Prevajalnik za `comm`

Oglejmo si implementacijo (različice) programskega jezika `comm` iz [PL Zoo](). Tako kot vsi jeziki v PL Zoo, je `comm` implementiran v programskem jeziku OCaml.

Jezik `comm` vsebuje:

- aritmetične in boolove izraze
- spremenljivke
  - deklaracija nove lokalne spremenljivke `let x := e in c`
  - nastavljanje vrednosti `x := e`
- pogojni stavek `if b then` c_1 `else` c_2 `done`
- zanka `while b do c done`
- ukaz `skip`
- sestavljani ukaz c_1 `;` c_2
- ukaz `print e`

Komentar:

```
new x := 2 + 3 in
  x := x + 1 ;
  if x < 7 then
    print x
  else
    skip
```

v Javi:

```
{ int x = 2 + 3 ;
  x = x + 1
  if (x < 7) {
    print (x);
  } else
  { }
}
```

Dogovoriti se moramo, kaj pomeni

`new x := e in` c_1 `;` c_2

Imamo dve možnosti:

1. `(new x := e in` c_1`)` `;` c_2 – `x` je veljaven samo v c_1
2. `new x := e in (`c_1 `;` c_2`)` – `x` je veljaven v c_1 in v c_2

Dogovorimo se, da velja 2. možnost.

Ogledamo si sestavne dele implementacije:

- abstraktna sintaksa je definirana s podatkovnimi tipi v `syntax.ml`

- konkretna sintaksa je opisana v `lexer.mll` in `parser.mly`; uporabimo generator parserjev Menhir
- preprost simulator procesorja z RAM in skladom najdemo v `machine.ml`
- prevajalnik iz `comm` v strojni jezik je v `compile.ml`
- glavni program je v `comm.ml`

Prevajalnik neposredno pretvori program v strojno kodo, ker je `comm` zelo preprost jezik. Prevajanje pravih programskih jezikov poteka preko večih stopenj, z vmesnimi jeziki. Vsak naslednji jezik je nekoliko bolj preprost in bližje strojni kodi.

Na primerih preizkusimo, kako se prevajajo programi.

Dokazovanje pravilnosti programov

Kako vemo, ali program deluje pravilno? Kako vemo, kakšen program želimo sestaviti?

Ločimo med **specifikacijo** in **implementacijo** programa:

- **Specifikacija** je opis, kaj naj želeni program počne.
- **Implementacija** je konkreten program, ki počne to, kar zahteva specifikacija.

Specifikacija je lahko podana bolj ali manj natančno, v človeškem jeziku ali zapisana v formalnem matematičnem jeziku. Zakaj potrebujemo specifikacije? Nekateri odgovori:

- da pridobimo opis programa, ki naj bi ga sestavili
- da lahko preverimo, ali je implementacija pravilna
- zagotovimo kompatibilnost med različnimi deli programske opreme

Danes bomo spoznali le majhen košček specifikacij, t.i. Hoarovo logiko, s katero izražamo dejstva o programih v ukaznem programskem jeziku in dokazujemo njihovo pravilnost.

Hoarova logika

V Hoarovi logiki pišemo *Hoarove trojice*

```
{ P } c { Q }
```

kjer sta `P`in `Q` logični formuli in `c` ukaz. Formuli `P` pravimo *predpogoj* (angl. *precondition*), formuli `Q` pravimo *končni pogoj* (ang. *postcondition*). V resnici poznamo dve različici trojic:

Delna pravilnost

```
{ P } c { Q }
```

pomeni

> Če velja `P` in če bo ukaz `c` končal, potem bo veljal `Q`

Popolna pravilnost

`[ P ] c [ Q ]`

pomeni

> Če velja `P`, potem se bo `c` končal in veljal bo `Q`.

Zapomnimo si: delna pravilnost ne zagotavlja, da se bo `c` končal, popolna pravilnost to zagotavlja.

Primer 1

Program `c` zamenja vrednosti spremenljivk `x` in `y`:

`{ x = m ∧ y = n } c { x = n ∧ y = m }`

Tu moramo predpostaviti, da sta `m` in `n` *duhova* (angl. ghost variable), se pravi spremenljivki, ki se ne pojavljata v `c`.

Primer 2

Program `c` poskrbi, da je `x` manjši ali enak `y`:

`{ true } c { x ≤ y }`

Ali znamo zapisati tak program? Da, na primer

`x := 0 ; y := 1`

Specifikacija to dovoli. Verjetno smo hoteli v resnici tole:

`{ x = m ∧ y = n } c { x = min(m, n) ∧ y = max(m, n) }`

Ko delamo s Hoarovo logiko, običajno pišemo pogoje in kodo navpično, da lahko med vrstice vrivamo pogoje.

```
{ x = m ∧ y = n }
c
{ x = min(m, n) ∧ y = max(m, n) }
```

Seveda potrebujemo nekakšna pravila sklepanja.

Pravila sklepanja

Za Hoarovo logiko veljajo naslednja pravila sklepanja.

Splošna pravila

Vedno smemo uporabiti veljavno logično in matematično sklepanje, na primer:

$$\frac{P' \Rightarrow P \qquad \{ P \}\ c\ \{ Q \} \qquad Q \Rightarrow Q'}{\{ P' \}\ c\ \{ Q' \}}$$

$$\frac{\{ P_1 \}\ c\ \{ Q_1 \} \qquad \{ P_2 \}\ c\ \{ Q_2 \}}{\{ P_1 \land P_2 \}\ c\ \{ Q_1 \land Q_2 \}}$$

Naj bodo `FV(P)` vse spremenljivke, ki se pojavljajo v formuli `P` (free variables) in `FA(c)` vse spremnljivke, ki jih `c` nastavlja (assigned variables). Na primer:

`FV(x ≤ y ∨ x > 0) = {x, y}`

`FA(if x < y then x := y + 3 else skip end) = { x }`

Pozor: ne sprašujemo, ali program dejansko spremeni vrednost, ampak le, ali se pojavlja v njem ukaz oblike `x := ...`:

`FA(if 5 < 3 then x := y + 3 else skip end) = { x }`

`FA(x := x) = { x }`

Velja pravilo:

$$\frac{\mathrm{FV}(P) \cap \mathrm{FA}(c) = \varnothing}{\{ P \}\ c\ \{ P \}}$$

Pravilo za `skip`

$$\frac{}{\{ P \}\ \mathtt{skip}\ \{ P \}}$$

Pravilo za pogojni stavek

$$\frac{\{ P \land b \}\ c_1\ \{ Q \} \qquad \{ P \land \neg b \}\ c_2\ \{ Q \}}{\{ P \}\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \mathtt{end}\ \{ Q \}}$$

Pravilo za c_1 ; c_2

$$\frac{\{ P \}\ c_1\ \{ Q \} \qquad \{ Q \}\ c_2\ \{ R \}}{\{ P \}\ c_1\ ;\ c_2\ \{ R \}}$$

Pravilo za zanko `while`

$$\frac{\{ P \land b \}\ c\ \{ P \}}{\{ P \}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ \{ \lnot b \land P \}}$$

Formuli `P` pravimo *invarianta* zanke `while`.

Pravilo za prirejanje

$$\frac{}{\{ P[x \mapsto e] \}\ x := e\ \{ P \}}$$

Zapis `P[x ↦ e]` pomeni "v formuli `P` zamenjaj pojavitve `x` z izrazmo `e`. Primer:

$$P \equiv x < n \land y + x = z$$

$$P[x \mapsto x+1] \quad \equiv \quad x+1 < n \land y + (x+1) = z$$

$$P[x \mapsto 0] \quad \equiv \quad 0 < n \land y + 0 = z$$

Operaciji, ki zamenja spremenljivko z nekim izrazom pravimo *substitucija*.

Popolna pravilnost

Vsa zgornja pravila, razen dveh, lahko predelamo v popolno pravilnost, na primer:

$$\frac{[\ P \land b\ ]\ c_1\ [\ Q\ ] \qquad [\ P \land \lnot b\ ]\ c_2\ [\ Q\ ]}{[\ P\ ]\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\ [\ Q\ ]}$$

Prva izjema je pravilo

$$\frac{FV(P) \cap FA(c) = \emptyset}{\{ P \}\ c\ \{ P \}}$$

Kaj je narobe z

$$\frac{FV(P) \cap FA(c) = \emptyset}{[\ P\ ]\ c\ [\ P\ ]}$$

Ne velja v naslednjem primeru:

```
[ 1 = 1 ]
while true do skip done
[ 1 = 1 ]
```

Zgornja trditev ne velja (ni res, da se program konča).

Zato pravilo predelamo takole:

$$\frac{FV(P) \cap FA(c) = \emptyset \qquad [\ R\ ]\ c\ [\ Q\ ]}{[\ R \land P\ ]\ c\ [\ Q \land P\ ]}$$

Pri zanki `while` zagotovimo, da se bo končala, tako da poiščemo količino, ki se zmanjšuje, a se ne more zmanjševati v nedogled. Na primer, to je lahko celoštevilska pozitivna vrednost.

Pozor: realna pozitivna vrednost se lahko zmanjšuje v nedogled:

`0.1 > 0.01 > 0.001 > 0.0001 > ...`

Pravilo za popolno pravilnost `while` se glasi:

Naj bo `e` količina, ki se ne more v nedogled zmanjševati (na primer naravno število):

$$\frac{[\ P \land b \land e = z\ ]\ c\ [P \land e < z\ ]}{[\ P\ ]\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}\ [\ \neg\ b \land P\ ]}$$

Stranski pogoji:

- količina `e` se ne more v nedogled zmanjševati
- `z` je duh, spremenljivka, ki se ne pojavlja nikjer drugje v `P`, `b` ali `c`.

Kako pa ta pravila v praksi uporabljamo? Poglejmo nekaj primerov.

Primeri

Primer 1

Zapiši s Hoarovo logiko:

1. Program `c` se ne ustavi.

2. Program `c` se ustavi.

Rešitev:

Poglejmo kar vse možne kombinacije:

- `{ true } c { true }` – vedno velja, ker `true` itak velja vedno

- `{ true } c { false }` — pravi "`c` se ne ustavi"

    - če velja `true` in če se bo ustavil `c`, bo veljalo `false`
    - če se bo ustavil `c`, bo veljalo `false`
    - s formulo: `c se ustavi ⇒ false`
    - spomnimo se: `P ⇒ false` je logično ekvivalentno `¬ P`
    - s formulo: `¬ (c se ustavi)` (upoštevamo zgornjo ekvivalenco)

    - z besedami: `c` se ne ustavi

- `{ false } c { true }` – vedno velja, ker iz `false` sledi karkoli
- `{ false } c { false }` – vedno velja, ker iz `false` sledi karkoli
- `[ true ] c [ true ]` – pomeni "`c` se ustavi"
    - če velja `true`, se bo `c` ustavil in veljalo bo `true`
    - `c` se bo ustavil in veljalo bo `true`
    - `c` se bo ustavil
- `[ true ] c [ false ]` – nikoli ne velja
    - če velja `true`, se bo `c` ustavil in veljalo bo `false`
    - s formulo: `true ⇒ "c se ustavi" ∧ false`
    - s formulo: `true ⇒ false`
    - s formulo: `false`
- `[ false ] c [ true ]` – vedno velja, ker iz `false` sledi karkoli
- `[ false ] c [ false ]` – vedno velja, ker iz `false` sledi karkoli

Koristni kombinaciji:

1.`{ true } c { false }` — pomeni "`c` se ne ustavi"

1. `[ true ] c [ true ]` – pomeni "`c` se ustavi"

Primer 1.5

Dokaži pravilnost programa:

```
{ true }
x := 7
{ x < 10 }
```

Zelo natančna rešitev:

```
{ true }
{ 7 < 10 }      –- P ≡ (x < 10)
{ P[x ↦ 7] }
x := 7
{ P }
{ x < 10 }
```

Praktična rešitev:

```
{ true }
x := 7
{ x = 7 }  — logično sklepanje
{ x < 10 }
```

Primer 1.75

```
{ i < n }    – naslednja vrstica je ekvivalentna tej
{ (i + 1) - 1 < n }
i := i + 1
{ i - 1 < n } – uporabili smo pravilo za prirejanje
{ i < n + 1 }
```

Primer 2 - 1.125

```
{ i < n }
i := i + j
{ i < n + j }
```

Primer 2

Dokaži pravilnost programa:

```
{ x ≤ y }
s := (x + y) / 2
{ x ≤ s ≤ y }
```

Rešitev:

```
{ x ≤ y }
s := (x + y) / 2
{ x ≤ y ∧ s = (x + y) / 2 } – logično sklepanje
{ x ≤ s ≤ y }
```

Varianta:

```
{ x < y }
s := (x + y) / 2
{ x < y ∧ s = (x + y) / 2 } – logično sklepanje lari fari
{ x < s < y }
```

NI RES! Protiprimer: `x = 4`, `y = 5`, `s = 4`.

Primer 3

Dokaži pravilnost programa:

```
[ b ≥ 0 ]
i := 0 ;
p := 1 ;
while i < b do
   p := p * a ;
   i := i + 1
done
[ p = a ^ b ]
```

Rešitev:

```
{ b ≥ 0 }
i := 0 ;
{ b ≥ 0 ∧ i = 0 }
p := 1 ;
{ b ≥ 0 ∧ i = 0 ∧ p = 1 }
{ p = a ^ i ∧ i ≤ b }
while i < b do
   { i < b ∧ p = a ^ i ∧ i ≤ b }
   { i < b ∧ p · a = a ^ (i+1) ∧ i ≤ b }
   p := p * a ;
   { i < b ∧ p = a ^ (i+1) ∧ i ≤ b }
   { i+1 < b+1 ∧ p = a ^ (i+1) ∧ i+1 ≤ b+1 }
   i := i + 1
   { i < b + 1 ∧ p = a ^ i ∧ i ≤ b+1 } — sklepamo: če i < b + 1,
   potem i ≤ b
   { p = a ^ i ∧ i ≤ b }
done
{ p = a ^ i ∧ i ≤ b ∧ ¬ (i < b) } — logično sklepamo: i ≤ b in b ≤ i
potem i = b
{ p = a ^ i ∧ b = i } — logično sklepamo
{ p = a ^ b }
```

Popolna pravilnost: potrebujemo količino `e`, ki se zmanjšuje in se ne more zmanjševati v nedogled. Predlog:

```
e ≡ b - i
```

To je celo število. Ali je nenegativno? Naša invarianta vsebuje dejstvo `i ≤ b`, od koder seveda sledi, da je `b - i ≥ 0`.

Zdaj bi moralo pazljivo dokazati, da se `e` res zmanjša. Pričakujemo, da `e` zmanjša za 1:

```
{ e = z }
while ...
  ...
done
{ e = z - 1 < z }
```