Teorija programskih jezikov

Andrej Bauer

9. januar 2014

Kazalo

1 OCaml **9**
1.1 Interaktivna zanka . . . 9
1.2 Prevajalnik . . . 10
1.3 Osnovno o OCamlu . . . 11
1.3.1 Tip `int` . . . 11
1.3.2 Tip `float` . . . 11
1.3.3 Tip `bool` . . . 12
1.3.4 Tipa `string` in `char` . . . 12
1.3.5 Tabele . . . 12
1.3.6 Tip `unit` . . . 13
1.3.7 Seznami . . . 13
1.3.8 Urejeni pari in urejene n-terice . . . 13
1.3.9 Zapisi . . . 14
1.3.10 Funkcije . . . 15
1.3.11 Lokalne, rekurzivne in hkratne definicije . . . 16
1.3.12 Vzorci . . . 17
1.3.13 Vsote . . . 18
1.3.14 Stavek `match` . . . 19
1.3.15 Definicije tipov . . . 20
1.3.16 Parametrični polimorfizem . . . 21
1.3.17 Izjeme . . . 21
1.4 Dva primera . . . 21
1.4.1 Asociativni seznami . . . 22
1.4.2 Dvojiška drevesa . . . 22
1.5 Naloge . . . 23

2 Aritmetični izrazi **25**
2.1 Konkretna in abstraktna sintaksa . . . 25
2.1.1 Od konkretne k abstraktni sintaksi . . . 27
2.1.2 Od abstraktne h konkretni sintaksi . . . 30
2.2 Evaluacija izrazov . . . 31
2.2.1 Semantika velikih korakov . . . 31
2.2.2 Semantika malih korakov . . . 34
2.3 Izrazi s spremenljivkami . . . 36
2.4 Naloge . . . 37

3 Funkcijski programski jezik **39**
3.1 MiniML . . . 39
3.2 Vezane in proste spremenljivke ter substitucija . . . 40
3.3 Preverjanje tipov . . . 41
3.4 Evaluacija . . . 43
3.5 Varnost MiniML . . . 44
3.5.1 Deljenje in napake ob izvajanju . . . 47
3.6 Učinkovita implementacija MiniML . . . 48
3.6.1 Abstraktni stroj . . . 48
3.6.2 Prevajalnik za MiniML . . . 50
3.7 Neučakani in leni jeziki . . . 51
3.7.1 Seznami . . . 52
3.7.2 Splošne rekurzivne definicije . . . 52
3.7.3 Programski jezik MiniHaskell . . . 54
3.8 Izpeljava tipov in parametrični polimorfizem . . . 55
3.8.1 Poly . . . 56
3.8.2 Izpeljava tipov in združevanje . . . 57
3.8.3 Operacijska semantika Poly . . . 60
3.9 Naloge . . . 60

4 Ukazni programski jezik **61**
4.1 Ukazni programski jezik . . . 61
4.1.1 Sintaksa . . . 61
4.1.2 Operacijska semantika . . . 62
4.2 Specifikacije in pravilnost programov . . . 63
4.3 Prevajalnik za ukazni jezik . . . 65
4.4 Naloge . . . 65

5 Objektni programski jezik **67**
5.1 Zapisi . . . 67
5.2 Podtipi . . . 68
5.2.1 Funkcijski programski jezik s podtipi . . . 69
5.3 Objekti kot zapisi . . . 73
5.3.1 Statična semantika . . . 75
5.3.2 Dinamična semantika . . . 76
5.4 Naloge . . . 76

6 Denotacijska semantika **81**
6.1 Kaj je denotacijska semantika . . . 81
6.2 Naivna semantika aritmetičnih izrazov . . . 81
6.2.1 Deljenje z nič in nedefinirana vrednost . . . 83
6.3 Semantika funkcijskega jezika z rekurzijo . . . 84
6.3.1 PCF . . . 85
6.3.2 Domene . . . 86
6.3.3 Semantika PCF . . . 90
6.4 Naloge . . . 91

Literatura **94**

Zanikanje odgovornosti

Ti zapiski vsebujejo napake.

Odkrivanje napak je sestavni del učnega procesa.

[verzija 9. januar 2014]

Poglavje 1

OCaml

Koncepte, ki se pojavljajo v programskih jezikih, je najlažje razumeti z njihovo *uporabo*. Zato bomo v prvi lekciji spoznali moderno zasnovan programski jezik, ki je opremljen z bogatim naborom programskih konstruktov. Ta jezik je *OCaml*, znan tudi kot *Objective Caml*.[1] Več o OCamlu najdete na `http://www.ocaml.org/`, kjer je na voljo tudi dokumentacija in učbeniki.

1.1 Interaktivna zanka

OCaml je opremljen s prevajalnikom v vmesno kodo (byte code) `ocamlc`, prevajalnikom v strojno kodo (native code) `ocamlopt` in z interkativno zanko `ocaml`. Posamezne module programa hranimo v datotekah s končnico `.ml`.[2]

Za preproste preizkuse uporabimo interaktivno zanko, ki jo poženemo z ukazom `ocaml`.[3] Vsak ukaz lahko vpišemo v eno ali več vrstic in ga vedno končamo z dvojnim podpičjem in pritiskom na `Enter`:

```
            Objective Caml version 3.08.1

# 2 + 2 ;;
- : int = 4
# 3 < 8 ;;
- : bool = true
```

OCaml izpiše odgovor in njegov tip. Z izrazom `let` definiramo vrednosti. V tem primeru OCaml izpiše ime nove vrednosti, njen tip in vrednost:

```
# let x = 3 + 19 ;;
val x : int = 22
# let y = 2 * x ;;
val y : int = 44
```

[1]V poštev bi prišla katerakoli različica iz družine programskih jezikov ML, pa tudi Haskell, `http://www.haskell.org` bi bil primeren. Priljubljeni programski jeziki kot so C/C++, java, perl, in python so povsem neprimerni, ker so bodisi siromašni bodisi na račun objektne orientiranosti zanemarjajo ostale koncepte.

[2]OCaml pozna tudi *signature* ali *vmesnike*, ki jih hranimo v datotekah s končnico `.mli`

[3]Za udobnejše delo poženemo interaktivno zanko znotraj XEmacs. Na Linuxu si lahko tudi namestimo ukaz `ledit` ali `rlwrap` in poženemo `ledit ocaml` ali `rlwrap ocaml`, ki interaktivno zanko oplemeniti z običajnimi ukazi za urejanje.

```
# x + y ;;
- : int = 66
```

Pozor! V ukaznih programskih jezikih kot je java in C/C++, smo navajeni, da lahko spremenljivkam menjamo vrednost. V OCamlu izraz `let x = ...` *ni* definicija spremenljivke, ampak definicija *(nespremenljive) vrednosti*. Se pravi, da vrednosti `x` ne moremo spremeniti. Sicer lahko `x` definiramo znova, a to pomeni, da stari `x` zavržemo in definiramo novega (ki ima lahko tudi nov tip). Seveda OCaml pozna tudi prave spremenljivke, ki jim lahko menjamo vrednosti (imenujejo se *reference*).

Zanko zaključimo z znakom za konec datoteke[4] ali z ukazom `#quit`, ki mu ne pozabimo dodati dveh podpičij:

```
# #quit;;

Process caml-toplevel finished
```

1.2 Prevajalnik

OCaml ima prevajalnik za vmesno kodo `ocamlc` in prevajalnik za strojno kodo `ocamlopt`. Seveda je strojna koda bistveno hitrejša od vmesne kode. Prednost vmesne kode je, da jo lahko z ukazom `#load` naložimo v interaktivno zanko in da je neodvisna od operacijskega sistema.

Programerksa tradicija zahteva, da je prvi program, ki ga napišemo v novem programskem jeziku, znani "Hello, world!":

hello.ml

```
1 print_endline "Hello,␣world!" ;;
```

Prevedemo ga z ukazom `ocamlbuild` in že ga lahko poženemo:

```
$ ocamlbuild hello.native
$ ./hello.native
Hello, world!
```

Ukaz `ocamlbuild` je inteligentni prevajalnik, ki zna prevesti tudi zahtevne programe. Z zgornjim ukazom smo prevedli program v strojno kodo, vmesno kodo pa bi dobili z `ocamlbuild hello.byte`.

Med prevajanjem `ocamlbuild` vse pomožne datoteke hrani v mapi `_build`, kjer se pojavi tudi program `hello.native`, nato pa se naredi povezava (soft link) iz `hello.native` na `_build/hello.native`. Operacijski sistem Microsoft Windows ne pozna povezav te vrste, zato moramo tam program poiskati v mapi `_build`.

Tudi v datoteki `.ml` posamezne ukaze in definicije ločimo z dvojnim podpičjem. Izjemoma smemo opustiti dvojno podpičje med dvema zaporednima definicijama in za zadnjim ukazom. Ker so skoraj vsi moduli zaporedja definicij, nekateri programerji izpuščajo dvojna podpičja, kadar je to le mogoče.

Program sestoji iz enega ali več *modulov* ali *struktur*. Imena modulov se vedno začnejo z veliko začetnico. Datoteka *`ime`*`.ml` predstavlja modul *`Ime`*. Na vrednost `x` definirano v modulu `M` se slikcujemo iz ostalih modulov z `M.x`. Ponazorimo to s primerom:

[4] Na sistemih Unix, MacOS in Linux je to `Ctrl-D`, na Microsoft Windows `Ctrl-Z`.

foo.ml

```
1 let x = 5 ;;
2 let y = 8 ;;
```

bar.ml

```
1 let z = Foo.x + Foo.y ;;
2 print_int z ;;
```

```
$ ocamlbuild bar.native
$ ./bar.native
13
```

1.3 Osnovno o OCamlu

Predpostavili bomo, da se je bralec že srečal s kakim splošnim programskim jezikom in da ima osnovno znanje programiranja. Zato le na hitro preletimo osnove lastnosti OCamla. Na tem mestu ne bomo ponavljali podatkov, ki so pregledno zapisane v dokumentaciji o OCamlu, ki jo najdete na `http://caml.inria.fr/pub/docs/manual-ocaml/`.

1.3.1 Tip `int`

Cela števila so predstavljena s tipom `int`. OCaml uporablja 30-bitno ali 62-bitno predznačeno aritmetiko,[5] odvisno od arhitekture CPU. Sicer deluje celoštevilska aritmetika po pričakovanjih.

1.3.2 Tip `float`

Števila s plavajočo vejico dvojne natančnosti so tipa `float`. Na kratko bomo takim številom rekli kar realna. Predvsem naj opozorimo bralca, da v OCamlu ne moremo mešati tipov – vsaka vrednost ima natanko en tip. To pomeni, da ne moremo neposredno seštevati števil tipa `int` in `float`, ampak moram eno od njih prej pretvoriti. Ker ima operator `+` tip `int -> int -> int`, ga lahko uporabljamo *samo* za seštevanje celih števil. Za seštevanje realnih števil imamo operator `+.` in na sploh imajo vsi realni aritemtični operatorji dodano piko: množenje je `*.`, odštevanje `-.`, deljenje `/.` itn. Nekaterim gre to izredno na živce, vendar so se načrtovalci OCamla zavestno odločili, da naj bo tako. S tem so v programskih jezik vnesli še dodatno stopnjo discipline.

Ponazorimo delovanje in nedelovanje aritmetike z nekaj primeri:

```
# 1.5 + 2.3 ;;
Characters 0-3:
  1.5 + 2.3 ;;
  ^^^
This expression has type float but is here used with type int
# 1.5 +. 2.3 ;;
- : float = 3.8
# 1 + 2.3 ;;
```

[5] Zakaj ni aritmetika 32-bitna ali 64-bitna? Ker OCaml porabi dodatne bite za pobiranje smeti (garbage collection). Na voljo je tudi prava 32-bitna in 64-bitna aritmetika, ki ju najdete v modulih `Int32` in `Int64`, medtem ko paket `num` ponuja cela in racionalna števila neomejene velikosti.

```
Characters 4-7:
  1 + 2.3 ;;
      ^^^
This expression has type float but is here used with type int
# float_of_int 1 +. 2.3 ;;
- : float = 3.3
```

1.3.3 Tip bool

Tip `bool` ima osnovni vrednosti `true` (resnično) in `false` (neresnično). Deluje po pričakovanjih, morda velja opozoriti le, da smemo pisati pogojne stavke, ki vrnejo poljubne vrednosti, medtem ko moramo v javi in C/C++ ločiti med stavkom `if` in aritmetičnim pogojnim stavkom $p ? a : b$:

```
# if 3 * 3 > 8 then "yes" else "no" ;;
- : string = "yes"
# if 3 * 3 > 8 then print_endline "yes" else print_endline "no" ;;
yes
- : unit = ()
```

V pogojnem stavku `if` ne moremo izpustiti `else`, saj bi to pomenilo, da odgovor ni definiran, kadar je pogoj neresničem. Izjemoma smemo izpustiti `else`, če je rezultat pogojnega stavka tipa `unit`, glej 1.3.6.

1.3.4 Tipa string in char

Nize znakov pišemo v dvojne narekovaje, posamezne znake v enojne:

```
# let s = "This string contains forty-two characters." ;;
val s : string = "This string contains forty-two characters."
# 'x' ;;
- : char = 'x'
```

Nize stikamo z operatorjem `^`, do `n`-tega znaka v nizu `s` pa dostopamo z `s.[n]`. Ostale funkcije za delo z nizi najdete v modulu `String`, opisano je v dokumentaciji v poglavju 20. Dolžino niza dobimo na primer s funkcijo `String.length`:

```
# s.[3] ;;
- : char = 's'
# String.length s ;;
- : int = 42
```

1.3.5 Tabele

Tabele elementov tipa t predstavlja tip t `array`. Tabelo lahko podamo tako, da njene elemente naštejemo med `[|` in `|]` ter jih ločimo s podpičji:

```
# [| 1; 3; 8; 0 |] ;;
- : int array = [|1; 3; 8; 0|]
# let a = [| "foo"; "bar"; "baz"; "qux" |] ;;
val a : string array = [|"foo"; "bar"; "baz"; "qux"|]
```

Vsi elementi v dani tabeli morajo imeti isti tip.[6] Funkcije za delo s tabelami najdemo v modulu `Array`.

Do `n`-tega elementa v tabeli `a` dostopamo z `a.(n)`. Tabele so *spremenljive vrednosti*. Vrednost `n`-tega elementa v tabeli `a` nastavimo na vrednost `x` z izrazom `a.(n)<-x`:

```
# a.(2);;
- : string = "baz"
# a.(2) <- "zblj" ;;
- : unit = ()
# a ;;
- : string array = [|"foo"; "bar"; "zblj"; "qux"|]
```

1.3.6 Tip `unit`

Tip `unit` predstavlja množico z enim elementom,[7] ki ga zapišemo kot prazno n-terico `()`. Ta tip ima podobno vlogo kot tip `void` v C/C++ in javi. Uporabimo ga, kadar ne želimo vrniti vrednosti, ali kadar želimo podati argument, ki ne vsebuje nobene informacije.

1.3.7 Seznami

Seznam elementov tipa *t* predstavlja tip *t* `list`. Elemente seznama naštejemo med oglatima oklepajema `[` in `]` in jih ločimo s podpičjem.

```
# [1; 2; 6; 8] ;;
- : int list = [1; 2; 6; 8]
# let r = ["foo"; "bar"; "baz"; "qux"] ;;
val r : string list = ["foo"; "bar"; "baz"; "qux"]
```

Funkcije za delo s seznami najdemo v modulu `List`. Osnovni operator za sestavljanje seznamov je `::`, ki dani element stakne z danim seznamom. Sezname lahko tudi stikamo z operatorjem `@`.

```
# "boing" :: r ;;
- : string list = ["boing"; "foo"; "bar"; "baz"; "qux"]
# [4; 5; 6] @ [1; 2; 3] ;;
- : int list = [4; 5; 6; 1; 2; 3]
```

1.3.8 Urejeni pari in urejene n-terice

Tip urejenih parov (x, y), kjer je x tipa `s` in y tipa `t`, v OCamlu predstavlja tip `s * t`. Urejene n-terice predstavlja tip $\mathtt{t}_1 * \cdots * \mathtt{t}_n$. Elemente urejenega para ali urejene n-terice pišemo med (neobvezne) oklepaje `(` in `)` in jih ločimo z vejico.

```
# (42, "foo") ;;
- : int * string = (42, "foo")
# 8, [1;4;5], 'c' ;;
- : int * int list * char = (8, [1; 4; 5], 'c')
```

[6] To se morda komu zdi slaba ideja. V resnici je to zelo dobra ideja. Če želimo imeti tabele "mešanega tipa" uporabimo vsote, glej 1.3.13.

[7] V resnici imamo dve vrednosti tipa `unit`: element `()` in *divergentno vrednost*, to je vrednost, ki jo ima program, ki se nikoli ne konča.

Prvi in drugi element urejenega para dobimo z operatorjema `fst` in `snd`, vendar ju redko uporabljamo. Ponavadi uporabimo *vzorce*, glej 1.3.12.

Pogosta napaka se pripeti, ko elemente seznama ločimo z vejicami namesto s podpičji. V tem primeru dobimo seznam z enim elementom, ki je n-terica:

```
# [1; 2; 3] ;;
- : int list = [1; 2; 3]
# [1, 2, 3] ;;
- : (int * int * int) list = [(1, 2, 3)]
```

Podobno opozorilo velja za tabele.

1.3.9 Zapisi

Urejene n-terice $(x_1, \ldots, x_n)$ so uporabne predvsem, ko je pomen posameznih komponent x_i očiten. Če pa je komponent več, ali je njihov pomen podoben, kaj hitro pride do zmede. Na primer, če bi podatke o osebi predstavili z urejeno sedmerico

$$(\mathit{ime}, \mathit{priimek}, \mathit{ulica}, \mathit{hišna\ številka}, \mathit{kraj}, \mathit{pošta}, \mathit{telefon}),$$

bi si programer s težavo zapomnil, kaj je pomen vsake posamezne komponente. To težavo rešimo tako, da damo komponentam smiselna imena. Podatkovni tip, s katerim to naredimo se imenuje *zapis* (angl. record type), posamezne komponente pa *polja*. V OCamlu podamo zapis z definicijo[8]

$$\texttt{type}\ t = \{\mathit{ime}_1 : \texttt{t}_1;\ \mathit{ime}_2 : \texttt{t}_2;\ \cdots;\ \mathit{ime}_n : \texttt{t}_\texttt{n}\}\ .$$

Vrednost tega tipa zapišemo tako, da v zavitih oklepajih naštejemo vrednosti posameznih komponent, pri čemer vrstni red *ni* pomemben:

$$\{\texttt{ime}_1 = v_1;\ \texttt{ime}_2 = v_2;\ \cdots;\ \texttt{ime}_n = v_n\}\ .$$

Na primer, podatkovni tip kompleksnih števil bi definirali z zapisom takole:

```
# type complex = { re : float ; im : float } ;;
type complex = { re : float; im : float; \brc
# { re = 1.0; im = 2.3 \brc ;;
- : complex = {re = 1.; im = 2.3\brc
# let w = { re = 1.0; im = 2.3 \brc ;;
val w : complex = {re = 1.; im = 2.3\brc
# let z = { im = 2.3; re = 1.0 \brc ;;
val z : complex = {re = 1.; im = 2.3\brc
# w = z ;;
- : bool = true
```

V zapisu `z` dostopamo do polja $\texttt{ime}_i$ z izrazom $\texttt{z.ime}_i$:

```
# z.re ;;
- : float = 1.
# {re = 2.5; im = -1.2}.im ;;
- : float = -1.2
```

Do polj v zapisu lahko dostopamo tudi z vzorci, glej 1.3.12.

Zapomnimo si, da so zapisi ekvivalentni urejenim n-tericam. Razlika je le v tem, da so v zapisu polja imenovana, v urejeni n-terici pa se zanašamo na vrstni red komponent.

[8] V C/C++ se zapis definira s `struct`, java zapisov ne pozna, lahko pa jih simuliramo z razredi.

1.3.10 Funkcije

Funkcije, ki sprejmejo argument tipa `s` in vrnejo rezultat tipa `t`, predstavlja tip `s -> t`. Funkcijo $x \mapsto e$, ki slika argument x v vrednost e zapišemo z izrazom

$$\texttt{function}\ x \to e\ ,$$

Funkcijo `f` uporabimo na argumentu `a` z izrazom `f a`.[9]

```
# function x -> 2 * x + 3 ;;
- : int -> int = <fun>
# (function x -> x * x - 10) 4 ;;
- : int = 6
# let g = function x -> x * x - 10 ;;
val g : int -> int = <fun>
# g 4 ;;
- : int = 6
```

Funkcijo `g` iz zgornjega primera lahko definiramo tudi takole:

```
# let g x = x * x + 10 ;;
val g : int -> int = <fun>
```

Funkcija večih argumentov, ki imajo tipe $\texttt{t}_1, \ldots, \texttt{t}_n$, ki vrača rezultat tipa `s`, ima tip

$$\texttt{t}_1 \to \texttt{t}_2 \to \cdots \to s\ .$$

Zapišemo jo lahko z vgnezdenimi `function`,

$$\texttt{function}\ x_1 \to \texttt{function}\ x_2 \to \cdots \to e\ ,$$

ali krajše $\texttt{fun}\ x_1\ x_2 \cdots x_n \to e$.

```
# function x -> function y -> x * x + y ;;
- : int -> int -> int = <fun>
# fun x y -> x * x + y ;;
- : int -> int -> int = <fun>
# let h x y = x * x + y ;;
val h : int -> int -> int = <fun>
# h 3 4 ;;
- : int = 13
```

Pogosto pride prav, da lahko pri funkciji večih spremenljivk nastavimo samo prvih nekaj argumentov in kot rezultat dobimo funkcijo, ki pričakuje preostale argumente:

```
# let f = h 7 ;;
val f : int -> int = <fun>
# f 4 ;;
- : int = 53
# f 5 ;;
- : int = 54
```

[9]Vidimo, da ni treba pisati oklepajev, v javi in C/C++ bi pisali `f(a)`. Tako lahko sicer pišemo tudi v OCamlu, a to ni potrebno in ni v navadi.

1.3.11 Lokalne, rekurzivne in hkratne definicije

Z izrazom `let` definiramo vrednosti v interaktivni zanki in v modulih. S tem dobimo *globalne* vrednosti, ki so vidne od svoje definicije naprej. Pogosto pa želimo znotraj kakega izraza definirati začasno, *lokalno* vrednost. To naredimo z izrazom

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ ,$$

ki pomeni, da ima x vrednost e_1 v izrazu e_2. Če je zunaj e_2 že definirana vrednost x, se ta ohrani in ni vidna znotraj e_2.

```
# let x = 1 ;;
val x : int = 1
# let x = 3 + 4 in (x + x) ;;
- : int = 14
# x ;;
- : int = 1
```

OCaml seveda omogoča tudi rekurzivne definicije. V tem primeru uporabimo `let rec` namesto `let`. Običajno definiramo rekurzivne funkcije, vendar OCaml dopušča tudi splošne rekurzivne definicije:

```
# let rec f n =
    if n = 0 then 1 else n * f (n-1) ;;
  val f : int -> int = <fun>
# let rec x = 1 :: 2 :: 5 :: x ;;
val x : int list =
  [1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1;
   2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2;
   5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5; 1; 2; 5;
   ...]
```

OCaml neskončnih in cikličnih podatkov ne izpisuje v nedogled, zato se v zgornjem primeru naveliča izpisovati ciklično definiran seznam. Včasih želimo hkrati definirati več vrednosti, kar naredimo z izrazom

$$\begin{aligned} &\texttt{let}\ x_1 = e_1 \\ &\texttt{and}\ x_2 = e_2 \\ &\quad\vdots \\ &\texttt{and}\ x_n = e_n\ . \end{aligned}$$

Če so hkrati definirane vrednosti med seboj rekurzivne, uporabimo `let rec`. Nekaj primerov:

```
# let x = 10 ;;
val x : int = 10
# let y = 7 ;;
val y : int = 7
# let x = y and y = x ;;
val x : int = 7
val y : int = 10
# let rec f n = (if n = 0 then 1 else 1 + g (n - 1))
      and g n = (if n = 0 then 2 else 2 * f (n - 1)) ;;
```

```
val f : int -> int = <fun>
val g : int -> int = <fun>
```

Rekurzivne in hkratne definicije so lahko tudi lokalne.

1.3.12 Vzorci

Izrazi `let`, `function` in `fun` imajo splošno obliko, v kateri lahko uporabimo *vzorce*. Na primer, če imamo urejeni par `p` in želimo njegovo prvo in drugo komponento shraniti v vrednosti `x` in `y`, to storimo z `let` in ustreznim vzorcem:

```
# let p = (42, "foo") ;;
val p : int * string = (42, "foo")
# let (x, y) = p ;;
val x : int = 42
val y : string = "foo"
```

Vzorci so lahko poljubno zakomplicirani. Če nas kak del izraza ne zanima, ga v vzorcu označimo s podčrtajem `_`:

```
# let (u, _, (v, _)) = (42, false, ('f', "bar")) ;;
val u : int = 42
val v : char = 'f'
```

Tudi zapise lahko prirejamo z vzorci. S tem lahko iz zapisa hkrati dobimo več komponent:

```
# type complex = { re : float; im : float } ;;
type complex = { re : float; im : float; }
# let z = {re = 1.2; im = 0.3} ;;
val z : complex = {re = 1.2; im = 0.3}
# let {re = x; im = y} = z ;;
val x : float = 1.2
val y : float = 0.3
```

Z vzorci lahko prirejamo tudi sezname:

```
# let x :: lst = [1; 2; 3; 4] ;;
Characters 4-12:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let x :: lst = [1; 2; 3; 4] ;;
      ^^^^^^^^
val x : int = 1
val lst : int list = [2; 3; 4]
```

V tem primeru smo `x` priredili glavo in `lst` rep seznama `[1;2;3;4]`. Ker se ob času prevajanja ne ve, ali bo izraz na desni strani prirejanja res neprazen seznam,[10] nas OCaml opozori, da bi lahko prišlo do napake. Namreč, praznega seznama `[]` ne moremo prirediti vzorcu `x::lst`, saj prazen seznam nima glave in repa:

```
# let x :: lst = [] ;;
Characters 4-12:
Warning: this pattern-matching is not exhaustive.
```

[10] V prikazanem primeru je seveda očitno, da ne bo prišlo do napake. Če pa bi imeli na desni klic funkcije, ki vrne seznam, ne bi vedeli, ali bo rezultat neprazen seznam, dokler se funkcija ne bi izvedla.

```
Here is an example of a value that is not matched:
[]
  let x :: lst = [] ;;
      ^^^^^^^^
Exception: Match_failure ("", 9, -47).
```

OCaml je izpisal opozorilo (imel je prav!) in v izvajanju je dejansko prišlo do napake, saj se je sprožila izjema `Match_failure`. Zato ocamlova opozorila jemljemo resno in dvakrat premislimo, ali se jim morda lahko izognemo.

Uporabo vzorcev v `function` bomo obravnavali v 1.3.14.

1.3.13 Vsote

Vsote so izredno uporaben podatkovni tip, ki ga večina programskih jezikov bodisi sploh nima (java, perl, python) bodisi so implementirane narobe (`union` v C/C++) in jih zato nihče ne uporablja. Posledica je kopica programerskih napak, ki jih ponavadi opišemo z besedami "pozabil sem obravnavati en primer".

Preden razložimo, kaj so vsote v programskem jeziku, pojasnimo, kaj so *disjunktne unije* v matematiki. Če imamo množici A in B, lahko tvorimo njuno unijo $A \cup B$. Pri tem se elementi, ki so skupni obema množicama, v uniji štejejo samo enkrat. Pogosto pa želimo množici A in B združiti tako, da se ne pomešata med seboj. V tem primeru tvorimo disjunktno unijo $A + B$, ki jo dobimo kot navadno unijo množic $\{0\} \times A$ in $\{1\} \times B$,

$$\begin{aligned} A + B &= \{0\} \times A \cup \{1\} \times B \\ &= \{(0, x) \mid x \in A\} \cup \{(1, y) \mid y \in B\} . \end{aligned}$$

Mešanje elementov A in B smo preprečili tako, da smo vsak element $x \in A$ spremenili v par $(0, x)$ in vsak element $y \in B$ v par $(1, y)$. Izbira 0 in 1 je pri tem povsem nepomembna, saj bi lahko uporabili kakršnakoli različna elementa. Pogosto je bolj praktično, če v disjunktni vsoti dovolimo poljubni oznaki namesto 0 in 1, saj lahko s tem predstavimo še dodatno informacijo. Zato definiramo

$$\mathtt{U}{:}A + \mathtt{V}{:}B = \{\mathtt{U}(x) \mid x \in A\} \cup \{\mathtt{V}(y) \mid y \in B\} .$$

Ta definicija je ekvivalenta definiciji $A + B$, le da dopušča, da namesto 0 in 1 sami izberemo *različni* oznaki `U` in `V`. Poleg tega namesto urejenih parov $(\mathtt{U}, x)$ in $(\mathtt{V}, y)$ pišemo kar formalne izraze $\mathtt{U}(x)$ in $\mathtt{V}(x)$.[11]

Ponazorimo uporabo disjunktne vsote s primerom. Denimo, da v spletni trgovini prodajamo artikle dveh vrst, majice in čevlje. Vsaka majica in vsak čevelj ima velikost, ki je celo število. Množica vseh artiklov je disjunktna vsota

$$\mathtt{artikel} = \mathtt{Majica}{:}\mathbb{Z} + \mathtt{Cevelj}{:}\mathbb{Z} .$$

Majico velikosti 42 predstavlja element $\mathtt{Majica}(42)$, čevelj velikosti 42 pa element $\mathtt{Cevelj}(42)$. Disjunktno vsoto dveh množic zlahka posplošimo na disjunktno vsoto končno mnogo množic $\mathtt{U}_1{:}A_1 + \cdots \mathtt{U}_n{:}A_n$.

V programskih jezikih ne govorimo o množicah ampak o *tipih*. V OCamlu torej *vsoto tipov* definiramo z izrazom

$$\mathtt{type}\ t = \mathtt{U}_1\ \mathtt{of}\ t_1 \mid \mathtt{U}_2\ \mathtt{of}\ t_2 \mid \cdots \mid \mathtt{U}_n\ \mathtt{of}\ t_n .$$

[11] Mislimo si lahko, da sta `U` in `V` vložitvi $\mathtt{U} : A \to (\mathtt{U}{:}A + \mathtt{V}{:}B)$ in $\mathtt{V} : B \to (\mathtt{U}{:}A + \mathtt{V}{:}B)$.

Oznake `U`$_i$ se morajo začeti z veliko začetnico in biti med seboj različne. Na primer:

```
# type artikel = Majica of int | Cevelj of int ;;
type artikel = Majica of int | Cevelj of int
# Majica 42 ;;
- : artikel = Majica 42
```

Omeniti velja dva posebna primera vsot. Če v vsoti nastopa tip `unit`, namesto `U of unit` pišemo kar `U`. S tem definiramo konstanto `U`. Na primer, če imamo tri vrste vozovnic, enosmerne, povratne in mesečne, to izrazimo z vsoto

```
# type vozovnica = Enosmerna | Povratna | Mesecna ;;
type vozovnica = Enosmerna | Povratna | Mesecna
```

Drugi primer uporabne vsote je tako imenovani *opcijski tip*. Pogosto želimo obravnavati tip `t` skupaj s posebno vrednostjo, ki pomeni "nedefinirano" ali "neveljavno".[12] To izrazimo v OCamlu z vsoto `Nedefinirano` | `Definirano of` t. Nedefinirano vrednost predstavlja konstanta `Nedefinirano`, medtem ko prave vrednosti zapišemo z izrazom `Definirano` x. Ker se take definicije pogosto pojavljajo, je OCaml že opremljen z opcijskim tipom t `option`, ki je definiran kot vsota

$$\texttt{None} \mid \texttt{Some of } t\ .$$

Vrednost `None` torej predstavlja nedefinirano vrednost, medtem ko `Some` x predstavlja defnirano vrednost x:

```
# Some "foo" ;;
- : string option = Some "foo"
# None ;;
- : 'a option = None
```

Tip izraza `None` je `'a option`. Imena tipov, ki se začnejo z navednico `'`, označujejo *poljubne tipe*, o čemer bomo še razpravljali v 1.3.16.

1.3.14 Stavek `match`

S stavkom `match` v OCamlu obravnavamo primere.[13] Splošna oblika

$$\begin{aligned}&\texttt{match } e \texttt{ with}\\ &\mid p_1 \to e_1\\ &\mid p_2 \to e_2\\ &\vdots\\ &\mid p_n \to e_n\ .\end{aligned}$$

pomeni: "Če e ustreza vzorcu p_1, potem e_1, sicer, če e ustreza vzorcu p_2, potem e_2, ..., sicer, če e ustreza vzorcu p_n, potem e_n". Če izraz e ne ustreza nobenemu vzorcu, se sproži izjema `Match_failure`. OCaml zna vedno izračunati, ali smo v stavku `match` obravnavali vse možnosti in nas opozori, če smo kakšno pozabili. S tem se izognemo celi vrsti napak.

Funkcijo, ki sešteje elemente danega seznama, zapišemo z `match` takole:

[12] V C/C++ je ta vrednost znana kot `NULL`, v javi je to objekt `null`, v perlu pa `undef`. Vendar so vsi ti primeri škodljivi, saj dopuščajo, da v programu pomotoma uporabimo tako neveljavno vrednost.

[13] Stavek `match` je do neke mere podoben stavku `switch` v C in C/C++.

```
# let rec vsota lst =
    match lst with
      | [] -> 0
      | x :: xs -> x + vsota xs ;;
val vsota : int list -> int = <fun>
# vsota [1; 2; 3; 10] ;;
- : int = 16
```

Funkcija obravnava dva primera: če seznam `lst` ustreza vzorcu `[]`, je prazen in je vsota 0, če pa seznam ustreza vzorcu `x::xs`, je sestavljen iz glave `x` in repa `xs`. V tem primeru rekurzivno izračunamo vsoto repa `xs` in ji prištejemo glavo `x`.

Tudi `function` dovoljuje, da obravnavamo več primerov, podobno kot pri `match`:

$$\begin{aligned} \texttt{function}\ & p_1 \to e_1 \\ & \mid p_2 \to e_2 \\ & \vdots \\ & \mid p_n \to e_n\ . \end{aligned}$$

Obvelja prvi vzorec, ki ustreza argumentu. Na primer, funkcijo, ki izračuna n-to Fibonaccijevo število definiramo z izrazom

```
# let rec fib = function
  | 0 -> 0
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2) ;;
val fib : int -> int = <fun>
# fib 6 ;;
- : int = 8
```

1.3.15 Definicije tipov

Omenili smo že, da lahko v OCamlu definiramo vsoto tipov in ji damo ime. Pravzaprav lahko definiramo poljubne tipe z izrazom

$$\texttt{type}\ t = \ldots$$

kjer namesto ... zapišemo tip, ki ga definiramo. Dovoljene so definicije rekurzivnih tipov[14] in hkratne definicije, ki jih zapišemo z izrazom

$$\begin{aligned} \texttt{type}\ & t_1 = \ldots \\ \texttt{and}\ & t_2 = \ldots \\ & \vdots \\ \texttt{and}\ & t_n = \ldots\ . \end{aligned}$$

[14] Pri rekurzivnih definicijah veljajo določene omejitve. Ne moremo na primer definirati `type t = t*t`. Če to res želimo, moramo uporabiti argument `-rectypes`, ko poženemo interaktivno zanko ali prevajalnik.

1.3.16 Parametrični polimorfizem

OCaml vedno sam izračuna vse tipe. Včasih pa je možnih tipov več, kot na primer pri funkciji `fun x -> x`, ki bi lahko bila tipa $\alpha \to \alpha$, kjer je α poljuben tip. Takim funkcijam pravimo *polimorfne*. OCaml pozna polimorfizem in izračuna tudi tipe polimorfnih funkcij:

```
# fun x -> x ;;
- : 'a -> 'a = <fun>
```

Tip `'a` je *parameter* in ga lahko zamenjamo s poljubnim tipom. Zato preberemo `'a -> 'a` kot tip funkcije, ki preslika argument poljubnega tipa `'a` v tip `'a`. V polimorfnem tipu lahko nastopa več parametrov, na primer

```
# fun (x, y) -> (y, x) ;;
- : 'a * 'b -> 'b * 'a = <fun>
```

Poleg polimorfnih funkcij v OCamlu nastopajo tudi ostale polimorfne vrednosti, na primer prazen seznam `[]` ima polimorfni tip `'a list`. Ko definiramo polimorfni tip, naštejemo njegove parametre pred imenom tipa. Za zgled definirijamo podatkovni tip dvojiških dreves, glej 1.4.2, ki imajo v listih podatek tipa `'a` in v vozliščih podatek tipa `'b`:

```
# type ('a, 'b) drevo =
    | List of 'a
    | Drevo of 'b * ('a,'b) drevo * ('a,'b) drevo ;;
type ('a, 'b) drevo =
  | List of 'a
  | Drevo of 'b * ('a, 'b) drevo * ('a, 'b) drevo
# Drevo("foo", List 42, Drevo ("bar", List 10, List 7)) ;;
- : (int, string) drevo =
Drevo ("foo", List 42, Drevo ("bar", List 10, List 7))
# Drevo('a', List [], List [1;4]) ;;
- : (int list, char) drevo = Drevo ('a', List [], List [1; 4])
```

1.3.17 Izjeme

Izjeme so posebni dogodki, ki jih sprožimo z ukazom `raise` in ulovimo z izrazom

$$
\begin{aligned}
&\texttt{try}\ e\ \texttt{with} \\
&\quad \mid p_1 \to e_1 \\
&\quad \mid p_2 \to e_2 \\
&\quad \vdots \\
&\quad \mid p_n \to e_n\ ,
\end{aligned}
$$

katerega vrednost je e, razen v primeru, da se sproži izjema I. V tem primeru je vrednost e_1, če izjema I ustreza vzorcu p_1, oziroma e_2, če I ustreza vzorcu p_2 itn.

1.4 Dva primera

V nadaljevanju bomo pogosto uporabljali asociativne sezname in drevesno predstavitev abstraktne sintakse. V tem razdelku si oglejmo, kaj so asociativni seznami in drevesa ter kako jih implementiramo v OCamlu.

1.4.1 Asociativni seznami

Asociativni seznam je seznam urejenih parov $[(x_1, y_1); (x_2, y_2); \ldots; (x_n, y_n)]$, ki predstavlja preslikavo $x_i \mapsto y_i$, ki preslika x_i v y_i. Pogosto x_i imenujemo *ključ* in y_i pripadajoča *vrednost*. Če se kateri od ključev x_i v seznamu pojavi večkrat, upoštevamo prvo pojavitev. Na primer, seznam $[(3, 1); (2, 5); (2, 3); (1, 8)]$ predstavlja preslikavo $1 \mapsto 8$, $2 \mapsto 5$, $3 \mapsto 1$.

Osnovni operaciji za delo z asociativnimi seznami sta:[15]

- `assoc` x ℓ, ki poišče vrednost, ki pripada ključu x v asociativnem seznamu ℓ.
- `add` x y ℓ, ki vrne asociativni seznam $(x, y) :: \ell$.

Drugo funkcijo zlahka sestavimo, saj je stikanje elementa s seznamom osnovna operacija v OCamlu, ki jo pišemo z dvojnim dvopičjem:

```
# let add x y lst = (x,y)::lst ;;
val add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

V funkciji `assoc` obravnavamo dva primera. Če je seznam prazen, ključa v njem gotovo ni in sprožimo izjemo `Not_found`. V drugem primeru je seznam sestavljen iz prvega elementa in preostanka. Če se ključ v prvem elementu ujema s tistim, ki ga iščemo, vrnemo pripadajočo vrednost, sicer nadaljujemo iskanje v preostanku:

```
# let rec assoc x lst =
    match lst with
      | [] -> raise Not_found
      | (x',y)::lst' -> if x = x' then y else assoc x lst'
  ;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

Preizkusimo delovanje `assoc`:

```
# assoc 2 [(3,1); (2,5); (2,3); (1,8)] ;;
- : int = 5
# assoc "Mojca" [("Mojca",12); ("Ana",10); ("Darko",5)] ;;
- : int = 12
# assoc "Eva" [("Mojca",12); ("Ana",10); ("Darko",5)] ;;
Exception: Not_found.
```

OCaml ima asociativne sezname že vgrajene v modulu `List`.

1.4.2 Dvojiška drevesa

Dvojiška drevesa tvorijo *induktivno* podatkovno strukturo, kar pomeni, da jih definiramo tako, da povemo pravila, s katerimi jih tvorimo. Pri tem smemo tvoriti nova drevesa iz že pridobljenih. V osnovni različici dvojiška drevesa tvorimo s praviloma:

1. List je drevo, označimo ga s konstano `List`.
2. Če sta d_1 in d_2 drevesi, lahko iz njiju tvorimo sestavljeno drevo, ki ga označimo `Drevo`(d_1, d_2).

[15] Glej tudi nalogo 1.4.

Če z D označimo množico vseh dvojiških, lahko zgornji pravili povzamemo z enačbo

$$D = \texttt{List} + \texttt{Drevo} : D \times D \ ,$$

kar v OCamlu zapišemo z *rekurzivnim* podatkovnim tipom:

```
type drevo =
  | List
  | Drevo of drevo * drevo
```

Za primer sestavimo funkcijo `velikost`, ki prešteje vozlišča v drevesu:

```
1 let rec velikost d =
2   match d with
3     | List -> 1
4     | Drevo (d1, d2) -> 1 + velikost d1 + velikost d2
```

Definicijo funkcije preberemo takole: velikost drevesa d je enaka 1, če je d list. Če je d sestavljeno drevo $\texttt{Drevo}(d_1, d_2)$, je njegova velikost za eno večja od vsote velikosti poddreves d_1 in d_2.

Običajno so bolj uporabna drevesa, pri katerih so listi ali vozlišča opremljeni še z dodatnimi podatki. Poleg tega ni nujno, da so drevesa vedno dvojiška – lahko so tudi trojiška, kombinirana ali celo s poljubnim številom potomcev. Primere raznih vrst dreves bomo srečali v nadaljevanju.

1.5 Naloge

Naloga 1.1 Na svoj računalnik naloži ocaml. Na voljo so že pripravljene distribucije za razne operacijske sisteme, dolgoročno pa se verjetno obrestuje uporaba paketnega managerja Opam, ki zna sam naložiti in prevesti OCaml ter na desetine programskih knjižnic zanj. Na operacijskem sistemu Microsoft Windows najprej potrebuješ Cygwin. Za resno uporabo boš potreboval še dober urejevalnik, na primer Emacs, ki ima način za urejanje programov v OCamlu (zahtevnejši uporabniki lahko namestijo še Emacs paket Tuareg, ki zna nekoliko več). Vse to najdeš na naslednjih spletnih mestih:

- OCaml: `http://www.ocaml.org/`
- OPAM: `http://opam.ocamlpro.com/`
- Cygwin: `http://www.cygwin.com/`
- Emacs: `http://www.gnu.org/software/emacs/`

Naloga 1.2 Podatkovni tip kompleksnih števil definiramo v OCamlu z

```
type complex = { re : float; im : float }
```

Sestavi funkcije za konjugirano vrednost, absolutno vrednost, seštevanje, odštevanje, množenje in deljenje kompleksnih števil. Nato si oglej še OCaml modul `Complex` iz standardne knjižnice za OCaml. Poišči tudi izvorno kodo za ta modul in jo primerjaj s svojo rešitvijo.

Naloga 1.3 Sestavi naslednje funkcije v OCamlu:

1. Funkcija `last`, ki sprejme seznam in vrne njegov zadnji element.

2. Funkcija `flatten`, ki sprejme seznam seznamov in jih stakne skupaj v en seznam.

3. Funkcija `zip`, ki sprejme seznama $[x_1; \ldots; x_n]$ in $[y_1; \ldots; y_n]$ in vrne seznam parov $[(x_1, y_1); \ldots, (x_n, y_n)]$. Če imata vhodna seznama različni dolžini, naj `zip` ignorira preostanek daljšega seznama.

Naloga 1.4 Sestavi funkcijo `remove`, ki sprejme ključ in asociativni seznam ter vrne nov asociativni seznam, v katerem je dani ključ in pripadajoče geslo izbrisano.

Naloga 1.5 Ta naloga je težja. Sestavi funkcijo `drevesa` : `int` → `drevo list`, ki vrne seznam vseh dreves z danim številom vozlišč. Sestavi še funkcijo `prestejDrevesa`, ki sprejme nenegativno celo število n in vrne število dvojiških dreves z n vozlišči.

Poglavje 2

Aritmetični izrazi

V tem poglavju obravnavamo celoštevilske aritmetične izraze, ki jih lahko razumemo kot zelo preprost programski jezik. Spoznali bomo osnove pojme iz teorije sintakse in teorije slovnic ter se naučili, kako aritmetične izraze evaluiramo.

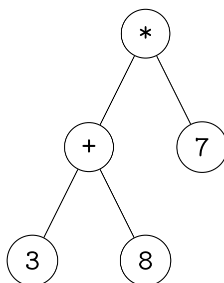
2.1 Konkretna in abstraktna sintaksa

Celoštevilske aritmetične izraze pozna vsakdo. Primeri takih izrazov so 11, $3 + 8 \cdot 7$, $(12 - 3) \cdot (1 + 2)$ ipd. Kako opišemo, kaj natančno je aritmetični izraz? Zakaj na primer je $3 \cdot (9 + 7)$ veljaven in $3 \cdot)9 + +7)$ neveljaven izraz? S takimi vprašanji se ukvarja teorija formalnih slovnic ali gramatik. Ker je naš namen povedati čim več o programskih jezikih, bomo predstavili le osnove te teorije.

Pravila, ki določajo, kateri izrazi v danem jeziku so veljavni, se imenuje *slovnična pravila*. Struktura, ki jo pravila določajo, se imenuje *sintaksa* jezika. V splošnem lahko izraze predstavimo povsem poljubno (malo vodoravno in malo navpično, ali pa kar v treh dimenzijah), ponavadi pa jih pišemo iz leve proti desni v ravni vrsti, pri čemer uporabimo končen nabor znakov. Izrazi so torej *zapisani* kot končni *nizi znakov*, denimo

```
(3 + 8) * 7 .
```

Ko pa razmišljamo o izrazih kot matematičnih objektih, je dosti bolje, če jih predstavimo abstraktno z *drevesi*. Zgornji izraz tako predstavimo z drevesom na sliki 2.1.



Slika 2.1: izraz `(3 + 8) * 7` predstavljen kot drevo

Imamo torej dve različni predstavitvi izrazov:

Konkretna sintaksa: izrazi so predstavljeni kot nizi znakov. To je običajni način za zapis izrazov. V konkretni sintaksi se pojavljajo oklepaji, nekatere operacije imajo prednost pred drugimi (denimo množenje pred seštevanjem), za vsako operacijo pa povemo, ali veže desno ali levo asociativno. (Operacija $*$ veže *desno* asociativno, če niz $a*b*c$ predstavlja $a*(b*c)$ in *levo* asociativno, če $a*b*c$ predstavlja $(a*b)*c$. Običajne aritmetične operacije so levo asociativne.)

Abstraktna sintaksa: izrazi so predstavljeni z drevesi. Oklepajev ne potrebujemo, prav tako ni treba razlagati, katere operacije imajo prednost in ali so levo ali desno asociativne, saj je struktura izraza v celoti razvidna iz drevesa. Abstraktna sintaksa je primerna podatkovna struktura za predstavitev in obdelavo izrazov.

Abstraktno sintakso podamo v *Backus-Naurjevi obliki (BNF)*,[1] iz katere je do neke mere razvidna tudi konkretna sintaksa. Aritmetične izraze zapišemo v BNF takole:

$$\begin{aligned} \text{Izraz } e &::= s \mid e+e \mid e-e \mid e\cdot e \mid e/e \mid -e \\ \text{Število } s &::= [\mathtt{0}-\mathtt{9}]^{+} \end{aligned}$$

Navpične črte pomenijo "ali", zapis $e+e$ pa pomeni, da smemo tvoriti nov izraz kot vsoto dveh izrazov.[2] Definicijo izraza preberemo takole: izraz je število, vsota, razlika, zmnožek, kvocient ali negacija izraza. Definicijo števila smo izrazili z *regularnim izrazom*, ki pravi, da je število neprazno zaporedje števk. Zapis $[0-9]$ pomeni "katerikoli znak med 0 in 9", znak $+$ pa pomeni "ena ali več ponovitev". Poznamo še znak $*$, ki pomeni nič ali več ponovitev in znak ?, ki pomeni nič ali eno pojavitev.

Iz abstraktne sintakse ni razvidno, katere aritmetične operacije imajo prednost, in tudi ne, ali so levo ali desno asociativne. Tako bi lahko izraz $2+3\cdot 8$ razumeli kot $(2+3)\cdot 8$ ali kot $2+(3\cdot 8)$. Manjka tudi pravilo, ki bi nam dovoljevalo uporabo oklepajev. Vse to spada v konkretno sintakso, ki jo bomo obravnavali v 2.1.2. Za zdaj definirajmo podatkovni tip `izraz`, ki povzema zgornjo definicijo:

```
1  type izraz =
2      Stevilo of int
3    | Plus of izraz * izraz
4    | Minus of izraz * izraz
5    | Krat of izraz * izraz
6    | Deljeno of izraz * izraz
7    | UnarniMinus of izraz
```

Drevo na sliki 2.1 predstavimo kot vrednost tipa `izraz` z

```
Krat (Plus (Stevilo 3, Stevilo 8), Stevilo 7)
```

Podatkovnega tipa `stevilo` nismo definirali, saj bomo uporabili kar vgrajeni tip celih števil `int`.

[1] BNF so izumili za specifikacijo sintakse Algola 60. Mi bomo uporabili samo slab približek pravega BNF, ki zadošča našim potrebam.

[2] Dejstvo, da smo dvakrat uporabili znak e *ne* pomeni, da morata biti člena v vsoti enaka. To lahko poudarimo tako, da namesto $e+e$ pišemo e_1+e_2.

2.1.1 Od konkretne k abstraktni sintaksi

Posvetimo se še konkretni sintaksi in vprašanju, kako od konkretne sintakse preidemo k abstraktni. Sestaviti želimo funkcijo, ki izraz, predstavljen z nizom znakov, razčleni in pretvori v drevo tipa `izraz`. Niz znakov, ki predstavlja izraz, na primer

`"(3 + 8) * 7"`,

najprej razbijemo na posamezne sestavne kose, ki jih imenujemo *leksemi*:[3]

`OKLEPAJ`, `STEVILO` 3, `PLUS`, `STEVILO` 8, `ZAKLEPAJ`, `KRAT`, `STEVILO` 7, `EOF` .

Temu postopku pravimo *leksična analiza*, funkciji, ki razbije niz na lekseme pa *lekser* (angl. lexer). Posebni leksem `EOF` označuje konec niza.[4] Presledkov ne štejemo za lekseme, čeprav so uporabni, saj razločujejo posamezne lekseme. V drugem koraku zaporedje leksemov razčlenimo in iz njih sestavimo drevesno predstavitev izraza. Funkcija, ki opravi ta del, se imenuje *parser*.

Ročno pisanje lekserja in parserja je zamudno opravilo, ki si ga bomo olajšali z orodjema `ocamllex` in `ocamlyacc`. Prvo iz opisa leksemov zgradi lekser, drugo pa iz opisa konkretne sintakse zgradi parser. Konkretno sintakso opišemo v datoteki s končnico `.mly` z naslednjimi podatki:

- naštejemo lekseme,
- podamo prioriteto[5] in asociativnost leksemov,
- podamo pravila, kako se zaporedje leksemov spremeni v podatkovno strukturo.

V datoteki s končnico `.mll` opišemo, kako se zaporedja znakov pretvori v lekseme. Konkretno sintakso izrazov opisuje datoteka `parser.mly`:

parser.mly

```
1  %{
2    open Syntax
3  %}
4
5  /* Lexemes */
6  %token <int> NUMERAL
7  %token PLUS
8  %token MINUS
9  %token TIMES
10 %token DIVIDE
11 %token UMINUS
12 %token LPAREN
13 %token RPAREN
14 %token EOF
15
16 /* Precedence and associativity */
17 %left PLUS MINUS
```

[3] V angleščini imenujejo osnovni kos tudi "token". Kakšen je ustaljeni slovenski prevod?
[4] `EOF` pomeni "end of file". Običajno nas zanimajo nizi, ki so zapisani v datotekah.
[5] Prioriteta pove, kateri leksemi vežejo bolj kot drugi. Na primer, + ima *nižjo* prioriteto kot ·.

```
18 %left TIMES DIVIDE
19 %nonassoc UMINUS
20
21 /* Top level rule */
22 %start toplevel
23 %type <Syntax.expression> toplevel
24
25 %%
26
27 /* Grammar */
28
29 toplevel: e = expression EOF
30   { e }
31 ;
32
33 expression:
34   | n = NUMERAL                                 { Numeral n }
35   | e1 = expression TIMES  e2 = expression  { Times (e1, e2) }
36   | e1 = expression PLUS   e2 = expression  { Plus (e1, e2) }
37   | e1 = expression MINUS  e2 = expression  { Minus (e1, e2) }
38   | e1 = expression DIVIDE e2 = expression  { Divide (e1, e2) }
39   | MINUS e = expression %prec UMINUS        { Negate e }
40   | LPAREN e = expression RPAREN             { e }
41 ;
```

Med simbola `%{` in `%}` vstavimo pomožne definicije. Mi smo vstavili ukaz `open Izraz`, ki naredi vidne definicije iz modula `Izraz` (definirali ga bomo kasneje, vanj pa bomo postavili definicijo podatkovnega tipa `izraz`). Tako lahko namesto `Izraz.`*`ime`* pišemo kar *`ime`*.

V drugem delu naštejemo vse lekseme, ki se lahko pojavijo v slovničnih pravilih. Nekateri leksemi so lahko opremljeni še z dodatnim podatkom. Tako je v našem primeru leksem `STEVILO` opremljen s podatkom tipa `int`, namreč s številom, ki ga leksem predstavlja.

V tretjem delu podamo prioriteto in associativnost leksemov. Lekseme naštejemo po vrsti od najnižje do najvišje prioritete. Uporabimo lahko določila `%left` za levo associativnost, `%right` za desno associativnost in `%nonassoc` za lekseme, ki nimajo associativnosti. Ni treba našteti vseh leksemov, ampak samo tiste, ki imajo prioriteto.

V četrtem in petem delu datoteke `.mly` podamo slovnična pravila, ki opisujejo konkretno sintakso. Vsako slovnično pravilo nam pove, kako zaporedja leksemov pretvorimo v vrednosti v ocamlu. Ker je pravil lahko več in so medsebojno odvisna, moramo našteti glavna ali *začetna* pravila in podati tip vrednosti, ki ga dobimo s takim pravilom. V našem primeru je glavno pravilo `main`, ki vrne vrednost tipa `Izraz.izraz`. Slovnično pravilo je oblike

$$
\begin{aligned}
&\mathit{ime} : \\
&\quad \mathit{simbol} \cdots \mathit{simbol} \qquad \{\mathit{izraz}\} \\
&\quad \mid \cdots \\
&\quad \mid \mathit{simbol} \cdots \mathit{simbol} \qquad \{\mathit{izraz}\}\ ,
\end{aligned}
$$

pri čemer je *ime* ime pravila, simboli so imena pravil in leksemov, *izraz* pa je izraz v ocamlu, v katerega pretvorimo pripadajoči niz simbolov. V *izraz* se sklicujemo na posamezne simbole z `$1`, `$2`, ... Na primer, pomen pravila

```
| izraz PLUS izraz  {\bro} Plus (\$1, \$3) {\brc}
```

je: “Zaporedje oblike $izraz_1$, PLUS, $izraz_2$ pretvori v vrednost `Plus($1,$3)`, pri čemer se `$1` nanaša na $izraz_1$ in `$3` na $izraz_2$.” Opozorimo še na pravilo

```
| MINUS izraz \%prec UMINUS {\bro} UnarniMinus \$2 {\brc}
```

v katerem določilo `%prec UMINUS` pove, da naj se to pravilo obravnava s prioriteto leksema `UMINUS`, čeprav v njem nastopa leksem `MINUS`, ki ima nižjo prioriteto. Vloga leksema `UMINUS` je torej le v tem, da določa prioriteto pravila, saj lekser znak `'-'` vedno pretvori v leksem `MINUS`.

Oglejmo si še datoteko `lexer.mll`, v kateri opišemo lekseme:

lexer.mll

```
1  {
2    open Parser
3  }
4
5  rule lexeme = parse
6    | [' ' '\t' '\r' '\n']  { lexeme lexbuf }
7    | ['0'-'9']+  { NUMERAL (int_of_string (Lexing.lexeme lexbuf)) }
8    | '+'         { PLUS }
9    | '-'         { MINUS }
10   | '*'         { TIMES }
11   | '/'         { DIVIDE }
12   | '('         { LPAREN }
13   | ')'         { RPAREN }
14   | eof         { EOF }
```

V prvem delu smo uporabili `open Parser`, da se lahko na imena leksemov sklicujemo neposredno. V splošnem datoteka `.mll` vsebuje več pravil `rule ...`, mi pa imamo le eno, saj je leksična struktura aritmetičnih izrazov zelo preprosta. Leksemi so opisani s pravilom `rule leksem`, ki ga preberemo takole:

1. presledek, tabulator in znak za novo vrsto preskoči,

2. neprazno zaporedje znakov med `'0'` in `'9'` pretvori v leksem `STEVILO` n, pri čemer n dobimo tako, da s funkcijo `int_of_string` dani niz števk pretvorimo v celo število,

3. znake `+`, `-`, `*`, `/`, `(` in `)` pretvori v pripadajoče lekseme,

4. konec niza pretvori v leksem `EOF`.

Datoteko `parser.mly` pretvorimo v datoteko `parser.ml` z ukazom `ocamlyacc parser.mly`, datoteko `lexer.mll` pa z ukazom `ocamllex lexer.mll` v datoteko `lexer.ml`. Obe datoteki `.ml` nato prevedemo po običajnem postopku. Končni rezultat sta modula `Parser` in `Lexer`, ki vsebujeta funkciji:

- `Lexer.leksem` razdeli dani niz znakov na posamezne lekseme. Te funkcije nikoli ne uporabimo neposredno, ampak jo podamo kot argument `Parser.main`.

- `Parser.main` spremeni niz znakov (konkretna sintaksa) v izraz v ocamlu (abstraktna sintaksa).

Niz znakov lahko podamo neposredno kot niz *str* z izrazom

`Parser.main Lexer.leksem (Lexing.from_string` *str*`)` ,

na primer:

```
# #load "lexer.cmo" ;;
# #load "parser.cmo" ;;
# #load "izraz.cmo" ;;
# open Izraz ;;
# Parser.main Lexer.leksem (Lexing.from_string "(3 + 8) * 7") ;;
- : Izraz.izraz = Krat (Plus (Stevilo 3, Stevilo 8), Stevilo 7)
```

Lahko pa tudi podamo odprto datoteko *ch*, iz katere parser prebere niz:

`Parser.main Lexer.leksem (Lexing.from_channel` *ch*`)` .

Tu smo spoznali le osnovno uporabo orodij `ocamlyacc` in `ocamllex`. Zahtevnejše primere bomo še videli. Seveda ne gre izgubljati besed, da je treba za dobro razumevanje prebrati dokumentacijo, glej `http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`.

2.1.2 Od abstraktne h konkretni sintaksi

Obratni problem, prehod od abstraktne h konkretni sintaksi, je bistveno lažji in zahteva le kanček programerske iznajdljivosti. Sestaviti želimo funkcijo `string_of_izraz`, ki dani izraz v drevesni obliki pretvori v niz znakov. Da bo naloga bolj zanimiva, zahtevajmo, da konkretni zapis vsebuje samo tiste oklepaje, ki so potrebni. Tako se mora na primer izraz

`Plus (Plus (Stevilo 1, Stevilo 2), Stevilo 3)`

pretvoriti v niz `"1 + 2 + 3"`, izraz

`Plus (Stevilo 1, Plus (Stevilo 2, Stevilo 3))`

pa v niz `"1 + (2 + 3)"`, saj je `+` levo asociativen. Podobno se mora izraz

`Plus (Krat (Stevilo 1, Stevilo 2), Stevilo 3)`

pretvoriti v niz `"1 * 2 + 3"` in ne v niz `"(1 * 2) + 3"`, ki vsebuje nepotrebne oklepaje.

Vsakemu tipu izraza priredimo celoštevilsko *prioriteto* tako, da operacijam z višjo prioriteto priredimo večje število:

$$\texttt{Plus} \mapsto 0\,, \qquad \texttt{Minus} \mapsto 0\,, \qquad \texttt{Krat} \mapsto 1\,,$$
$$\texttt{Deljeno} \mapsto 1\,, \qquad \texttt{UnarniMinus} \mapsto 2\,, \qquad \texttt{Stevilo} \mapsto 3\,.$$

Vsakemu izraz priredimo prioriteto, ki je enaka prioriteti operacije v korenu izraza (zunanji operaciji). Nato definiramo pomožno funkcijo `to_str` n e, ki pretvori izraz e v niz s. Če je prioriteta izraza e manjša od n, potem niz s obdamo z oklepaji, sicer pa ne. S primernimi rekurzivnimi klici `to_str` lahko tako kontroliramo, kdaj se oklepaji pojavijo in kdaj ne. Končni izdelek je funkcija `string_of_izraz`:

```
1  let string_of_izraz e =
2    let rec to_str n e =
3      let (m, str) = match e with
4          Stevilo n ->      (3, string_of_int n)
5        | UnarniMinus e ->  (2, "-" ^ (to_str 0 e))
6        | Krat(e1,e2) ->    (1, (to_str 1 e1)^"␣*␣"^(to_str 2 e2))
7        | Deljeno(e1,e2) -> (1, (to_str 1 e1)^"␣/␣"^(to_str 2 e2))
8        | Plus(e1, e2) ->   (0, (to_str 0 e1)^"␣+␣"^(to_str 1 e2))
9        | Minus(e1,e2) ->   (0, (to_str 0 e1)^"␣-␣"^(to_str 1 e2))
10     in
11       if m < n then "(" ^ str ^ ")" else str
12   in
13     to_str (-1) e
```

Pomožno funkcijo `to_str` smo definirali z lokalno definicijo, ki je veljavna samo znotraj `string_of_izraz`. S tem sledimo pomembnemu programskemu načelu, da naj bodo vrednosti vidne samo tam, kjer jih potrebujemo. Funkcijo preizkusimo:

```
# string_of_izraz (Plus (Stevilo 1, Plus (Stevilo 2, Stevilo 3))) ;;
- : string = "1 + (2 + 3)"
# string_of_izraz (Plus (Plus (Stevilo 1, Stevilo 2), Stevilo 3)) ;;
- : string = "1 + 2 + 3"
# string_of_izraz (Krat (Plus (Stevilo 3, Stevilo 8), Stevilo 7)) ;;
- : string = "(3 + 8) * 7"
```

2.2 Evaluacija izrazov

Osnovna naloga v zvezi z aritmetičnimi operacijami je *evaluacija* ali izračun vrednosti izraza. Čeprav evaluacija celoštevilskih aritmetičnih izrazov ni pretirano komplicirana zadeva, jo bomo obravnavali podrobno, saj se bomo tako kasneje lažje spoprijeli z bolj kompliciranimi primeri evaluacije.

2.2.1 Semantika velikih korakov

Evaluacija je postopek, ki slika izraz v njegovo celoštevilsko vrednost. Dejstvo, da se izraz e evaluira v celo število n zapišemo z relacijo $\hookrightarrow$,

$$e \hookrightarrow n\ .$$

Taki vrsti evaluacije pravimo semantika *velikih korakov*, ker $\hookrightarrow$ določa, kako iz začetnega izraza e končni rezultat. Druga vrsta evaluacije je semantika *malih korakov*, glej 2.2.2, pri katerih povemo, kako naredimo le en korak v evaluaciji.

Pravila za izračun funkcije $\hookrightarrow$ podamo kot *pravila sklepanja*. Pravilo sklepanja je oblike

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_n}{Q}$$

in ga preberemo: "če veljajo predpostavke $P_1, \ldots, P_n$, potem velja A". Posebno pravilo sklepanja je *aksiom*, ki nima predpostavk in je oblike $\overline{Q}$. Pravila za evaluacijo aritmetičnih

izrazov izrazimo s pravili sklepanja takole:

$$\frac{}{\texttt{Stevilo}\ n \hookrightarrow n} \qquad \frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2}{\texttt{Plus}(e_1, e_2) \hookrightarrow n_1 + n_2}$$

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2}{\texttt{Minus}(e_1, e_2) \hookrightarrow n_1 - n_2} \qquad \frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2}{\texttt{Krat}(e_1, e_2) \hookrightarrow n_1 \cdot n_2}$$

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow n_2 \qquad n_2 \neq 0}{\texttt{Deljeno}(e_1, e_2) \hookrightarrow n_1 \div n_2} \qquad \frac{e \hookrightarrow n}{\texttt{UnarniMinus}\ e \hookrightarrow -n}$$

Pri pravilu za deljenje smo z $\div$ označili celoštevilsko deljenje. Pomembna podrobnost pri tem pravilu je dodatni pogoj $n_2 \neq 0$, ki pove, da smemo pravilo uporabiti samo, kadar delimo z neničelnim številom. Kaj se zgodi, ko pride do deljenja z nič bomo obravnavali v 3.5.1. Zaenkrat si zapomnimo, da ne obstaja število k, da bi veljalo $\texttt{Deljeno}(1, 0) \hookrightarrow k$. Pravimo, da evaluacija deljenja z nič *blokira*.

Pravila za evaluacijo brez težav pretvorimo v funkcijo `eval`, ki izračuna vrednost izraza:

```
1  let rec eval = function
2      Stevilo n -> n
3    | Plus (e1, e2) ->
4        let n1 = eval e1 and n2 = eval e2 in n1 + n2
5    | Minus (e1, e2) ->
6        let n1 = eval e1 and n2 = eval e2 in n1 - n2
7    | Krat (e1, e2) ->
8        let n1 = eval e1 and n2 = eval e2 in n1 * n2
9    | Deljeno (e1, e2) ->
10       let n1 = eval e1 and n2 = eval e2 in
11         if n2 <> 0 then n1 / n2 else failwith "Deljenje␣z␣nic"
12   | UnarniMinus e ->
13       let n = eval e in -n
```

To bi lahko zapisali nekoliko krajše tudi:

```
1  let rec eval = function
2      Stevilo n         -> n
3    | Plus (e1, e2)     -> eval e1 + eval e2
4    | Minus (e1, e2)    -> eval e1 - eval e2
5    | Krat (e1, e2)     -> eval e1 * eval e2
6    | Deljeno (e1, e2) ->
7        let n2 = eval e2 in
8          if n2 <> 0 then eval e1 / n2 else failwith "Deljenje␣z␣nic"
9    | UnarniMinus e     -> -(eval e)
```

Funkcijo preizkusimo:

```
# eval (Krat (Plus (Stevilo 3, Stevilo 8), Stevilo 7)) ;;
- : int = 77
```

Sestavimo sedaj lekser, parser, izraze in evaluacijo v preposto računalo, ki prebere izraz in izpiše njegovo vrednost. Najprej združimo definicijo izrazov in funkcije za delo z njimi v modul `Syntax`:

syntax.ml

```
(** Abstract syntax. *)

(** Arithmetical expressions. *)
type expression =
  | Numeral of int (** non-negative integer constant *)
  | Plus of expression * expression  (** Addition [e1 + e2] *)
  | Minus of expression * expression (** Difference [e1 - e2] *)
  | Times of expression * expression (** Product [e1 * e2] *)
  | Divide of expression * expression (** Quotient [e1 / e2] *)
  | Negate of expression (** Opposite value [-e] *)

(** Conversion of expresions to strings. *)
let string_of_expression e =
  let rec to_str n e =
    let (m, str) = match e with
        Numeral n       ->    (3, string_of_int n)
      | Negate e        ->    (2, "-" ^ (to_str 0 e))
      | Times (e1, e2)  ->    (1, (to_str 1 e1) ^ "␣*␣" ^ (to_str 2 e2))
      | Divide (e1, e2) ->    (1, (to_str 1 e1) ^ "␣/␣" ^ (to_str 2 e2))
      | Plus (e1, e2)   ->    (0, (to_str 0 e1) ^ "␣+␣" ^ (to_str 1 e2))
      | Minus (e1, e2)  ->    (0, (to_str 0 e1) ^ "␣-␣" ^ (to_str 1 e2))
    in
      if m < n then "(" ^ str ^ ")" else str
  in
    to_str (-1) e
```

Nato napišemo še glavni program `calc`, ki iz standardnega vhoda sprejema nize znakov, jih pretvori v izraze, evaluira in vrednost izpiše na standardni izhod. Poleg tega ulovi še nekatere izjeme, ki se zgodijo ob napakah, in izpiše ustrezna opozorila:

calc.ml

```
module Calc = Zoo.Toplevel(struct
  type toplevel = Syntax.expression
  type environment = unit
  let name = "calc"
  let options = []
  let help_directive = None
  let initial_environment = ()
  let prompt = "Calc>␣"
  let more_prompt = ">␣"
  let read_more _ = false
  let file_parser = None
  let toplevel_parser = Parser.toplevel Lexer.lexeme

  let exec _ interactive () e =
    let n = Eval.eval e in
      if interactive then print_endline (string_of_int n)
end) ;;

Calc.main ()
```

Celoten program bi lahko prevedli ročno. Bolj praktično pa je, da uporabimo program

`make`, ki mu v datoteko `Makefile` napišemo pravila, kako se program prevede. Na tem mestu ne bomo razlagali podrobnosti v zvezi z `Makefile`. Kogar to zanima, si lahko ogleda `calc/Makefile` v gradivu, ki je priloženu tem zapiskom. Preizkusimo končni izdelek:

```
\$ ./calc
Kalkulator. Pritisni Ctrl-D za konec.
> (3 + 8) * 7
77
> 10 - 5 - 3
2
> 1/0
Napaka: Deljenje z nic.
> 1 + (((
Napaka: napacen vnos.
> zblj
Napaka: napacen vnos.
>
Na svidenje.
```

Vse to smo dosegli z dobrimi 100 vrsticami izvorne kode! Resnici na ljubo pa moramo povedati, da bi morali za resno uporabo dodati vsaj boljšo obravnavo napak. Kadar je uporabnikov vnos nepravilen, bi moral program povedati, kakšna in kje je napaka v izrazu.

2.2.2 Semantika malih korakov

Običajni računalniki izvajajo računske korake enega za drugim. Relacija $\hookrightarrow$, ki določa vrednost aritmetičnega izraza, nam ne pove, kako pridemo do vrednosti z zaporedjem osnovnih ukazov, saj je definirana rekurzivno, kar odraža tudi njena implementacija `Izraz.eval`.

S semantiko *malih korakov* podamo pravila za evaluacijo izrazov z relacijo $\mapsto$,

$$e \mapsto e' \,,$$

ki pomeni "Izraz e se pretvori v izraz e' v enem računskem koraku". Nekateri izrazi so končni rezultati – iz njih ne pelje računski korak, saj je računanje zaključeno. Takim izrazom pravimo *vrednosti*. V našem primeru so vrednosti številske konstante, kasneje bomo pri ostalih programskih jezikih spoznali še druge. Torej:

$$\text{Vrednost } v ::= \texttt{Stevilo}\, n \,.$$

Pravila za $\mapsto$ so vsa narejena po istem kopitu, zato jih pojasnimo na primeru seštevanja. Osnovnošolec bi računal takole:

$$(2 \cdot 3 - 1) + (8 - 7) = (6 - 1) + (8 - 7) = 5 + (8 - 7) = 5 + 1 = 6.$$

Vsoto $e_1 + e_2$ izračunamo tako, da najprej izračunamo e_1, nato e_2 in seštejemo dobljena rezultata. Od tod razberemo pravila za $\mapsto$:

$$\frac{e_1 \mapsto e_1'}{\texttt{Plus}(e_1, e_2) \mapsto \texttt{Plus}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\texttt{Plus}(\texttt{Stevilo}\, n_1, e_2) \mapsto \texttt{Plus}(\texttt{Stevilo}\, n_2, e_2')}$$

$$\frac{}{\texttt{Plus}(\texttt{Stevilo}\, n_1, \texttt{Stevilo}\, n_2) \mapsto \texttt{Stevilo}(n_1 + n_2)}$$

Tako zapisana pravila nas prisilijo, da vedno najprej izračunamo e_1 in šele nato e_2, saj drugo pravilo dopušča računski korak v e_2 le v primeru, da je e_1 že vrednost $\texttt{Stevilo}\ n_1$.

Preostala pravila za $\mapsto$ se glasijo:

$$\frac{e_1 \mapsto e_1'}{\texttt{Minus}(e_1, e_2) \mapsto \texttt{Minus}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\texttt{Minus}(\texttt{Stevilo}\ n_1, e_2) \mapsto \texttt{Minus}(\texttt{Stevilo}\ n_2, e_2')}$$

$$\frac{}{\texttt{Minus}(\texttt{Stevilo}\ n_1, \texttt{Stevilo}\ n_2) \mapsto \texttt{Stevilo}(n_1 - n_2)}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{Krat}(e_1, e_2) \mapsto \texttt{Krat}(e_1', e_2)} \qquad \frac{e_2 \mapsto e_2'}{\texttt{Krat}(\texttt{Stevilo}\ n_1, e_2) \mapsto \texttt{Krat}(\texttt{Stevilo}\ n_2, e_2')}$$

$$\frac{}{\texttt{Krat}(\texttt{Stevilo}\ n_1, \texttt{Stevilo}\ n_2) \mapsto \texttt{Stevilo}(n_1 \cdot n_2)}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{Deljeno}(e_1, e_2) \mapsto \texttt{Deljeno}(e_1', e_2)}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{Deljeno}(\texttt{Stevilo}\ n_1, e_2) \mapsto \texttt{Deljeno}(\texttt{Stevilo}\ n_2, e_2')}$$

$$\frac{n_2 \neq 0}{\texttt{Deljeno}(\texttt{Stevilo}\ n_1, \texttt{Stevilo}\ n_2) \mapsto \texttt{Stevilo}(n_1 \div n_2)}$$

Evaluacija aritmetičnega izraza e poteka kot zaporedje računskih korakov

$$e \mapsto e' \mapsto e'' \mapsto \cdots$$

ki vodi do treh možnosti:

1. zaporedje pripelje do vrednosti in se ustavi,
2. zaporedje pripelje do izraza, ki ni vrednost in nima naslednjega koraka – pravimo, da evaluacija *blokira*,
3. zaporedje računskih korakov je neskončno – pravimo, da evaluacija *divergira*.

Prva možnost je običajna, druga se zgodi, če pride do deljenja z nič ali če je semantika slabo definirana (na primer, če bi pozabili podati pravila za odštevanje), tretja pa se pri aritmetičnih izrazih ne more zgoditi. Kasneje bomo obravnavali programske jezike, pri katerih lahko pride tudi do divergence.

S pomočjo relacije $\mapsto$ definiramo relacijo $\mapsto^*$,

$$e \mapsto^* e' ,$$

ki pomeni “e se pretvori v e' v nič ali več računskih korakih”:

$$\frac{e \mapsto e}{e \mapsto^* e} \qquad \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}$$

Zvezo med semantiko velikih in malih korakov pove naslednji izrek.

Izrek 2.1 *Za vsak aritmetični izraz e velja $e \hookrightarrow n$ natanko tedaj, ko $e \mapsto^* \texttt{Stevilo}\ n$.*

Dokaz. Glej nalogo 2.6. ∎

2.3 Izrazi s spremenljivkami

Razširimo sedaj aritmetične izraze s spremenljivkami. Abstraktni sintaksi dodamo določilo za spremenljivke:

$$\begin{aligned} \text{Izraz } e &::= s \mid x \mid e+e \mid e-e \mid e\cdot e \mid e/e \mid -e \\ \text{Število } s &::= [\mathtt{0}-\mathtt{9}]^+ \\ \text{Spremenljivka } x &::= [\mathtt{a}-\mathtt{zA}-\mathtt{Z}]^+ \,. \end{aligned}$$

Iz pravil je razvidno, da je spremenljivka neprazno zaporedje velikih in malih črk. Popravimo še podatkovni tip `izraz`:

```
type izraz =
  | Stevilo of int
  | \textit{Spremenljivka of string}
  | Plus of izraz * izraz
  | Minus of izraz * izraz
  | Krat of izraz * izraz
  | Deljeno of izraz * izraz
  | UnarniMinus of izraz
```

Na primer, izraz $2 \cdot x + 5$ predstavimo z vrednostjo

```
Plus (Krat (Stevilo 2, Spremenljivka "x"), Stevilo 5) .
```

Ko izraz evaluiramo, moramo poznati vrednosti spremenljivk, ki v njem nastopajo. Preslikavi, ki spremenljivke preslika v njihove vrednosti, pravimo *okolje* (angl. environment). Predstavimo ga z asociativnim seznamom $[(x_1, k_1); \ldots; (x_n, k_n)]$, ki pomeni, da je vrednost spremenljivke x_i celo število k_i. Če je η okolje in x_i spremenljivka, naj $\eta(x_i)$ pomeni vrednost, ki jo ima x_i v okolju η. Evaluacija izrazov s spremenljivkami je funkcija, ki sprejme poleg izraza še okolje, v katerem izraz evaluiramo. Zapis

$$\eta \mid e \hookrightarrow n$$

pomeni: "v okolju η je vrednost izraza e enaka n." Nova pravila za evaluacijo so zelo podobna starim, le da dodamo okolje kot dodatni argument in zapišemo še pravilo za računanje vrednosti spremenljivke:

$$\frac{}{\eta \mid \mathtt{Stevilo}\; n \hookrightarrow n} \qquad \frac{\eta(x) = n}{\eta \mid \mathtt{Spremenljivka}\; x \hookrightarrow n}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid \mathtt{Plus}(e_1, e_2) \hookrightarrow n_1 + n_2} \qquad \frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid \mathtt{Minus}(e_1, e_2) \hookrightarrow n_1 - n_2}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2}{\eta \mid \mathtt{Krat}(e_1, e_2) \hookrightarrow n_1 \cdot n_2} \qquad \frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2 \qquad \eta \mid n_2 \neq 0}{\eta \mid \mathtt{Deljeno}(e_1, e_2) \hookrightarrow n_1 \div n_2}$$

$$\frac{\eta \mid e \hookrightarrow n}{\eta \mid \mathtt{UnarniMinus}\; e \hookrightarrow -n}$$

Na primer, če je $\eta = [(\texttt{"x"}, 7); (\texttt{"y"}, 12)]$ je vrednost izraza $\texttt{Plus}(\texttt{Spremenljivka "x"}, \texttt{Stevilo } 5)$ v okolju η enaka 13:

$$\dfrac{\dfrac{\eta(\texttt{"x"}) = 7}{\eta \mid \texttt{Spremenljivka "x"} \hookrightarrow 7} \qquad \dfrac{}{\texttt{Stevilo } 5 \hookrightarrow 5}}{\eta \mid \texttt{Plus}(\texttt{Spremenljivka "x"}, \texttt{Stevilo } 5) \hookrightarrow 13}$$

Program `calc` razširimo v program `calc2`, v katerem lahko definiramo vrednosti spremenljivk z ukazom *x* `=` *e* in spremenljivke uporabimo v izrazih. Izvorno kodo programa si lahko ogledate v priloženem gradivu, program pa deluje takole:

```
Kalkulator s spremenljivkami. Pritisni Ctrl-D za konec.
> x = 5
x = 5
> y = x + 2
y = 7
> x + y * y
54
> z + 13
Napaka: neznana spremenljivka z.
```

2.4 Naloge

Naloga 2.2 Z uporabo pravil na strani 36 izpelji do vseh podrobnosti evaluacijo izraza $(\mathrm{y} + 3) \cdot (\mathrm{x} - \mathrm{x})$ v okolju $[(\mathrm{x}, 7); (\mathrm{y}, 8)]$.

Naloga 2.3 Pravila za evaluacijo ne predvidevajo, kako se obravnava deljenje z 0. Ena rešitev je, da bi izrazom dodali konstanti `posInf` in `negInf`, ki predstavljata pozitivno in negativno "neskončnost", ter pravili za evaluacijo:

$$\frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow 0 \qquad n_1 \geq 0}{\texttt{Deljeno}(e_1, e_2) \hookrightarrow \texttt{posInf}} \qquad \frac{e_1 \hookrightarrow n_1 \qquad e_2 \hookrightarrow 0 \qquad n_1 < 0}{\texttt{Deljeno}(e_1, e_2) \hookrightarrow \texttt{negInf}}$$

Zapiši *smiselna* pravila za računanje vsot, razlik, zmožkov in kvocientov, v katerih nastopata `posInf` in `negInf`. Kateri od naslednjih aritmetičnih zakonov se pri tem ohranijo:

1. asociativnost seštevanja: $(x + y) + z = x + (y + z)$,
2. asociativnost množenja: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,
3. komutativnost seštevanja: $x + y = y + x$,
4. komutativnost množenja: $x \cdot y = y \cdot x$,
5. distributivnost: $(x + y) \cdot z = x \cdot z + y \cdot z$.

Naloga 2.4 Definiraj abstraktno sintakso in pravila za evaluacijo izrazov, v katerih nastopajo števila s plavajočo vejico (tip `float`). Dodaj tudi osnovne elementarne funkcije sin, cos, exp in log. Nato vse skupaj še implementiraj.

Naloga 2.5 Dokaži, da evaluacija aritmetičnega izraza semantiko malih korakov nikoli ne divergira. Namig: dokaži, da iz $e \hookrightarrow e'$ sledi, da ima drevo e' manj vozlišč kot e.

Naloga 2.6 Ta naloga je težja. Dokaži izrek 2.1. Namig: izrek dokaži z indukcijo po strukturi izraza e.

Poglavje 3

Funkcijski programski jezik

Obravnavamo preprost funkcijski programski jezik in dokažemo, da je *varen*: program, ki ima tip, med izvajanjem ne more povzročiti napake.

Za *čisti* funkcijski programski jezik običajno štejemo programski jezik, v katerem so funkcije enakovredne drugim vrednostim in ki nima efektov. Funkcije so enakovredne drugim vrednostim, če jih lahko sprejemamo kot argumente in vračamo kot rezultate funkcij in jih na sploh uporabljamo kot navadne vrednosti.[1] *Efekt* ali *stranski učinek* je vsak programski konstrukt, s katerim lahko dosežemo, da dva zaporedna klica funkcije z enakimi argumenti vrneta različna rezultata.[2] V praksi je čistih funkcijskih programskih jezikov malo, saj niso pretirano koristni, ker ne morejo komunicirati z zunanjim svetom. Tipični (nečisti?) funkcijski programski jeziki, kot so Haskell, ML in scheme, zato vsebuejo efekte. Od teh je programski jezik Haskell "najčistejši", ker efekte predstavi kot *monade*. To so matematične strukture, s katerimi opišemo efekte in jih ločimo od čisto funkcijskega dela programskega jezika. S tem sledimo načelu, da naj bodo posamezni konstrukti v programskem jeziku med seboj neodvisni. Neodvisne koncepte lahko namreč obravnavamo ločeno, kar olajša njihovo učenje in analizo.[3] V tem poglavju bomo obravnavali čisti funkcijski programski jezik, efekti in monade pa bodo morali počakati na drugo priložnost.

3.1 MiniML

Programski jezik MiniML je minimalni primer čistega funkcijskega jezika, saj poleg funkcij pozna le dva osnovna tipa, `int` in `bool`. Zapišimo abstraktno sintakso MiniML, tokrat v

[1] V C/C++ sicer lahko uporabljamo *kazalce* na funkcije, samih funkcij pa ne moremo vračati kot rezultate. Zato C/C++ ne šteje za funkcijski jezik.

[2] Primeri efektov so: komunikacija z uporabnikom, izjeme, spremenljivke, ki jim lahko menjamo vrednost, paralelizem, nedeterminizem (naključna izbira) idr.

[3] Primeri kršitev načela ločenosti konceptov: v C ne moremo brati s standardnega vhoda, če prej ne razumemo kazalcev; v C++ in v Javi ne moremo definirati podatkovnega tipa seznam, ne da bi prej razumeli objekte in razrede.

bolj ohlapni obliki BNF:

$$
\begin{aligned}
\text{Tip } \tau &::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \\
\text{Izraz } e &::= n \mid x \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \\
&\quad\ \texttt{true} \mid \texttt{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid \\
&\quad\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \\
&\quad\ \texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e \mid e_1\ e_2 \\
\text{Število } n &::= \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \\
\text{Spremenljivka } x &::= x \mid y \mid z \mid \cdots
\end{aligned}
$$

Zaradi lažje berljivosti smo uporabili indekse in pisali $e_1 + e_2$ namesto $e + e$, v pravilu za funkcije pa smo celo uporabili f in x za oznako spremenljivk. Prav tako nismo podali natančnih pravil za števila in spremenljivke, ki si itak razvidna iz konkretne sintakse.

Osnovna tipa `int` in `bool` predstavljata cela števila in boolove vrednosti. Vrednosti tipa $\tau_1 \to \tau_2$ so funkcije, ki sprejmejo argument tipa τ_1 in vrnejo vrednost tipa τ_2. Operacija $\to$ veže desno asociativno in jo v programu zapišemo kot `->`. Tako je vrednost tipa $\tau_1 \to \tau_2 \to \tau_3$ funkcija, ki sprejme argument tipa τ_1 in nato še argument tipa τ_2 ter vrne vrednost tipa τ_3, med tem ko je vrednost tipa $(\tau_1 \to \tau_2) \to \tau_3$ funkcija, ki sprejme kot argument funkcijo tipa $\tau_1 \to \tau_2$ in vrne vrednost tipa τ_3.

V MiniML lahko tvorimo aritmetične izraze z množenjem, seštevanjem in odštevanjem. Namenoma smo izpustili deljenje, ki ga bomo obravnavali v 3.5.1. Z izrazi oblike $e_1 = e_2$ in $e_1 < e_2$ primerjamo cela števila. Sintakso izraza $\texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e$, s katerim definiramo funkcijo, bomo pojasnili v 3.2. Izraz $e_1\ e_2$ je aplikacija funkcije e_1 na argumentu e_2.

3.2 Vezane in proste spremenljivke ter substitucija

V nekaterih izrazih se pojavljajo tako imenovane *vezane* spremenljivke. Na primer, v izrazih

$$\sum_{i=0}^{n} a_i, \qquad \int_0^1 f(t)\, dt, \qquad \forall x \in A.\, \phi(x)\ .$$

so spremenljivke i, t in x *vezane*. To pomeni, da so "nevidne" zunaj izraza in da jih lahko vedno preimenujemo, ne da bi spremenili pomen izraza (seveda se novo ime ne sme mešati z ostalimi spremenljivkami, ki nastopajo v izrazu): izraza $\int_0^1 f(t)\, dt$ in $\int_0^1 f(x)\, dx$ štejemo za *enaka*, ker se razlikujeta le v imenu vezane spremenljivke. Spremenljivki, ki ni vezana, pravimo *prosta*. Izrazu, v katerem ni prostih spremenljivk, pravimo *zaprt* izraz. V programskih jezikih zaprtim izrazom pravimo *programi*.[4]

Substitucija je osnovna sintaktična operacija, v kateri *proste* spremenljivke zamenjamo z izrazi. Zapis

$$e[x_1 \to e_1, \ldots, x_n \to e_n]$$

pomeni: "v izrazu e *hkrati* zamenjaj proste spremenljivke x_1 z e_1, x_2 z e_2, ... in x_n z e_n." Pri tem moramo upoštevati, da proste spremenljivke, ki nastopajo v $e_1, \ldots, e_n$, s substitucijo ne smejo postati vezane. Če bi se to lahko zgodilo, moramo vezane spremenljivke

[4] V logiki zaprtim izjavam pravimo *stavki*.

v e prej preimenovati. Nekaj primerov bo razjasnilo ta pravila:

$$
\begin{aligned}
(x + y + 1)[x \to 2] &= 2 + y + 1 \, , \\
(x + y^2 + 1)[x \to y, y \to x] &= y + x^2 + 1 \\
((x + y^2 + 1)[x \to y])[y \to x] &= x + x^2 + 1 \, , \\
(x + \textstyle\int_0^1 x \cdot y \, , dx)[x \to 2] &= 2 + \textstyle\int_0^1 x \cdot y \, , dx \, , \\
(\textstyle\int_0^1 x \cdot y \, dx)[y \to x^2] &= \textstyle\int_0^1 t \cdot x^2 \, dt \, .
\end{aligned}
$$

Tudi v MiniML nastopajo vezane spremneljivke. V izrazu

$$\texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e$$

sta spremenljivki f in x vezani. Izraz predstavlja funkcijo, ki sprejme argument x tipa τ_1 in vrne rezultat e tipa τ_2. V podizrazu e lahko nastopata spremenljivki x in f, pri čemer se f sklicuje na funkcijo samo in nam omogoča, da zapišemo tudi rekurzivne funkcije. Na primer, funkcijo, ki izračuna vsoto prvih n naravnih števil, definiramo z izrazom

$$\texttt{fun}\ f(n : \texttt{int}) : \texttt{int}\ \texttt{is}\ \texttt{if}\ n = 0\ \texttt{then}\ 0\ \texttt{else}\ n + f(n - 1) \, .$$

S tem izrazom *nismo* definirali vrednosti `f`, ampak smo samo izrazili funkcijo *brez imena*, ki izračuna vsoto prvih n števil. V MiniML sploh ne moremo definirati nikakršnih vrednosti.[5]

3.3 Preverjanje tipov

Ni vsak izraz v MiniML smiselen, denimo `if` 2 `then` 5 `else` 8. Želeli bi, da prevajalnik ali tolmač za MiniML zavrne take izraze, saj zanje že v naprej vemo, da bodo ob izvajanju privedli do napake. V ta namen preverimo, ali je izraz pravilno sestavljen glede na tipe svojih podizrazov. Tako lahko `if` 2 `then` 5 `else` 8 zavrnemo, ker podizraz 2 ni tipa `bool`.

Tip izraza je v splošnem odvisen od tipov spremenljivk, ki se v njem pojavljajo. Na primer, brez podatka o tipu spremenljivke x ne moremo ugotoviti, ali izraz $x + 5$ ima tip `int`. Preslikavi, ki spremenljivke slika v pripadajoče tipe, pravimo *kontekst*. Ker imamo v programu vedno opravka le s končno mnogo spremenljivkami, lahko kontekst predstavimo kot končen seznam parov

$$x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \, .$$

Ta prestavitev pomeni, da ima x_i tip τ_i. Kontekste običajno označujemo z velikimi grškimi črkami Γ, Δ, Θ, ... Če se v kontekstu ista spremenljivka pojavi večkrat, za veljavno vzamemo njeno prvo pojavitev. Če ima glede na kontekst Γ spremenljivka x tip τ, pišemo $\Gamma(x) = \tau$.

Tromestna relacija

$$\Gamma \mid e : \tau$$

[5]Implementacija MiniML sicer omogoča globalne definicije, saj je pisanje programov brez definicij izjemno nadležno.

pomeni da ima izraz e tip τ glede na kontekst Γ. Definirana je z naslednjimi pravili sklepanja:

$$\frac{\Gamma(x) = \tau}{\Gamma \mid x : \tau} \qquad \frac{}{\Gamma \mid n : \mathtt{int}} \ (n \text{ je celo število})$$

$$\frac{\Gamma \mid e_1 : \mathtt{int} \qquad \Gamma \mid e_2 : \mathtt{int}}{\Gamma \mid e_1 + e_2 : \mathtt{int}} \ (\text{podobno za } - \text{ in } \cdot)$$

$$\frac{}{\Gamma \mid \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \mid \mathtt{false} : \mathtt{bool}}$$

$$\frac{\Gamma \mid e_1 : \mathtt{int} \qquad \Gamma \mid e_2 : \mathtt{int}}{\Gamma \mid e_1 = e_2 : \mathtt{bool}} \qquad \frac{\Gamma \mid e_1 : \mathtt{int} \qquad \Gamma \mid e_2 : \mathtt{int}}{\Gamma \mid e_1 < e_2 : \mathtt{bool}}$$

$$\frac{\Gamma \mid e_1 : \mathtt{bool} \qquad \Gamma \mid e_2 : \tau \qquad \Gamma \mid e_3 : \tau}{\Gamma \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

$$\frac{x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \mid e : \tau_2}{\Gamma \mid \mathtt{fun}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \mid e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid e_2 : \tau_1}{\Gamma \mid e_1\ e_2 : \tau_2}$$

Oglejmo si pobližje pravilo za preverjanje pravilnosti tipov v funkciji. Izraz $\mathtt{fun}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e$ ima tip $\tau_1 \to \tau_2$, če ima podizraz e tip τ_2 pri predpostavki, da ima x tip τ_1 in f tip $\tau_1 \to \tau_2$. To je edino pravilo, ki spremeni kontekst, saj vanj doda informacijo o tipu argumenta, $x : \tau_1$, in o tipu funkcije, $f : \tau_1 \to \tau_2$. To je potrebno zato, ker se lahko v podizrazu e spremenljivki x in f pojavita prosto, čeprav sta v celotnem izrazu vezani.

Pravila za preverjanje tipov imajo pomembno lastnost *inverzije*. Če velja $\Gamma \mid e : \tau$, potem lahko do tega sklepa pridemo z natanko enim od zgornjih pravil. Katero pravilo pride v upoštev, je razvidno iz abstraktne sintakse izraza e. Na primer, če je $e = e_1 + e_2$, lahko sklepamo, da ima e tip `int` edino na podlagi pravila za tip vsote. Sami se lahko prepričate, da imamo za vsako alternativo v abstraktni sintaksi natanko eno pripadajoče pravilo sklepanja za tip izraza. Inverzijo s pridom uporabljamo v raznih dokazih, kot je na primer naslednji.

Izjava 3.1 *V danem kontekstu Γ ima izraz e največ en tip.*

Dokaz. Izrek dokažemo z indukcijo po strukturi izraza e in uporabimo inverzijo. Če je e spremenljivka in ima tip τ, je ta enolično določen s kontekstom Γ. Če je e število ali Boolova vrednost, ima lahko samo tip `int`. Podobno lahki primeri so še aritmetična operacija, ki ima lahko samo tip `int`, primerjava, ki ima kvečjemu tip `bool`, in funkcija $\mathtt{fun}\ x(c : \tau_1) : \tau_2\ \mathtt{is}\ e$, ki ima kvečjemu tip $\tau_1 \to \tau_2$. Preostaneta nam še pogojni stavek in aplikacija, ki ju obravnavamo z indukcijskim korakom.

Denimo, da ima $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ v kontekstu Γ tip. Zaradi inverzije ima isti tip tudi e_2, ki pa ima po indukcijski predpostavki en sam tip. Torej ima tudi e natanko en tip.

Če ima $e = e_1 e_2$ v kontekstu Γ tip τ, potem ima zaradi inverzije e_1 v istem kontekstu tip $\sigma \to \tau$ za neki tip. Po indukcijski predpostavki za e_1 velja, da je $\sigma \to \tau$ enolično določen, zato je tudi τ enolično določen. ∎

Pravila za preverjanje tipov nam torej določajo funkcijo, ki v danem kontekstu Γ izračuna tip izraza e. Funkcija ni vedno definirana, saj nimajo vsi izrazi tipa. Implementacijo funkcije si lahko ogledate v priloženi izvorni kodi za MiniML.

3.4 Evaluacija

Posvetimo se še pravilom za evaluacijo izrazov v MiniML. Smiselno je izvajati samo *programe*, to je zaprte izraze. Če namreč izraz vsebuje kako prosto spremenljivko, potem ne poznamo njenega tipa in vrednosti, zato bi bilo tak izraz nesmiselno izvajati.

Uporabili bomo semantiko malih korakov, ker želimo natančno razumeti, kako poteka evaluacija programa. Kakor v 2.2.2 definiramo relaciji

$$p \mapsto p' , \qquad\qquad p \mapsto^* p' ,$$

kjer $p \mapsto p'$ pomeni "p preide v p' v enem računskem koraku" in $p \mapsto^* p'$ pomeni "p preide v p' v enem ali več računskih korakih". Relacijo $\mapsto^*$ izpeljemo iz $\mapsto$ s praviloma

$$\frac{p \mapsto p'}{p \mapsto^* p'} \qquad \frac{p \mapsto p' \quad p' \mapsto^* p''}{p \mapsto^* p''} .$$

Izvajanje programa p predstavimo kot zaporedje posameznih računskih korakov $p \mapsto p' \mapsto p'' \mapsto p''' \mapsto \cdots$. Tako zaporedje je lahko tudi neskončno in v tem primeru pravimo, da p *divergira*. Izvajanje programa je končano, če program doseže *vrednost*, ki je bodisi število, `true`, `false`, ali funkcija:

$$\text{Vrednost } v ::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e .$$

Vrednosti so torej možni končni rezultati programa.

Sedaj definirajmo še funkcijo $p \mapsto p'$. Ker je p zaprt izraz, ne more biti spremenljivka. Če je število, `true` ali `false`, je že vrednost in nima naslednjega računskega koraka. Za seštevanje imamo tri pravila:

$$\frac{n = n_1 + n_2}{n_1 + n_2 \mapsto n} \qquad \frac{e_1 \mapsto e_1'}{e_1 + e_2 \mapsto e_1' + e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1 + e_2 \mapsto n_1 + e_2'}$$

Prvo pravilo pove, da vsoto števil izračunamo tako, da števili seštejemo, drugi dve pravili pa, da v splošni vsoti najprej izračunamo prvi in nato drugi seštevanec. Podobna pravila veljajo za množenje in odštevanje, videli smo jih že v 2.2.2:

$$\frac{n = n_1 - n_2}{n_1 - n_2 \mapsto n} \qquad \frac{e_1 \mapsto e_1'}{e_1 - e_2 \mapsto e_1' - e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1 - e_2 \mapsto n_1 - e_2'}$$

$$\frac{n = n_1 \cdot n_2}{n_1 \cdot n_2 \mapsto n} \qquad \frac{e_1 \mapsto e_1'}{e_1 \cdot e_2 \mapsto e_1' \cdot e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1 \cdot e_2 \mapsto n_1 \cdot e_2'}$$

Pravila za primerjanje celih števil se glasijo

$$\frac{n_1 = n_2}{n_1 = n_2 \mapsto \texttt{true}} \qquad \frac{n_1 \neq n_2}{n_1 = n_2 \mapsto \texttt{false}} \qquad \frac{e_1 \mapsto e_1'}{e_1 = e_2 \mapsto e_1' = e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1 = e_2 \mapsto n_1 = e_2'}$$

$$\frac{n_1 \geq n_2}{n_1 < n_2 \mapsto \texttt{false}} \qquad \frac{n_1 < n_2}{n_1 < n_2 \mapsto \texttt{true}} \qquad \frac{e_1 \mapsto e_1'}{e_1 < e_2 \mapsto e_1' < e_2} \qquad \frac{e_2 \mapsto e_2'}{n_1 < e_2 \mapsto n_1 < e_2'}$$

Pravila za izvajanje pogojnega stavka:

$$\frac{}{\texttt{if true then } e_2 \texttt{ else } e_3 \mapsto e_2} \qquad \frac{}{\texttt{if false then } e_2 \texttt{ else } e_3 \mapsto e_3}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mapsto \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3}$$

Pomembno je, da *ne* izvedemo obeh vej pogojnega stavka. Pravila za izvajane funkcije ni, saj so funkcije vrednosti. Pravila za izvajanje aplikacije se glasijo:

$$\frac{v_1 = \texttt{fun } f(x : \tau_1) : \tau_2 \texttt{ is } e}{v_1\ v_2 \mapsto e[x \to v_2, f \to v_1]} \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{e_1\ e_2 \to e_1\ e_2'}$$

Primer izvajanja programa:

$$\texttt{if } 2 < 3 \texttt{ then } 7 + 2 \texttt{ else } 8 \mapsto \texttt{if true then } 7 + 2 \texttt{ else } 8 \mapsto 7 + 2 \mapsto 9$$

Izvajanje je doseglo vrednost 9, ki nima naslednjega računskega koraka. Pravila, ki smo jih podali, so *deterministična*, kar pomeni, da je naslednji računski korak enolično določen, če obstaja.

3.5 Varnost MiniML

Pravila za izvajanje lahko uporabimo na poljubnih zaprtih izrazih, tudi takih, ki so nesmiselni:

$$\texttt{if } 2 + 3 \texttt{ then } 7 \texttt{ else } 8 \mapsto \texttt{if } 5 \texttt{ then } 7 \texttt{ else } 8\ .$$

Naslednjega koraka v evaluaciji ni. V poštev bi lahko prišla le pravila za evaluacijo pogojnega stavka, vendar jih ne moremo uporabiti. Dve izmed njih pričakujeta, da je pogoj `true` ali `false`, tretje pa, da ima pogoj naslednji računski korak. Izraz 5 ne ustreza nobeni od teh zahtev. Pri evaluaciji programa p lahko torej nastopijo tri možnosti:

- *p se evaluira v vrednost v:* obstaja končno zaporedje $p \mapsto p' \mapsto p'' \mapsto \cdots \mapsto v$, kjer je v vrednost. Vrednosti seveda nimajo naslednjega koraka v izvajanju.

- *p blokira:* obstaja končno zaporedje $p \mapsto p' \mapsto p'' \mapsto \cdots \mapsto q$, kjer q *ni* vrednost in nima naslednjega koraka v izvajanju.

- *p divergira:* obstaja neskončno zaporedje $p \mapsto p' \mapsto p'' \mapsto \cdots$

Divergenca programov je v splošnem programskem jeziku neizbežna,[6] evaluacija v vrednost je seveda zaželena, medtem ko bi se programom, ki blokirajo, radi izognili. Le-ti so namreč primerki programov, ki ob izvajanju sprožijo napako.[7] Ker od programerjev seveda ne moremo pričakovati, da bodo pisali samo delujoče programe, je naloga programskega jezika, da prepreči izvajanje programov, ki blokirajo. Najboljše zagotovilo, da programski jezik res ima to lastnost, je ustrezen matematični izrek.

Izrek 3.2 (Varnost) *Če program v MiniML ima tip, bodisi divergira bodisi se evaluira v vrednost.*

Izrek o varnosti nam pove, da lahko programe, ki imajo tip, izvajamo brez skrbi, saj prevajalnik ali tolmač za MiniML zagotovi varnost tako, da zavrne programe brez tipa. Dokaz izreka o varnosti razdelimo na dva dela, napredek in ohranitev.[8] Dokaz izreka o varnosti je na strani 46.

Izrek 3.3 (Napredek) *Če ima program p tip, je p vrednost, ali pa obstaja tak program p', da velja $p \mapsto p'$.*

Dokaz. Izrek si zapomnimo takole: "veljaven program je bodisi vrednost bodisi ima naslednji računski korak." Izrek dokažemo z indukcijo po strukturi programa p. Ker so si vsi primeri podobni, obravnavamo le enega, ostale pustimo za vajo.

Če je p vsota $p = p_1 + p_2$ in ima tip, potem je to nujno `int`, prav tako pa imata p_1 in p_2 tip `int`. Po indukcijski predpostavki je vsak od p_1 in p_2 bodisi vrednost bodisi ima naslednji računski korak. To nam da tri možnosti: (1) če $p_1 \mapsto p_1'$, potem $p \mapsto p_1' + p_2$, (2) če je p_1 vrednost in $p_2 \mapsto p_2'$, potem $p \mapsto p_1 + p_2'$ in (3) če sta p_1 in p_2 vrednosti, sta nujno enaka nekima številoma $p_1 = n_1$ in $p_2 = n_2$, torej velja $p \mapsto n$, kjer je $n = n_1 + n_2$.

Po istem kopitu dokažemo tudi ostale primere. Iz predpostavke, da ima p tip in da ni vrednost, sklepamo, da imajo tipe tudi njegovi sestavni deli, zato so po indukcijski predpostavki bodisi vrednosti bodisi imajo naslednji korak v evaluaciji. Od tod sledi, da ima tudi p naslednji korak. ∎

Izrek 3.4 (Ohranitev) *Če ima program p tip τ in velja $p \mapsto p'$, ima tudi p' tip τ.*

Dokaz. Izrek o ohranitvi si zapomnimo takole: "med evaluacijo se tip veljavnega programa ohranja". Tudi ta izrek dokažemo z indukcijo po strukturi programa p. Ker p po predpostavki ima naslednji računski korak, ni vrednost.

Če je $p = p_1 + p_2$, je $\tau =$ `int`, zaradi inverzije pa imata tudi p_1 in p_2 tip `int`. Računski korak $p \mapsto p'$ sledi iz enega od treh pravil za evaluacijo vsote:

- Če $p_1 \mapsto p_1'$, po indukcijski predpostavki velja, da ima p_1' tip `int`, torej ima tudi $p_1' + p_2$ tip `int`. Ker pa je evaluacija enolična, velja $p' = p_1' + p_2$.
- Če je p_1 vrednost in $p_2 \mapsto p_2'$, po indukcijski predpostavki velja, da ima p_2' tip `int`. Torej ima tip `int` tudi izraz $p' = p_1 + p_2'$.

[6] Dokazati je mogoče, da vsak programski jezik, v katerem lahko izrazimo vse izračunljive funkcije $\mathbb{N} \to \mathbb{N}$, vsebuje divergentne programe.

[7] V praksi se takšne napake kažejo kot neveljavni dostopi do pomnilnika, neveljavni strojni ukazi in druge oblike nedovoljenih ali nedefiniranih stanj procesorja, zaradi katerih se program "sesuje".

[8] Ta pristop k dokazovanju varnosti programskih jezikov je populariziral Robert Harper s sloganom "varnost = napredek + ohranitev".

- Če sta p_1 in p_2 vrednosti, je $p_1 = n_1$ in $p_2 = n_2$ za neki števili n_1 in n_2. Tedaj je tudi p' število, namreč $n = n_1 + n_2$ in ima tip `int`.

Podobno poteka dokaz za ostale aritmetične operacije in primerjave števil, prav tako ni nič novega pri pogojnem stavku. Funkcij ni treba obravnavati, ker so vrednosti, tako da nam ostane le še aplikacija.

Če je $p = p_1 p_2$, z inverzijo sklepamo, da ima p_1 tip $\tau_1 \to \tau_2$ in p_2 tip τ_1 za neki τ_1. Spet obravnavamo tri primere, glede na to ali p_1 in p_2 imata naslednji računski korak. Od teh je zanimiv le primer, ko sta oba vrednosti, se pravi $p_1 = v_1 = \texttt{fun}\ f(x : \tau_1) : \tau\ \texttt{is}\ e$ in $p_2 = v_2$. V tem primeru je $p' = e[x \to v_2, f \to v_1]$. Dokaz, da ima p' tip τ sledi iz dvakratne uporabe naslednje leme. ∎

Lema 3.5 *Substitucija ohranja tipe: če velja* $\Gamma_1, x : \sigma, \Gamma_2 \mid e : \tau$ *in* $\Gamma_1, \Gamma_2 \mid e_1 : \sigma$*, potem* $\Gamma_1, \Gamma_2 \mid e[x \to e_1] : \tau$*.*

Dokaz. Tu smo privzeli, da se x ne pojavalja v Γ_1, saj moramo vedno obravnavati prvo pojavitev x v kontekstu. Dokaz poteka z indukcijo po strukturi izraza e, ki jo razdelimo na štiri vrste primerov:

- Če je e število, `true` ali `false`, opazimo, da velja $\Gamma_1, \Gamma_2 \mid e : \tau$ (se pravi, $x : \sigma$ lahko odstranimo iz konteksta), in $e[x \to e_1] = e$.

- Če je e spremenljivka, je bodisi $e = x$ bodisi $e = y \neq x$. V prvem primeru velja $\tau = \sigma$ in $e[x \to e_1] = e_1$, ki res ima tip $\sigma = \tau$ v kontekstu Γ_1, Γ_2. V drugem primeru velja $\Gamma_1, \Gamma_2 \mid y : \tau$ in $y[x \to e_1] = y$.

- Če je e aritmetični izraz, primerjava, pogojni stavek ali aplikacija, je dokaz preprost in ga prepuščamo za vajo.

- Če je $e = \texttt{fun}\ f(y : \tau_1) : \tau_2\ \texttt{is}\ e'$ velja $\tau = \tau_1 \to \tau_2$ in predpostaviti smemo $x \neq y$ in $x \neq f$, sicer vezani spremenljivki f in y preimenujemo. Od tod sledi

  $$(\texttt{fun}\ f(y : \tau_1) : \tau_2\ \texttt{is}\ e')[x \to e_1] = \texttt{fun}\ f(y : \tau_1) : \tau_2\ \texttt{is}\ (e'[x \to e_1])\ .$$

  Ker velja $\Gamma_1, x : \sigma, \Gamma_2, f : \tau_1 \to \tau_2, y : \tau_1 \mid e' : \tau_2$, po indukcijski predpostavki sledi, $\Gamma_1, \Gamma_2, y : \tau_1, f : \tau_1 \to \tau_2 \mid e'[x \to e_1] : \tau_2$, od koder smemo sklepati $\Gamma_1, \Gamma_2 \mid: \texttt{fun}\ f(y : \sigma) : \tau_1\ \texttt{is}\ (e'[x \to e_1]) : \tau_1 \to \tau_2$.

∎

Končno dokažimo še izrek o varnosti MiniML.

Dokaz izreka 3.2. Denimo, da ima program p tip τ. Če p divergira, izrek velja, sicer pa obstaja končno zaporedje

$$p = p_0 \mapsto p_1 \mapsto \cdots \mapsto p_n\ ,$$

kjer p_n nima naslednjega koraka v evaluaciji. Dokazati moramo, da je p_n vrednost. Ker ima p_0 tip τ in $p_0 \mapsto p_1$, ima po izreku o ohranitvi tudi p_1 tip τ. Enak razmislek pokaže, da imajo tip τ tudi p_2, p_3 in tako naprej do p_n. Torej je p_n program, ki ima tip τ in nima naslednjega koraka v evaluaciji. Po izreku o napredku je p_n vrednost. ∎

3.5.1 Deljenje in napake ob izvajanju

MiniML je idealiziran programski jezik. Od praktično uporabnega jezika ga loči marsikaj, predvsem pa dejstvo, da ne vsebuje nobenih operacij, ki bi lahko povzročile napako ob izvajanju. Realistični programski jeziki komunicirajo z zunanjim svetom, zato se lahko med izvajanjem programa marsikaj zatakne. Pojavi se vprašanje, kako naj se program v takih primerih odzove.

MiniML smo opremili s seštevanjem, odštevanjem in množenjem. Deljenje smo namenoma izpustili, ker morebitno deljenje z nič povzroči napako ob izvajanju. Težavo lahko razrešimo na nekaj načinov:

1. Idealna rešitev bi seveda bila, če bi prevajalnik preveril, ali bo v programu prišlo do deljenja z nič, in bi že ob prevajanju javil tako napako. Na žalost je to nemogoče, saj nas teorija izračunljivosti uči, da je *v splošnem* nemogoče preveriti, ali bo dani program naredil dano operacijo. Poleg tega je lahko izvajanje programa odvisno od podatkov, ki so ob času prevajanja neznani, na primer uporabnikovi premiki miške. Prevajalnik zato ne more vedno napovedati, ali bo prišlo do napake.

2. Namesto, da bi bilo deljenje z nič nedefinirano, uvedemo posebno vrednost $\pm\infty$ in definiramo $e/0 = \pm\infty$. To rešitev podpira IEEE standard za aritmetiko v pomični vejici. Če v javi ali ocamlu evaluiramo `1.0 /. 0.0`, dobimo odgovor `infinity`. Slabost te rešitve je predvsem, da pokvari osnovne lastnosti aritmetičnih operacij, kot so komutativnost, asociativnost in distributivnost. To privede do napak, saj programerji te lastnosti običajno privzamejo.

3. V definiciji evaluacije $\mapsto$ ne povemo, kako se evaluira deljenje z nič. To odločitev prepustimo tistemu, ki MiniML implementira na konkretnem računalniku. Čeprav ta pristop v nekaterih primerih omogoča praktično rešitev, je s stališča načrtovanja programskega jezika slab. Programski jezik, ki ni *do potankosti* definiran, ni prenosljiv in zahteva od programerja poznavanje posebnosti posameznih prevajalnikov in arhitektur.

4. Če pride do deljenja z nič, se sproži *izjema*, to je posebno sporočilo o napaki, ki prekine izvajanje programa.

Odločimo se za zadnjo rešitev. Splošne izjeme so učinkovit mehanizem za javljanje napak in posebnih dogodkov v programu. Omejimo se samo na najpreprostejšo verzijo izjeme, to je *napako*. V ta namen uvedemo posebno vrednost `error`,

$$\text{Vrednost } v ::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\, f(x : \tau_1) : \tau_2 \;\texttt{is}\; e \mid \texttt{error}\,,$$

ki ob izvajanju "požre" vse ostale vrednosti:

$$\frac{}{\texttt{error} + e_2 \mapsto \texttt{error}} \qquad \frac{}{v_1 + \texttt{error} \mapsto \texttt{error}}$$

(podobna pravila za $\cdot$, $-$, $=$ in $<$)

$$\frac{}{\texttt{if error then}\; e_2 \;\texttt{else}\; e_3 \mapsto \texttt{error}} \qquad \frac{}{\texttt{error}\; e_2 \mapsto \texttt{error}} \qquad \frac{}{v_1\; \texttt{error} \mapsto \texttt{error}}$$

Sedaj lahko dodamo MiniML deljenje e_1/e_2. Pravila za preverjanje tipov prepuščamo za vajo. Pravila za evaluacijo so (z $\div$ označimo celoštevilsko deljenje):

$$\frac{e_1 \mapsto e_1'}{e_1/e_2 \mapsto e_1'/e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1/e_2 \mapsto v_1'/e_2'} \qquad \frac{n_2 \neq 0 \qquad n = n_1 \div n_2}{n_1/n_2 \mapsto n} \qquad \frac{}{n_1/0 \mapsto \texttt{error}}$$

Ostane še vprašanje, kakšen tip ima vrednost `error`. Ker se lahko pojavi kjerkoli v programu, nam preostane ena sama možnost – `error` ima *vsak* tip:

$$\frac{}{\Gamma \mid \texttt{error} : \tau} \ .$$

Izrek o varnosti MiniML nam je zagotavljal, da veljavni programi ne blokirajo. Enak izrek velja tudi za MiniML obogaten z `error`.

Izrek 3.6 (Varnost MiniML+`error`) *Če program v MiniML z* error *ima tip, bodisi divergira bodisi se evaluira v vrednost.*

Dokaz. Dokaz poteka enako kot dokaz izreka 3.2 o varnosti MiniML, le da je ena izmed možnih vrednosti `error`. Podrobnosti prepuščamo za vajo. ∎

3.6 Učinkovita implementacija MiniML

Kot smo že omenili, so pravila za evauacijo MiniML deterministična, zato jih zlahka prepišemo v funkcijo, ki evaluira dani program.[9] Tako funkcijo sicer lahko uporabimo kot osnovo za tolmač, vendar je neučinkovita iz vsaj dveh razlogov. V vsakem koraku najprej poiščemo podizraz, v katerem izvedemo naslednji računski korak. Tako iskanje je nepotrebno, saj lahko program prevedemo tako, da je vedno jasno, kaj je naslednji korak. Drugi razlog za neučinkovitost je pravilo za evaluacijo aplikacije, ki zahteva, da argument substituiramo v telo funkcije. Tudi to je potratna operacija, saj je treba ob vsakem klicu funkcije pregledati njeno celotno telo. V tem razdelku se posvetimo *prevajalniku* za MiniML, ki program prevede v kodo za učinkovit in preprost *abstraktni stroj*.

3.6.1 Abstraktni stroj

Abstraktni stroj, ki ga bomo opisali, je načeloma podoben sodobnim procesorjem: sestoji iz treh skladov in po vrsti izvaja *preproste* ukaze. Do pravega stroja ga loči predvsem poenostavljena obravnava spremenljivk in dejstvo, da se zanaša na ocamlovo upravljanje s pomnilnikom. Stroj sestoji iz treh skladov:

Sklad okvirjev: okvir je zaporedje ukazov, ki predstavlja telo funkcije ali vejo v pogojnem stavku. Stroj po vrsti izvaja ukaze v prvem okvirju na skladu. Ko teh zmanjka, preide na naslednji okvir. Kontrolna ukaza za pogojni stavek in klic podprograma lahko na sklad postavita nov okvir.[10]

[9] Glej funkcijo `Miniml.eval` v priloženi implementaciji MiniML.

[10] Da ne boste mislili, da ocaml dejansko kopira naokoli celotne sezname ukazov. Ko dodamo ali odvzamemo okvir s sklada, je potrebno spremeniti samo enega ali dva kazalca v pomnilniku.

Sklad vrednosti: stroj računa s *strojnimi vrednostmi*, ki so cela števila, Boolove vrednosti in zaprtja, ki predstavljajo funkcije in bomo o njih še govorili. Aritmetične in druge operacije vzamejo vrednosti z vrha sklada in postavijo rezultat nazaj na sklad. Paziti moramo, da se na vrhu sklada vedno znajdejo ravno prave vrednosti.

Sklad okolij: okolje je seznam $[(x_1, m_1), \ldots, (x_k, m_k)]$, ki pove, kakšne strojne vrednosti imajo spremenljivke $x_1, \ldots, x_k$. Trenutno veljavno okolje je tisto, ki je na vrhu sklada okolij. Okolje je treba zamenjati pri klicu in vračanju iz podprograma.

Ker je MiniML funkcijski programski jezik, lahko funkcije sprejemamo in vračamo kot argumente, zato mora tudi abstraktni stroj obravnavati funkcije kot običajne vrednosti. Funkcijo $\mathtt{fun}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e$ predstavimo z zaporedjem ukazov, ki jih dobimo, ko prevedemo telo e. Vendar pa to ni vse, saj se lahko v e pojavijo spremenljivke. Na primer, v izrazu

$$(\mathtt{fun}\ f(x : \mathtt{int}) : \mathtt{int}\ \mathtt{is}\ ((\mathtt{fun}\ g(y : \mathtt{int}) : \mathtt{int}\ \mathtt{is}\ x + y)\ 5))\ 3$$

se funkcija $\mathtt{fun}\ g(y : \mathtt{int}) : \mathtt{int}\ \mathtt{is}\ x + y$ sklicuje na spremenljivko x, katere vrednost moramo ohraniti. Zato funkcijo prevedemo v urejeno trojico (x, c, η), ki se imenuje *zaprtje*, pri čemer je x ime argumenta, okvir c je prevedeni e in η okolje, v katerem funkcija nastopa. Ko funkcijo kličemo, η razširimo z vrednostjo argumenta x, ga postavimo na sklad okolij ter postavimo c na sklad okvirjev. Zadnji ukaz v c odstrani okolje s sklada.

Trenutno sranje abstraktnega stroja predstavimo s trojico skladov (F, S, E), kjer je F sklad okvirjev, S sklad strojnih vrednosti in E sklad okolij. Strojni ukazi so preproste preslikave, ki sprejmejo trenutno stanje (F, S, E) in vrnejo novo stanje (F', S', E'). Če ukaz u stanje (F, S, E) spremeni v (F', S', E'), to zapišemo

$$(F, S, E) \xrightarrow{u} (F', S', E')\ .$$

Ukaze delimo v dve skupini:

Računski ukazi bodisi postavijo vrednost na sklad, bodisi računajo z vrednostmi, ki so že na skladu. Sklada okvirjev in sklada okolij ne spreminjajo:

- celoštevilska konstanta $(F, S, E) \xrightarrow{\mathtt{int}\,n} (F, n :: S, E)$, kjer je $n \in \mathbb{Z}$,
- boolova konstanta, $(F, S, E) \xrightarrow{\mathtt{bool}\,b} (F, b :: S, E)$, kjer je $b \in \{\mathtt{true}, \mathtt{false}\}$,
- ukaz $\mathtt{Closure}(f, x, c)$ na sklad postavi zaprtje:

  $$(F, S, \eta :: E) \xrightarrow{\mathtt{Closure}(f,x,c)} (F, z :: S, \eta :: E)\ ,$$

  kjer je $z = (x, c, (f, z) :: \eta)$,
- vrednost spremenljivke $(F, S, \eta{::}E) \xrightarrow{\mathtt{Var}\,x} (F, \eta(x){::}S, \eta{::}E)$, kjer je $\eta(x)$ vrednost spremenljivke x glede na okolje η,
- množenje $(F, n_1 :: n_2 :: S, E) \xrightarrow{\mathtt{Mult}} (F, (n_2 \cdot n_1) :: S, E)$,
- seštevanje $(F, n_1 :: n_2 :: S, E) \xrightarrow{\mathtt{Add}} (F, (n_2 + n_1) :: S, E)$,
- odštevanje $(F, n_1 :: n_2 :: S, E) \xrightarrow{\mathtt{Sub}} (F, (n_2 - n_1) :: S, E)$,
- enakost $(F, n_1 :: n_2 :: S, E) \xrightarrow{\mathtt{Equal}} (F, (n_2 = n_1) :: S, E)$,

- manjše $(F, n_1 :: n_2 :: S, E) \stackrel{\texttt{Less}}{\longrightarrow} (F, (n_2 < n_1) :: S, E)$.

Ukaza za primerjavo na sklad postavita strojno vrednost `true` ali `false`.

Kontrolni ukazi nadzorujejo izvajanje pogojnih stavkov in klice funkcij:

- pogojni stavek $\texttt{Branch}(c_1, c_2)$ opisujeta pravili

$$(F, \texttt{true} :: S, E) \stackrel{Branch(c_1,c_2)}{\longrightarrow} (c_1 :: F, S, E) ,$$
$$(F, \texttt{false} :: S, E) \stackrel{Branch(c_1,c_2)}{\longrightarrow} (c_2 :: F, S, E) .$$

- klic funkcije s sklada vzame vrednost argumenta in zaprtje ter na sklad okvirjev postavi okvir iz zaprtja in na sklad okolij okolje iz zaprtja, razširjeno z vrednostjo argumenta:

$$(F, v :: (x, c, \eta) :: S, E) \stackrel{\texttt{Call}}{\longrightarrow} (c :: F, S, ((x, v) :: \eta) :: E) ,$$

- ukaz `PopEnv` odstrani okolje s sklada: $(F, S, \eta :: E) \stackrel{\texttt{PopEnv}}{\longrightarrow} (F, S, E)$.

Abstraktni stroj izvaja ukaze iz okvirja na vrhu sklada okvirjev. Ko teh zmanjka, preide na naslednji okvir na skladu. Stroj se ustavi, če je sklad okvirjev prazen. V tem primeru je *končni rezultat* vrednost na vrhu sklada vrednosti. Delovanje stroja formalno opišemo s preslikavo

$$(F, S, E) \mapsto (F', S', E')$$

ki napravi en korak v izvajanju:

$$\frac{}{([\,] :: F, S, E) \mapsto (F, S, E)} \qquad \frac{(c :: F, S, E) \stackrel{u}{\longrightarrow} (F', S', E')}{((u :: c) :: F, S, E) \mapsto (F', S', E')} .$$

Preslikava $(F, S, E) \mapsto^* (F', S', E')$ je tranzitivna ovojnica $\mapsto$:

$$\frac{(F, S, E) \mapsto (F', S', E')}{(F, S, E) \mapsto^* (F', S', E')} \qquad \frac{(F, S, E) \mapsto (F', S', E') \qquad (F', S', E') \mapsto^* (F'', S'', E'')}{(F, S, E) \mapsto^* (F'', S'', E'')}$$

Strojni program je okvir c. Pravimo, da tak program *izračuna vrednost* v, če velja $([c], [\,], [\,]) \mapsto^* ([\,], [v], [\,])$. Če pa obstaja neskončno zaporedje računskih korakov $([c], [\,], [\,]) \mapsto \cdots \mapsto \cdots$, pravimo, da strojni program c *divergira*.

3.6.2 Prevajalnik za MiniML

Prevajalnik iz programov za MiniML v strojne programe je preslikava C, ki izraz preslika v zaporedje strojnih ukazov. Definirana je z naslednjimi pravili, pri čemer z $\ell_1 @ \ell_2$ označimo

stikanje seznamov ℓ_1 in ℓ_2:

$$
\begin{aligned}
C(x) &= [\mathtt{Var}\, x] \\
C(n) &= [\mathtt{int}\, n] \\
C(\mathtt{true}) &= [\mathtt{bool}\;\mathtt{true}] \\
C(\mathtt{false}) &= [\mathtt{bool}\;\mathtt{false}] \\
C(e_1 \cdot e_2) &= C(e_1) \mathbin{@} C(e_2) \mathbin{@} [\mathtt{Mult}] \\
C(e_1 + e_2) &= C(e_1) \mathbin{@} C(e_2) \mathbin{@} [\mathtt{Add}] \\
C(e_1 - e_2) &= C(e_1) \mathbin{@} C(e_2) \mathbin{@} [\mathtt{Sub}] \\
C(e_1 = e_2) &= C(e_1) \mathbin{@} C(e_2) \mathbin{@} [\mathtt{Equal}] \\
C(e_1 < e_2) &= C(e_1) \mathbin{@} C(e_2) \mathbin{@} [\mathtt{Less}] \\
C(\mathtt{if}\; e_1 \;\mathtt{then}\; e_2 \;\mathtt{else}\; e_3) &= C(e_1) \mathbin{@} [\mathtt{Branch}(C(e_2), C(e_3))] \\
C(\mathtt{fun}\; f(x : \tau_1) : \tau_2 \;\mathtt{is}\; e) &= [\mathtt{Closure}(f, x, C(e) \mathbin{@} [\mathtt{PopEnv}])] \\
C(e_1\; e_2) &= C(e_1) \mathbin{@} C(e_2) \mathbin{@} [\mathtt{Call}]
\end{aligned}
$$

Na primer, program $(\mathtt{fun}\; f(x : \mathtt{int}) : \mathtt{int}\; \mathtt{is}\; 3 - x)\; 5$ se prevede v zaporedje ukazov

$$[\mathtt{Closure}(f, x, [\mathtt{int}\, 3, \mathtt{Var}\, x, \mathtt{Sub}, \mathtt{PopEnv}]), \mathtt{int}\, 5, \mathtt{Call}]\ .$$

Ostane se vprašanje, ali prevajalnik deluje pravilno. Kaj sploh to pomeni, nam pove naslednji izrek.

Izrek 3.7 *Prevajalnik C deluje* pravilno*: če je p program tipa τ, potem strojni program $C(p)$ divergira natanko tedaj, ko divergira p, in izračuna strojno vrednost v natanko tedaj, ko se p evaluira v vrednost v.*

Dokaz. Dokaz je dokaj zahteven in ga opustimo. ∎

3.7 Neučakani in leni jeziki

Programske jezike, predvsem funkcijske, v grobem delimo na neučakane in lene:

Neučakan jezik v aplikaciji $e_1\; e_2$ najprej evaluira oba izraza, e_1 in e_2, in šele nato izvede aplikacijo, ne glede na to, ali funkcija e_1 dejansko uporabi arugment e_2. Prav tako neučakan jezik vedno evaluira celotno podatkovno strukturo, denimo, v urejenem paru (e_1, e_2) evaluira obe komponentu, celo v primeru, da ene od njih sploh ne uporabimo.

Len jezik evaluira samo tiste izraze, ki jih *mora* evaluirati, da lahko napreduje v računskem postopku. Tako v aplikaciji $e_1\; e_2$ evaluira samo e_1, da dobi funkcijo, ki ji nato poda neevaluiran izraz e_2. Izraz e_2 se evaluira samo, če ga telo funkcije dejansko uporabi. Prav tako v urejenem paru (e_1, e_2) ne evaluira komponent, dokler jih ne potrebuje.

Čeprav se na prvi pogled zdi, da so leni jeziki "pametnejši", ker ne računajo ničesar po nepotrebnem, je v praksi večina programskih jezikov neučakanih. Razlogov za to je več, najpomembnejša pa sta dva: v prisotnosti stranskih učinkov je lažje razumeti, kaj se

bo zgodilo v neučakanem jeziku, poleg tega pa je lene jezike nekoliko težje učinkovito impementirati. Zato neučakani jeziki običajno v hitrosti prekašajo lene.

V tem razdelku se posvetimo preprostemu lenemu jeziku MiniHaskell, ki je narejen po vzoru lenega jezika Haskell. Da bo jezik bolj zabaven, ga opremimo s seznami in splošnimi rekurzivnimi definicijami.

3.7.1 Seznami

Tip seznamov, ki imajo elemente tipa τ, označimo s $\tau\,\mathtt{list}$:

$$\text{Tip } \tau ::= \cdots \mid \tau\,\mathtt{list}\;.$$

Sezname tvorimo na dva načina: naredimo prazen seznam $[\tau]$ tipa $\tau\,\mathtt{list}$,[11] ali staknemo skupaj element x (glavo) in seznam ℓ (rep) v novi seznam $x :: \ell$. Seznam razstavimo z izrazom `match`:

$$\mathtt{match}\;\ell\;\mathtt{with}\;[\tau] \to e_1 \mid x :: y \to e_2\;.$$

Če je ℓ prazen seznam, je vrednost izraza e_1, če pa je sestavljen iz glave x in repa y, je vrednost e_2. Spremenljivki x in y sta v izrazu e_2 vezani. Abstraktni sintaksi dodamo tri vrste izrazov:

$$\text{Izraz } e ::= \cdots \mid [\tau] \mid e_1 :: e_2 \mid (\mathtt{match}\;e_1\;\mathtt{with}\;[\tau] \to e_2 \mid x :: y \to e_3)\;.$$

V ocamlu lahko elemente seznama naštejemo z izrazom $[e_1; e_2; \ldots; e_n]$, ki ga moramo v naši sintaksi napisati nekoliko bolj okorno $e_1 :: e_2 :: \cdots :: e_n :: [\tau]$. Če bi želeli, bi lahko parserju parserju dodali pravilo, ki bi prvi zapis samodejno pretvorilo v drugega.

Morda se sprašujete, zakaj nismo za osnovni operaciji na seznamu proglasili "glava" in "rep", ki vrneta glavo in rep seznama. Težava je v tem, da sta glava in rep praznega seznama nedefinirani. Torej bi morali vpeljati v programski jezik še pojem napake `error`. S tem bi kršili pomembno načelo, da naj bodo posamezni koncepti v programskem jeziku med seboj neodvisni, saj bi bil pojem seznama odvisen od pojma napake.

Pravila za preverjanje tipov seznamov so:

$$\frac{}{\Gamma \mid [\tau] : \tau\,\mathtt{list}} \qquad \frac{\Gamma \mid e_1 : \tau \qquad \Gamma \mid e_2 : \tau\,\mathtt{list}}{\Gamma \mid e_1 :: e_2 : \tau\,\mathtt{list}}$$

$$\frac{\Gamma \mid e_1 : \tau\,\mathtt{list} \qquad \Gamma \mid e_2 : \sigma \qquad \Gamma, x : \tau, y : \tau\,\mathtt{list} \mid e_3 : \sigma}{\Gamma \mid (\mathtt{match}\;e_1\;\mathtt{with}\;[\tau] \to e_2 \mid x :: y \to e_3) : \sigma}$$

V 3.7.3 bomo podali še lena pravila za evaluacijo seznamov.

3.7.2 Splošne rekurzivne definicije

Rekurzivno definicija funkcije f lahko obravnavamo kot *enačbo* z neznanko f,

$$f = \Phi(f)\;. \tag{3.1}$$

[11] Namesto pričakovenega zapisa $[\,]$ za prazne sezname uporabljamo izraz $[\tau]$, ker je iz njega razviden tip. Lahko bi tudi pisali $[\,]_\tau$ ali kaj podobnega, vendar bi to bilo nepraktično v konkretni sintaksi. V programskih jezikih, ki dopuščajo parametrični polimorfizem, lahko pišemo $[\,]$, kar pomeni "prazen seznam poljubnega tipa".

Na primer, funkcija fakulteta $\mathtt{fak}(n) = 1 \cdot 2 \cdots n$ je rešitev enačbe $\mathtt{fak} = \Phi(\mathtt{fak})$, kjer je Φ funkcija definirana s predpisom

$$\Phi(f)(n) = \mathtt{if}\ n = 0\ \mathtt{then}\ 1\ \mathtt{else}\ n \cdot f(n-1)\ .$$

Enačba (3.1) tudi pomeni, da je rekurzivno definirana funkcija f *negibna točka* funkcije Φ. V lenih programskih jezikih lahko z rekurzivnimi definicijami podamo tudi neskončne sezname in ostale vrednosti, ki niso funkcije. Na primer, neskončni seznam $1 :: 2 :: 1 :: 2 :: \cdots$ je rešitev ℓ enačbe

$$\ell = 1 :: 2 :: \ell\ .$$

V programski jezik dodamo *splošne rekurzivne definicije*:

$$\text{Izraz } e ::= \cdots \mid \mathtt{rec}\ x : \tau\ \mathtt{is}\ e\ .$$

Izraz $\mathtt{rec}\ x : \tau\ \mathtt{is}\ e$ pomeni "vrednost tipa τ, ki je rešitev enačbe $x = e$", pri čemer je x vezana spremenljivka v izrazu e. Na primer, seznam ℓ iz zgornjega primera definiramo kot

$$\ell = \mathtt{rec}\ \mathtt{lst} : \mathtt{int}\,\mathtt{list}\ \mathtt{is}\ 1 :: 2 :: \mathtt{lst}\ .$$

Pravilo za preverjanje tipa rekurzivne definicije se glasi

$$\frac{\Gamma, x : \tau \mid e : \tau}{\Gamma \mid (\mathtt{rec}\ x : \tau\ \mathtt{is}\ e) : \tau}\ .$$

Pravilo za evaluacijo rekurzivne definicije se na prvi pogled zdi nenavadno:

$$\frac{}{\mathtt{rec}\ x : \tau\ \mathtt{is}\ e \mapsto e[x \to \mathtt{rec}\ x : \tau\ \mathtt{is}\ e]}$$

Če namesto e pišemo $e(x)$ in $x_0 = \mathtt{rec}\ x : \tau\ \mathtt{is}\ e(x)$, velja $x_0 = e(x_0)$. Zgornje pravilo za evaluacijo pove, da se x_0 evaluira v $e(x_0)$, kar sploh ni tako presenetljivo.

Ker smo uvedli splošne rekurzivne definicije, ne potrebujemo več posebnih rekurzivnih definicij funkcij. Zato poenostavimo izraze za zapise funkcij:

$$\text{Izraz } e ::= \cdots \mid \mathtt{fun}\ x : \tau \to e\ .$$

Izraz $\mathtt{fun}\ x : \tau \to e$ pomeni "funkcija, ki preslika argument x tipa τ v e", pri čemer je x vezana spremenljivka v e. Take funkcije niso rekurzivne, saj ne morejo klicati same sebe. Pravilo za preverjanje tipa se poenostavi, prav tako pravilo za evaluacijo aplikacije:

$$\frac{\Gamma, x : \tau_1 \mid e : \tau_2}{\Gamma \mid \mathtt{fun}\ x : \tau_1 \to e : \tau_1 \to \tau_2} \qquad \frac{}{(\mathtt{fun}\ x : \tau \to e)\ v_2 \mapsto e[x \to v_2]}$$

Kar smo v MiniML pisali kot $\mathtt{fun}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e$ zapišemo zdaj s kombinacijo rekurzivne definicije in funkcije,

$$\mathtt{rec}\ f : \tau_1 \to \tau_2\ \mathtt{is}\ \mathtt{fun}\ x : \tau_1 \to e\ .$$

3.7.3 Programski jezik MiniHaskell

Programski jezik MiniHaskell je funkcijski programski jezik z naslednjo abstraktno sintakso:

$$
\begin{aligned}
\text{Tip } \tau &::= \mathtt{int} \mid \mathtt{bool} \mid \tau\,\mathtt{list} \mid \tau_1 \to \tau_2 \\
\text{Izraz } e &::= x \mid n \mid \mathtt{true} \mid \mathtt{false} \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \\
&\quad e_1 = e_2 \mid e_1 < e_2 \mid \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \mid \\
&\quad [\tau] \mid e_1 :: e_2 \mid (\mathtt{match}\, e_1\, \mathtt{with}\, [\tau] \to e_2 \mid x :: y \to e_3) \\
&\quad \mathtt{fun}\, x : \tau \to e \mid e_1\, e_2 \mid \mathtt{rec}\, x : \tau\, \mathtt{is}\, e
\end{aligned}
$$

Pravila za preverjanje tipov so taka kot v MiniML, za sezname in rekurzivne definicije pa smo jih ravnokar podali. Spremembe glede na MiniML so predvsem v definiciji vrednosti in pravilih za evaluacijo. Ker MiniHaskell ničesar ne računa po nepotrebnem, se pri evaluaciji seznama ustavi takoj, ko ugotovi, ali je senzam prazen ali sestavljen. Tako za vrednosti proglasimo sestavljene sezname oblike $e_1 :: e_2$, kjer sta e_1 in e_2 poljubna programa:

$$\text{Vrednost } v ::= n \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{fun}\, x : \tau \to e \mid [\tau] \mid e_1 :: e_2\,.$$

Pravila za evaluacijo aritmetičnih izrazov, primerjave in pogojnega stavka so v MiniML in MiniHaskell enaka. Razlikujeta se v evaluaciji aplikacije, saj MiniHaskell ne evaluira argumenta:

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \qquad \frac{}{(\mathtt{fun}\, x : \tau \to e)\; e_2 \mapsto e[x \to e_2]}$$

Na ta način izraz e_2 evaluiramo samo takrat, ko je to zares potrebno. Vendar pa je naivna implementacija MiniHaskell tudi zelo neučinkovita, ker se hitro zgodi, da po nepotrebnem evaluiramo e_2 večkrat. Na primer, lahko se zgodi, da v naslednjem programu po nepotrebnem trikrat računamo vrednost $3 + 7$:

$$
\begin{aligned}
&(\mathtt{fun}\, x : \mathtt{int} \to x + x + x)(3 + 7) \mapsto (3 + 7) + (3 + 7) + (3 + 7) \mapsto \\
&\qquad 10 + (3 + 7) + (3 + 7) \mapsto 10 + 10 + (3 + 7) \mapsto 20 + (3 + 7) \mapsto 20 + 10 \mapsto 30\,.
\end{aligned}
$$

Zapišimo še lena pravila za evaluacijo seznamov:

$$\frac{e_1 \mapsto e_1'}{(\mathtt{match}\, e_1\, \mathtt{with}\, [\tau] \to e_2 \mid x :: y \to e_3) \mapsto (\mathtt{match}\, e_1'\, \mathtt{with}\, [\tau] \to e_2 \mid x :: y \to e_3)}$$

$$\frac{}{(\mathtt{match}\, [\tau]\, \mathtt{with}\, [\tau] \to e_2 \mid x :: y \to e_3) \mapsto e_2}$$

$$\frac{}{(\mathtt{match}\, e_1' :: e_1''\, \mathtt{with}\, [\tau] \to e_2 \mid x :: y \to e_3) \mapsto e_3[x \to e_1', y \to e_1'']}$$

Sodobni programski jeziki pogosto kombinirajo neučakano in leno evaluacijo. Jeziki, ki so v osnovi neučakani, so pogosto obogateni z lenimi podatkovnimi strukturami, in obratno, leni programski jeziki so obogateni z neučakanimi podatkovnimi strukutrami.

3.8 Izpeljava tipov in parametrični polimorfizem

Programska jezik MiniML in MiniHaskell smo v 3.3 opremili s preverjanjem tipov, da bi v 3.5 lahko zagotovili varnost tistih programov, ki imajo tip. Čeprav je to zaželena lastnost, nas tipi tudi omejujejo. Tak primer je funkcija

$$\mathtt{map}\, f\, (x_1 :: x_2 :: x_3 :: \cdots) = (f\, x_1) :: (f\, x_2) :: (f, x_3) :: \cdots\ .$$

Ko jo zapišemo v MiniHaskell, moramo določiti tip f in tip elementov x_i. Tako dobimo veliko različnih kopij `map`, ki pravzaprav počnejo "isto":

```
1  let map1 = rec map : (int -> int) -> int list -> int list is
2    fun f : (int -> int) ->
3      fun l : int list ->
4        match l with
5            [int] -> [int]
6          | x::xs -> (f x)::(map f xs)
7
8  let map2 = rec map : (bool -> bool) -> bool list -> bool list is
9    fun f : (bool -> bool) ->
10     fun l : bool list ->
11       match l with
12           [bool] -> [bool]
13         | x::xs -> (f x)::(map f xs)
```

Za programerja je neugodno, če mora pisati enako kodo večkrat samo zato, ker ga omejuje programski jezik. V tem razdelku si oglejmo, kako lahko programerju olajšamo delo, če jezik razširimo s *parametričnimi tipi*.

Zapišimo funkcijo `map` brez tipov in se vprašajmo, ali bi jih znali izračunati:

```
1  rec map is fun f -> fun l ->
2    match l with
3        [] -> []
4      | x::xs -> (f x)::(map f xs)
```

Nastavimo *sistem enačb*, ki jih razberemo iz zgornjega programa. V programu nastopajo spremenljivke `map`, `f`, `l`, `x` in `xs`, ki jim dodelimo neznane tipe

$$\mathtt{map} : X\ , \quad \mathtt{f} : Y\ , \quad \mathtt{l} : Z\ , \quad \mathtt{x} : U\ , \quad \mathtt{xs} : V\ .$$

Ker `l` uporabimo kot seznam v izrazu `match`, mora biti $Z = Z_1\ \mathtt{list}$ za neki tip Z_1. Iz izraza `match` sledi tudi, da je $U = Z_1$ in $V = Z$. Iz izraza `rec` se vidi, da je $X = Y \to Z \to X_1$ za neki tip X_1. Ker nastopa izraz `f x`, velja še $Y = Y_1 \to Y_2$ in $U = Y_1$. Ker nastopa `map f xs` kot drugi argument ::, sklepamo $X_1 = Y_2\ \mathtt{list}$. Tako smo dobili naslednje enačbe:

$$\begin{aligned} Z &= Z_1\ \mathtt{list} \\ U &= Z_1 \\ V &= Z \\ X &= Y \to Z \to X_1 \\ Y &= Y_1 \to Y_2 \\ U &= Y_1 \\ X_1 &= Y_2\ \mathtt{list}\ . \end{aligned}$$

V 3.8.2 bomo podali algoritem, ki reši tak sistem enačb. Zaenkrat ga rešimo "na roko". Uporabimo drugo enačbo in namesto Z_1 povsod pišemo U. Iz predzadnje enačbe nato sledi $U = Y_1$, zato namesto Y_1 povsod pišemo U, zaradi zadnje enačbe pa lahko X_1 nadomestimo z Y_2 `list`. Nazadnje vstavimo še $Y = U \to Y_2$ v enačbo za X in $Z = U\ \mathtt{list}$, da dobimo

$$
\begin{aligned}
X &= (U \to Y_2) \to U\ \mathtt{list} \to Y_2\ \mathtt{list} \\
Y &= U \to Y_2 \\
Z &= U\ \mathtt{list} \\
U &= U \\
V &= Z\ .
\end{aligned}
$$

Vidimo, da ima `map` tip

$$(U \to Y_2) \to U\ \mathtt{list} \to Y_2\ \mathtt{list}$$

pri čemer sta U in Y_2 poljubna tipa, ki jima pravimo *parametra*. Običajno parametre pišemo z grškimi črkami, zato bi tip `map` zapisali

$$\mathtt{map} : (\alpha \to \beta) \to \alpha\ \mathtt{list} \to \beta\ \mathtt{list}\ ,$$

v programskem jeziku pa nadomestili grške črke z navadnimi črkami, pred katere postavimo znak `'`:

```
map : ('a -> 'b) -> 'a list -> 'b list
```

Postopku, s katerim ugotovimo, kakšnega tipa je izraz, pravimo *izpeljava* ali *rekonstrukcija tipov*. Izpeljava tipov programerju olajša delo, saj mu ni treba podati tipov spremenljivk in funkcij.[12]

3.8.1 Poly

Definirajmo programski jezik, ki je nadgradnja MiniHaskell s *parametričnim polimorfizmom* in *izpeljavo tipov*. Tipom dodamo *parametre* in spotoma še produkte:

$$
\begin{aligned}
\text{Parameter } \alpha &::= \alpha \mid \beta \mid \gamma \mid \cdots \\
\text{Tip } \tau &::= \alpha \mid \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau\ \mathtt{list}
\end{aligned}
$$

Ker bo Poly sam izpeljal tipe, spremenimo sintakso za funkcije, rekurzivne definicije in sezname tako, da tipov ni več treba pisati:

$$
\begin{aligned}
\text{Izraz } e ::= {} & x \mid n \mid \mathtt{true} \mid \mathtt{false} \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \\
& e_1 = e_2 \mid e_1 < e_2 \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \mid \\
& [\,] \mid e_1 :: e_2 \mid (\mathtt{match}\ e_1\ \mathtt{with}\ [\,] \to e_2 \mid x :: y \to e_3) \\
& \mathtt{fun}\ x \to e \mid e_1\ e_2 \mid \mathtt{rec}\ x\ \mathtt{is}\ e
\end{aligned}
$$

Programe v Poly pišemo povsem enako kot v MiniHaskell, le da nikoli ne pišemo tipov, saj jih Poly izračuna sam:

[12] Velja tudi nasproten pogled: program, v katerem tipi niso zapisani, težje obvladamo, saj tipi *dokumentirajo* program. Čeprav je to res, je v zmogljivih programskih jeziki. Tak pogled imajo ponavadi tisti, ki uporabljajo jezike s preprostimi tipi ali objekti. V funkcijskem jeziku so lahko tipi komplicirani in jih programer težko sam izračuna.

```
Poly> fun x -> (x,x)
- : 'a -> 'a * 'a = <fun>
Poly> fun x -> x + 1
- : int -> int = <fun>
Poly> rec lst is []::lst
- : ('a list) list = [] :: [] :: [] :: ...
```

3.8.2 Izpeljava tipov in združevanje

V Poly tipov ne preverjamo, ampak jih *izpeljemo*. Vendar takoj naletimo na vprašanje, *kateri* tip je treba izpeljati, saj ima lahko program v Poly več kot en tip, na primer:

$$\begin{aligned}
\texttt{fun}\ x \to x &\quad : \quad \texttt{int} \to \texttt{int} \\
\texttt{fun}\ x \to x &\quad : \quad \alpha\ \texttt{list} \to \alpha\texttt{list} \\
\texttt{fun}\ x \to x &\quad : \quad \beta \to \beta \\
&\ldots
\end{aligned}$$

Za Poly velja, da ima program, ki ima vsaj en tip, vedno tudi *glavni tip*. To je tak tip, da so vsi ostali le njegov poseben primer. V zgornjem primeru je glavni tip $\beta \to \beta$, saj lahko ostale dobimo tako, da za β vstavljamo vrednosti $\beta = \texttt{int}$, $\beta = \alpha\ \texttt{list}$, ... Postopek za izpeljavo tipa bo izračunal glavni tip.

Izpeljava glavnega tipa programa e poteka v dveh korakih:

1. Izračunamo tip τ, v katerem lahko nastopajo parametri, in sistem enačb $\mathcal{E}$.

2. Sistem $\mathcal{E}$ rešimo in rešitve vstavimo v τ, da dobimo tip programa e.

Prvi korak vedno uspe, drugi pa ne nujno, saj se lahko zgodi, da sistem $\mathcal{E}$ nima rešitve.

Pravila za izpeljavo tipa in sistema enačb zapišemo z sodbo

$$x_1 : \tau_1, \ldots, x_n\, \tau_n \mid e : \tau\ \ /_{\{\alpha_1,\ldots,\alpha_m\}}\ \ \mathcal{E}\ ,$$

ki pomeni "izraz e ima v kontekstu $x_1 : \tau_1, \ldots, x_n : \tau_n$ tip τ, pri čemer je treba upoštevati rešitev sistema enačb $\mathcal{E}$ v neznankah $\alpha_1, \ldots, \alpha_m$. Če sistem $\mathcal{E}$ nima rešitve, izraz nima tipa." Kontekst krajše označimo z Γ in množico neznank z A. Pravila sklepanja za konstante in spremenljivke so preprosta:

$$\frac{\Gamma(x) = \tau}{\Gamma \mid x : \tau\ \ /_{\emptyset}\ \ \emptyset} \qquad\qquad \frac{}{\Gamma \mid n : \texttt{int}\ \ /_{\emptyset}\ \ \emptyset}$$

$$\frac{}{\Gamma \mid \texttt{true} : \texttt{bool}\ \ /_{\emptyset}\ \ \emptyset} \qquad\qquad \frac{}{\Gamma \mid \texttt{false} : \texttt{bool}\ \ /_{\emptyset}\ \ \emptyset}$$

Pravila za aritmetične operacije in primerjave so bolj zapletena:

$$\frac{\Gamma \mid e_1 : \tau_1\ \ /_{A_1}\ \ \mathcal{E}_1 \qquad \Gamma \mid e_2 : \tau_2\ \ /_{A_2}\ \ \mathcal{E}_2 \qquad A_1 \cap A_2 = \emptyset}{\Gamma \mid e_1 + e_2 : \texttt{int}\ \ /_{A_1 \cup A_2}\ \ \mathcal{E}_1 \cup \mathcal{E}_2 \cup \{\tau_1 = \texttt{int}, \tau_2 = \texttt{int}\}}\ \text{(podobno za } e_1 \cdot e_2,\ e_1 - e_2\text{)}$$

$$\frac{\Gamma \mid e_1 : \tau_1\ \ /_{A_1}\ \ \mathcal{E}_1 \qquad \Gamma \mid e_2 : \tau_2\ \ /_{A_2}\ \ \mathcal{E}_2 \qquad A_1 \cap A_2 = \emptyset}{\Gamma \mid e_1 = e_2 : \texttt{bool}\ \ /_{A_1 \cup A_2}\ \ \mathcal{E}_1 \cup \mathcal{E}_2 \cup \{\tau_1 = \texttt{int}, \tau_2 = \texttt{int}\}}\ \text{(podobno za } e_1 < e_2\text{)}$$

Zadnji dve pravili sta dokaj zapleteni, pravila, ki sledijo, pa še bolj. Ali ni tak zapis pravil preveč kompliciran? Da boste lahko presodili sami, jih tokrat zapišimo neformalno, za vajo pa jih sami prepišite v formalno obliko:

Spremenjivka x: tip spremenljivke poiščemo v kontekstu Γ, ne dobimo nobenih enačb.

Celo število n: tip je `int`, ne dobimo nobenih enačb.

Boolovi konstanti `true` in `false`: tip je `bool`, ne dobimo nobenih enačb.

Aritmetična operacija $e_1 * e_2$, $* \in \{+, -, \cdot\}$: Izračunamo tip e_1 in dobimo τ_1 s sitemom enačb $\mathcal{E}_1$. Izračunamo tip e_2 in dobimo τ_2 s sistemom enačb $\mathcal{E}_2$. Pri tem poskrbimo, da sistema $\mathcal{E}_1$ in $\mathcal{E}_2$ nimata skupnih spremenljivk (vedno, ko izberemo novo spremenljivko, pazimo, da se ne pojavi kje drugje). Tip $e_1 * e_2$ je `int` z enačbami $\mathcal{E}_1$, $\mathcal{E}_2$ in $\tau_1 = \texttt{int}$, $\tau_2 = \texttt{int}$.

Primerjava $e_1 = e_2$ ali $e_1 < e_2$: Izračunamo tip e_1 in dobimo τ_1 s sitemom enačb $\mathcal{E}_1$. Izračunamo tip e_2 in dobimo τ_2 s sistemom enačb $\mathcal{E}_2$. Pri tem poskrbimo, da sistema $\mathcal{E}_1$ in $\mathcal{E}_2$ nimata skupnih spremenljivk. Tip $e_1 = e_2$ (ali $e_1 < e_2$) je `bool` z enačbami $\mathcal{E}_1$, $\mathcal{E}_2$ in $\tau_1 = \texttt{int}$, $\tau_2 = \texttt{int}$.

Pogojni stavek `if` e_1 `then` e_2 `else` e_3: Izračunamo tipe e_1, e_2, e_3 ter dobimo tipe τ_1, τ_2, τ_3 s pripadajočimi sistemi enačb $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$. Pri tem poskrbimo, da sistemi enačb nimajo skupnih spremenljivk. Tip pogojnega stavka je τ_2 z enačbami $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$, $\tau_1 = \texttt{bool}$, $\tau_2 = \tau_3$.

Prazen seznam $[\,]$: Izberemo nov parameter (spremenljivko) α. Tip $[\,]$ je α `list` brez enačb.

Sestavljen seznam $e_1 :: e_2$: Izračunamo tipa e_1 in e_2 ter dobimo tipa τ_1 in τ_2 s sistemoma enačb $\mathcal{E}_1$ in $\mathcal{E}_2$. Pri tem poskrbimo, da $\mathcal{E}_1$ in $\mathcal{E}_2$ nimata skupnih spremenljivk. Tip $e_1 :: e_2$ je τ_2 z enačbami $\mathcal{E}_1$, $\mathcal{E}_2$, $\tau_1\ \texttt{list} = \tau_2$.

Izraz `match` e_1 `with` $[\,] \to e_2 \mid x :: y \to e_3$: Izračunamo tipe e_1, e_2, e_3 ter dobimo tipe τ_1, τ_2, τ_3 s pripadajočimi sistemi enačb $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$. Ko računamo tip e_3 v kontekst dodamo $x : \alpha$ in $y : \alpha\ \texttt{list}$, pri čemer je α parameter, ki se ne pojavlja nikjer drugje. Poskrbimo tudi, da sistemi $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$ nimajo skupnih spremenljivk. Tip izraza je τ_2 z enačbami $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$, $\tau_1 = \alpha\ \texttt{list}$, $\tau_2 = \tau_3$.

Funkcija `fun` $x \to e$ Izberemo nov parameter α, ki se ne pojavlja nikjer drugje. Izračunamo tip τ in enačbe $\mathcal{E}$ za izraz e, pri čemer v kontekst dodamo $x : \alpha$. Tip funkcije je $\alpha \to \tau$ z enačbami $\mathcal{E}$.

Aplikacija $e_1\ e_2$: Izračunamo tipa e_1 in e_2 ter dobimo tipa τ_1 in τ_2 s sistemoma enačb $\mathcal{E}_1$ in $\mathcal{E}_2$. Pri tem poskrbimo, da $\mathcal{E}_1$ in $\mathcal{E}_2$ nimata skupnih spremenljivk. Izberemo nov parameter α, ki se ne pojavlja nikjer drugje. Tip izraza $e_1\ e_2$ je α z enačbami $\mathcal{E}_1$, $\mathcal{E}_2$, $\tau_1 = \tau_2 \to \alpha$.

Rekurzija `rec` x `is` e: Izberemo nov parameter α, ki se ne pojavlja nikjer drugje. Izračunamo tip τ in enačbe $\mathcal{E}$ za izraz e, pri čemer v kontekst dodamo $x : \alpha$. Tip izraza je τ z enačbami $\mathcal{E}$, $\tau = \alpha$.

Algoritem za reševanje sistema enačb se imenuje *združevanje*. Vhodni podatek je množica enačb

$$\mathcal{E} = \{L_1 = D_1, \dots, L_n = D_n\} \; .$$

Izhodni podatek je bodisi "ni rešitve" bodisi rešitev sistema

$$\alpha_1 = e_1, \dots, \alpha_k = e_k$$

pri čemer so $\alpha_1, \dots, \alpha_k$ neznanke, ki se pojavljajo v sistemu enačb (če v rešitvi kakšna neznanka ni omenjena, to pomeni, da ima poljubno vrednost). Postopek za reševanje je naslednji:

1. Nastavi rešitev na prazen seznam $r = [\,]$.
2. Ponavljaj, doker je $\mathcal{E} \neq \emptyset$: iz $\mathcal{E}$ odstrani poljubno enačbo $L_i = D_i$ in:
   (a) če sta L_i in D_i enaka, nadaljuj, sicer
   (b) če je $L_i = \alpha_j$ in se α_j ne pojavi v D_i, povsod v $\mathcal{E}$ in r zamenjaj α_j z D_i, nato v r dodaj $\alpha_j = D_i$, sicer
   (c) če je $D_i = \alpha_j$ in se α_j ne pojavi v L_i, povsod v $\mathcal{E}$ in r zamenjaj α_j z L_i, nato v r dodaj $\alpha_j = L_i$, sicer
   (d) če je $D_i = D_{i,1} \to D_{i,2}$ in $L_i = L_{i,1} \to L_{i,2}$, dodaj v $\mathcal{E}$ enačbi $D_{i,1} = L_{i,1}$ in $D_{i,2} = L_{i,2}$, sicer
   (e) če je $D_i = D_{i,1} \times D_{i,2}$ in $L_i = L_{i,1} \times L_{i,2}$, dodaj v $\mathcal{E}$ enačbi $D_{i,1} = L_{i,1}$ in $D_{i,2} = L_{i,2}$, sicer
   (f) če je $D_i = D'_i\ \texttt{list}$ in $L_i = L'_i\ \texttt{list}$, dodaj v $\mathcal{E}$ enačbo $D'_i = L'_i$, sicer
   (g) če ne velja nobena od zgornjih možnosti, vrni "ni rešitve".
3. Vrni r.

Izpeljavo tipov ponazorimo s primerom. Izpeljimo tip izraza

$$\texttt{fun}\ f \to (f(\texttt{fun}\ x \to x + 3)) \; . \tag{3.2}$$

Izraz je funkcija, zato izberemo nov parameter α in izračunamo tip aplikacije $f(\texttt{fun}\ x \to (x + 3))$ v kontekstu $f : \alpha$. Tip f je potem α, tip funkcije $\texttt{fun}\ x \to x + 3$ pa je enak $\beta \to \texttt{int}$ z enačbo $\beta = \texttt{int}$. Tip aplikacije je γ z enačbama $\beta = \texttt{int}$, $\alpha = (\beta \to \texttt{int}) \to \gamma$. Izraz (3.2) ima torej tip

$$\alpha \to \gamma \; ,$$

z enačbama

$$\beta = \texttt{int} \; ,$$
$$\alpha = (\beta \to \texttt{int}) \to \gamma \; .$$

Sistem rešimo z združevanjem. Odstranimo prvo enačbo, povsod nadomestimo β z $\texttt{int}$ in v rešitev dodamo $\beta = \texttt{int}$. Preostane enačba $\alpha = (\texttt{int} \to \texttt{int}) \to \gamma$, ki jo dodamo v rešitev, ki se torej glasi

$$\beta = \texttt{int} \; ,$$
$$\alpha = (\texttt{int} \to \texttt{int}) \to \gamma \; ,$$

pri čemer je γ poljuben parameter. Ko vstavimo rešitev v (3.2) dobimo tip

$$((\texttt{int} \to \texttt{int}) \to \gamma) \to \gamma \; .$$

3.8.3 Operacijska semantika Poly

Pravila za evaluacijo programov v MiniML in MiniHaskell niso v ničemer odvisna od tipov. Navsezadnje lahko evaluiramo tudi program, ki sploh nima tipa. Zato so pravila za evaluacijo Poly *enaka* pravilom za evaluacijo MiniHaskell. Treba je le ustrezno spremeniti sintakso in pobrisati oznake tipov iz funkcij, rekurzivnih definicij in praznega seznama.

3.9 Naloge

Naloga 3.8 Z uporabo pravil za preverjanje tipov ugotovi, kakšen tip ima MiniML program

```
(fun f(n:int):int is if n < 1 then 1 else n * f(n-1)) 5
```

nato pa s pravili za evaluacijo izračunaj njegovo vrednost.

Naloga 3.9 Razširi MiniML s tipom `unit` in vrednostjo `()` tipa `unit`. Zgleduj se po Ocamlovem tipu `unit`. Popraviti je treba definicijo tipov, izrazov, preverjanja tipov in pravil za evaluacijo.

Naloga 3.10 Če v MiniML pobrišemo eno izmed pravil za evaluacijo, ali še vedno veljata izreka o napredku 3.3 in ohranitvi 3.4?

Naloga 3.11 Če v MiniML dodamo pravili za evaluacijo $\texttt{false} + n \mapsto n$ in $\texttt{true} + n \mapsto n + 1$, ali še vedno veljata izreka o napredku 3.3 in ohranitvi 3.4?

Naloga 3.12 MiniML dodaj operaciji konjunkcija $e_1 \,\texttt{and}\, e_2$ in disjunkcija $e_1 \,\texttt{or}\, e_2$. Popravi abstraktno sintakso izrazov, preverjanje tipov in pravila za evaluacijo.

Naloga 3.13 Pravimo, da ima zaporedje $x_0, x_1, x_2, \ldots$ *periodičen rep*, če obstajata takšna $n, m \in \mathbb{N}$, da za vse $i \in \mathbb{N}$ velja $x_{n+i} = x_{n+i+m}$. Povedano z drugimi besedami, tako zaporedje se od nekod naprej ponavlja. V MiniHaskell definiraj neskončen seznam Boolovih vrednosti, ki *nima* periodičnega repa.

Naloga 3.14 Zapiši izreka o ohranitvi in napredku za MiniHaskell. Ali veljata?

Naloga 3.15 MiniHaskell dodamo nov osnovni tip `magic` in ne dodamo nobenih novih izrazov, prav tako ne spreminjamo pravil za preverjanje tipov in evaluacijo. V tako razširjenem MiniHaskell zapiši program, ki ima tip `magic`.

Naloga 3.16 Navedi primer programa v Poly, ki

1. nima tipa,
2. ima natanko en tip,
3. ima natanko dva tipa,
4. ima neskončno tipov.

Naloga 3.17 Zapiši pravila za evaluacijo Poly.

Poglavje 4

Ukazni programski jezik

V tem poglavju obravnavamo preprost *ukazni* jezik. V ukaznih jezikih s programom upravljamo *stanje* pomnilnika, ki sestoji iz *spremenljivk*, ki jim lahko spreminjamo vrednosti. V drugem delu spoznamo pravila sklepanja, s katerimi dokazujemo pravilnost programov, na zadnje pa na kratko omenimo še, kako bi ukaznemu jeziku dodali procedure.

4.1 Ukazni programski jezik

Obravnavamo preprost ukazni programski jezik, opremljen s celoštevilskimi aritmetičnimi izrazi, Boolovimi izrazi, (lokalnimi) celoštevilskimi spremenljivkami in kontrolnimi ukazi (zaporedje ukazov, pogojni stavek in zanka `while`).

4.1.1 Sintaksa

Sintaksa ukaznega programskega jezika nekoliko spominja na pascal:

$$
\begin{aligned}
\text{Aritmetični izraz } e &::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \cdot e_2 \mid e_1/e_2 \\
\text{Boolov izraz } b &::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{not}\ b \mid b_1\ \mathtt{and}\ b_2 \mid b_1\ \mathtt{or}\ b_2 \mid \\
&\quad\ e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2 \\
\text{Ukaz } c &::= \mathtt{skip} \mid c_1 \,;\, c_2 \mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \\
&\quad\ \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{done} \mid \mathtt{newvar}\ x := e\ \mathtt{in}\ c \mid x := e \mid \mathtt{print}\ e
\end{aligned}
$$

Ta jezik je tako preprost, da ni potrebno preverjati pravilnosti tipov, saj so edini tipi, ki nastopajo `int`, `bool` in `comm` (tip ukaza). Če parser sprejme izraz, ima ta pravilen tip. Ko bomo dodali procedure, pa bo treba dodati tudi tipe.

Preden podamo formalno operacijsko semantiko jezika, pojasnimo intuitivni pomen ukazov:

- `skip` je ukaz, ki ne naredi ničesar,
- $c_1 \,;\, c_2$ izvede ukaz c_1 in nato še ukaz c_2,
- `if` b `then` c_1 `else` c_2 je pogojni stavek,
- `while` b `do` c `done` je običajna zanka `while`,

- `newvar` $x := e$ `in` c je definicija lokalne spremenljivke x z začetno vrednostjo e, spremenljivka pa je veljavna v ukazu c,
- $x := e$ nastavi vrednost spremenljivke x na vrednost izraza e,
- `print` e izpiše vrednost izraza e.

4.1.2 Operacijska semantika

Operacijska semantika, oziroma pravila za izvajanje programov, sestoji iz treh delov: evaluacija aritmetičnega izraza, evaluacija Boolovega izraza in evaluacija ukaza.

Evaluacijo aritmetični izrazov s spremenljivkami smo obravnavali v 2.3, zato je tu ne bomo ponavljali. Osvežimo si spomin: če je $\eta = [(x_1, v_1); \ldots; (x_n; v_n)]$ *okolje*, v katerem imajo spremenljivke x_i vrednosti v_i, zapis $\eta \mid e \hookrightarrow v$ pomeni, da se v okolju η aritmetični izraz e evaluira v vrednost v.

Tudi za evaluacijo Boolovih izrazov uporabimo enak zapis: $\eta \mid b \hookrightarrow v$ pomeni, da se Boolov izraz b v okolju η evaluira v Boolovo vrednost v. Pravila za evaluacijo Boolovih izrazov niso zapletena in jih puščamo za vajo, glej nalogo 4.1.

Preostanejo še pravila za izvajanje ukazov. Ker ima ukaz `print` učinek, ki ga drugi ukazi nimajo, ga bomo obravnavali nazadnje. Ukaz se od aritmetičnega izraza razlikuje po tem, da nima vrednosti, ampak *učinek* na trenutno okolje. Zato lahko na ukaz c gledamo kot na funkcijo, ki slika okolja v nova okolja. Na primer, ukaz $x := x + 1$ je funkcija, ki dano okolje $[\ldots; (x, v); \ldots]$ preslika v okolje $[\ldots; (x, v + 1); \ldots]$. Nekateri ukazi se izvedejo v enem koraku in se končajo, drugi pa v enem koraku nekaj napravijo z okoljem, nato pa nadaljujejo z izvajanjem. Zato podamo pravila v dveh oblikah:

$$(\eta, c) \mapsto \eta' , \qquad\qquad (\eta, c) \mapsto (\eta', c') .$$

Prvo pravilo pove, da izraz c v enem koraku konča in okolje η preslika v okolje η'. Drugo pravilo pove, da ukaz c v enem koraku preslika okolje η v okolje η', izvajanje pa se nadaljuje z ukazom c'.

Preden zapišemo pravila za izvajanje ukazov, potrebujemo še operacije na okoljih, s katerimi bomo izrazili učinke ukazov. Če je $\eta = [(x_1, v_1); \ldots; (x_n, v_n)]$, pišemo

- $\eta(x)$ za vrednost, ki jo ima spremenljivka x v okolju η. Če se x v η pojavi večkrat, obvelja prva vrednost,
- $(x, v) :: \eta$ je okolje η razširjeno s spremenljivko x, ki ima vrednost v,
- $\eta[x \to v]$ je okolje η, v katerem smo vrednost spremenljivke x *zamenjali* z v. Če se x v η pojavi večkrat, zamenjamo vrednost prve pojavitve.
- $\eta \setminus x$ dobimo, ko iz okolja η odstranimo prvo pojavitev spremenljivke x, skupaj z njeno vrednostjo.

Pravila za izvajanje ukazov se glasijo:

$$\frac{}{(\eta, \texttt{skip}) \mapsto \eta} \qquad \frac{(\eta, c_1) \mapsto \eta'}{(\eta, c_1 \,;\, c_2) \mapsto (\eta', c_2)} \qquad \frac{(\eta, c_1) \mapsto (\eta', c_1')}{(\eta, c_1 \,;\, c_2) \mapsto (\eta', c_1' \,;\, c_2)}$$

$$\frac{\eta \mid b \hookrightarrow \texttt{true}}{(\eta, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \mapsto (\eta, c_1)} \qquad \frac{\eta \mid b \hookrightarrow \texttt{false}}{(\eta, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \mapsto (\eta, c_2)}$$

$$\frac{\eta \mid b \hookrightarrow \texttt{false}}{(\eta, \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}) \mapsto \eta} \qquad \frac{\eta \mid b \hookrightarrow \texttt{true}}{(\eta, \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}) \mapsto (\eta, c\,;\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done})}$$

$$\frac{\eta \mid e \hookrightarrow v}{(\eta, x := e) \mapsto \eta[x \to v]}$$

Preostane nam še pravilo za lokalne spremenljivke, za katerega bi najprej pomislili, da se glasi

$$\frac{\eta \mid e \hookrightarrow v}{(\eta, \texttt{newvar}\ x := e\ \texttt{in}\ c) \mapsto ((x, v) :: \eta, c)} \quad ?$$

Lokalno spremenljivko x z ustrezno začetno vrednostjo torej dodamo okolju η in v novem okolju izvedemo c. Vendar to ne bo prav, saj mora biti x *lokalna* spremnljivka, ki "izgine", ko c konča izvajanje, zgornje pravilo pa spremenljivke nikoli ne odstrani iz okolja. Da bomo lahko to storili, potrebujemo nov ukaz `delete`:

$$\text{Ukaz } c ::= \cdots \mid \texttt{delete}\ x\ .$$

To je *interni* ukaz, do katerega programer nima dostopa. Pravili za lokalne spremenljivke in `delete` sta

$$\frac{\eta \mid e \hookrightarrow v}{(\eta, \texttt{newvar}\ x := e\ \texttt{in}\ c) \mapsto ((x, v) :: \eta, c\,;\texttt{delete}\ x)} \qquad \frac{}{(\eta, \texttt{delete}\ x) \mapsto \eta \setminus x}$$

Nazadnje povejmo še, kako zapišemo pravilo za izvajanje ukaza `print`, ki povzroči učinek posebne vrste, saj izpiše število na zaslon. V ta namen operacijski semantiki dodamo računske korake

$$(\eta, c) \overset{!n}{\mapsto} \eta'\ , \qquad (\eta, c) \overset{!n}{\mapsto} (\eta', c')\ .$$

Oznaka $\overset{!n}{\mapsto}$ pomeni, da ukaz c pri izvajanju na zaslon izpiše število n. Pravilo za izvajanje ukaza `print` lahko sedaj zapišemo takole:

$$\frac{\eta \mid e \hookrightarrow n}{(\eta, \texttt{print}\ e) \overset{!n}{\mapsto} \eta}$$

4.2 Specifikacije in pravilnost programov

Trditev je izjava P zapisana v logiki prvega reda. *Speficikacija* je predpis, kaj naj bi program počel. Program, ki *zadošča* ali *ustreza* specifikaciji, je *pravilen* glede na to specifikacijo. V tem razdelku obravnavamo pravilnost in specifikacije za ukazni jezik brez ukaza `print`.

V ukaznem jeziku poznamo dve vrsti specifikacij:

Delna pravilnost je predpis $\{P\}\ c\ \{Q\}$, ki pomeni: "Če velja P in izvedemo ukaz c, bo c divergiral ali pa končal in veljalo bo Q."

Popolna pravilnost je predpis $[P]\ c\ [Q]$, ki pomeni: "Če velja P in izvedemo ukaz c, bo c končal in veljalo bo Q."

Razlika med delno in popolna pravilnostjo je torej v tem, ali zahtevamo, da mora ukaz c končati.

Nekatera pravila sklepanja o pravilnosti programov zahtevajo pozorno obravnavo spremenljivk. V ta namen definiramo funkciji FV in FA. Za dani izraz, ukaz, ali izjavo e je $\mathrm{FV}(e)$ množica prostih spremenljivk, ki se pojavijo v e. Za ukaz c je $\mathrm{FV}(c)$ definiran s predpisi

$$\begin{aligned}
\mathrm{FV}(\texttt{skip}) &= \emptyset \\
\mathrm{FV}(c_1\,;c_2) &= \mathrm{FV}(c_1) \cup \mathrm{FV}(c_2) \\
\mathrm{FV}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) &= \mathrm{FV}(e) \cup \mathrm{FV}(c_1) \cup \mathrm{FV}(c_2) \\
\mathrm{FV}(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}) &= \mathrm{FV}(b) \cup \mathrm{FV}(e) \\
\mathrm{FV}(\texttt{newvar}\ x := e\ \texttt{in}\ c) &= \mathrm{FV}(e) \cup (\mathrm{FV}(c) \setminus \{x\}) \\
\mathrm{FV}(x := e) &= \{x\} \cup \mathrm{FV}(e)\ .
\end{aligned}$$

Funkcija $\mathrm{FA}(c)$ vrne množico tistih spremenljivk, ki bi jim ukaz c lahko spremenil vrednost:

$$\begin{aligned}
\mathrm{FA}(\texttt{skip}) &= \emptyset \\
\mathrm{FA}(c_1\,;c_2) &= \mathrm{FA}(c_1) \cup \mathrm{FA}(c_2) \\
\mathrm{FA}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) &= \mathrm{FA}(c_1) \cup \mathrm{FA}(c_2) \\
\mathrm{FA}(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}) &= \mathrm{FA}(c) \\
\mathrm{FA}(\texttt{newvar}\ x := e\ \texttt{in}\ c) &= \mathrm{FA}(c) \setminus \{x\} \\
\mathrm{FA}(x := e) &= \{x\}\ .
\end{aligned}$$

Večina pravil sklepanja ima dve inačici, eno za delno in eno za popolno pravilnost, ki se razlikujeta samo v oklepajih $\{\cdots\}$ in $[\cdots]$. Zato bomo našteli le pravila za delno pravilnost. Pripadajoča pravila za popolno pravilnost dobimo tako, da zavite oklepaje nadomestimo z oglatimi. Izjema so pravila CSP, SCT, WHP in WHT, ki *nimajo* dveh inačic.

Naštejmo najprej splošna pravila sklepanja (v oglatih oklepajih je zapisana okrajšava, s katero se sklicujemo na pravilo):

$$\frac{P \Rightarrow Q \qquad \{Q\}\ c\ \{R\}}{\{P\}\ c\ \{R\}}\ [\mathrm{SP}] \qquad \frac{\{P\}\ c\ \{Q\} \qquad Q \Rightarrow R}{\{P\}\ c\ \{R\}}\ [\mathrm{WC}]$$

$$\frac{\{P_1\}\ c\ \{Q_1\} \qquad \{P_2\}\ c\ \{Q_2\}}{\{P_1 \wedge P_2\}\ c\ \{Q_1 \wedge Q_2\}}\ [\mathrm{CA}] \qquad \frac{\{P_1\}\ c\ \{Q_1\} \qquad \{P_2\}\ c\ \{Q_2\}}{\{P_1 \vee P_2\}\ c\ \{Q_1 \vee Q_2\}}\ [\mathrm{DA}]$$

$$\frac{\mathrm{FV}(P) \cap \mathrm{FA}(c) = \emptyset}{\{P\}\ c\ \{P\}}\ [\mathrm{CSP}] \qquad \frac{[Q]\ c\ [R] \qquad \mathrm{FV}(P) \cap \mathrm{FA}(c) = \emptyset}{[Q \wedge P]\ c\ [R \wedge P]}\ [\mathrm{CST}]$$

Zadnji pravili povesta, da ukaz c ohranja pravilnost trditve P, če ne spremeni vrednosti spremenljivk, ki prosto nastopajo v P.

Pravila za `skip`, zaporedno izvajanje in pogojni stavek so preprosta:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ [\mathrm{SK}] \qquad \frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\ ;\ c_2\ \{R\}}\ [\mathrm{SQ}]$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}\ [\mathrm{CD}]$$

V pravilih za delno in popolno pravilnost zanke `while` nastopa trditev I, ki ji pravimo *invarianta* za zanko. Pravili namreč povesta, da se I ohrani, če enkrat izvedemo telo zanke. V pravilu za popolno pravilnost nastopa še nenegativna količina e, ki se na vsakem koraku zmanjša, od koder smemo sklepati, da se zanka ustavi:

$$\frac{\{I \wedge b\}\ c\ \{I\}}{\{I\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ \{I \wedge \neg b\}}\ [\mathrm{WHP}]$$

$$\frac{[I \wedge b \wedge e = v]\ c\ [I \wedge e < v] \qquad I \wedge b \Rightarrow e \geq 0}{[I]\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ [I \wedge \neg b]}\ (v \notin \mathrm{FV}(I, b, e, c))$$

Preostaneta še pravili za lokalne spremenljivke in prirejanje vrednosti:

$$\frac{}{\{P[x \to e]\}\ x := e\ \{P\}}\ [\mathrm{AS}] \qquad \frac{\{P\}\ s\ ;\ x := e\ ;\ c\ \{Q\} \qquad x \notin \mathrm{FV}(Q)}{\{P\}\ s\ ;\ \texttt{newvar}\ x := e\ \texttt{in}\ c\ \{Q\}}\ [\mathrm{DC}]$$

V pravilu *DC* lahko ukaz s izpustimo.

Izpeljano pravilo je tako pravilo, ki že sledi iz ostalih pravil sklepanja in ga načeloma ne potrebujemo. Vseeno pa so nekatera izpeljana pravila dokaj uporabna. Na primer, pravilo

$$\frac{x \notin \mathrm{FV}(e)}{\{\texttt{true}\}\ x := e\ \{x = e\}} \qquad \text{(EQ)}$$

izpeljemo takole. Denimo, da se x ne pojavi v izrazu e. V AS vzamemo $P \equiv x = e$ in dobimo $\{e = e\}\ x := e\ \{x = e\}$. Če zamenjamo $e = e$ z ekvivalentno izjavo `true`, dobimo (EQ).

4.3 Prevajalnik za ukazni jezik

(Še ni napisano.)

4.4 Naloge

Naloga 4.1 Napiši pravila za evaluacijo Boolovih izrazov v ukaznem jeziku. Pravila zapiši v obliki $\eta \mid b \hookrightarrow v$, ki pomeni, da se v okolju η Boolov izraz b evaluira v vrednost v. Seveda se smeš pri evaluaciji izrazov kot je $7 + 3 < x + 15$ sklicevati na evaluacijo aritmetičnih izrazov.

Naloga 4.2 V spodnjih specifikacijah nadomesti vprašaje z izjavami, ki naredijo specifikacije veljavne. Tvoje izjave morajo biti čim šibkejše, kar pomeni, da zahtevajo samo to, kar je potrebno, da bo specifikacija izpolnjena:[1]

$$[?]\ x := x + 1\ ;\ y := y - 1\ [y = x^2]$$
$$\{?\}\ \texttt{while true do skip done}\ \{\texttt{false}\}$$
$$[?]\ \texttt{while true do skip done}\ [\texttt{false}]$$
$$[?]\ \texttt{while not}(i = 10)\ \texttt{do}\ i := i + 2\ \texttt{done}\ [i = 10]$$

Naloga 4.3 Dokaži, da velja:

$$[x \geq 0 \land x = x_0 \land y = y_0]$$
$$\texttt{while not}(x = 0)\ \texttt{do}\ x := x - 1\ ;\ y := y + x\ \texttt{done}$$
$$[y = y_0 + x_0 \cdot (x_0 - 1)/2]$$

Naloga 4.4 Dokaži, da velja:

$$[n > 0 \land s = 0]$$
$$\texttt{newvar}\ i := 1\ \texttt{in while}\ i < 2 \cdot n\ \texttt{do}\ s := s + i\ ;\ i := i + 2\ \texttt{done}$$
$$[s = n^2]$$

Naloga 4.5 Iz pravil sklepanja za pravilnost programov izpelji naslednje pravilo:

$$\frac{\{P\}\ c_1\ \{R\} \qquad \{Q\}\ c_2\ \{R\}}{\{(b \implies P) \land (\neg b \implies Q)\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{R\}}$$

Naloga 4.6 Sestavi ukaz u, ki ne vsebuje spremenljivk x_0, y_0 in z_0 ter zadošča specifikaciji

$$[x = x0 \land y = y0 \land z = z0]\ u\ [x \leq y \leq z \land \{x, y, z\} = \{x_0, y_0, z_0\}]$$

Dokaži, da tvoj u res zadošča tej specifikaciji.

[1] Izjava P je *šibkejša* od izjave Q, če velja $Q \implies P$. Najšibkejša izjava je `true` in najmočnejša `false`.

Poglavje 5

Objektni programski jezik

5.1 Zapisi

V 1.3.9 smo že spoznali podatkovni tip zapis, ki je podoben podatkovnemu tipu urejenih n-teric, le da so posamezna polja poimenovana. Tako na primer tip $\{\mathtt{re} : \mathtt{float}, \mathtt{im} : \mathtt{float}\}$ predstavlja kompleksna števila. Kompleksno število $2.5 + 3.8\,i$ zapišemo $\{\mathtt{re} = 2.5, \mathtt{im} = 3.8\}$. V splošnem ima zapis in pripadajoča vrednost poljubno število polj:[1]

$$\{\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_n = e_n\} \quad \text{in} \quad \{\ell_1 : \tau_1, \ell_2 : \tau_2, \ldots, \ell_n : e_n\}\ .$$

Krajše to zapišemo z $\{\ell_i = e_i\}_{i=1}^n$ in $\{\ell_i : \tau_i\}_{i=1}^n$. Oznake ℓ_i so *imena polj*. Običajno so veljavna imena polj sestavljena iz črk in številk, tako kot spremenljivke. V zapisu ne smemo uporabiti istega imena večkrat. Če je $n = 0$, dobimo *prazen zapis* $\{\}$.

Poglejmo, kako zapise dodamo programskemu jeziku, kakršen je MinML. Najprej dodamo podatkovni tip zapis:

$$\text{Tip } \tau ::= \cdots \ \mid\ \{\ell_i : \tau_i\}_{i=1}^n$$

Tu še nismo postavili zahteve, da morajo biti imena polj v zapisu med seboj različna. To bomo naredili pri preverjanju tipov.

Abstraktni sintaksi izrazov dodamo operacijo za tvorjenje zapisov in za *projekcijo* polja iz zapisa:

$$\text{Izraz } e ::= \cdots \ \mid\ \{\ell_i = e_i\}_{i=1}^n \ \mid\ e.\ell$$

Izraz $e.\ell$ pomeni "polje ℓ v zapisu e", na primer $\{\mathtt{re} = 2.5, \mathtt{im} = 3.8\}.\mathtt{re}$ je 2.5.

Pravilom za preverjanje tipov dodamo pravili za zapise:

$$\frac{\text{za } i, j = 1, \ldots, n\text{: če } i \neq j \text{ potem } \ell_i \neq \ell_j \qquad \text{za } i = 1, \ldots, n\text{: } \Gamma \mid e_i : \tau_i}{\Gamma \mid \{\ell_i = e_i\}_{i=1}^n : \{\ell_i : \tau_i\}_{i=1}^n}$$

$$\frac{\Gamma \mid e : \{\ell_i : \tau_i\}_{i=1}^n \qquad 1 \leq k \leq n}{\Gamma \mid e.\ell_k : \tau_k}$$

Prvo pravilo pove, da ima zapis veljaven tip, če so imena polj paroma različna in če imajo polja ustrezne tipe. Drugo pravilo pove, da ima projekcija $e.\ell_k$ tip τ_k, če je e zapis, ki vsebuje polje imenovano ℓ_k tipa τ_k.

[1] Pozor, v ocamlu so poja ločena s podpičji namesto z vejicami.

Preden povemo, kako zapise evaluiramo, moramo razmisliti, ali so zapisi vrednosti, se pravi, kdaj je zapis končni rezultat nekega računa. Zapis $\{x = 2 + 3, y = 7 - 2\}$ zagotovo *ni* vrednost, saj lahko izračunamo vrednosti njegovih polj, $\{x = 5, y = 5\}$. Vidimo, da je zapis vrednost, če so njegova polja vrednosti:

$$\text{Vrednost } v ::= \cdots \mid \{\ell_i = v_i\}_{i=1}^n$$

Zapis evaluiramo tako, da po vrsti evaluiramo polja od leve proti desni:

$$\frac{e_k \mapsto e'_k}{\{\ell_1 = v_1, \ldots, \ell_{k-1} = v_k, \ell_k = e_k, \ldots, \ell_n = e_k\} \mapsto \{\ell_1 = v_1, \ldots, \ell_{k-1} = v_k, \ell_k = e'_k, \ldots, \ell_n = e_k\}}$$

Pravilo pove, da izvedemo naslednji korak v k-tem polju, če tak korak obstaja in če so vsa polja pred k-tim že vrednosti. Tako pravilo bi verjetno pričakoval programer, s stališča prevajalnika pa bi bilo bolje, če vrstnega reda evaluacije ne bi predpisali, saj bi s tem prevajalniku pustili več možnosti za optimizacijo. Evaluacija projekcije je preprosta, saj zapis evaluiramo do vrednosti, nato pa izluščimo ustrezno polje:

$$\frac{e \mapsto e'}{e.\ell \mapsto e'.\ell} \qquad \frac{v = \{\ell_i = v_i\}_{i=1}^n \qquad 1 \leq k \leq n}{v.\ell_k \hookrightarrow v_k}$$

5.2 Podtipi

V programskem jeziku se lahko zgodi, da smemo vrednosti tipa τ zamenjati z vrednostmi tipa σ, pri čemer moramo morebiti opraviti še ustrezno pretvorbo. Na primer, v javi lahko vedno uporabimo število tipa `int` namesto števila tipa `float`, saj bo java sama pretvorila celo število v število v plavajoči vejici. Drug primer v javi izhaja iz hierarhije razredov. Če je A podrazred razreda B, lahko objekte tipa A uporabljamo, kot da bi bili objekti tipa B. Pri tem ni potrebna nobena pretvorba. Koncept, ki ga tu spoznavamo je *podtip*.

Kadar lahko vrednosti tipa σ uporabljamo, kot da bi bile tipa τ (z morebitno pretvorbo iz enega tipa v drugega), pravimo, da je σ *podtip* τ in zapišemo

$$\sigma \leq \tau\ .$$

Relacija "podtip" $\leq$ zadošča naslednjim splošnim pravilom:

$$\frac{}{\tau \leq \tau} \qquad \frac{\rho \leq \sigma \qquad \sigma \leq \tau}{\rho \leq \tau} \qquad \frac{\Gamma \mid e : \sigma \qquad \sigma \leq \tau}{\Gamma \mid e : \tau}$$

Prvo pravilo pravi, da je $\leq$ refleksivna relacija, drugo pa, da je tranzitivna. Taki relaciji pravimo *šibka urejenost*.[2] Tretje pravilo se imenuje *posplošitev* (angl. subsumption) in pravi, da lahko tip izraza vedno posplošimo na nadtip.

Zanimivo je vprašanje, kdaj je funkcijski tip $\sigma_1 \to \sigma_2$ podtip $\tau_1 \to \tau_2$. Denimo, da imamo funkcijo $f : \sigma_1 \to \sigma_2$. Kdaj jo lahko uporabimo, kot da bi imela tip $\tau_1 \to \tau_2$? Zagotovo mora funkcija vračati vrednosti, ki jih lahko uporabljamo, kot da bi imele tip τ_2.

[2] Če bi imeli še *antisimetričnost*, $\sigma \leq \tau \land \tau \leq \sigma \implies \sigma = \tau$, bi bil $\leq$ *delna urejenost*.

Od tod sledi, da mora veljati $\sigma_2 \leq \tau_2$. Če naj f sprejema argumente tipa τ_1, se morajo ti obnašati kot vrednosti tipa σ_1. Zato mora veljati $\tau_1 \leq \sigma_1$ in splošno pravilo se glasi:

$$\frac{\tau_1 \leq \sigma_1 \qquad \sigma_2 \leq \tau_2}{\sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2}$$

Pozor, v zgornjem pravilu sta domeni in kodomeni obravnavani različno, saj imamo $\tau_1 \leq \sigma_1$ in $\sigma_2 \leq \tau_2$.

Podtipi so uprabni tudi pri zapisih. Poznamo tri vrste podtipov zapisov, od katerih ima vsak svoje prednosti in slabosti. Najprej imamo *podtip zapisa po širini* (angl. record width subtyping). Ideja je, da imamo tip zapisa za podtip, če vsebuje kvečjemu še kakšna dodatna polja, saj lahko taka polja vedno ignoriramo:

$$\frac{}{\{\ell_i : \tau_i\}_{i=1}^{n+k} \leq \{\ell_i : \tau_i\}_{i=1}^{n}}$$

Objektni programski jeziki ponavadi uporabljajo podtipe zapisov po širini, saj ima podrazred vsa polja, ki jim ima nadrazred, in morebiti še kakšna dodatna.

Druga vrsta so *podtipi zapisov po globini.* Tu imamo en zapis za podtip drugega, če imata oba enako število polj z enakimi imeni, tipi v prvem pa so podtipi tipov v drugem zapisu:

$$\frac{\text{za } i = 1, \dots, n\text{:} \quad \sigma_i \leq \tau_i}{\{\ell_i : \sigma_i\}_{i=1}^{n} \leq \{\ell_i : \tau_i\}_{i=1}^{n}}$$

Nazadnje poznamo še *podtip zapisa s permutacijo*, ki pravi, da je prvi zapis podtip drugega, če se razlikujeta samo v vrstnem redu polj:

$$\frac{\pi \text{ permutacija na } \{1, \dots, n\}}{\{\ell_i : \tau_i\}_{i=1}^{n} \leq \{\ell_{\pi(i)} : \tau_{\pi(i)}\}_{i=1}^{n}}$$

Mnogi programski jeziki, na primer ocaml, namesto podtipov zapisov s permutacijo vzamejo dva zapisa za *enaka*, če se razlikujeta samo v vrstnem redu polj. Vse tri vrste podtipov zapisov lahko združimo v eno samo "super" pravilo, ki ga bomo spoznali v 5.2.1.

Zgornja pravila za podtipe so z matematičnega stališča elegantna, vendar so za preverjanje tipov v prevajalniku neuporabna, ker iz strukture izraza ni razvidno, katero pravilo naj uporabimo. Na primer, pravili o tranzitivnosti in posplošitvi načeloma vedno prideta v poštev, kar je povsem nekoristno, če želimo sestaviti algoritem. V naslednjem razdelku bomo imeplementirali funkcijski programski jezik z zapisi in podtipi, v katerem bomo zgornja pravila zapisali v drugačni, a ekvivalentni obliki, ki je uporabna pri preverjanju tipov.

5.2.1 Funkcijski programski jezik s podtipi

Abstraktna sintaksa

Implementirajmo zapise in podtipe v preprostem funkcijskem programskem jeziku Sub. Poleg osnovnih tipov `int` in `bool`, imamo še funkcije in zapise:

$$\text{Tip } \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_i : \tau_i\}_{i=1}^{n}$$

Abstraktna sintaksa izrazov se glasi:

$$\begin{aligned}\text{Izraz } e ::= {} & x \mid n \mid \texttt{true} \mid \texttt{false} \mid \\ & e_1 + e_2 \mid e_1 - e_2 \mid e_1 \cdot e_2 \mid e_1/e_2 \mid \\ & e_1 = e_2 \mid e_1 < e_2 \mid e_1 \,\texttt{and}\, e_2 \mid e_1 \,\texttt{or}\, e_2 \mid \texttt{not}\, e_1 \\ & \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \mid \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 \\ & \texttt{fun}\, f(x : \tau_1) : \tau_2 \,\texttt{is}\, e \mid e_1\, e_2 \mid \{\ell_i = e_i\}_{i=1}^n \mid e.\ell\end{aligned}$$

Sub smo opremili še z *lokalnimi definicijami* $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, ki nam bodo olajšale pisanje programov.

Statična semantika

Kot smo napovedali, moramo pravila za preverjanje tipov zapisati v obliki, ki je primerna za razvoj algoritma, ki izračuna tip danega izraza. Prva težava, ki nastopi je ta, da ima lahko dani izraz več tipov. Kako naj torej zapišemo pravila, da bodo enolično določala tip izraza? Iz zagate se rešimo tako, da sploh ne poskušamo določiti vseh tipov, ki jih ima dani izraz, ampak samo enega. Pri celih številih in Boolovih vrednostih nimamo izbire, saj nimajo nobenega podtipa in nadtipa. Pri zapisih izračunamo *najmanjši* tip, ki ga dani zapis ima, pri funkcijah pa *najmanjši tip z dano domeno*. Pravila zapišemo tako, da so čim bolj podobna običajnim pravilom brez podtipov, le da na ustreznih mestih dodamo preverjanje podtipov. V pravilih, ki so zbrana na sliki 5.1 na strani 71, smo uporabili okrajšavo

$$\Gamma \mid e : \sigma \leq \tau,$$

ki pomeni

$$\Gamma \mid e : \sigma \quad \text{in} \quad \sigma \leq \tau\ .$$

Preostanejo še pravila za preverjanje podtipov. Tudi tu ne uporabimo splošnih pravil iz 5.2, ampak ekvivalentna pravila, ki so primerna za algoritem:

$$\frac{\tau = \sigma}{\tau \leq \sigma} \qquad \frac{\tau_1 \leq \sigma_1 \qquad \sigma_2 \leq \tau_2}{\sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2}$$

$$\frac{\forall i \in \{1, \dots, n\}.\, \exists j \in \{1, \dots, m\}.\, (\ell'_i = \ell_j \wedge \sigma_j \leq \tau_i)}{\{\ell_i : \sigma_i\}_{i=1}^m \leq \{\ell'{}_i : \tau_i\}_{i=1}^n} \tag{5.1}$$

Pravilo za podtipe zapisov je "super" pravilo, ki smo ga že omenili v 5.2. To pravilo pravi, da je $\rho = \{\ell_i : \sigma_i\}_{i=1}^m$ podtip $\xi = \{\ell'{}_i : \tau_i\}_{i=1}^n$, če vsako polje ℓ'_i, ki nastopa v ξ, nastopa tudi v ρ kot neki $\ell_j = \ell'_i$, in je pripadajoči tip σ_j podtip τ_i.

Dinamična semantika

V definiciji MinML smo zapisali operacijsko semantiko z relacijo $p \mapsto p'$, glej 3.4. Taki semantiki pravimo tudi "semantika majhnih korakov", saj predpisuje posamezne korake v izvajanju programa. Pravila za evaluacijo aritmetičnih izrazov, glej 2.2, pa so bila podana z relacijo $e \hookrightarrow v$, ki je predpisovala, da je končni rezultat evaluacije izraza e vrednost v. Tej vrsti semantike pravimo "semantika velikih korakov", ker nam neposredno pove le, kako dobimo končni rezultat, ne pa tudi vseh vmesnih korakov v izračunu.

$$\frac{\Gamma(x) = \tau}{\Gamma \mid x : \tau} \qquad \frac{}{\Gamma \mid n : \mathtt{int}} \qquad \frac{}{\Gamma \mid \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \mid \mathtt{false} : \mathtt{bool}}$$

$$\frac{\Gamma \mid e_1 : \tau \leq \mathtt{int} \qquad \Gamma \mid e_2 : \sigma \leq \mathtt{int}}{\Gamma \mid e_1 + e_2 : \mathtt{int}} \quad \text{(podobno za } -, \cdot \text{ in } /\text{)}$$

$$\frac{\Gamma \mid e_1 : \tau \leq \mathtt{bool} \qquad \Gamma \mid e_2 : \sigma \leq \mathtt{bool}}{\Gamma \mid e_1 \ \mathtt{and}\ e_2 : \mathtt{bool}} \qquad \frac{\Gamma \mid e_1 : \tau \leq \mathtt{bool} \qquad \Gamma \mid e_2 : \sigma \leq \mathtt{bool}}{\Gamma \mid e_1 \ \mathtt{or}\ e_2 : \mathtt{bool}}$$

$$\frac{\Gamma \mid e : \tau \leq \mathtt{bool}}{\Gamma \mid \mathtt{not}\ e : \mathtt{bool}} \qquad \frac{\Gamma \mid e_1 : \tau \leq \mathtt{bool} \qquad \Gamma \mid e_2 : \tau \qquad \Gamma \mid e_3 : \tau}{\Gamma \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \mid e_1 : \tau \leq \mathtt{int} \qquad \Gamma \mid e_2 : \sigma \leq \mathtt{int}}{\Gamma \mid e_1 = e_2 : \mathtt{bool}} \qquad \frac{\Gamma \mid e_1 : \tau \leq \mathtt{int} \qquad \Gamma \mid e_2 : \sigma \leq \mathtt{int}}{\Gamma \mid e_1 < e_2 : \mathtt{bool}}$$

$$\frac{f : \tau_1 \to \tau_2, x : \tau_1, \Gamma \mid e : \tau_2}{\Gamma \mid \mathtt{fun}\ f(x : \tau_1) : \tau_2\ \mathtt{is}\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \mid e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid e_2 : \sigma \leq \tau_1}{\Gamma \mid e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \mid e_1 : \sigma \qquad x : \sigma, \Gamma \mid e_2 : \tau}{\Gamma \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau} \qquad \frac{\Gamma \mid e : \{\ell_i : \tau_i\}_{i=1}^n \qquad 1 \leq k \leq n}{\Gamma \mid e.\ell_k : \tau_k}$$

$$\frac{\text{za } i, j = 1, \ldots, n\text{: če } i \neq j \text{ potem } \ell_i \neq \ell_j \qquad \text{za } i = 1, \ldots, n\text{: } \Gamma \mid e_i : \tau_i}{\Gamma \mid \{\ell_i = e_i\}_{i=1}^n : \{\ell_i : \tau_i\}_{i=1}^n}$$

Slika 5.1: Preverjanje tipov v Sub.

Ker sta Sub in MinML precej podobna, bi lahko pravila za evaluacijo v dobršnji meri prepisali od MinML. Da bo snov bolj zanimiva, jih raje podajmo v semantiki velikih korakov z okolji, s čemer bo algoritem, ki sledi tem pravilom, bistveno bolj učinkovit.

Da bomo lahko podali pravilo za evaluacijo funkcij, moramo jeziku dodati *zaprtja*, ki smo jih spoznali v 3.6.1 in v domači nalogi MinScheme:

$$\text{Izraz } e ::= \cdots \mid \texttt{closure}(\eta, x, e) \;.$$

Programer nima neposrednega dostopa do zaprtij. Zaprtja so vrednosti, kakor tudi števila, Boolove konstante in zapisi vrednosti,

$$\text{Vrednost } v ::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{closure}(\eta, x, e) \mid \{\ell_i = v_i\}_{i=1}^{n} \;.$$

Operacijsko semantiko Sub izrazimo z relacijo $\eta \mid e \hookrightarrow v$, ki pomeni, da se v okolju η izraz e evaluira v končno vrednost v. Kot vedno je okolje η seznam $[(x_1, v_1); \ldots, (x_n, v_n)]$ ki spremenljivkam priredi njihove vrednosti. Pravila so zbrana na sliki 5.2. Iz njih je razvidno, da je Sub neučakan programski jezik.

$$\frac{\eta(x) = v}{\eta \mid x \hookrightarrow v} \qquad \frac{}{\eta \mid n \hookrightarrow n} \qquad \frac{}{\eta \mid \texttt{true} \hookrightarrow \texttt{true}} \qquad \frac{}{\eta \mid \texttt{false} \hookrightarrow \texttt{false}}$$

$$\frac{\eta \mid e_1 \hookrightarrow n_1 \qquad \eta \mid e_2 \hookrightarrow n_2 \qquad n = n_1 + n_2}{\eta \mid e_1 + e_2 \hookrightarrow n} \quad \text{(podobno za } -, \cdot, /\text{)}$$

$$\frac{\eta \mid e_1 \hookrightarrow b_1 \qquad \eta \mid e_2 \hookrightarrow b_2}{\eta \mid e_1 \;\texttt{and}\; e_2 \hookrightarrow b_1 \wedge b_2} \quad \text{(podobno za } \texttt{or}\text{)} \qquad \frac{\eta \mid e \hookrightarrow b}{\eta \mid \texttt{not}\; e \hookrightarrow \neg b}$$

$$\frac{\eta \mid e_1 \hookrightarrow \texttt{true} \qquad \eta \mid e_2 \hookrightarrow v}{\eta \mid \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 \hookrightarrow v} \qquad \frac{\eta \mid e_1 \hookrightarrow \texttt{false} \qquad \eta \mid e_3 \hookrightarrow v}{\eta \mid \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 \hookrightarrow v}$$

$$\frac{c = \texttt{closure}(((f, c), \eta), x, e)}{\eta \mid (\texttt{fun}\; f(x : \tau_1) : \tau_2 \;\texttt{is}\; e) \hookrightarrow c} \qquad \frac{}{\eta \mid \texttt{closure}(\eta, x, e) \hookrightarrow \texttt{closure}(\eta, x, e)}$$

$$\frac{\eta \mid e_1 \hookrightarrow v_1 \qquad (x, v_1) :: \eta \mid e_2 \hookrightarrow v_2}{\eta \mid \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 \hookrightarrow v_2}$$

$$\frac{\eta \mid e_1 \hookrightarrow \texttt{closure}(\eta', x, e) \qquad \eta \mid e_2 \hookrightarrow v_2 \qquad (x, v_2) :: \eta' \mid e \hookrightarrow v}{\eta \mid e_1\; e_2 \hookrightarrow v}$$

$$\frac{\text{za } i = 1, \ldots, n\text{:} \quad \eta \mid e_i \hookrightarrow v_i}{\eta \mid \{\ell_i = e_i\}_{i=1}^{n} \hookrightarrow \{\ell_i = v_i\}_{i=1}^{n}} \qquad \frac{\eta \mid e \hookrightarrow \{\ell_i = v_i\}_{i=1}^{n} \qquad 1 \leq k \leq n}{\eta \mid e.\ell_k \hookrightarrow v_k}$$

Slika 5.2: Evaluacija v Sub.

5.3 Objekti kot zapisi

Že prej smo omenili, da lahko na zapise gledamo kot na objekte. Če želimo iz zapisov narediti prave objekte, jih moramo še nekoliko razširiti. V tem razdelku si bomo ogledali jezik Ob, ki je objektna razširitiev jezika Sub. Ker bomo obravnavali objektni jezik, bomo zapisom rekli *objekti*, čeprav so še vedno le zapisi. Poljem v objektu, ki so funkcije, bomo rekli tudi *metode*.

Naštejmo nekaj osnovnih značilnosti objektov:

1. Objekti tvorijo hierarhijo.
2. Objekt hrani *spremenljivo stanje*, ki se spreminja, ko kličemo objektne metode.
3. Objektne metode se lahko z rezervirano besedo `this` sklicujejo na objekt, na katerem so klicane.
4. Objekt lahko vsebuje privatne komponente, ki niso neposredno dosegljive izven objekta.

Hierarhijo objektov bomo predstavii s podtipi. V popularnih objektnih jezikih, kot sta Java in C++, je hierarhija predstavljena z razredi in relacijo "podrazred". V razredni hierarhiji programer pri definiciji razreda eksplicitno določi njegov nadrazred. Naš jezik, ki namesto razredne hierarhije uporablja podtipe, deluje drugače, saj programer nikjer ne določi, kateri tipi so nadtipi danega tipa – to je določeno z definicijo relacije podtip $\leq$. V tem pogledu je Ob bolj podoben programskemu jeziku javascript.

Zapisi v jeziku Sub ne hranijo spremenljivega stanja, saj ne moremo spreminjati vrednosti polj. Ker želimo objekte, ki hranijo stanje, dodamo ukaze za spreminjanje vrednosti polj. S tem se tudi spremeni dinamična semantika jezika, kar bomo videli kasneje.

Tako kot v javi in C++ se bodo lahko tudi v Ob objekti sklicevali sami nase z rezervirano besedico `this`.

Privatnost nekaterih zapisov v objektu pa dosežemo z *askripcijo* ali pretvorbo tipov. To je ukaz, ki tip izraza spremeni v njegov nadtip in s tem povzroči, da so nekateri deli izraza privatni.

Iz povedanega sledi, da je Ob razširitev Sub, zato ne bomo ponavljali celotne definicije jezika, ampak se bomo osredotočili samo na spremembe.

Ker se lahko na objekt `this` sklicujejo samo metode ne pa tudi polja v objektu, strogo ločimo polja od metod. Objekt je torej zapis, ki sestoji iz polj $\ell_i = e_i$, označenih z besedico `val` in metod $m_j : \tau'_j = f_j$, označenih z besedico `method`:

$$\{\mathtt{val}\ \ell_1 = e_1 \ldots\ \mathtt{val}\ \ell_k = e_k\ \mathtt{method}\ m_1 : \tau'_1 = f_1 \ldots\ \mathtt{method}\ m_n : \tau'_n = f_n\}$$

To krajše zapišemo $\{\mathtt{val}\ \ell_i = e_i|_{i=1}^k\ \mathtt{method}\ m_j : \tau'_j = f_j|_{j=1}^n\}$. Zapis metode $m_j : \tau'_j = f_j$ pomeni, da metoda m_j vrne vrednost tipa τ'_j, pri čemer je f_j definicija metode. Pri poljih tipa ni treba navajati, ker ga Ob izračuna sam. Tipov metod ne more izračunati, ker se lahko sklicujejo same nase.[3] Tip objekta zapišemo

$$\{\mathtt{val}\ \ell_1 : \tau_1 \ldots\ \mathtt{val}\ \ell_k : \tau_k\ \mathtt{method}\ m_1 : \tau'_1 \ldots\ \mathtt{method}\ m_n : \tau'_n\}$$

[3] Pravzaprav bi lahko Ob računal tudi tipe metod, a bi morali zaradi tega bistveno raširiti tipe, zapletel pa bi se tudi postopek preverjanja tipov.

ali krajše $\{\texttt{val}\ \ell_i : \tau_i|_{i=1}^k\ \texttt{method}\ m_j : \tau'_j|_{j=1}^n\}$. Tipi v Ob so torej:

$$\text{Tip } \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \{\texttt{val}\ \ell_i : \tau_i|_{i=1}^k\ \texttt{method}\ m_j : \tau'_j|_{j=1}^n\}$$

Priložena implementacija Ob dopušča še *definicije tipov*. V programu lahko definiramo nov tip, na primer,

```
type counter = {method reset:{}, get:int}
```

in se nato sklicujemo na tip `counter`, kar omogoči pisanje preglednejših programov. Definicij tipov v statični semantiki ne bomo obravnavali, kogar pa to zanima, lahko reši nalogo 5.8.

Abstraktna sintaksa izrazov v Ob je podobna tisti v Sub, od nje pa se razlikuje v zadnjih dveh vrsticah naslednje definicije:

$$\begin{aligned}\text{Izraz } e ::= {} & x \mid n \mid \texttt{true} \mid \texttt{false} \mid \\ & e_1 + e_2 \mid e_1 - e_2 \mid e_1 \cdot e_2 \mid e_1/e_2 \mid \\ & e_1 = e_2 \mid e_1 < e_2 \mid e_1\ \texttt{and}\ e_2 \mid e_1\ \texttt{or}\ e_2 \mid \texttt{not}\ e_1 \mid \\ & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\ & \texttt{fun}\ x : \tau \to e \mid e_1\ e_2 \mid \\ & \{\texttt{val}\ \ell_i = e_i|_{i=1}^k\ \texttt{method}\ m_j : \tau'_j = f_j|_{j=1}^n\} \mid e.\ell \mid e\#m \mid \\ & \texttt{this} \mid e_1.\ell := e_2 \mid e_1 \,;\, e_2 \mid e\ \texttt{as}\ \tau\end{aligned}$$

Ob se od Sub razlikuje v tem, da ima namesto zapisov objekte, da dostopamo do komponente ℓ z izrazom $e.\ell$ in do metode m z izrazom $e\#m$. Poleg tega pozna Ob še izraz `this`, nastavljanje vrednost komponente $e_1.\ell := e_2$, izvajanje zaporednih ukazov $e_1 \,;\, e_2$ in askripicjo e `as` τ.

Izraz `this` se nanaša na objekt, v katerem se pojavi. V naslednjem primeru metoda `get` dostopa do polja `n` preko `this`:

```
Ob> let a = {val n = 3 method get : int = this.n}
a : {val n : int method get : int} = {val n = 3 method get}
Ob> a#get
- : int = 3
```

Seveda bi lahko do polja n dostopali kar neposredno:

```
Ob> a.n
- : int = 3
```

Kako preprečimo takšne dostope, bomo videli malo kasneje. Ukaz $e_1.\ell := e_2$ nastavi polje ℓ v objektu e_1 na vrednost e_2:

```
Ob> let b = {val n = 2}
b : {val n : int} = {val n = 2}
Ob> b.n := 2 + 3
- : {} = {}
Ob> b
- : {val n : int} = {val n = 5}
```

Več zaporednih ukazov ločimo s podpičjem.

Če ima izraz e tip τ in je $\tau \leq \sigma$, z izrazom e `as` σ spremenimo tip e v σ. To je uporabno predvsem v primeru, ko je e objekt in želimo nekatera njegova polja skriti. Denimo, da bi želeli implementirati števec, to je objekt tipa

```
type counter = {reset : {}, get : int}
```

Ko pokličemo metodo `get` dobimo vrednost, ki je za eno večja kot prejšnjič. Metoda `reset` števec postavi na 0. V Ob lahko to implementiramo takole:

```
let c = {
  val n = 0
  method reset : {} = this.n := 0
  method get : int  = this.n := this.n+1 ; this.n
}
```

Polje `n` je v objektu `c` dostopno, saj ima `c` tip

```
{val n : int method reset : {} method get : int}
```

Zares nam Ob dovoli, da `c.n` spreminjamo po mili volji:

```
Ob> c.n := 7
- : {} = {}
```

Če želimo to preprečiti, z določilom $\cdots$ `as counter` spremenimo tip `c`. Ker `counter` ne omenja polja `n`, to ni več dostopno:

```
let c = {
  val n = 0
  method reset : {} = this.n := 0
  method get : int  = this.n := this.n+1 ; this.n
} as counter
```

Sedaj se `c` res obnaša po pričakovanjih:[4]

```
Ob> let c = { ... } as counter
c : counter = {val n = 0 method reset method get}
Ob> c#get
- : int = 1
Ob> c#get
- : int = 2
Ob> c.n
type error: polje n ne obstaja
```

5.3.1 Statična semantika

Pravila za preverjanje tipov v Ob so podobna kot tista v Sub. Namesto pravil za preverjanje tipov zapisov ima Ob naslednja pravila za preverjanje tipov objektov.

Zaradi lažjega zapisa definirajmo okrašjavi

$$o = \{\texttt{val}\ \ell_i = e_i|_{i=1}^{k}\ \texttt{method}\ m_j : \tau'_j = f_j|_{j=1}^{n}\}$$
$$\tau_o = \{\texttt{val}\ \ell_i : \tau_i|_{i=1}^{k}\ \texttt{method}\ m_j : \tau'_j|_{j=1}^{n}\}$$

Pravilo za tip objekta o se v Ob glasi

$$\frac{\begin{array}{l}\text{vsi } \ell_i,\ m_j \text{ so paroma različni}\\ \text{za } i = 1,\ldots,k\text{: } \Gamma \mid e_i : \tau_i\\ \text{za } j = 1,\ldots,n\text{: } \texttt{this} : \tau_o, \Gamma \mid f_j : \tau'_j\end{array}}{\Gamma \mid o : \tau_o}\ .$$

[4]Ob izpiše vsa polja, tudi tista, ki smo jih "skrili".

Poleg tega imamo še nova pravila za projekcijo polja, klic metode, prirejanje, podpičje in askripcijo:

$$\frac{\Gamma \mid e : \tau_o \qquad 1 \leq p \leq k}{\Gamma \mid e.\ell_p : \tau_p} \qquad \frac{\Gamma \mid e : \tau_o \qquad 1 \leq p \leq n}{\Gamma \mid e\#\mu_p : \tau'_p}$$

$$\frac{\Gamma \mid e_1 : \tau_o \qquad 1 \leq p \leq n \qquad \Gamma \mid e_2 : \tau_p}{\Gamma \mid e_1.\ell_p := e_2 : \{\}} \qquad \frac{\Gamma \mid e_1 : \sigma \qquad \Gamma \mid e_2 : \tau}{\Gamma \mid e_1 \,;\, e_2 : \tau}$$

$$\frac{\Gamma; \theta \mid e : \tau \qquad \tau \leq \sigma}{\Gamma; \theta \mid e \;\texttt{as}\; \sigma : \sigma}$$

5.3.2 Dinamična semantika

Dinamična semantika Ob je nekoliko bolj zapletena kot dinamična semantika Sub, ker imajo objekti spremenljiva polja. Najprej potrebujemo pojma *lokacija*[5] in *stanje*, ki sta analogna pojmoma pomnilniški naslov in pomnilnik. Lokacije so elementi števno neskončne množice $L = \{l_0, l_1, l_2, \ldots\}$. Zares ni pomembno, kaj pravzaprav so elementi L, najlažje pa si je predstavljati, da so možni naslovi pomnilnika. Stanje Σ je končna preslikava iz lokacij v vrednosti. Natančneje:

$$\begin{aligned}
\text{Lokacija } \lambda &::= l_0 \mid l_1 \mid l_2 \mid \cdots \\
\text{Vrednost } v &::= n \mid \texttt{true} \mid \texttt{false} \mid \\
&\qquad \texttt{closure}(\eta, x, e) \mid \{\texttt{val}\ \ell_i = \lambda_i|_{i=1}^{k}\ \texttt{method}\ m_j = (\eta_j, f_j)|_{j=1}^{n}\} \\
\text{Stanje } \Sigma &::= \lambda_1 \mapsto v_1, \ldots, \lambda_n \mapsto v_n \\
\text{Okolje } \eta &::= x_1 : v_1, \ldots, x_n : v_n
\end{aligned}$$

Objekt se torej evaluira v zapis, ki sestoji iz lokacij λ_i (te povedo, kje v pomnilniku so shranjena polja objekta) in vrednosti (η_j, f_j) (te predstavljajo definicije metod, pri čemer si zapomnimo tudi okolje η_j, v katerem je bila metoda evaluirana). Osnovna sodba v semantiki Ob je

$$\Sigma; \eta \mid e \hookrightarrow v \mid \Sigma'$$

in pomeni: "Če v stanju Σ in okolju η evaluiramo e, dobimo vrednost v in novo stanje Σ'." Večino pravil iz Sub iz slike 5.2, zlahka priredimo Ob, glej sliko 5.3.

5.4 Naloge

Naloga 5.1 Ali za funkcijski programski jezik z zapisi, opisan v 5.2.1, velja izrek o varnosti 3.2? Če velja, ga dokaži.

Naloga 5.2 Za naslednje tipe zapisov ugotovi, ali so podtipi drug drugega. Za vsak podtip tudi ugotovi, katero vrsto podtipov potrebuješ (po širini, po globini, s permutacijo,

[5] V ocamlu je to referenca in v C/C++ kazalec.

$$\frac{\eta(x) = v}{\Sigma; \eta \mid x \hookrightarrow v \mid \Sigma} \qquad \frac{\eta(\texttt{this}) = v}{\Sigma; \eta \mid \texttt{this} \hookrightarrow v \mid \Sigma} \qquad \frac{}{\Sigma; \eta \mid n \hookrightarrow n \mid \Sigma}$$

$$\frac{}{\Sigma; \eta \mid \texttt{true} \hookrightarrow \texttt{true} \mid \Sigma} \qquad \frac{}{\Sigma; \eta \mid \texttt{false} \hookrightarrow \texttt{false} \mid \Sigma}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow n_1 \mid \Sigma' \qquad \Sigma'; \eta \mid e_2 \hookrightarrow n_2 \mid \Sigma'' \qquad n = n_1 + n_2}{\Sigma; \eta \mid e_1 + e_2 \hookrightarrow n \mid \Sigma''} \quad \text{(podobno za } -, \cdot, /\text{)}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow b_1 \mid \Sigma' \qquad \Sigma'; \eta \mid e_2 \hookrightarrow b_2 \mid \Sigma''}{\Sigma; \eta \mid e_1 \,\texttt{and}\, e_2 \hookrightarrow b_1 \wedge b_2 \mid \Sigma''} \quad \text{(podobno za } \texttt{or}\text{)} \qquad \frac{\Sigma; \eta \mid e \hookrightarrow b \mid \Sigma'}{\Sigma; \eta \mid \texttt{not}\, e \hookrightarrow \neg b \mid \Sigma'}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow \texttt{true} \mid \Sigma' \qquad \Sigma'; \eta \mid e_2 \hookrightarrow v \mid \Sigma''}{\Sigma; \eta \mid \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \hookrightarrow v \mid \Sigma''} \quad \text{(podobno za } \texttt{false}\text{)}$$

$$\frac{}{\Sigma; \eta \mid (\texttt{fun}\, x : \tau \to e) \hookrightarrow \texttt{closure}(\eta, x, e) \mid \Sigma} \qquad \frac{}{\Sigma; \eta \mid \texttt{closure}(\eta, x, e) \hookrightarrow \texttt{closure}(\eta, x, e) \mid \Sigma}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow v_1 \mid \Sigma' \qquad \Sigma'; (x, v_1) :: \eta \mid e_2 \hookrightarrow v_2 \mid \Sigma''}{\Sigma; \eta \mid \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 \hookrightarrow v_2 \mid \Sigma''}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow \texttt{closure}(\eta', x, e) \mid \Sigma' \qquad \Sigma'; \eta \mid e_2 \hookrightarrow v_2 \mid \Sigma'' \qquad \Sigma''; (x, v_2) :: \eta' \mid e \hookrightarrow v \mid \Sigma'''}{\Sigma; \eta \mid e_1\; e_2 \hookrightarrow v \mid \Sigma'''}$$

$$\frac{\lambda_1, \ldots, \lambda_n \notin \mathrm{dom}(\Sigma) \qquad \Sigma^0 = \Sigma \qquad \text{za } i = 1, \ldots, k\text{:}\;\; \Sigma^{i-1}; \eta \mid e_i \hookrightarrow v_i \mid \Sigma^i}{\Sigma; \eta \mid \{\texttt{val}\, \ell_i = e_i|_{i=1}^k \,\texttt{method}\, m_j : \tau_j' = f_j|_{j=1}^n\} \hookrightarrow \{\texttt{val}\, \ell_i = \lambda_i|_{i=1}^k \,\texttt{method}\, m_j = (\eta, f_j)|_{j=1}^n\} \mid \Sigma^k[\lambda_1 \mapsto v_1, \ldots, \lambda_n \mapsto v_k]}$$

$$\frac{\Sigma; \eta \mid e \hookrightarrow \{\texttt{val}\, \ell_i = \lambda_i|_{i=1}^k \,\texttt{method}\, m_j = (\eta_j, f_j)|_{j=1}^n\} \mid \Sigma' \qquad 1 \le p \le k \qquad \Sigma'(\lambda_p) = v}{\Sigma; \eta \mid e.\ell_p \hookrightarrow v \mid \Sigma'}$$

$$\frac{\begin{array}{l}\Sigma; \eta \mid e \hookrightarrow o \mid \Sigma' \\ o = \{\texttt{val}\, \ell_i = \lambda_i|_{i=1}^k \,\texttt{method}\, m_j = (\eta_j, f_j)|_{j=1}^n\} \qquad 1 \le p \le n \qquad \Sigma'; \eta_j \mid f_j \hookrightarrow v \mid \Sigma''\end{array}}{\Sigma; \eta \mid e\#m_p \hookrightarrow v \mid \Sigma''}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow v_1 \mid \Sigma' \qquad \Sigma'; \eta \mid e_2 \hookrightarrow v_2 \mid \Sigma''}{\Sigma; \eta \mid e_1 \,;\, e_2 \hookrightarrow v_2 \mid \Sigma''}$$

$$\frac{\Sigma; \eta \mid e_1 \hookrightarrow \{\ell_i = \lambda_i\}_{i=1}^{|}\Sigma' \qquad 1 \le k \le n \qquad \Sigma'; \eta \mid e_2 \hookrightarrow v_2 \mid \Sigma''}{\Sigma; \eta \mid e_1.\ell_k := e_2 \hookrightarrow \;\mid \Sigma''[\lambda_k \mapsto v_2]} \qquad \frac{\Sigma; \eta \mid e \to v \mid \Sigma'}{\Sigma; \eta \mid e \,\texttt{as}\, \tau \to v \mid \Sigma'}$$

Slika 5.3: Evaluacija v Ob.

ali ustrezno kombinacijo teh treh):

$$\begin{aligned}
\rho &= \{\} , \\
\sigma &= \{\mathtt{a} : \mathtt{int}, \mathtt{b} : \{\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int}\}\} , \\
\tau &= \{\mathtt{b} : \{\mathtt{y} : \mathtt{int}, \mathtt{x} : \mathtt{int}\}\} , \\
\mu &= \{\mathtt{b} : \{\mathtt{y} : \mathtt{int}\}\} .
\end{aligned}$$

Naloga 5.3 Zapiši vse tipe, ki jih ima izraz $\{\mathtt{a} = \{\mathtt{x} = 8, \mathtt{y} = 0\}\}$ v programskem jeziku, v katerem veljajo vsa splošna pravila za podtipe in vsa tri pravila za podtipe zapisov, glej 5.2.

Naloga 5.4 Ali v programskem jeziku Sub obstaja neskončna padajoča veriga podtipov $\tau_1 \geq \tau_2 \geq \tau_3 \geq \cdots$, ki so si med seboj vsi različni? Kaj pa neskončna rastoča veriga $\sigma_1 \leq \sigma_2 \leq \sigma_3 \leq \cdots$?

Naloga 5.5 Izpelji pravilo (5.1) za podtipe zapisov iz pravil za podtipe zapisov po širini, globini in permutaciji, glej 5.2.

Naloga 5.6 Sub je neučakan programski jezik. Definiraj jezik LazySub, ki se od Sub razlikuje le po tem, da je len. Kaj je treba spremeniti v definiciji Sub in kako?

Naloga 5.7 Denimo, da bi v Ob dovolili sintakso, v kateri metodam v objektu ni treba določiti tipa. Pojasni, v kakšne težave bi zašli, ko bi poskusili preveriti tip izraza $\{\mathtt{method}\ f = \mathtt{this}\#f\}$.

Naloga 5.8 Statični semantiki Ob dodaj definicije tipov, ki jih pozna priložena implementacija Ob. Abstraktni sintaksi tipov dodamo še imena,

$$\text{Tip } \tau ::= \cdots \mid t ,$$

abstraktni sintaksi izrazov pa definicije tipov

$$\text{Izraz } e ::= \cdots \mid \mathtt{type}\ t = \tau .$$

Statična semantika preverja tip izraza e v *dvojnem kontekstu* $\mathrm{T};\Gamma$. Tu je Γ običajni kontekst spremenljivk, $\mathrm{T} = [(t_1, \tau_1), \ldots, (t_n, \tau_n)]$ pa kontekst definicij tipov. Ključno pravilo se glasi

$$\frac{\mathrm{T}(t) = \tau \qquad \mathrm{T};\Gamma \mid e : \tau}{\mathrm{T};\Gamma \mid e : t} ,$$

sam pa premisli, ali pride še do kakšnih drugih sprememb. Dinamična semantika ostane nespremenjena, saj nima zveze s tipi.

Naloga 5.9 S pravili za dinamično semantiko Ob izračunaj vrednost izraza

```
let a = {val n = 23} in
let b = a in
  (a.n := 42; b.n)
```

Nato preveri, ali se implementacija Ob sklada s tvojo ugotovitvijo.

Naloga 5.10 S pravili za dinamično semantiko Ob izračunaj vrednost izraza

```
let a = {val n = 2 method get:int = this.n} in
let b = {val n = 5 method get:int = a#get} in
 b#get
```

Nato preveri, ali se implementacija Ob sklada s tvojo ugotovitvijo.

Naloga 5.11 Java in C++ imata ukaz `new` in *konstruktorje*, s katerimi naredimo nov objekt. V Ob naredimo objekt preprosto tako, da naštejemo njegova polja in metode. Kljub temu pa je včasih koristno imeti funkcijo, ki vrne nov primerek danega objekta. V Ob sestavi funkcijo `newCounter` tipa $\to$ `counter`, ki pri vsakem klicu vrne nov primerek števca (za definicijo tipa `counter` glej stran 74). Primer uporabe:

```
Ob> let c1 = newCounter {}
c1 : counter = {val n = 0 method reset method get}
Ob> let c2 = newCounter {}
c2 : counter = {val n = 0 method reset method get}
Ob> c2#get
- : int = 1
Ob> c2#get
- : int = 2
Ob> c1#get
- : int = 1
```

Naloga 5.12 Ugotovi, kateri od naslednjih programov v Ob imajo tip in kakšnega:

(a) `{} as {val n:int}`

(b) `{val n = 3} as {}`

(c) `{val n = 3 method get:int = n}`

(d) `{val n = 3 method get:int = this.n}`

(e) `{val n = this.n}`

Naloga 5.13 Funkcije v Ob niso rekurzivne. Ker pa so objekti rekurzivni, saj se lahko z `this` sklicujejo sami nase, lahko vseeno definiramo rekurzivne funkcije. Pojasni, kako to naredimo. Na primer, kako bi v Ob definirali vrednost `f` : `int` $\to$ `int`, ki predstavlja rekurzivno funkcijo $f : \mathbb{N} \to \mathbb{N}$, definirano s predpisom

$$f(0) = 0 , \qquad f(n+1) = 2 \cdot f(n)^2 + 1 \ ?$$

Naloga 5.14 Ta naloga je zahtevna. Ali je z objekti v Ob mogoče definirati sezname celih števil? Očitna implementacija ne deluje, saj rekurzivne definicije tipov niso dovoljene:

```
Ob> type seznam = {val prazen:bool val glava:int val rep:seznam}
unknown type seznam
```

Razširi Ob tako, da bo dovoljeval rekurzivne definicije tipov in objektno implementiraj sezname celih števil.

Poglavje 6

Denotacijska semantika

6.1 Kaj je denotacijska semantika

Semantika programskega jezika priredi sestavnim delom jezika (tipom, kontekstom, izrazom) njihov *pomen*. Ker ima programski jezik lahko več vrst pomenov, poznamo več vrst semantik. Spoznali smo že *statično* in *dinamično* semantiko – prva za pomen izraza vzame njegov tip, druga pa pove, kako program izvajamo (zato jo pogosto imenujemo tudi *operacijska* semantika).

V tem poglavju na programe ne bomo gledali kot na navodila za izvajanje računskih ukazov, ampak bomo razmišljali o njihovem *matematičnem pomenu*. Da bomo lažje razumeli, zakaj je matematični pomen programa pomemben, se vprašajamo, kaj v MinML pomeni program

$$\texttt{fun}\ f(n : \texttt{int}) : \texttt{int is if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f(n - 1).$$

Krajši premislek nam pove, da je to funkcija fakulteta, ki za dani $n \geq 0$ izračuna zmnožek $1 \cdot 2 \cdots n$. Toda kako to sploh vemo? Sam po sebi je program le zaporedje znakov ali abstraktno sintaktično drevo. Statična semantika nam pove samo njegov tip $\texttt{int} \to \texttt{int}$, dinamična pa, kako program z zaporedjem sintaktičnih transformacij predelamo v vrednost. Nikjer ni niti besedice o kaki matematični funkciji. Kljub temu si programerji mislijo, da zgornji program predstavlja matematično funkcijo. Očitno imamo pred seboj teoretično vrzel: kako programu priredimo ustrezen matematični pomen?

Enega od odgovor na to vprašanje ponuja *denotacijska semantika*, ki sestavnim delom programskega jezika priredi ustrezne matematične strukture. V nadeljevanju si bomo najprej ogledali denotacijsko semantiko aritmetičnih izrazov. Ta ne zahteva nobene posebne matematične veščine. Ko pa se vprašamo, kakšen je matematični pomen rekurzivnih programov, se stvari zapletejo.

6.2 Naivna semantika aritmetičnih izrazov

Ko na papir zapišemo celoštevilski aritmetični izraz, na primer

$$x^2 + 2y + 1\ ,$$

si ga običajno predstavljamo kot funkcijo spremenljivk, ki nastopajo v izrazu. Zgornji izraz tako predstavlja funkcijo dveh spremenljivk x in y, ki slika (x, y) v vrednost $x^2 + 2y + 1$.

Če izraz ne vsebuje nobene spremenljivke, predstavlja neko celo število. Ta naivni pogled na odnos med nizom znakov na papirju in njihovim matematičnim pomenom opredelimo bolj natančno.

V denotacijski semantiki označimo pomen s tako imenovanim *semantičnim oklepajem* $[\![-]\!]$: pomen izraza e je $[\![e]\!]$, pomen tipa τ je $[\![\tau]\!]$, pomen konteksta Γ je $[\![\Gamma]\!]$ itn.

Kot že dobro vemo, je nespametno obravnavati izraz, v katerem nastopajo spremenljivke, brez ustreznega konteksta.[1] Zato vzemimo aritmetični izraz e v kontekstu $x_1 : \mathtt{int}, \ldots, x_n : \mathtt{int}$,

$$x_1 : \mathtt{int}, \ldots, x_n : \mathtt{int} \mid e : \mathtt{int} \;, \tag{6.1}$$

in se vprašajmo, kakšen je njegov matematični pomen. Sestavnim delom (6.1) priredimo matematične strukture. Tip `int` razumemo kot množico celih števil:

$$[\![\mathtt{int}]\!] = \mathbb{Z} \;.$$

Kontekst predstavlja urejeno n-terico spremenljivk tipa `int`:

$$[\![x_1 : \mathtt{int}, \ldots, x_n : \mathtt{int}]\!] = \underbrace{\mathbb{Z} \times \cdots \times \mathbb{Z}}_{n} = \mathbb{Z}^n \;.$$

Pomen sodbe (6.1) je funkcija

$$[\![x_1 : \mathtt{int}, \ldots, x_n : \mathtt{int} \mid e : \mathtt{int}]\!] : \mathbb{Z}^n \to \mathbb{Z} \;,$$

ki jo definiramo induktivno glede na strukturo izraza e. Zaradi lažje berljivosti označimo $\Gamma \equiv x_1 : \mathtt{int}, \ldots, x_n : \mathtt{int}$ in za $\vec{t} = (t_1, \ldots, t_n) \in \mathbb{Z}^n$ definiramo:

$$\begin{aligned}
[\![\Gamma \mid x_i : \mathtt{int}]\!](\vec{t}) &= t_i && (1 \le i \le n) \\
[\![\Gamma \mid k : \mathtt{int}]\!](\vec{t}) &= k && (k \in \mathbb{Z} \text{ konstanta}) \\
[\![\Gamma \mid e_1 + e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) + [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid e_1 - e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) - [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid e_1 \cdot e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) \cdot [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid -e : \mathtt{int}]\!](\vec{t}) &= -[\![\Gamma \mid e : \mathtt{int}]\!](\vec{t}) \;.
\end{aligned}$$

Ker je zgornji zapis dokaj kompliciran, si oglejmo podrobneje pomen vsote $e_1 + e_2$. V izrazih e_1 in e_2 lahko nastopajo spremenljivke $x_1, \ldots, x_n$. Definicija nam pove pomen izraza $e_1 + e_2$, če za vrednosti spremenljivk vstavimo cela števila $t_1, \ldots, t_n$: pomen izraza $e_1 + e_2$ je vsota pomenov e_1 in e_2. Podobno velja za ostale operacije. Na primer, pomen $x : \mathtt{int}, y : \mathtt{int} \mid x \cdot y + 1$ pri $x = 2$, $y = 3$ izračunamo takole:

$$\begin{aligned}
&[\![x : \mathtt{int}, y : \mathtt{int} \mid x \cdot y + 1 : \mathtt{int}]\!](2,3) = \\
&\qquad [\![x : \mathtt{int}, y : \mathtt{int} \mid x \cdot y : \mathtt{int}]\!](2,3) + [\![x : \mathtt{int}, y : \mathtt{int} \mid 1 : \mathtt{int}]\!](2,3) = \\
&\qquad [\![x : \mathtt{int}, y : \mathtt{int} \mid x : \mathtt{int}]\!](2,3) \cdot [\![x : \mathtt{int}, y : \mathtt{int} \mid y : \mathtt{int}]\!](2,3) + 1 = \\
&\qquad\qquad 2 \cdot 3 + 1 = 7 \;.
\end{aligned}$$

[1] Težava je v tem, da so iz izraza razvidne le tiste spremenljivke, ki se dejansko pojavijo v izrazu. Lahko pa imamo še kakšne spremenljivke, od katerih izraz ni odvisen, a jih vseeno želimo upoštevati.

Zdi se, da nismo naredili prav nič, saj smo definirali, da "plus pomeni plus" in "krat pomeni krat"! Vendar je treba zapis pravilno razumeti. Na levi strani defincije + pomeni znak, zapisan na papirju ali v programu, na desni pa označuje matematično opreacijo seštevanje celih števil. Ker je semantika aritmetičnih izrazov zelo preposta, je morda v tem trenutku težko razumeti, zakaj tako kompliciramo.

Povzemimo: denotacijska semantika celoštevilskih aritmetičnih izrazov priredi tipu `int` množico celih števil. Celoštevilski izraz e v kontekstu $x_1 : \mathtt{int}, \ldots, x_n : \mathtt{int}$ predstavlja n-členo funkcijo $\mathbb{Z}^n \to \mathbb{Z}$.

6.2.1 Deljenje z nič in nedefinirana vrednost

V prejšnjem razdelku smo podali denotacijsko semantiko aritmetičnih s seštevanjem, odštevanjem, množenjem in nasprotno vrednostjo. Ko želimo definirati še pomen celoštevilskega deljenja $\div$, naletimo na vprašanje, kaj storiti glede deljenja z nič. Ker le-to ni definirano, bi lahko pomen izraza e v kontekstu Γ definirali kot *delno*[2] funkcijo $\mathbb{Z}^n \rightharpoonup \mathbb{Z}$. Izkaže se, da je nekoliko drugačna, a ekvivalentna rešitev boljša pri obravnavi splošnih programskih jezikov. Namesto, da bi rekli, da je $1 \div 0$ nedefinirano, si mislimo, da je $1 \div 0$ enako posebni vrednosti *dno* $\bot$, ki pomeni "nedefinirano".

Za poljubno množico A naj bo *dvig* $A_\bot$ množica

$$A_\bot = A + \{\bot_A\} ,$$

ki vsebuje poleg elementov A še poseben element *dno* $\bot_A$. Zakaj se $A_\bot$ imenuje "dvig" in $\bot_A$ "dno" bo pojasnjeno v 6.3.2. Kadar je očitno, za katero množico gre, pišemo $\bot$ namesto $\bot_A$.

Če je $f : A \to B$ poljubna funkcija, je *dvig* $f_\bot : A_\bot \to B_\bot$ funkcija

$$f_\bot(x) = \begin{cases} \bot_B & \text{če je } x = \bot_A \\ f(x) & \text{če je } x \in A \,. \end{cases}$$

Podobno za funkcijo dveh spremenljivk $g : A \times B \to C$ definiramo $g_{\bot,\bot} : A_\bot \times B_\bot \to C_\bot$ s predpisom

$$g_{\bot,\bot}(x, y) = \begin{cases} \bot_C & \text{če je } x = \bot_A \text{ ali } y = \bot_B \\ g(x, y) & \text{če je } x \in A \text{ in } y \in B \,. \end{cases}$$

Celoštevilsko deljenje $\div$ definiramo kot funkcijo $\mathbb{Z}_\bot \times \mathbb{Z}_\bot \to \mathbb{Z}_\bot$ s predpisom

$$n \div m = \begin{cases} k & \text{če je } k, n \in \mathbb{Z},\ m \in \mathbb{Z} \setminus \{0\} \text{ in } 0 \leq n - k \cdot m < m \\ \bot & \text{če je } n = \bot,\ m = \bot \text{ ali } m = 0 \end{cases}$$

Denotacijska semantika celoštevilskih aritmetičnih izrazov s celoštevilskim deljenjem se

[2]Delna funkcija $f : A \rightharpoonup B$ je funkcija $f : A' \to B$ z domeno $A' \subseteq A$.

glasi:

$$
\begin{aligned}
[\![\mathtt{int}]\!] &= \mathbb{Z}_\bot \\
[\![\Gamma \mid e : \mathtt{int}]\!] &: \mathbb{Z}_\bot^n \to \mathbb{Z}_\bot \\
[\![\Gamma \mid x_i : \mathtt{int}]\!](\vec{t}) &= t_i \\
[\![\Gamma \mid k : \mathtt{int}]\!](\vec{t}) &= k \\
[\![\Gamma \mid e_1 + e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) +_{\bot,\bot} [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid e_1 - e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) -_{\bot,\bot} [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid e_1 \cdot e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) \cdot_{\bot,\bot} [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid -e : \mathtt{int}]\!](\vec{t}) &= -_\bot [\![\Gamma \mid e : \mathtt{int}]\!](\vec{t}) \\
[\![\Gamma \mid e_1 \div e_2 : \mathtt{int}]\!](\vec{t}) &= [\![\Gamma \mid e_1 : \mathtt{int}]\!](\vec{t}) \div [\![\Gamma \mid e_2 : \mathtt{int}]\!](\vec{t}) \;.
\end{aligned}
$$

Za razliko od semantike v 6.2 poleg celih števil tu dopuščamo tudi vrednost $\bot_\mathbb{Z}$, ki pomeni "nedefinirano". Ker so lahko vrednosti podizrazov enake $\bot_\mathbb{Z}$ smo namesto običajnih aritmetičnih operacij $+$, $-$, $\cdot$ in unarni $-$ uporabili njihove dvige $+_{\bot,\bot}$, $-_{\bot,\bot}$ in $\cdot_{\bot,\bot}$ in $-_\bot$.

Denotacijska semantika aritmetičnih izrazov ima pomembno lastnost: pomen izraza je funkcija pomenov njegovih podizrazov. Pravimo, da je taka semantika *kompozicijska*, saj lahko izračunamo pomen izraza tako, da sestavimo skupaj (komponiramo) pomene njegovih sestavnih delov.

6.3 Semantika funkcijskega jezika z rekurzijo

V tem rezdelku obravnavamo denotacijsko semantiko funkcijskega programskega jezika PCF (Programming Language for Computable Functions), ki je nekakšna "laboratorijska miš" teorije programskih jezikov. Jezik PCF je podmnožica MinHaskell zato ga bomo zlahka definirali in razumeli.

Preden se posvetimo denotacijski semantiki PCF, opozorimo na težavo, ki nas čaka. Denimo, da bi želeli razširiti naivno semantiko aritmetičnih izrazov na splošen funkcijski programski jezik: za pomen tipa τ določimo ustrezno množico vrednosti $[\![\tau]\!]$, ki po potrebi vsebuje tudi nedefinirano vrednost $\bot_\tau$, in sicer

$$
\begin{aligned}
[\![\mathtt{int}]\!] &= \mathbb{Z}_\bot \\
[\![\mathtt{bool}]\!] &= \{\mathtt{true}, \mathtt{false}\}_\bot \\
[\![\tau_1 \to \tau_2]\!] &= [\![\tau_2]\!]^{[\![\tau_1]\!]} \;.
\end{aligned}
$$

Tu $[\![\tau_2]\!]^{[\![\tau_1]\!]}$ označuje *eksponentno množico*.[3] Pomen konteksta $\Gamma \equiv x_1 : \tau_1, \ldots, x_n : \tau_n$ je kartezični produkt pomenov tipov τ_i,

$$[\![\Gamma]\!] \equiv [\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \;.$$

Pomen izraza $\Gamma \mid e : \tau$ je funkcija $[\![\Gamma]\!] \to [\![\tau]\!]$, ki jo definiramo glede na strukturo e. Tako funkcijo bi zares lahko definirali za vse sestavne dele MinHaskell, zataknilo pa bi se pri

[3] Eksponentna množica B^A je množica vseh funkcij iz A v B.

pomenu rekurzivnih definicij. Denimo nasprotno, da bi nam to uspelo. Tedaj bi bil pomen izraza

$$\mathtt{f} : \mathtt{int} \to \mathtt{int} \mid (\mathtt{rec}\; x : \mathtt{int}\; \mathtt{is}\; \mathtt{f}\, x) : \mathtt{int}$$

neka funkcija $\phi : (\mathbb{Z}_\perp)^{\mathbb{Z}_\perp} \to \mathbb{Z}_\perp$. Ker v MinHaskell velja

$$(\mathtt{rec}\; x : \mathtt{int}\; \mathtt{is}\; \mathtt{f}\, x) = \mathtt{f}\,(\mathtt{rec}\; x : \mathtt{int}\; \mathtt{is}\; \mathtt{f}\, x)\;,$$

mora za ϕ in za *vsako* funkcijo $f : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ seveda veljati

$$\phi(f) = f(\phi(f))\;,$$

Povedano z drugimi besedami, ϕ izračuna *negibno točko*[4] funkcije f. Tak ϕ pa ne more obstajati, saj funkcija $g : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$, definirana s predpisom

$$g(n) = \begin{cases} 0 & \text{če je } n \neq 0 \\ 1 & \text{če je } n = 0 \end{cases}$$

nima nobene negibne točke! Ta razmislek nam pove, da matematični pomen splošnega programskega jezika z rekurzijo ni tako preprost, kot bi to pričakovali na prvi pogled.

6.3.1 PCF

MinHaskell vsebuje sezname, celoštevilsko aritmetiko in osnovne logične operacije. Ker se želimo osredotočiti na rekurzivne definicije, ga najprej nekoliko oklestimo.

Programski jezik PCF je leni funkcijski jezik, ki je poenostavljena verzija MinHaskell. Tipi v PCF so

$$\text{Tip } \tau ::= \mathtt{nat} \mid \mathtt{bool} \mid \tau_1 \to \tau_2\;.$$

Glede na MinHaskell smo opustili sezname in cela števila nadomestili z naravnimi. Izrazi v PCF so podani z naslednjo abstraktno sintakso:

$$\begin{aligned}\text{Izraz } e ::= {} & 0 \mid \mathtt{true} \mid \mathtt{false} \mid x \mid \mathtt{succ}\, e \mid \mathtt{pred}\, e \mid \mathtt{iszero}\, e \mid \\ & \mathtt{if}\; e_1\; \mathtt{then}\; e_2\; \mathtt{else}\; e_3 \mid \mathtt{fun}\; x : \tau \to e \mid e_1\, e_2 \mid \mathtt{rec}\; x : \tau\; \mathtt{is}\; e\end{aligned}$$

V primerjavi z MinHaskell smo opustili aritmetične operacije in primerjave ter jih nadomestili s funkcijami naslednik `succ`, predhodnik `pred` in testom `iszero`, ki vrne `true`, če je argument enak nič. Prav tako smo ohranili le celoštevilsko konstanto 0.

V PCF so naravna števila predstavljena v "eniškem sistemu": $0 = 0$, $1 = \mathtt{succ}\,0$, $2 = \mathtt{succ}\,(\mathtt{succ}\,0)$ itn. Za čitlivejši zapis definiramo okrajšavo

$$\overline{n} = \underbrace{\mathtt{succ}(\mathtt{succ}(\cdots(\mathtt{succ}}_{n}\,0)\cdots))\;.$$

Pravila za preverjanje tipov v PCF so enaka kot v MinHaskell za konstante 0, `false` in `true`, za spremenljivke, pogojni stavek, funkcije, aplikacijo in rekurzivne definicije, glej 3.7.3. K tem dodamo še pravila za `succ`, `pred` in `isezero`:

$$\frac{\Gamma \mid e : \mathtt{nat}}{\Gamma \mid \mathtt{succ}\, e : \mathtt{nat}} \qquad \frac{\Gamma \mid e : \mathtt{nat}}{\Gamma \mid \mathtt{pred}\, e : \mathtt{nat}} \qquad \frac{\Gamma \mid e : \mathtt{nat}}{\Gamma \mid \mathtt{iszero}\, e : \mathtt{bool}}$$

[4] Točka $x \in A$ je *negibna točka* za funkcijo $f : A \to A$, če velja $x = f(x)$.

Vrednosti v PCF so:

$$\text{Vrednost } v ::= \overline{n} \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\, x : \tau \to e\,.$$

Pravila za evaluacijo pogojnega stavka, aplikacije in rekurzivne definicije prevzamemo od MinHaskell, zraven pa dodamo še pravila

$$\frac{e \mapsto e'}{\texttt{pred}\, e \mapsto \texttt{pred}\, e'} \qquad \frac{}{\texttt{pred}\, 0 \mapsto 0} \qquad \frac{}{\texttt{pred}\, \overline{n+1} \mapsto \overline{n}}$$

$$\frac{e \mapsto e'}{\texttt{iszero}\, e \mapsto \texttt{iszero}\, e'} \qquad \frac{}{\texttt{iszero}\, 0 \mapsto \texttt{true}} \qquad \frac{n > 0}{\texttt{iszero}\, \overline{n} \mapsto \texttt{false}}\,.$$

6.3.2 Domene

V tem razdelku predstavimo *domene*. To so matematične strukture z lastnostmi, ki jih potrebujemo za zadovoljivo semantiko PCF. Obstaja več vrst domen, ki se uporabljajo za denotacijsko semantiko raznih programskih jezikov. Tu bomo obravnavali *ω-polne delne ureditve (ω-cpo*[5]*)*, ki so najpreprostejše od vseh domen.

Vemo, da za denotacijsko semantiko PCF ne moremo uporabiti naivnega pristopa, v katerem tipe interpretiramo kot običajne množice in izraze kot funkcije med množicami. Množice moramo obogatiti z dodatno strukturo in funkcije omejiti tako, da bomo lahko interpretirali tudi rekurzivne definicije.

Osnovna ideja teorije domen je, da je matematični pomen podatkovnega tipa delno urejena množica, zato moramo najprej ponoviti nekaj osnovnih pojmov iz teorije urejenosti.

Definicija 6.1 *Delno urejena množica* $(D, \leq)$ je množica D z relacijo $\leq$, ki je:

1. refleksivna: $x \leq x$,
2. tranzitivna: iz $x \leq y$ in $y \leq z$ sledi $x \leq z$,
3. antisimetrična: iz $x \leq y$ in $y \leq x$ sledi $x = y$.

Funkcija $f : D \to E$ med delno urejenima množicama je *monotona*, če ohranja relacijo urejenosti: iz $x \leq_D y$ sledi $f(x) \leq_E f(y)$.

Definicija 6.2 Element $x \in D$ v delno urejeni množici $(D, \leq)$ se imenuje *dno* ali *najmanjši element*, če za vse $y \in D$ velja $x \leq y$.

Nima vsaka delno urejena množica najmanjšega elementa. Na primer, pozitivna realna števila urejena z običajno relacijo "manjše ali enako", nimajo dna (nič ni dno, ker ni pozitivno realno število).

V denotacijski semantiki tip τ v programskem jeziku interpretiramo kot (določene vrste) delno urejeno množico D. Elementi D predstavljajo informacijo o vrednostih tipa τ. Delna urejenost $\leq$ je mera za količino informacije: $x \leq y$ pomeni, da x vsebuje manj informacije kot y. Tako na primer dno delno urejene množice predstavlja "ničelno informacijo" ali "nedefinirano vrednost".

[5]Kratica ω-cpo pomeni "ω-chain complete partial order".

Definicija 6.3 Naj bo $(D, \leq)$ delno urejena množica in $S \subseteq D$. Pravimo, da je $x \in D$ *spodnja meja* množice S, če velja $x \leq y$ za vsak $y \in S$. Podobno je $z \in D$ *zgornja meja* množice S, če velja $y \leq z$ za vsak $y \in S$.

Definicija 6.4 Naj bo $(D, \leq)$ delno urejena množica in $S \subseteq D$. Pravimo, da je $x \in D$ *natančna spodnja meja* ali *infimum* množice S, če je spodnja meja za S in za vsak $x' \in D$, ki je spodnja meja za S, velja $x' \leq x$. Pravimo, da je $z \in D$ *natančna zgornja meja* ali *supremum* množice S, če je zgornja meja za S in za vsak $z' \in D$, ki je zgornja meja za S, velja $z \leq z'$.

Infimum dane množice S ne obstaja nujno. Kadar pa obstaja, je enolično določen in ga označimo z $\bigwedge S$ ali $\inf S$. Podobno velja za supremum, ki ga označimo z $\bigvee S$ ali $\sup S$. Namesto $\bigwedge\{x, y\}$ in $\bigvee\{x, y\}$ pišemo $x \wedge y$ in $x \vee y$.

Definicija 6.5 Naraščajoče zaporedje $x_0 \leq x_1 \leq x_2 \leq \cdots$ v delno urejeni množici $(D, \leq)$ se imenuje *veriga*.

Definicija 6.6 Delno urejena množica $(D, \leq)$ je *ω-polna*, če ima vsaka veriga v D supremum.

Definicija 6.7 *Domena* je ω-polna delno urejena množica z dnom. Funkcija $f : D \to E$ med domenama D in E je *zvezna*, če je monotona in ohranja supremume verig: za vsako verigo $(x_i)_{i \in \mathbb{N}}$ v D velja

$$f(\textstyle\bigvee_{i \in \mathbb{N}} x_i) = \textstyle\bigvee_{i \in \mathbb{N}} f(x_i).$$

Izrek 6.8

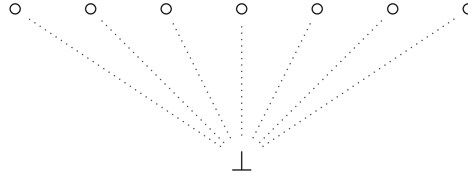
1. *Identiteta je zvezna funkcija.*

2. *Konstantna funkcija je zvezna.*

3. *Kompozitum zveznih funkcij je zvezna funkcija.*

Dokaz. Dokaz prepuščamo za nalogo. ■

Zakaj so domene primerne za denotacijsko semantiko tipov? Denimo, da je denotacijski pomen tipa τ neka domena $[\![\tau]\!] = D$. Tedaj dno domene predstavlja divergentni program tipa τ, na primer `rec` $x : \tau$ `is` x. V splošnem pa si predstavljamo, da dobimo pomen programa $p : \tau$ kot supremum verige $x_0 \leq x_1 \leq \cdots$ v D, pri čemer elementi verige x_i izražajo informacijo o p, ki jo pridobimo, ko računamo vrednost p z zaporedjem evaluacijskih korakov. Kadar so vrednosti tipa τ "končne", denimo če je $\tau =$ `bool` ali $\tau =$ `int`, lahko veriga približkov doseže supremum že po končno mnogo korakih. Ko pa imamo opravka z neskončnimi domenami, na primer $\tau =$ `int` $\to$ `bool`, se prav lahko zgodi, da je veriga $x_0 \leq x_1 \leq \cdots$ neskončna, saj potrebujemo za opis vrednosti tipa τ neskončno količino informacije.



Slika 6.1: Dvig $A_\bot$

Konstrukcije domen

Poljubno množico A lahko pretvorimo v domeno $A_\bot$ z operacijo *dvig*, ki smo jo že videli v 6.2.1. Definiramo

$$A_\bot = A + \{\bot_A\}$$

in na $A_\bot$ uvedemo relacijo $\leq$ s predpisom

$$x \leq y \iff x = \bot \lor x = y \; . \tag{6.2}$$

Domena $A_\bot$ je predstavljena na sliki 6.1. Elementi množice A so med seboj neprimerljivi, pod njimi pa je dno $\bot_A$. Slika tudi pojasnjuje izraz *dvig*: elemente množice A "dvignemo" in pod njih postavimo dno. Domeni oblike $A_\bot$ pravimo tudi *ploščata domena*.

Izjava 6.9 *Dvig $f_\bot : A_\bot \to B_\bot$ je zvezna funkcija.*

Dokaz. Dokaz prepuščamo za nalogo. ■

Če sta $(D, \leq_D)$ in $(E, \leq_E)$ domeni, je tudi njun kartezični produkt $D \times E$ domena, urejena po komponentah:

$$(x, y) \leq_{D\times E} (x', y') \iff x \leq_D x' \land y \leq y' \; .$$

Zlahka preverimo, da je $\leq_{D\times E}$ delna ureditev. Dno domene $D \times E$ je par $(\bot_D, \bot_E)$. Supremum verige $(x_0, y_0) \leq (x_1, y_1) \leq \cdots$ izračunamo po komponentah,

$$\bigvee_{i\in\mathbb{N}}(x_i, y_i) = (\bigvee_{i\in\mathbb{N}} x_i, \bigvee_{i\in\mathbb{N}} y_i) \; .$$

Naslednji izrek nam pogosto olajša delo pri dokazovanju, da je funkcija dveh argumentov zvezna.

Izrek 6.10 *Naj bodo D_1, D_2 in E domene. Funkcija $f : D_1 \times D_2 \to E$ je zvezna natanko tedaj, ko je zvezna v vsakem argumentu posebej, se pravi: za vsak $x \in D_1$ in verigo $y_0 \leq y_1 \leq \cdots$ v D_2 je $\bigvee_{i\in\mathbb{N}} f(x, y_i) = f(x, \bigvee_{i\in\mathbb{N}} y_i)$ in za vsako verigo $x_0 \leq x_1 \leq \cdots$ v D_1 in $y \in D_2$ je $\bigvee_{i\in\mathbb{N}} f(x_i, y) = f(\bigvee_{i\in\mathbb{N}} x_i, y)$.*

Dokaz. Če je f zvezna, je očitno zvezna tudi v vsakem argumentu posebej. Denimo torej, da je f zvezna v vsakem argumentu posebej, in dokažimo, da je f zvezna. Za poljubno verigo $(x_0, y_0) \leq (x_1, y_1) \leq \cdots$ v $D_1 \times D_2$ velja, upoštevaje zveznost f v vsakem argumentu posebej:

$$f(\bigvee_{i\in\mathbb{N}}(x_i, y_i)) = f(\bigvee_{i\in\mathbb{N}} x_i, \bigvee_{j\in\mathbb{N}} y_j) = \bigvee_{i\in\mathbb{N}} f(x_i, \bigvee_{j\in\mathbb{N}} y_j) = \bigvee_{i\in\mathbb{N}} \bigvee_{j\in\mathbb{N}} f(x_i, y_j) = \bigvee_{i\in\mathbb{N}} f(x_i, y_i) \; .$$

V zadnjem koraku zgornje izpeljave smo uporabili naslednjo lemo. ■

Lema 6.11 *Naj bo* $(x_{i,j})_{i,j \in \mathbb{N}}$ *tako dvojno zaporedje v domeni* D*, da iz* $i \leq i'$ *in* $j \leq j'$ *sledi* $x_{i,j} \leq x_{i',j'}$*. Tedaj je* $\bigvee_{i \in \mathbb{N}} (\bigvee_{i \in \mathbb{N}} x_{i,j}) = \bigvee_{i \in \mathbb{N}} x_{i,i}$*.*

Dokaz. Prepuščamo za nalogo. ∎

Naj bosta D in E domeni. Definiramo

$$[D \to E] = \{f : D \to E \mid f \text{ zvezna}\}$$

in uvedemo relacijo $\leq_{[D \to E]}$ s predpisom

$$f \leq_{[D \to E]} g \iff \forall x \in D.\, f(x) \leq_E g(x) \,.$$

Tedaj je $([D \to E], \leq_{[D \to E]})$ delna ureditev, ki je tudi domena. Dno je funkcija $\bot_{[D \to E]} : D \to E$, ki slika vse elemente v $\bot_E$. Supremum verige funkcij $f_0 \leq f_1 \leq \cdots$ pa izračunamo po točkah:

$$(\bigvee_{i \in \mathbb{N}} f_i)(x) = \bigvee_{i \in \mathbb{N}} f_i(x) \,.$$

Evaluacija $e : [D \to E] \times D \to E$ je funkcija $e(f, x) = f(x)$. Iz izreka 6.10 hitro sledi, da je evaluacija zvezna funkcija.

Naj bo $f : D \times E \to F$ zvezna funkcija. Definirajmo *transponirano funkcijo* $\widetilde{f} : D \to [E \to F]$ s predpisom $\widetilde{f}(x)(y) = f(x, y)$. Tudi transponirana funkcija je zvezna, saj za vsako verigo $x_0 \leq x_1 \leq \cdots$ v D in $y \in E$ velja

$$\widetilde{f}(\bigvee_{i \in \mathbb{N}} x_i)(y) = f(\bigvee_{i \in \mathbb{N}} x_i, y) = \bigvee_{i \in \mathbb{N}} f(x_i, y) = \bigvee_{i \in \mathbb{N}} \widetilde{f}(x_i)(y) = (\bigvee_{i \in \mathbb{N}} \widetilde{f}(x_i))(y) \,.$$

Izrek o negibni točki

V tem razdelku dokažemo izrek, zaradi katerega smo pravzaprav definirali domene in ki nam omogoča, da v domenah interpretiramo splošne rekurzivne definicije.

Izrek 6.12 (o negibni točki) *Vsaka zvezna funkcija* $f : D \to D$ *ima najmanjšo negibno točko.*

Dokaz. Za $n \geq 0$ in $x \in D$ pišemo

$$f^n(x) = \underbrace{f(f(\cdots f}_{n}(x) \cdots)) \,.$$

Torej je $f^0(x) = x$, $f^1(x) = f(x)$, $f^2(x) = f(f(x))$ itd. Trdimo, da je zaporedje $\bot_D, f(\bot_D), f^2(\bot_D), \ldots$ veriga v D, kar dokažemo z indukcijo. Očitno je $\bot_D \leq f(\bot_D)$, saj je $\bot_D$ najmanjši element v D. Denimo, da smo že dokazali $f^i(\bot_D) \leq f^{i+1}(\bot_D)$. Ker je f zvezna, je tudi monotona, zato velja $f^{i+1}(\bot_D) = f(f^i(\bot)_D) \leq f(f^{i+1}(\bot_D)) = f^{i+2}(\bot_D)$, s čimer je dokazan indukcijski korak. Element $x = \bigvee_{i \in \mathbb{N}} f^i(\bot_D)$ je negibna točka f:

$$f(x) = f(\bigvee_{i \in \mathbb{N}} f^i(\bot_D)) = \bigvee_{i \in \mathbb{N}} f(f^i(\bot_D)) = \bigvee_{i \in \mathbb{N}} f^{i+1}(\bot_D) = \bigvee_{i \in \mathbb{N}} f^i(\bot_D) = x \,.$$

Denimo, da je $y \in D$ tudi negibna točka za f. Z indukcijo pokažemo, da velja $f^i(\bot_D) \leq y$. Za $i = 0$ je to očitno. Če velja $f^i(\bot_D) \leq y$, potem je $f^{i+1}(\bot_D) = f(f^i(\bot_D)) \leq f(y) = y$, kar dokazuje indukcijski korak. Ker torej velja $f^i(\bot_D) \leq y$ za vse $i \in \mathbb{N}$, velja tudi $x = \bigvee_{i \in \mathbb{N}} f^i(\bot_D) \leq y$. Torej je x najmanjša negibna točka f. ∎

Definirajmo preslikavo $\mathsf{fix}_D : [D \to D] \to D$, ki zvezni funkciji $f : D \to D$ priredi njeno najmanjšo negibno točko,

$$\mathsf{fix}_D(f) = \bigvee_{i \in \mathbb{N}} f^i(\bot_D) .$$

Izjava 6.13 *Preslikava* $\mathsf{fix}_D : [D \to D] \to D$ *je zvezna.*

Dokaz. Prepuščamo za nalogo. ∎

6.3.3 Semantika PCF

Nazadnje podajmo denotacijsko semantiko PCF. Osnovna ideja, da je izraz $e : \tau$ v kontekstu Γ funkcija spremenljivk iz Γ, še vedno velja. Vendar pa dopuščamo kot veljavne samo *zvezne* funkcije med domenami.

Tipe interpretiramo kot domene:[6]

$$\begin{aligned} [\![\mathtt{bool}]\!] &= \{\mathsf{t}, \mathsf{f}\}_\bot \\ [\![\mathtt{nat}]\!] &= \mathbb{N}_\bot \\ [\![\tau_1 \to \tau_2]\!] &= [[\![\tau_1]\!] \to [\![\tau_2]\!]] . \end{aligned}$$

Interpretacija konteksta $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ je kartezični produkt domen

$$[\![\Gamma]\!] = [\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] .$$

Sodbo $\Gamma \mid e : \tau$ interpretiramo kot zvezno funkcijo

$$[\![\Gamma \mid e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!] ,$$

ki je definirana glede na strukturo izraza e. Naj bo $\vec{t} = (t_1, \ldots, t_n) \in [\![\Gamma]\!]$. Če je e ena izmed konstant, je pomen jasen:

$$\begin{aligned} [\![\Gamma \mid 0 : \mathtt{nat}]\!](\vec{t}) &= 0 \\ [\![\Gamma \mid \mathtt{true} : \mathtt{bool}]\!](\vec{t}) &= \mathsf{t} \\ [\![\Gamma \mid \mathtt{false} : \mathtt{bool}]\!](\vec{t}) &= \mathsf{f} \end{aligned}$$

Te definicije res določajo zvezne funkcije, saj je konstantna funkcija zvezna. Pomen spremenljivke je ustrezna projekcija, ki je tudi zvezna funkcija:

$$[\![\Gamma \mid x_i : \tau_i]\!](\vec{t}) = t_i .$$

Naslednik, predhodnik in primerjanje z ničlo določajo naslednje funkcije:

$$[\![\Gamma \mid \mathtt{succ}\, e : \mathtt{nat}]\!](\vec{t}) = \begin{cases} n + 1 & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = n \in \mathbb{N} \\ \bot & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = \bot \end{cases}$$

$$[\![\Gamma \mid \mathtt{pred}\, e : \mathtt{nat}]\!](\vec{t}) = \begin{cases} 0 & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = 0 \\ n - 1 & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = n \in \mathbb{N} \setminus \{0\} \\ \bot & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = \bot \end{cases}$$

$$[\![\Gamma \mid \mathtt{iszero}\, e : \mathtt{bool}]\!](\vec{t}) = \begin{cases} \mathsf{t} & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = 0 \\ \mathsf{f} & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) \in \mathbb{N} \setminus \{0\} \\ \bot & \text{če je } [\![\Gamma \mid e : \mathtt{nat}]\!](\vec{t}) = \bot \end{cases}$$

[6] V domeni $\{\mathsf{t}, \mathsf{f}\}_\bot$ sta t in f dve različni izbrani konstanti in ni pomembno, kako ju poimenujemo.

Da so tako definirane funkcije zvezne, sledi iz izjave 6.9. Nekoliko manj očitno je, da tudi naslednja definicija pomena pogojnega stavka da zvezno funkcijo:

$$[\![\Gamma : (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : \tau]\!](\vec{t}) = \begin{cases} [\![\Gamma \mid e_2 : \tau]\!](\vec{t}) & \text{če je } [\![\Gamma \mid e_1 : \texttt{bool}]\!](\vec{t}) = \texttt{t} \\ [\![\Gamma \mid e_3 : \tau]\!](\vec{t}) & \text{če je } [\![\Gamma \mid e_1 : \texttt{bool}]\!](\vec{t}) = \texttt{f} \\ \bot & \text{če je } [\![\Gamma \mid e_1 : \texttt{bool}]\!](\vec{t}) = \bot \end{cases}$$

Denotacijski pomen funkcij dobimo s transpozicijo:

$$[\![\Gamma \mid (\texttt{fun}\ x : \tau \to e) : \tau \to \sigma]\!](\vec{t})(t_0) = [\![x : \tau, \Gamma \mid e : \sigma]\!](t_0, \vec{t})$$

Ker je funkcija na desni strani zgornje enačbe zvezna, je zvezna tudi njena transponiranka na levi strani. Aplikacijo interpretiramo kot aplikacijo, ki je zvezna funkcija:

$$[\![\Gamma \mid e_1\, e_2 : \sigma]\!](\vec{t}) = ([\![\Gamma \mid e_1 : \tau \to \sigma]\!](\vec{t}))([\![\Gamma \mid e_2 : \tau]\!](\vec{t}))$$

Nazadnje nam preostane še pomen splošne rekurzivne definicije. Tu uporabimo funkcijo fix, ki zvezni funkciji priredi njeno najmanjšo negibno točko:

$$[\![\Gamma \mid (\texttt{rec}\ x : \tau\ \texttt{is}\ e) : \tau]\!](\vec{t}) = \mathsf{fix}_{[\![\tau]\!]}(t_0 \mapsto [\![x : \tau, \Gamma \mid e : \tau]\!](t_0, \vec{t}))\ .$$

V zgornji definiciji zapis $t_0 \mapsto \cdots$ označuje funkcijo, ki sprejme argument $t_0 \in [\![\tau]\!]$.

Opazimo, da je semantika PCF kompozicijska.

Ustreznost semantike PCF

V prejšnjem razdelku smo definirali denotacijsko semantiko PCF. Vprašati pa se moramo, ali ima ta semantika kakšne dobre lastnosti. Kako sploh vemo, da nismo definirali nekaj povsem neuporabnega?

Programa e_1 in e_2 tipa τ v PCF sta *ekvivalentna*, če ju ne moremo ločiti z Boolovim testom. Natančneje, za vsak program $t : \tau \to \texttt{bool}$ velja, da bodisi $t\, e_1$ in $t\, e_2$ divergirata, bodisi se oba evaluirata v isto Boolovo vrednost, $t\, e_1 = t\, e_2$.

Pravimo, da je semnatika (ne nujno denotacijska) *ustrezna*, če za vsaka programa e_1 in e_2 velja: če je $[\![\cdot \mid e_1 : \tau]\!] = [\![\cdot \mid e_2 : \tau]\!]$, sta e_1 in e_2 ekvivalentna. Povedano z drugimi besedami, ustrezna semantika ne izenačuje programov, ki jih lahko razločimo z Boolovim testom.

Izrek 6.14 (ustreznost semantike) *Denotacijska semantika PCF iz 6.3.3 je* ustrezna.

Dokaz. Dokaz je dokaj zahteven in presega okvire teh zapiskov, zato ga opuščamo. ∎

6.4 Naloge

Naloga 6.15 Izračunaj denotacijski pomen celoštevilskega izraza $0 \cdot (1 \div 0)$ v praznem kontekstu.

Naloga 6.16 Kateri od naslednjih izrazov imajo enak denotacijski pomen:

(a) $x : \texttt{int} \mid (x \cdot x - 1) \div (x - 1) : \texttt{int}$

(b) $x : \texttt{int} \mid x + 1 : \texttt{int}$

(c) $x : \texttt{int} \mid x \cdot x + 1 : \texttt{int}$

(d) $x : \texttt{int}, y : \texttt{int} \mid x \cdot x + 1 : \texttt{int}$

(e) $x : \texttt{int} \mid (x + 1) \cdot (x + 1) - 2 \cdot x : \texttt{int}$

Naloga 6.17 V PCF definiraj funkcijo $\texttt{add} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$, ki izračuna vsoto danih naravnih števil.

Naloga 6.18 V PCF definiraj funkcijo $\texttt{mul} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$, ki izračuna zmnožek danih naravnih števil.

Naloga 6.19 V PCF definiraj funkcijo $\texttt{sub} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$, ki za vse $m, n \in \mathbb{N}$ zadošča pogoju

$$\texttt{sub}\,\overline{m}\,\overline{n} = \begin{cases} \overline{m - n} & \text{če je } m \geq n \\ 0 & \text{če je } m < n \end{cases}$$

Naloga 6.20 V PCF definiraj funkcijo $\texttt{eq} : \texttt{nat} \to \texttt{nat} \to \texttt{bool}$, ki za vse $m, n \in \mathbb{N}$ zadošča

$$\texttt{eq}\,\overline{m}\,\overline{n} = \begin{cases} \texttt{true} & \text{če je } m = n \\ \texttt{false} & \text{sicer} \end{cases}$$

Naloga 6.21 Dokaži, da v mreži[7] velja $x \leq y$ natanko tedaj, ko je $x \vee y = y$, in natanko tedaj, ko je $x \wedge y = x$.

Naloga 6.22 Naj bo $f : D \to E$ preslikava med domenama in denimo, da za vsako verigo $x_0 \leq x_1 \leq \cdots$ v D velja, da obstaja supremum zaporedja $f(x_0), f(x_1), \ldots$ v E ter $f(\bigvee_{i \in \mathbb{N}}) = \bigvee_{i \in \mathbb{N}} f(x_i)$. Dokaži, da je f zvezna preslikava. (Dokazati je treba monotonost f, ohranjanje supremumov je že predpostavljeno.)

Naloga 6.23 Podaj primer monotone funkcije med dvema domenama, ki ni zvezna.

Naloga 6.24 Dokaži, da je dvig $(A_\perp, \leq)$, kjer je $\leq$ definirana s predpisom (6.2), res domena.

Naloga 6.25 Ali velja izrek 6.10 za realne funkcije? Ali je res, da je funkcija $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ zvezna natanko tedaj, ko je zvezna v vsakem argumentu posebej? Če je res, dokaži, sicer poišči protiprimer.

Naloga 6.26 Dokaži lemo 6.11.

Naloga 6.27 Dokaži izjavo 6.13.

Naloga 6.28 Za naslednje množice z relacijo preveri, ali so delno urejene in ali so domene:

1. odprti interval $(0, 1)$, urejen z relacijo $\leq$ (običajna ureditev realnih števil),

[7] Mreža je delno urejena množica, v kateri obstajajo supremumi in infimumi končnih množic.

2. zaprti interval $[0,1]$, urejen z relacijo $\leq$,
3. množica $[0,1) \cup (2,3]$, urejena z relacijo $\leq$,
4. množica $[0,1) \cup [2,3]$, urejena z relacijo $\leq$,
5. množica $[0,1] \cap \mathbb{Q}$, urejena z relacijo $\leq$.

Naloga 6.29 Zaprti interval $[0,1]$, urejen z relacijo $\leq$, je domena. Za funkcije $f,g : [0,1] \to [0,1]$ in $h,k : [0,1] \times [0,1] \to [0,1]$ preveri, ali so zvezne:

$$f(x) = \begin{cases} 0 & x < 1/2 \\ 1 & x \geq 1/2 \end{cases} \qquad g(x) = \begin{cases} 0 & x \leq 1/2 \\ 1 & x > 1/2 \end{cases}$$

$$h(x,y) = (x+y)/2 \qquad k(x,y) = |x-y|$$

Naloga 6.30 Izračunaj najmanjšo negibno točko funkcije

$$F : [\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]$$

definirane s predpisom

$$F(f)(n) = \begin{cases} \perp & \text{če je } n = \perp \\ 0 & \text{če je } n = 0 \\ (2n+1) +_{\perp,\perp} f(n-1) & \text{če je } n > 0. \end{cases}$$

Nato izračunaj še najmanjšo negibno točko funkcije G, ki ima isti tip kot F in je definirane s predpisom

$$G(f)(n) = \begin{cases} \perp & \text{če je } n = \perp \\ 0 & \text{če je } n = 0 \\ (2n+1) +_{\perp,\perp} f(n) & \text{če je } n > 0. \end{cases}$$

Naloga 6.31 Pravimo, da je domena D *retrakt* domene E, če obstajata zvezni funkciji *prerez* $s : D \to E$ in *retrakcija* $r : E \to D$, za kateri velja $r(s(x)) = x$ za vsak $x \in D$. Dokaži, da je s injektivna in r surjektivna preslikava.

Naloga 6.32 Dokaži, da je domena D rektrakt domene $D \times E$.

Naloga 6.33 Naj bodo $D_1, D_2, \ldots$ domene. Dokaži, da je tudi neskončni kartezični produkt $\prod_{i \in \mathbb{N}} D_i$ domena z ureditvijo po komponentah. Elementi kartezičnega produkta $\prod_{i \in \mathbb{N}} D_i$ so neskončne terice $(x_i)_{i \in \mathbb{N}}$, pri čemer je $x_i \in D_i$.

Naloga 6.34 Dokaži, da je domena D_i rektrakt domene $\prod_{i \in \mathbb{N}} D_i$.

Naloga 6.35 Ta naloga je težja. Ali obstaja taka domena U, da je vsaka končna domena retrakt domene U? Glej nalogo 6.31 za definicijo retrakta. Namig: obstaja taka števna družina končnih domen, da je vsaka končna domena izomorfna eni od domen iz družine.

Naloga 6.36 Ta naloga je zelo težka. Naj bo $D = [\mathbb{N}_\perp \to \mathbb{N}_\perp]$. Dokaži, da je $D \times D$ retrakt D. Nato dokaži še, da je $[D \to \mathbb{N}_\perp]$ retrakt D. Glej nalogo 6.31 za definicijo retrakta.

Literatura