | None -> try ys)
             in try ys))
  | val u -> (fun cs -> if u = v then Some cs else None)
  | finally f -> f []
```

Prestreznik vsako operacijo pretvori v funkcijo, ki sprejme asociativni seznam trenutno aktivnih odločitev. Elementi tega seznama so pari oblike ključ/vrednost, funkcija `assoc` pa za dan ključ vrne pripadajočo vrednost (ali `None`, če v

seznamu ne obstaja element s tem ključem). Operacijski člen najprej preveri, če je za odločitveno točko že shranjena izbira; v tem primeru preprosto pokliče kontinuacijo z isto izbiro. Sicer nadaljuje enako kot prestreznik `amb` in pokliče kontinuacijo z vsako možno izbiro, dokler ena ne uspe. Člen `val` preveri, ali končna vrednost evaluacije ustreza želeni in v tem primeru vrne seznam izbir, ki so pripeljale do tega rezultata. Člen `finally` pa kot v prejšnjih primerih, ki uporabljajo ta pristop, na koncu pokliče tako konstruirano funkcijo s praznim seznamom aktivnih izbir.

Delovanje tega prestreznika lahko preizkusimo na izračunu, ki poišče pitagorejsko trojico.

```
let s = new selection in
with select s true handle
  let a = s#select ("a", [5;6;7;8]) in
  let b = s#select ("b", [9;10;11;12]) in
  let c = s#select ("c", [13;14;15;16]) in
    a*a + b*b = c*c
```

Rezultat tega programa je `Some [("c",13); ("b",12); ("a",5)]`.

Poglavje 4

Opozorila za prirejanja vzorcev

Ena od značilnosti, ki jih eff deli s programskimi jeziki družine ML, je analiza primerov s prirejanjem vzorcev (angl. *pattern matching*). Vzorec je struktura, ki opisuje množico vrednosti. Osnovna primera vzorcev sta `4`, ki predstavlja množico $\{4\}$ z eno samo vrednostjo, in `_` (joker ali angl. *wildcard*), ki predstavlja vse možne vrednosti. Vzorce lahko gnezdimo in jih tako naredimo poljubno kompleksne. Vzorec `(2, _, "foo")` tako opisuje množico vseh urejenih trojk, ki imajo na prvem mestu vrednost 2 in na zadnjem niz `foo`.

Analiza primerov je izraz, ki vsebuje seznam vzorcev $p_1, \ldots, p_n$ in pripadajočih podizrazov $e_1, \ldots, e_n$. Kanoničen primer je stavek *match*, ki je v splošnem sledeče oblike.

```
match e with
  | p1 → e1
  | p2 → e2
  ...
  | pn → en
```

Pri izvajanju se poišče prvi vzorec p_i, s katerim se ujema izraz e. Rezultat analize je potem izraz e_i. Če se e ne ujema z nobenim od podanih vzorcev, pride do napake med izvajanjem, zaradi česar je pomembno, da je seznam vzorcev $p_1, \ldots, p_n$ *izčrpen*. To pomeni, da vsako možno vrednost, ki jo lahko ima izraz e, pokriva vsaj en vzorec. Koristno je torej, če lahko prevajalnik opozori programerja, da je izpustil kakšno možnost. Prav tako bi radi vedeli, če je kakšen vzorec p_i *nekoristen* – vsaka vrednost, ki jo pokriva, se ujema z vsaj enim od vzorcev $p_1, \ldots, p_{i-1}$.

V tem poglavju najprej opišemo predstavitev vrednosti in vzorcev, ki jo bomo uporabljali, in formaliziramo pojma izčrpnega in nekoristnega vzorca. Nato predstavimo algoritem [5], ki odkrije tovrstne napake, dokažemo njegovo

pravilnost in opišemo implementacijo za programski jezik eff. Podamo tudi specializirano verzijo algoritma za ugotavljanje izčrpnosti, ki poišče primer vrednosti, ki je ne pokrije noben vzorec v seznamu.

4.1 Predstavitev vrednosti in vzorcev

Vsak podatkovni tip v effu je definiran z množico konstruktorjev, ki lahko imajo različno število argumentov. Množici konstruktorjev s tipi pripadajočih argumentov pravimo tudi *podpis* podatkovnega tipa. Primer tipa je dvojiško drevo, rekurzivno definirano s podpisom

```
type tree = Leaf | Node of tree * tree.
```

Ta definicija pove, da tip `tree` vsebuje dva konstruktorja: `Leaf` z nič argumenti in `Node` z dvema argumentoma istega tipa. Konstruktorjem, ki ne sprejmejo nobenega argumenta, pravimo konstante. V tabeli 4.1 je podanih nekaj primerov tipov, predstavljenih na tak način.

boolove vrednosti	`type bool = False | True`
cela števila	`type int = 0 | 1 | -1 | 2 | -2 | ...`
seznami števil	`type list = Empty | Cons of int * list`
n-terice števil	`type tuple`$_n$ `= Tuple`$_n$ `of` $\underbrace{\texttt{int} * \cdots * \texttt{int}}_{n}$

Tabela 4.1: Predstavitev nekaterih podatkovnih tipov

Množica konstruktorjev nekega tipa je lahko neskončna. To nam omogoča, da recimo tip `int` predstavimo z množico konstant, ki ustrezajo celim številom. Enako velja tudi za tip `float` realnih števil in tip `string` nizov znakov. Velja omeniti, da noben od teh tipov ni v resnici neskončen – tip `float` je interno predstavljen s štirimi bajti in je tako omejen na 2^{32} možnih vrednosti. Tip celih števil je prav tako običajno predstavljen s štirimi ali osmimi bajti. Število različnih vrednosti tipa `string` pa je omejeno s količino spomina, ki je na voljo izvajalnemu okolju.

Takšna predstavitev tipov je poenostavljena, saj med drugim ne opisuje parametričnega polimorfizma [5]. Kljub temu zadošča za naše potrebe v tem poglavju in ima to prednost, da lahko vsako vrednost predstavimo na uniformen način z aplikacijo konstruktorja ustreznega tipa na n-terico argumentov:

$$v ::= c(v_1, v_2, \ldots, v_n).$$

Skozi to poglavje bomo predpostavili, da za vsak tip obstaja vsaj ena vrednost s tem tipom. Za eff to sicer ne velja povsem, saj definira prazen tip `empty`, ki ga potrebuje za opis izjem. Praktična posledica tega je, da naš algoritem v določenih primerih vrne neveljavna opozorila; to je bolj podrobno opisano v razdelku o implementaciji. Ker predpostavka velja za vse tipe, ki jih lahko določi uporabnik, so tako primeri redki. Prav tako se ne more zgoditi, da algoritem ne bi vrnil opozorila, ko bi ga moral. Ta pomanjkljivost zato ni preveč omejujoča.

Vzorci opisujejo skupne značilnosti množic vrednosti in so lahko *enostavni* (joker) ali *sestavljeni* (aplikacija konstruktorja na n-terico argumentov oziroma podvzorcev). Vzorci, ki predstavljajo konstante, imajo konstruktor brez argumentov in jih štejemo med sestavljene. Formalno vzorce opišemo s produkcijo

$$p ::= \_ \mid c(p_1, p_2, \ldots, p_n).$$

Vzorci običajno vsebujejo tudi spremenljivke. Množica vrednosti, ki ustrezajo nekemu vzorcu, se ne spremeni, če vse spremenljivke v njem zamenjamo z jokerji. Zaradi tega jih obravnavamo na enak način.

Poleg tega v nadaljevanju predpostavimo, da se tipi vzorcev in vrednosti, ki jih primerjamo, ujemajo. Konkretno to pomeni, da vzorec _ ne pokriva *vseh* vrednosti, temveč le tiste, ki so istega tipa. V dejanski implementaciji se naš algoritem požene šele po tem, ko se izpeljejo in preverijo tipi izrazov v programu, zato je ta predpostavka smiselna. Elementom množice vrednosti, ki jo opisuje nek vzorec, pravimo tudi *instance* tega vzorca. Ta pojem opišemo z relacijo $\preceq$, ki jo definiramo rekurzivno.

Definicija 1. Naj bosta vzorec p in vrednost v istega tipa. Relacija $p \preceq v$ je potem induktivno definirana s pravili:

$$\begin{aligned} \_ &\preceq v \\ c(p_1, \ldots, p_n) \preceq c(v_1, \ldots, v_n) \quad &\Leftrightarrow \quad (p_1, \ldots, p_n) \preceq (v_1, \ldots, v_n) \\ (p_1, \ldots, p_n) \preceq (v_1, \ldots, v_n) \quad &\Leftrightarrow \quad \forall i \in \{1, \ldots, n\} : p_i \preceq v_i \end{aligned}$$

Zadnja vrstica definira relacijo $\preceq$ na vektorjih vzorcev in vrednosti. S pomočjo tega razširimo definicijo relacije $\preceq$ tudi na matrike vzorcev. Matrika vzorcev predstavlja unijo množic (vektorjev) vrednosti, ki jih opisujejo njene vrstice. Instance matrike vzorcev P so torej vsi vektorji vrednosti, ki jih pokrije kakšna izmed vrstic v P.

Definicija 2. Naj bo P matrika vzorcev z m vrsticami in n stolpci ter $\vec{v}$ vektor vrednosti dolžine n. Potem velja:

$$P \preceq \vec{v} \Leftrightarrow \exists i : P^i \preceq \vec{v}$$

Analiza primerov v `match` stavkih poteka od prvega do zadnjega vzorca in se konča takrat, ko naleti na vzorec, s katerim se ujema vhodna vrednost. To lahko na naraven način predstavimo z matriko vzorcev P tako, da vsakemu primeru priredimo vrstico v P. Potem pravimo, da i-ta vrstica *filtrira* vektor vrednosti $\vec{v}$ natanko takrat, ko je $\vec{v}$ instanca i-te vrstice in nobene pred njo; matrika P filtrira vektor $\vec{v}$, če ga filtrira ena izmed njenih vrstic. Matrika vzorcev P ima v taki predstavitvi zmeraj natanko en stolpec, saj eff vektorje vzorcev in vrednosti v analizi primerov avtomatično ovije v konstruktor za n-terice. Za primer vzemimo naslednji `match` stavek.

```
match 1,"foo" with
  | 1,_ -> ...
  | _,"foo" -> ...
  | _ -> ...
```

Ustreza mu matrika vzorcev P velikosti 3×1, ki izgleda tako:

$$P = \begin{bmatrix} \texttt{(1,\_)} \\ \texttt{(\_,"foo")} \\ \texttt{\_} \end{bmatrix}$$

V uvodu opisani napaki pri analizi primerov lahko sedaj izrazimo z uporabo matrik in vektorjev vzorcev.

Definicija 3. Matrika vzorcev P je *izčrpna* natanko tedaj, ko vsako vrednost ustreznega tipa filtrira vsaj ena vrstica v P.

Definicija 4. V matriki vzorcev P je i-ta vrstica *nekoristna* natanko tedaj, ko vsako njeno instanco filtrira vsaj ena vrstica, ki je v P pred njo.

Izkaže se, da se splača razmišljati o *koristnih* (angl. *useful*) vektorjih vzorcev, saj lahko z uporabo tega pojma ti definiciji poenotimo. Vektor vzorcev $\vec{q}$ dolžine n je koristen za matriko vzorcev P velikosti $m \times n$, če pokriva vsaj kakšno vrednost, ki ni instanca nobene vrstice v P. Na tem mestu še enkrat poudarimo predpostavko, da se tipi vrstic v P in vektorjev $\vec{q}$ ter $\vec{v}$ ujemajo, in definiramo predikat $\mathcal{U}$, ki zajame pojem koristnosti:

$$\mathcal{U}(P, \vec{q}) = \exists \vec{v} : P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v}.$$

Sedaj lahko definiciji 3 in 4 izrazimo s pomočjo predikata $\mathcal{U}$ na sledeč način. Matrika P je izčrpna natanko tedaj, ko ne velja $\mathcal{U}(P, (\_ \cdots \_))$. Vektor samih jokerjev je namreč koristen za vsako matriko z istim številom stolpcev, ki ne pokriva vseh možnih vrednosti. Prav tako je i-ta vrstica P^i nekoristna natanko tedaj, ko ne velja $\mathcal{U}(P^{[1,\ldots,i)}, P^i)$, kjer je $P^{[1,\ldots,i)}$ matrika prvih $i-1$ vrstic iz P.

4.2 Algoritem za preverjanje koristnosti

V tem razdelku predstavimo algoritem useful [5], ki izračuna, ali je vektor vzorcev $\vec{q}$ dolžine n koristen za dano matriko vzorcev P velikosti $m \times n$. Kot smo videli, lahko s pomočjo takega algoritma in primernimi izbirami vhodnih vektorjev preverimo, če je dana matrika vzorcev izčrpna in ali vsebuje nekoristne vrstice. Postopek poteka rekurzivno po stolpcih vektorja $\vec{q}$ in matrike P ter poskuša najti instanco vektorja $\vec{q}$, ki je P ne filtrira.

Osnovni primer ($n = 0$). Obstaja samo en prazen vektor vrednosti. Matrika velikosti $m \times 0$ ga zagotovo filtrira (in torej pokriva vse možne vrednosti), če ima vsaj eno vrstico. Povedano drugače: vektor $\vec{q}$ je koristen za matriko P natanko tedaj, ko je $m = 0$. Tako definiramo

$$\mathsf{useful}(P, ()) = (m = 0).$$

Indukcija ($n > 0$). Če imata P in $\vec{q}$ vsaj en stolpec, se rekurzivni klic izvrši glede na to, ali je prvi vzorec q_1 iz $\vec{q}$ sestavljen ali enostaven. Posebej obravnavamo ti dve možnosti.

1. Vzorec q_1 je sestavljen, torej je oblike $q_1 = c(r_1, \ldots, r_a)$. Vse instance vektorja vzorcev $\vec{q}$ imajo na prvem mestu konstruktor c, zato jih lahko v matriki P filtrirajo le tiste vrstice, ki imajo na prvem mestu vzorec s tem konstruktorjem ali joker. Definiramo *specializirano* matriko $\mathcal{S}(c, P)$, kjer ohranimo le take vrstice iz P in jim prvi vzorec zamenjamo s seznamom njegovih argumentov. Spodaj je prikazana specializacija i-te vrstice glede na prvi vzorec v tej vrstici.

$$\mathcal{S}(c,P)^i = \begin{cases} (r_1, \ldots, r_a, P_2^i, \ldots, P_n^i) & \text{če } P_1^i = c(r_1, \ldots, r_a) \\ (\underbrace{\_, \ldots, \_}_{a}, P_2^i, \ldots, P_n^i) & \text{če } P_1^i = \_ \\ \text{ni vrstice} & \text{sicer} \end{cases}$$

   Na enak način definiramo tudi specializacijo za vektor vzorcev ali vrednosti. Ta ima smisel le takrat, ko je prva komponenta vektorja sestavljena s konstruktorjem c (prvi primer) ali enostavni vzorec (drugi primer). Problem koristnosti vektorja $\vec{q}$ za matriko P lahko s pomočjo teh definicij poenostavimo:

$$\mathsf{useful}(P, \vec{q}) = \mathsf{useful}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q})).$$

Specializirana matrika in vektor vzorcev imata $a+n-1$ stolpcev, kjer je a število argumentov konstruktorja c. Velikost vhodnih podatkov se zato pri tem klicu lahko poveča – zmanjša se le, če je c konstanta, ki ne sprejme nobenega argumenta. Zmeraj pa se zmanjša strukturna kompleksnost vektorja $\vec{q}$ in vrstic v P, zaradi česar je problem na desni enostavnejši.

2. Sicer je vzorec q_1 enostaven oziroma joker. Označimo njegov tip s t in množico vseh konstruktorjev tega tipa s Σ_t. Če obstaja kakšna instanca vektorja vzorcev $\vec{q}$, ki je matrika P ne filtrira, mora na prvem mestu imeti konstruktor iz Σ_t. Zato preverimo koristnost vektorja $\vec{q}$ za vsako specializacijo matrike P za te konstruktorje. Če je za nek $c \in \Sigma_t$ vektor $\mathcal{S}(c, \vec{q})$ koristen za matriko $\mathcal{S}(c, P)$, je tudi $\vec{q}$ koristen za P oziroma

$$\mathsf{useful}(P, \vec{q}) = \bigvee_{c \in \Sigma_t} \mathsf{useful}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q})).$$

Recimo, da obstaja konstruktor $c \in \Sigma_t$, ki se ne pojavi kot korenski konstruktor v prvem stolpcu matrike P. Instance vektorja $\vec{q}$ s tem konstruktorjem v prvi komponenti lahko torej filtrirajo le tiste vrstice iz P, ki imajo na prvem mestu jokerja. Zato je dovolj, če preverimo koristnost le za te vrstice. Definiramo *privzeto* matriko $\mathcal{D}(P)$, kjer ohranimo le take vrstice in jim odstranimo prvo komponento.

$$\mathcal{D}(P)^i = \begin{cases} (P_2^i, \ldots, P_n^i) & \text{če } P_1^i = \_ \\ \text{ni vrstice} & \text{sicer} \end{cases}$$

Posebni primer, ko prvi stolpec v P ne vsebuje vseh konstruktorjev iz Σ_t, lahko obravnavamo učinkoviteje z uporabo samo enega rekurzivnega klica

$$\mathsf{useful}(P, \vec{q}) = \mathsf{useful}(\mathcal{D}(P), (q_2, \ldots, q_n)).$$

4.2.1 Pravilnost

Pokazati želimo, da algoritem useful, definiran v prejšnjem razdelku, pravilno izračuna vrednost predikata $\mathcal{U}$ za vse veljavne vhodne podatke – to so pari matrik in vektorjev vzorcev enake širine, ki se ujemajo v tipih istoležnih stolpcev. Pred tem vpeljemo naslednjo pomožno trditev o instancah specializiranih in privzetih matrik.

Lema 1. *Za vsak vektor vrednosti* $\vec{v} = (c(w_1, \ldots, w_a), v_2, \ldots, v_n)$ *in matriko vzorcev* P *z ujemajočimi tipi istoležnih stolpcev velja*

$$P \preceq \vec{v} \Leftrightarrow \mathcal{S}(c, P) \preceq \mathcal{S}(c, \vec{v}). \tag{4.1}$$

Prav tako za vsak vektor vrednosti $\vec{v} = (v_1, v_2, \ldots, v_n)$ *velja*

$$P \not\preceq (v_1, v_2, \ldots, v_n) \Rightarrow \mathcal{D}(P) \not\preceq (v_2, \ldots, v_n). \tag{4.2}$$

Dokaz. Dokaz sledi neposredno iz definicij, zato tukaj za zgled dokažemo samo prvo točko. Vektor $\vec{v}$ je instanca matrike P natanko tedaj, ko P vsebuje vrstico P^i, za katero velja

$$P^i = (c(r_1, \ldots, r_a), p_2, \ldots, p_n) \preceq (c(w_1, \ldots, w_a), v_2, \ldots, v_n).$$

Uporabimo definiciji relacije $\preceq$ in funkcije $\mathcal{S}$, da dobimo ekvivalentno izjavo

$$\mathcal{S}(c, P^i) = (r_1, \ldots, r_a, p_2, \ldots, p_n) \preceq (w_1, \ldots, w_a, v_2, \ldots, v_n) = \mathcal{S}(c, \vec{v}).$$

Specializirana matrika $\mathcal{S}(c, P)$ torej filtrira vrednost $\mathcal{S}(c, \vec{v})$ natanko tedaj, ko P filtrira $\vec{v}$. Dokaz za drugo točko poteka podobno. □

S pomočjo te leme sedaj dokažemo naslednjo trditev, ki nam zagotavlja, da algoritem useful pravilno določi koristnost vektorja vzorcev za dano matriko.

Trditev 1. *Za vsako matriko vzorcev* P *in vektor vzorcev* $\vec{q}$*, ki se ujemata po številu in tipih stolpcev, velja*

$$\mathcal{U}(P, \vec{q}) = \mathsf{useful}(P, \vec{q}).$$

Dokaz. Če je $\vec{q}$ prazen vektor vzorcev, ima natanko eno instanco $\vec{v} = ()$ oziroma prazen vektor vrednosti. Po predpostavki so vrstice matrike P prav tako prazne, zato vsaka vrstica filtrira $\vec{v}$. $\mathcal{U}(P, \vec{q})$ torej velja natanko tedaj, ko P nima nobene vrstice, kar pa je ravno definicija algoritma useful za osnovni primer.

Če ima $\vec{q}$ vsaj en stolpec, moramo dokazati, da rekurzivni klici, ki definirajo algoritem useful, ohranjajo resničnost predikata $\mathcal{U}$. Spet obravnavamo ločeno primera, ko je prvi vzorec q_1 iz $\vec{q}$ sestavljen ali enostaven.

1. Če je $q_1 = c(r_1, \ldots, r_a)$, potem uporabimo prvo točko leme 1 za P in $\vec{q}$, da pokažemo ekvivalenco:

$$\begin{aligned}\mathcal{U}(P, \vec{q}) &\Leftrightarrow \exists \vec{v} = (c(w_1, \ldots, w_a), v_2, \ldots, v_n) : P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v} \\ &\Leftrightarrow \exists \vec{v} : \mathcal{S}(c, P) \not\preceq \mathcal{S}(c, \vec{v}) \wedge \mathcal{S}(c, \vec{q}) \preceq \mathcal{S}(c, \vec{v}) \\ &\Leftrightarrow \mathcal{U}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q})).\end{aligned}$$

2. Če je vzorec q_1 joker tipa t, označimo s Σ_t množico vseh konstruktorjev tipa t in s Σ množico vseh korenskih konstruktorjev tega tipa, ki se pojavijo v prvem stolpcu matrike P. Glede na to, ali je $\Sigma = \Sigma_t$, ločimo dva primera.

   (a) $\Sigma = \Sigma_t$, torej se vsak konstruktor tipa t vsaj enkrat pojavi kot korenski konstruktor v prvem stolpcu matrike P. Dokaz poteka praktično enako kot za prvo točko:

$$\begin{aligned}\mathcal{U}(P, \vec{q}) &\Leftrightarrow \exists \vec{v} = (c(w_1, \ldots, w_a), v_2, \ldots, v_n) : P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v} \\ &\Leftrightarrow \exists \vec{v} : \mathcal{S}(c, P) \not\preceq \mathcal{S}(c, \vec{v}) \wedge \mathcal{S}(c, \vec{q}) \preceq \mathcal{S}(c, \vec{v}) \\ &\Leftrightarrow \exists c \in \Sigma_t : \mathcal{U}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q})) \\ &\Leftrightarrow \bigvee_{c_i \in \Sigma_t} \mathcal{U}(\mathcal{S}(c_i, P), \mathcal{S}(c_i, \vec{q})).\end{aligned}$$

   (b) $\Sigma_t - \Sigma \neq \emptyset$, torej obstaja konstruktor $c \in \Sigma_t$, ki se ne pojavi kot korenski konstruktor v prvem stolpcu matrike P. Pokazati moramo, da velja naslednja ekvivalenca:

$$\mathcal{U}(P, (q_1, q_2, \ldots, q_n)) \Leftrightarrow \mathcal{U}(\mathcal{D}(P), (q_2, \ldots, q_n)).$$

   Implikacija v desno smer je lažja, zato jo pokažemo najprej. Če velja $\mathcal{U}(P, \vec{q})$, mora obstajati vrednost $\vec{v} = (v_1, v_2, \ldots, v_n)$, da

$$P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v}.$$

   Uporabimo drugo točko leme in definicijo relacije $\preceq$, da dobimo

$$\mathcal{D}(P) \not\preceq (v_2, \ldots, v_n) \wedge (q_2, \ldots, q_n) \preceq (v_2, \ldots, v_n).$$

   To pa je ravno definicija predikata $\mathcal{U}$ za privzeto matriko $\mathcal{D}(P)$ in vektor $(q_2, \ldots, q_n)$ in smo končali.
   Sedaj pokažemo še implikacijo v drugo smer. Predpostavimo torej, da velja $\mathcal{U}(\mathcal{D}(P), (q_2, \ldots, q_n))$. To pomeni, da obstaja tak vektor vrednosti $\vec{v'} = (v_2, \ldots, v_n)$, da

$$\mathcal{D}(P) \not\preceq \vec{v'} \wedge (q_2, \ldots, q_n) \preceq \vec{v'}.$$

   Po predpostavki obstaja konstruktor $c \notin \Sigma$ tipa t, ki se ne pojavi v prvem stolpcu matrike P. Prav tako smo predpostavili, da ima vsak tip vsaj eno vrednost, zato mora obstajati tudi vrednost

$v_1 = c(w_1, \ldots, w_a)$ tipa t. Vrednost $\vec{v} = (v_1, \ldots, v_n)$ lahko filtrirajo kvečjemu tiste vrstice iz P, ki imajo v prvi komponenti vzorec s konstruktorjem c ali jokerja. Ker se c ne pojavi kot korenski konstruktor v prvem stolpcu P in $\mathcal{D}(P)$ ne filtrira vrednosti $(v_2, \ldots, v_n)$, velja $P \not\preceq \vec{v}$. Ker $(q_2, \ldots, q_n)$ filtrira $(v_2, \ldots, v_n)$ in $q_1 = \_$, lahko zaključimo

$$\exists \vec{v} : P \not\preceq \vec{v} \wedge \vec{q} \preceq \vec{v} \Rightarrow \mathcal{U}(P, \vec{q}).$$

□

4.2.2 Preverjanje izčrpnosti s protiprimeri

Opozorilo o izpuščenih možnostih pri analizi primerov se da izboljšati tako, da izpišemo tudi primer nepokrite vrednosti. To uporabniku omogoči, da hitreje odkrije in popravi napako. Tu predstavimo različico algoritma useful, ki preveri, če je dana matrika izčrpna. Nov algoritem exhaustive sprejme matriko P z n stolpci in deluje na podoben način kot useful, le da za vektor vzorcev $\vec{q}$ zmeraj vzame vektor n-tih jokerjev. Če je matrika P izčrpna, je rezultat algoritma `None`, sicer pa `Some` p, kjer p opisuje podmnožico vrednosti, ki jih P ne pokriva.

Osnovni primer ($n = 0$). Če matrika P vsebuje vsaj eno vrstico, vrnemo `None`, sicer pa `Some []`, ki je prazen vektor vzorcev z eno samo instanco ().

Indukcija ($n > 0$). V tem primeru preverjamo koristnost vektorja vzorcev $\vec{q} = (\_, \ldots, \_)$ za dano matriko. Prva komponenta q_1 je torej zmeraj joker, kar ustreza drugi točki induktivnega koraka algoritma useful. Spet označimo s t tip vzorca q_1 in s Σ množico vseh korenskih konstruktorjev v prvem stolpcu matrike P. Glede na to, ali Σ vsebuje vse konstruktorje tipa t, obravnavamo dva primera.

1. Če Σ vsebuje vse konstruktorje tipa t, preverimo izčrpnost specializacije P za vsak $c \in \Sigma$. Če vsi rekurzivni klici exhaustive$(\mathcal{S}(c, P))$ vrnejo `None`, je P izčrpna in algoritem vrne `None`. Sicer lahko izvajanje končamo ob prvem klicu, ki za nek c vrne vektor vzorcev $(r_1, \ldots, r_a, p_2, \ldots, p_n)$. Iz tega dobimo primer nepokritega vektorja tako, da prvih a komponent ovijemo s konstruktorjem c. Rezultat algoritma je potem `Some` $(c(r_1, \ldots, r_a), p_2, \ldots, p_n)$.

2. Če Σ ne vsebuje vseh konstruktorjev tipa t, moramo preveriti samo privzeto matriko $\mathcal{D}(P)$. Če je privzeta matrika izčrpna, je izčrpna tudi matrika P in vrnemo `None`. Sicer klic exhaustive$(\mathcal{D}(P))$ vrne vektor vzorcev

$(p_2, \ldots, p_n)$. V tem primeru vrnemo `Some` $(c(\_, \ldots, \_), p_2, \ldots, p_n)$ za poljubno izbran $c \notin \Sigma$. V posebnem primeru, ko je Σ prazna množica, pa lahko vrnemo tudi `Some` $(\_, p_2, \ldots, p_n)$ z namenom, da bo končni rezultat obsegal čim širšo množico vrednosti.

4.3 Implementacija

V tem razdelku predstavimo nekaj značilnosti konkretne implementacije algoritmov useful in exhaustive za programski jezik eff. Konstruktorji vzorcev so opisani z naslednjim podatkovnim tipom.

```
type ctor =
  | Tuple of int
  | Record of string list
  | Variant of string * bool
  | Const of const
  | Wildcard
```

Konstruktorje n-teric opišemo s številom komponent n, pri zapisih pa shranimo seznam imen posameznih polj. Pri vzorcih zapisov programerju ni treba določiti vseh polj, navede pa jih lahko v poljubnem vrstnem redu. Ko mora algoritem specializirati vzorec zapisa, postavi na manjkajoča mesta jokerje, vrstni red polj pa določi glede na vrstni red v definiciji tipa zapisa. Za primer vzemimo zapis, definiran z `{a:int; b:int; c:int}` in njegov konstruktor označimo s c. Vzorca `{b=1}` in `{a=_; b=1; c=_}` potem predstavljata isto množico vrednosti tega tipa. Spodaj je prikazan primer specializacije matrike za konstruktor c.

$$P = \begin{bmatrix} \texttt{\{a=1\}} & \ldots \\ \texttt{\{b=\_; c=3\}} & \ldots \\ \texttt{\{c=3; b=2; a=4\}} & \ldots \end{bmatrix} \rightarrow \mathcal{S}(c, P) = \begin{bmatrix} \texttt{1} & \texttt{\_} & \texttt{\_} & \ldots \\ \texttt{\_} & \texttt{\_} & \texttt{3} & \ldots \\ \texttt{4} & \texttt{2} & \texttt{3} & \ldots \end{bmatrix}$$

Na ta način zagotovimo, da se tipi oziroma pomeni komponent v istem stolpcu ujemajo. Iz specializirane vrstice in konstruktorja c je zato možno dobiti nazaj vrstico, ki je ekvivalentna originalni. To je pomembno za algoritem exhaustive, ko gradi protiprimer za izčrpnost.

Konstruktorje vsot predstavimo z imenom oznake in podatkom o tem, ali konstruktor sprejme argument. Za primer vzamemo tip seznama števil, ki ga lahko implementiramo z vsoto

```
type list = Nil | Cons of int * list.
```

V tem primeru prvi konstruktor opišemo z `Variant ("Nil", false)`, drugega pa z `Variant ("Cons", true)`. Vgrajeni seznami v effu so v resnici predstavljeni na podoben način z vsotami, zato jih ni potrebno obravnavati posebej.

Primitivne konstante v effu so boolove vrednosti, nizi znakov in cela ter realna števila. Vse njihove možne vrednosti zajame podatkovni tip `const`, prikazan spodaj, ki uporablja OCamlove vgrajene tipe. Konstantni konstruktorji so opisani preprosto z vrednostjo, ki jo predstavljajo in ne sprejmejo nobenega argumenta.

```
type const =
  | Integer of int
  | String of string
  | Boolean of bool
  | Float of float
```

Vektorji vzorcev so shranjeni kot seznami, matrike pa kot seznami seznamov. Matrika brez vrstic je tako predstavljena kot prazen seznam, iz katerega ni mogoče določiti števila njenih stolpcev. Algoritem useful$(P, \vec{q})$ zato predpostavi, da se matrika P po številu in tipih stolpcev vedno ujema z vektorjem vzorcev $\vec{q}$. Algoritmu exhaustive(P, n) pa moramo zmeraj podati število stolpcev kot parameter n, s pomočjo katerega lahko določi, ali je bil dosežen osnovni primer tudi takrat, ko P ne vsebuje nobene vrstice.

V effu sta algoritma useful in exhaustive implementirana v modulu `check`. Da bi za seznam vzorcev dolžine m preverili izčrpnost in koristnost posameznih vzorcev v njem, pokličemo funkcijo `check_patterns`. Ta začne s prazno matriko P. Za vsak vzorec $\vec{q}$ iz vhodnega seznama pokliče useful$(P, \vec{q})$, ki preveri njegovo koristnost glede na trenutno vsebino matrike P. Če je $\vec{q}$ koristen, ga doda v P, sicer izpiše opozorilo in nadaljuje z nespremenjeno matriko. Ko doseže konec seznama, pokriva matrika P natanko iste vrednosti kot vzorci v vhodnem seznamu. Na koncu preveri še izčrpnost tako dobljene matrike s klicem exhaustive$(P, 1)$. Rezultat je enak, kot če bi preverjali izčrpnost originalnega seznama, saj so bili izpuščeni le nekoristni vzorci.

Predpogoj za oba algoritma je, da so tipi vzorcev izračunani in da so vzorci v istem stolpcu enakih tipov. Nekoristnost in izčrpnost v effu zato preverjamo v modulu `infer` takoj za tem, ko se določijo tipi. Poleg stavkov `match` preverjamo tudi izčrpnost posameznih vzorcev v stavkih oblike

`let` p `=` e`,`

kjer želimo zagotoviti, da se vzorec p lahko prilagodi vsem možnim vrednostim izraza e. Za ta namen lahko pokličemo funkcijo `is_irrefutable`, ki je enakovredna klicu `check_patterns` na seznamu `[`p`]` z enim samim vzorcem.

Pri samem izpisu opozoril smo se zgledovali po OCamlu. Nekaj primerov opozoril je prikazanih v dodatku A, kjer predstavimo tudi rezultate analize primerov kode v effu, ki so priloženi prototipni implementaciji tolmača.

Poglavje 5

Zaključki in nadaljnje delo

V prvem delu diplomske naloge smo najprej predstavili programski jezik eff. Pri tem smo posebno pozornost posvetili konstruktom, s katerimi obravnava računske učinke. Na raznih primerih smo nato pokazali, kako eff implementira vhod/izhod, izjeme in reference, poleg tega pa še, kako lahko s pomočjo lastnih učinkov in prestreznikov tipične probleme rešujemo na nove načine. Osredotočili smo se na uporabo nedeterminizma za lažje obvladovanje problemov, ki zahtevajo rekurzivno preiskovanje.

V drugem delu smo opisali rešitev praktičnega problema generiranja opozoril o neizčrpnosti obravnave primerov. Predstavili smo uporabljen algoritem in dokazali njegovo pravilnost. Naša implementacija prevede algoritem v programsko kodo bolj ali manj neposredno. Prednost tega je, da je preprosta in lahko razumljiva, saj ne vnaša dodatne kompleksnosti, še zmeraj pa pravilno zazna morebitne napake pri pisanju vzorcev. Naš cilj je torej bil dosežen. V nadaljevanju opišemo omejitve naše implementacije in možnosti za razširitve ter nadaljnje delo.

Čeprav iskanje protiprimerov v primeru napake ni nujno, lahko programerju na ta način močno olajšamo delo. Temu ustrezno naša rešitev zmeraj poišče vrednost, ki je ne pokrije noben vzorec v obravnavi primerov. Napako potem odpravimo tako, da dodamo ustrezni vzorec. Omejitev implementacije je, da pri vsakem opozorilu vrne le en protiprimer. V nekaterih situacijah bi to lahko izboljšali tako, da bi vrnili vse možnosti, ki niso pokrite. To velja predvsem za vsote, ki imajo zmeraj končno mnogo konstruktorjev. Ta omejitev ne vpliva bistveno na uporabnost, saj lahko isti rezultat dosežemo tako, da postopoma dodajamo manjkajoče primere in dobivamo nova opozorila, dokler ne pokrijemo vseh možnosti.

Nekateri jeziki vzorce razširijo tako, da lahko v njih uporabljamo logično

disjunkcijo. To nam omogoča, da pišemo vzorce oblike $p_1 \mid p_2$, ki se ujemajo z vsemi vrednostmi, ki jih pokriva kateri od podvzorcev p_1 in p_2. Tako lahko v obravnavi primerov združimo tiste vzorce, za katere želimo izvajanje programa nadaljevati na enak način. Ker eff takih vzorcev ne podpira in bi zaradi njih bila naša rešitev precej bolj zapletena, smo jih izpustili. Seveda bo potrebno implementacijo nadgraditi v primeru, če eff v prihodnosti dobi podporo za tako združevanje vzorcev.

Eff zaenkrat sicer nima prevajalnika, v splošnem pa lahko podatek o izčrpnosti uporabimo za optimizacijo prevajanja obravnave primerov [3]. Če namreč vemo, da je izčrpna, lahko zavržemo tiste veje programa, ki med izvajanjem sprožijo napako, kadar se konkretna vrednost ne ujema z nobenim od podanih vzorcev.

Za konec si oglejmo še podoben problem, ki pa je specifičen za eff. Struktura prestreznikov je podobna obravnavi primerov s stavkom `match`, le da imajo namesto vzorcev seznam operacij, ki jih prestrezajo. Operacija, ki je ne ujame noben prestreznik in nima definiranega privzetega obnašanja v viru, povzroči napako med izvajanjem. Pojavi se vprašanje, ali ima smisel tudi v konstruktih `with` h `handle` c preverjati, če h ujame vse operacije, ki se pojavijo v izračunu c. Ker lahko prestreznike gnezdimo, je možno in pravzaprav običajno, da določene operacije prepustimo prestreznikom na višjih nivojih. Po drugi strani bi bilo koristno vedeti, ali lahko v celotnem programu pride do neobravnavane operacije. Na obe vprašanji pa je v splošnem nemogoče odgovoriti s preprosto sintaktično analizo.

Dodatek A

Primeri opozoril

Najprej si ogledamo, kako izgledajo opozorila za različne vrste napak pri prirejanjih vzorcev v splošnem. V ta namen definiramo tip

```
type foo = A | B | C.
```

Prva skupina napak pri analizi primerov je, da kakšne možne vrednosti ne pokrije noben od podanih vzorcev. Za stavek

```
match A with
  | A -> true
```

naš algoritem vrne naslednje opozorilo.

```
Warning: This pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
B
```

To napako odpravimo tako, da v stavek `match` dodamo člen za označeno manjkajočo vrednost. V naslednji iteraciji dobimo opozorilo še za manjkajoč konstruktor `C`. Ko dodamo še tega, je analiza primerov popolna. Alternativno bi lahko uporabili jokerja, ki se ujema z vsemi vrednostmi.

Nasprotno lahko v analizi primerov podamo „preveč“ vzorcev, v smislu, da kateri izmed njih ne pokrije nobene nove vrednosti.

```
match A with
  | A -> 0
  | B -> 1
  | C -> 2
  | _ -> 9
```

V zgornjem stavku je joker na koncu nekoristen, saj vsako možno vrednost tipa `foo` pokrijejo prejšnje tri vrstice. Algoritem v tem primeru vrne opozorilo

```
Warning (line 5, char 5): This match case is unused.
```

Poleg izvorne kode tolmača za eff je nekaj datotek s testi in primeri programov. Te smo preverili z našim algoritmom, ki je izpisal le eno opozorilo, za stavek

```
let x::y = [1;2;3;4] in (x,y).
```

V stavku `let` nastopa le en vzorec. V tem primeru je to `x::y`, ki opisuje množico nepraznih seznamov. Spremenljivki `x` priredi vrednost prvega elementa, `y` pa preostali del seznama. To ni problem, kadar je seznam, ki ga prirejamo, znan vnaprej. Če pa se zgodi, da prirejamo prazen seznam, pride do napake med izvajanjem. Izpisano opozorilo za ta primer je tako

```
Warning: This pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
```

Ob vsakem zagonu tolmač naloži modul `pervasives`, ki vsebuje definicije vgrajenih tipov učinkov in vrsto uporabnih funkcij. Tudi v tem modulu ni nobene tovrstne napake.

Literatura

[1] H. Abelson, G. J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: Mit Press, 1996, str. 426–437.

[2] A. Bauer, M. Pretnar. (mar. 2012). Programming with Algebraic Effects and Handlers. *Computing Research Repository* [Online]. 1203.1539. Dostopno na: `http://arxiv.org/abs/1203.1539`.

[3] F. Le Fessant, L. Maranget, "Optimizing Pattern Matching," v *Intl. Conference on Functional Programming*, Florence, Italy, 2001, str. 26–37.

[4] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon. (2011). *The OCaml system (release 3.12): Documentation and user's manual* [Online]. Dostopno na: `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[5] L. Maranget, "Warnings for pattern matching," *Journal of Functional Programming*, št. 17, zv. 3, str. 387–421, 2007.