VEGOVA
ELEKTROTEHNIŠKO-RAČUNALNIŠKA
STROKOVNA ŠOLA IN GIMNAZIJA
LJUBLJANA

Razvoj prevajalnika za Bitrout, programski jezik na primitivni arhitekturi

(računalništvo in informatika)

Raziskovalna naloga

Avtor: Adrian Sebastian Šiška, G 4. B
Mentor: Aleš Volčini

Ljubljana, februar 2023

Kazalo

Slike

1 Zahvala

Zahvaljujem se vsem, ki so mi pomagali pri raziskovanju in pisanju. Še posebej se zahvaljujem mentorju Alešu Volčiniju za podporo in potrpežljivost. Zahvalil bi se tudi mojima prijateljema Oliverju Wagnerju (oliwerix.com) in Antonu L. Šijancu (šijanec.eu), ki sta s svojimi implementacijami emulatorja MUHI[**?**] popestrila moje delo.

2 Povzetek

V raziskovalni nalogi sem si zamilil nov programski jezik imenovan Bitrout. Bitrout je preprost proceduralen jezik narejen za MUHI[**?**] arhitekturo opisano v moji prejšnji raziskovalni nalogi.

Ključne besede:

3 Uvod

Zaradi unikatnih limitacij moje arhitekture MUHI, ki sem jo popisal v prejšnji raziskovalni nalogi, so prevodi ostalih programskih jezikov (npr. C) netrivialni. Za potrebe arhitekure MUHI sem pripravil svoj programski jezik, ki sem ga poimenoval Bitrout.

3.1 Hipoteze

V tej nalogi bom osredotočil na sledeče hipoteze.

1. Bitrout je Turing-kompleten jezik.
2. Prevajalnik za Bitrout prevaja v strojno kodo MUHI.
3. Struktura jezika Bitrout omogoča popolne optimizacije.
4. Sistem podatkovnih tipov jezika Bitrout omogoča implicitno navajanje tipov.

4 Jezik

Bitrout preprost proceduralen jezik. Zamislil sem si ga, ker sem potreboval majhen in preprost, vendar zmogljiv, jezik za MUHI arhitekturo. Za shranjevanje Bitrout programov uporabljam datoteke s končnico `.brt`.

4.1 Podatkovni tipi

Cilj tega jezika je postati to, kar je C postal za *x86* in kupico drugih arhitektur. Menim, da je pomembno, da se že na nivoju sistema podatkovnih tipov prilagodimo prednostim slabostim arhitekture. V mojem primeru to pomeni tip na osnovi bita, kar seveda ne zadostuje za višje nivojski jezik. (Če bi vse spremenljivke bile enobitnega tipa, bi delo z njimi bilo še težje kot v zbirniku, ki sem ga izdal prejšnje leto.) Kot naslednji najboljši kompromis sem si zamislil seznam bitov. Vsi podatki, ne glede na tip in ahitekturo se prevedejo v bite[1]. Zaradi potencialnih optimizacij, preprostosti jezika in preprostosti dodeljevalnika spomina sem se odločil, da bom izbral seznam bitov, kot moj edini tip. (`{1,0,1,1}`) Edina nejasnost je le še dolžina seznamov. Za to poskrbim, da je izpeljana iz načina uporabe[2] in zahtevam, da je znana v času prevajanja.

```
{
  var moj_float;
  int_to_float 42 moj_float;
}
```

V zgornjem primeru si lahko zamislimo, da imamo knjižnico za števila s plavajočo vejico. Smiselen zapis funkcije `int_to_float` bi lahko bil `int_to_float (int: 32 | float: 32)`. Ker vemo, da so vse ostale dolžine razen 32 nesmiselne, lahko rečemo da `my_float` mora biti dolg 32.

[1] Obstajajo redke izjeme.
[2] Le v času prevajanja.

4.2 Procedure

Bitrout se primarno ukvarja z procedurami in podajanjem podatkov med njimi. Na najvišjem nivoju programa lahko definiramo le proceduro ali dodamo še eno datoteko.

```
include "std.brt";
fn moja_procedura(){
  // vsebina
}
```

4.2.1 Vgrajeni ukazi

```
{
  a I J; // NAND ukaz med I in J
  x K L; // XOR ukaz med K in L
}
```

Ta ukaza sta izbrana zaradi možnosti dobesednega prevoda v strojni jezik. Enočrkovna sintaksa je ohranjena iz zbirnika.

Za oba ukaza velja, da sta oba argumenta vhodna, rezultat je pa shranjen[3] na drugega. Podana seznama morata biti iste dolžine, ukaza pa bosta navidezno[4] uporabila bitno verzijo operatorja.

4.3 Klicni argumenti

Ob vsakem klicu funkcije je potrebno podati vse zahtevane argumente. Argumenti imajo za : lahko specifikacijo dolžine. Te so lahko preproste numerične konstante, ali pa spremenijo fiunkcijo v „bitno“ operacijo. Bitne operacije so ekvivalentne „for each“ zanki. Omejitev je seveda, da že v času prevajanja mora biti znan tip, po katerem iteriramo.

V seznamu elementov lahko eno izmed vejic zamenjam z |, ki je meja med argumenti, ki so le vhodni in jih funkcija ne spreminja, ter argumenti, uporabljenimi za vračanje podatkov iz funkcije. Slednji lahko služijo tudi kot vhodni argumenti.

```
fn funkcija(A, B:1, C:1b, D:1r | E:2){
  // A - argument brez zahteve glede dolžine
  // B - argument dolžine 1
  // C - bitni argument (najprej bit z najmanjšo težo)
  // D - bitni argument (najprej bit z največjo težo)
  // E - izhodni argument dolžine 2
}
```

[3] Staro vrednost zamenja z novo.

[4] Arhitektura ne podpira tipov daljših od 1 bit, zato prevedemo v mnogo enobitnih ukazov

Spodaj vidimo primer bitne funkcije, ki sprejema 16-bitne vrednosti. Podamo ji lahko eno ali več 16-bitnih vrednosti, ki jih nato prevajalnik prevede kot ustrezno število klicev z eno 16-bitno vrednostjo.

```
fn natisni_črko(a:16b){
   ...}

// ...
{
  natisni_črko "abcd";
}
```

⇒

```
{
  natisni_črko "cd";
  natisni_črko "ab";
}
```

Namesto `b` lahko uporabimo tudi `r`, če želimo obdelavo argumentov v obratnem vrstem redu.

4.4 Spremenljivke

Kot v vsakem dobrem jeziku, lahko tudi v Bitrout-u ustvarimo spremenljivke. To dosežemo z rezervirano besedo `var`.

```
{
  var A;              // spremenljivka z arbitrarno dolžino in s privzeto
     vrednostjo 0
  var B = 1;          // spremenljivka z vrednostjo 1 podana kot decimalno število[5]
  var C: 14;          // spremenljivka z določeno dolžino
  var D: 12 = 12;     // spremenljivka z določeno dolžino in vrednostjo
}
```

Pred začetkom optimizacij moramo poznati dolžine vseh spremenljivk.

Iz tega razloga zgornji odlomek sam zase ni pravilen, saj dolžina `A` in `B` nista znana in ne moreta biti izpeljana. V tem primeru moramo spremenljivki A in B uporabiti še v neki funkciji, ki zahteva element določene dolžine. (npr. seštevanje z 14-bitno spremenljivko, v tem primeru dotedaj neznana dolžina spremenljivke postane znana in nespremenljiva.) Spodnja deklaracija prav tako sama zase ne zadostuje, saj dolžina E v času prevajanja ni znana.

```
{
  var E = 16;
  set E;
}
```

4.5 Konstante

Konstante so sestavljene iz spremenljivk, intervalov bitov (`B[1..2]`), numeričnih konstant, nizov (`"abc..."`), seznamov bitov (`{1,0,1}`) in stikov med njimi (`a ++ {0,1}`). Originalno so bile dodane v jezik za lažje nastavljanje spremenljivk na znano vrednost.

[5] Desetiški literali nimajo eksplicitne dolžine asociirane z njimi, zato smo v tem primeru, podobno kot v prejšnjem, prepuščeni prevajalcu, da določi dolžino. Ta gotovo ne sme biti manjša kot literal sam

```
{
  var i = "a b c";
  funkcija i;
}
```

Vendar se mi je zdela smiselna tudi deklaracija na mestu samem:

```
{
  funkcija "a b c";
}
```

4.5.1 Numerični literali

Veljavna števila so desetiška števila, ki nimajo predhodnih ničel, šestnajstiška števila (`0x10` ⇒ `16`) in binarna števila (`0b1010`).[6] Šestnajstiška in binarna števila imajo s seboj močno asociirano dolžino. Če npr. spremenljivki priredimo vrednost `0b01100` potem obenem z vrednosto določimo tudi, da je njena dolžina 5 bitov. Več o tem kasneje.

V vsa števila smemo vstaviti _ na katerokoli mesto[7] (`0b1001_1010_1100_1101`). Števila so v procesu prevajanja spremenjena v sezname bitov, kjer je prvi bit v seznamu bit z najmanjšo težo. (`(10) -> {0, 1, 0, 1}`)

4.6 Intervali bitov

Simboli lahko predstavljajo sezname bitov različnih dolžin. Za operiranje z subsetom seznama sem izbral sintakso intervalov bitov[8]. (`Simbol[1..10]`) Veljavne intervali so tudi tisti, ki imajo izpuščeno spodnjo oz. zgornjo mejo intervala[9]. To implicitno pomeni začetni oz. končni element seznama. (`Simbol[..10]` ali `Simbol[1..]`) Zgornja meja je tako kot pri `python`-u indeks elementa, ki bi sledil zadnjemu:

```
S[2:5]
// ⇒
S[2] ++ S[3] ++ S[4]
```

Sintaksa `Simbol[`n`]` je tudi podprta in se prevede v `Simbol[`$n..n+1$`]`.

4.7 Vhod in izhod

Za vhod in izhod sta rezervirana ukaza `get` (za pridobivanje podatkov) in `set` za pošiljanje podatkov. Problem na katerega hitro naletimo, je scenarij v katerem uporabnik ne poda nobenega vhoda.

V osnovni arhitekturi se zanašamo na `INPUT_AVAILABLE` in `OUTPUT_AVAILABLE` bite. Specifikacija arhitekture MUHI preprečuje branje iz vhoda in pisanje na izhod, še preden bi preverili, ali sta na voljo. To pomeni, da se lahko zgodi, da klic `get` ali `set` ukaza ne uspe.

[6] V moji implementaciji se zanašam na pitonski tolmač števil

[7] Razen pred 0x ali 0b prefiks.

[8] Ang. Slice

[9] Če izpustimo oba indeksa, je to ekvivalentno vačanju celotnega seznama, kar se mi zdi kot dober razlog za sintaktično napako.

Za preprostost najosnovnejših gradnikov jezika sem se odločil, da bo privzeto obnašanje, da čaka, dokler mu ne uspe[10]. Tako obnašanje zagotovi, da ne bo prihajalo do napak, povezanih z omenjeno omejitvijo arhitekture. To nam seveda omeji načine na katere lahko komuniciramo z zunanjim svetom.

```
{
  :loop
  var B: 8;
  get B; // ← Če na standardnem vhodu ni več podatkov, bo program tukaj večno
      stal.
  set B; // ← Če na drugi strani ni prejemnika, ki bi podatek pobral, bo program
      prav tako obtičal tukaj.
  goto loop;
}
```

Primer programa, ki tvori povratno zanko (ekvivalenten bash program `$ cat`).

4.8 Oznake in ukaz GOTO

V MUHI arhitekturi absolutni skoki trajajo razmeroma veliko časa (2 do 33 ciklov)[11]. Zaradi tega moramo paziti kje in kdaj skačemo drugam v programu. Da preprečim programe, ki bodo sestavleni le iz gradnikov naslova[12], sem poskrbel, da so sicer na voljo programerju, vendar jih mora eksplicitno vstaviti.

Vsak ukaz `goto` kaže na `:  oznako`. Ta mora biti v istem ali višjem dosegu (ang. *scope*).

4.9 Obravnavanje napak med izvajanjem programa

Izjeme so danes eden izmed najpomembnejših delov jezika. Zaradi standardnega vhoda in izhoda sem si zamislil jezikovno konvencijo za opis napak. Žal so te napake manj podobne *Rust*-ovim `std::result`[?] in bolj *C*-jevemu `errno`[?]. Tako kot C tudi moj jezik nima nobenega mehanizma, ki bi prisilil uporabnika, da obvezno preveri uspešnost klica.

Za zapis funkcije ki lahko vrne napako predlagam dodatek `_eeee`... na koncu imena, kjer je število `e` enako številu različnih napak. (`set_e`) Za vsako napako mora funkcija sprejeti še enobitni izhodni argument. (npr. `fn set_e (A| E:1){...}`) Če vrednost spremenljivke podane v izhodni element ni 0, se je zgodila napaka. Vse funkcije, ki prejmejo argument za napako, ga smejo nastaviti na 1, ne pa na 0, da se lahko napake propagirajo čez več neuspelih klicev.

```
{
  //⋮
  var E = 0;
  syscall arg E;
  set arg E;
  syscall arg E;
  // Katerkoli izmed vmesnih klicov je lahko nastavil E na 1
```

[10] To obnašanje se bo najverjetneje spremenilo v verziji 2.0.0.

[11] Trajanje je odvisno od implementacije skoka, saj niso vgrajeni v arhitekturo.

[12] Ena izmed možnosti je, da v spominu zgradiš 16 bitni naslov in ga prekopiraš na PC.

}

Pričakovano je, vendar ne obvezno, da funkcija v primeru, da je prišlo do napake, naredi čim manj.

4.10 Primeri in razlage

4.11 Omejitve

Največja omejitev jezika izhaja iz same arhitekture. Mnogo dejstev o programu (npr. globina rekurzije) moramo spoznati že v analizi programa. Ne le da to naredi vprašanje, ali je nek program prevedljiv, neizračunljivo, ampak za nameček še poskrbi, da mnogo postopkov ne moremo na najbolj očiten način napisati, kajti obstaja razlika v obnašanju med konstantami in vrednostmi, znanimi v izvajalnem okolju.

5 Prevajalnik

Za implementacijo prevajalnika sem vzel jezik *python*, vendar v retrospektivi to morda ni bila najboljša ideja. Prejšnje leto sem se v veliki meri zanašal na *python*-ov `eval()` za interpretacijo konstant. Implementiral sem celo pitonski blok, kar pomeni, da je interpretacija nezane zbirne kode lahko nevarna. To leto se nisem zanašan na *python*-ovo dinamično evaluacijo, sem imel obilo problemov z vnaprej nepoznanimi tipi podatkov (ang. *duck typing*).

5.1 Leksikalna analiza

Za interpretacijo jezika je standardna praksa, da najprej zlepimo posamezne znake v smiselne elemente v jeziku (`'i', 'f'` ⇒ `"if"`). Ta proces se imenuje leksikalna analiza. Navdih za zgradbo lekserja sem vzel iz knjige *Crafting interpreters*[**?**].

5.1.1 Individualni znaki

Jezik uporablja naslednje individualne znake:

- `( )` zapis argumentov pri definiciji funkcije,
- `{ }` zapis literala seznama in dosega,
- `[ ]` zapis rezine simbola,
- `:` navajanje dolžine,
- `;` zaključek stavka,
- `=` nastavljanje spremenljivk na privzeto vrednost,
- `|` mejo med vhodnimi in vhodno-izhodnimi argumenti,
- `,` mejo med argumeni v definiciji funkcije,
- `-` negiranje števil v rezinah.[13]

5.1.2 Večsimbolni operatorji

Jezik uporablja dva večsimbolna operatorja:

[13]Ne dovolim rabe tega znaka pred konstantami v programu, ker nočem zahtevati določenega zapisa negativnih števil.

`..` zapis območja v rezinah,
`++` spajanje vrednosti.

5.1.3 Literali

`Identifikator` spremenljivke in funkcije;
Sestavljen mora biti iz katerega koli veljavnega utf-8 niza, ki ne vsebuje sledečih znakov `() "'{}[]:;,.=+-/*|` ter ascii znakov LF in CR.

`Niz` zapis literalov v obliki zaporedja znakov (ang. string);
Ti se začnejo in končajo z `"`.

`Celo_število` zapis števil v različnih bazah;
števila se morajo začeti s števko, lahko vsebujejo prefix, ki določi številski sistem (`0` in (`'x'` ali `'b'`)) in lahko na kateremkoli mestu (razen prvem ali zadnjem) vsebujejo `_` za lažje branje števila.

5.1.4 Ključne besede

Vnaprej rezervirane ključne besed v jeziku so:

`fn` začetek definicije funkcije,
`if` začetek if stavka,
`var` definicijo nove spremenljivke,
`else` element if stavka,
`goto` ukaz goto,
`include` vključevanje druge datoteke.

5.2 Sintaktična analiza

Za sintaktično analizo ne izvajam ničesar posebnega. Pohlepno požiram simbole glede na različne kontekste, razen na določenih mestih, kjer imam do 1 znak vnaprejšnjega vpogleda. Na koncu dobim abstraktno sintaktično drevo.

Ker vse `include` stavke takoj dodam na seznam datotek potrebnih analize, je „celoten“ program le seznam definicij funkcij. Problem s takim pristopom je, da bi lahko izpustil sintaktično in semantično analizo za funkcije, ki jih nikoli ne pokličem.

```
———exampleRule110.brt———
(fn print (A:1B)
if A set "@" else set " "
)

(fn iter (*c)
var o = {1} ++ c ++ {1}
rule110 c ++ {1, 1} {1, 1} ++ c o
is o[1..-1] c
)

(fn main ()
: loop
var C: 16 = 0(_)
print C
set "\n"
iter C
goto loop
)
```

Slika 1: Pisna reprezentacija Abstraktnega sintaktičnega drevesa.

5.3 Razločevanje spremenljivk

Razločevanje spremenljivk se zgodi v dveh korakih.

V prvem se se sprehodimo[14] čez vse omenjene dosege iz katerih zberem čim več namigov o dolžini.

V drugem program poženemo v „emulatorju“, v procesu zapišemo kaj se zgodi vsaki spremenljivki. Spremenljivke, ki so kadarkoli natisnjene, tvorijo „veje“.

5.4 Sistem namigov

Vsaka vrstica, ki uporabi spremenljivko, nam lahko namigne kaj o njeni dolžini. Iz tega sem sestavil hierarhijo namigov. Vsak naslednji doda neko informacijo prejšnjemu.

5.4.1 LenHintExact

`LenHintExact` je najpreprostejši „namig“, saj nam navede gotovo dolžino

```
fn primer(A:42){...}
fn main(){
  var B;
  primer B; // ← B je gotovo dolžine 42, saj so vse ostale dolžine nesmiselne
}
```

5.4.2 LenHintMin

`LenHintMin(0)` je privzeta vrednost vseh spremenljivk, ko nastanejo, saj vemo da gotovo ne zavzamejo negativnega prostora. Namig tega tipa nastane tudi ob navajanju

[14] Ta sprehod se lahko zgodi večkrat in upošteva tudi podatke v nedosegljivih vejah.

desetiškega literala, saj je znano koliko prostora najmanj zavzame.

```
fn main(){
  var B = 65; // ← B je gotovo dolg vsaj 7 bitov, vendar je lahko več.
}
```

5.4.3 LenHintDivisible

`LenHintDivisible` nastane, ko izvemo da je neka dolžina deljiva s številom. najpreprostejši primer tega je bitna operacija ki upošteva večje sklope. ker je `LenHintDivisible` nižje na hierarhiji kot `LenHintMin`, mora vsebovati vse prej podane informacije. v tem primeru to pomeni, da tudi omeji število z nižjo mejo.

```
fn primer(a:4b){...}
fn main(){
  var b;
  primer b; // ← dolžina b je gotovo večkratnik 4.
}
```

5.4.4 LenHintRelation

`LenHintRelation` je najzapletenejši namig, ker se edini ukvarja s spremenljivkami samimi. Opisuje razmerje med dolžinami (`(a:4b, b:8b)` $\Rightarrow a : b = 4 : 8$). Nastane ob bitni funkciji, ki sprejme več bitnih argumentov.

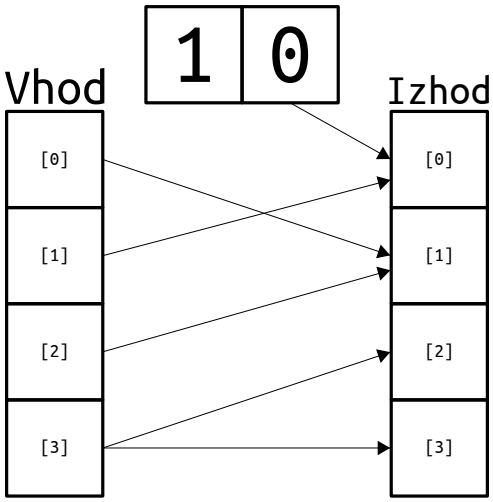
```
fn primer(a:4b, b:2b){...}
fn main(){
  var b; // ← Dolžina b je gotovo večkratnik 4.
  var c; // ← Dolžina b je gotovo večkratnik 2.
  primer b c; // ← Nastane relacija, ki obljublja, da bo za vsaka 2 bita c-ja b
      imel 4 bite in obratno.
}
```

Najpreprostejša metoda za razrešitev sistemov relacij je, da izvemo natančno dolžino ene izmed spremenljivk. Nato zračunamo število ponovitev in iz tega izpeljemo natančne dolžine vseh ostalih.

5.5 Emulacija

Cilj tega koraka je, da za vsak `set` ukaz zgradimo seznam odvisnosti. To dosežem s tem, da pri emulaciji beležim vire posameznih podatkov. Edini viri podatkov v jeziku so standardni vhod, konstante in začetno stanje pomnilnika[15]. Konstante so v tem koraku že znane, zato ne predstavljajo problemov. Če predvidevamo, da je začetno stanje vedno nič, potem z njim obravnavamo kot s konstanto, sicer pa vedno nastavimo spomin na nič pred uporabo. Standardni vhod mi povzroča največ problemov, saj se s vsakim bitom število začetnih mest podvoji. Problem eksponentne rasti rahlo oblažim, s pomočjo bitne notacije binarnih funkcij. Več o tem kasneje. Na koncu dobim seznam vseh `set` ukazov in vrednosti, ki jih potrebujejo.

[15]Zaenkrat se zanašam, da so vsa mesta v pomnilniku 0. Palnirano je, da bo verzija 2.0.0 dodala varen način, ki ne bo delal te predpostavke.



Slika 2: Visualni prikaz seznama odvsnosti

5.6 Dodeljevalnik spomina

Za dodelitev spomina se sprehodimo po sezamu `set` ukazov in vsakemu bitu izpeljanemu iz `get` ukaza določimo življensko dobo. Ta pristop najbolje deluje za programe, ki se obnašajo kot filtri. Pomembno je, da preverimo, da je vedno količina rezerviranega spomina vedno manjša od tega, ki ga imamo na voljo.

Zanimiva posledica arhitekture je, rešitev problema razdroblenosti. Ker so vsi tipi poravnani na dolžino bita in ker se nikoli ne zanašam na lokacije bitov v spominum, jih lahko postavim kamorkoli.

6 Optimizacije

6.1 Bitna notacija

Že lansko leto sem opazil, da lahko seznam logičnih operacij opišem z celim številom. Tak zapis sem imenoval bitni zapis. Uporaben, kot način poenostavljanja binarnih funkcij.

Za prvi primer vzamimo enobitno spremenljivko `A`. Njena identiteta v tem sistemu je $\texttt{10}_2$. Kasneje več o načinih generiranja identitet, saj je za ta primer pomembno le, da opazimo, da že iz zapisa vidimo vse možnosti `A` spremenljivke. Če vzamemo eniški komplement `A` dobimo vrednost $\texttt{01}_2$. Intuitivno opazimo, da tam kjer je bila začetna vrednost 1 je sedaj 0 in obratno. Če še enkrat negiramo vrednost, kot pričakovano, dobimo originalno število, saj vemo, da $\neg\neg A = A$ drži.

Ugotovil sem, da se v tem sistemu ne da predstaviti iste funkcije na dva različna načina. Za indentifikacijo podvojenih funkcij je dovolj le prevod v ta sistem.

Bitna notacija svoje lastnosti obdrži tudi, če uporabimo več simbolov in izvajamo operacije med njimi. Pri rabi večih simbolih pa naletimo na največji problem take notacije. Za vsak nov simbol se podvojijo dolžine identitet.

Za drugi primer vzamimo dve enobitni spremenljivki: `A` z identiteto $\texttt{1010}_2$ in `B` z identiteto $\texttt{1100}_2$ Če predpostavimo, da so lastnosti sistema, ki sem jih dosedaj opisal resnične, lahko z bitnimi operacijami dokažemo De Morganova pravila.

$\neg (A \wedge B) = (\neg A \vee \neg B)$ / Spremenljivke menjamo z njihovimi identitetami.

$\neg (1010 \wedge 1100) = (\neg 1010 \vee \neg 1100) \Rightarrow$

$\neg (1000) = (0101 \vee 0011) \Rightarrow$

$0111 = 0111$

Vzorec, ki ga jaz uporabljam za generacijo je da za vsak n-ti simbol uporabim 2^n zaporednih enk in ničel v njegovem zapisu. Opazil sem, da bi tudi gray kode delovale za ta namen, vendar nisem še opazil nobenega praktičnega razloga za njihovo uporabo.

6.2 Redukcija vej

Za to optimizacijo se zanašam na vpogledno tabelo. To tabelo naredi program, ki sem ga izdal že lansko leto. Za rabo vpogledne tabele je potrebno le pretvoriti funkcijo v bitno notacijo.

Omejitev tega pristopa je možna podvojitev dela in podatkov med vegami. Tega problema lansko leto nisem imel, ker sem tako analizo izvajal za več registrov hkrati. Ta pristop je hitro nehal delovati, saj sem izračunal, da za shrambo veh programov, ki vzamejo 4 bite in vrnejo 4 bite, potrebujem 2 eksabajta rama.

7 Zaključek

Naslednjič bi raje uporabil jezik z statičnimi tipi. Namesto *python*-a bi raje izbral kakšen moderen tipiziran jezik npr. Rust. Sicer pa sem mnenja, da je ta projekt velik korak v razvoju MUHI ekosistema.

7.1 Hipoteze

1. Bitrout je Turing-kompleten jezik.
2. Prevajalnik za Bitrout prevaja v strojno kodo MUHI.
3. Struktura jezika Bitrout omogoča popolne optimizacije.
4. Sistem podatkovnih tipov jezika Bitrout omogoča implicitno navajanje tipov.

Lahko potrdim 1. hipotezo, saj je v datoteki `exampleRule110.brt` napisan program, ki izrisuje celični avtomat s pravilom 110. Lahko potrdim tudi 2. hipotezo, saj 2 ločena MUHI emulatorja uspešno izvajata prevedene programe. 3. hipoteza je uvržena, ker imajo za moj pristop računalniki še zmeraj premalo rama. 4. Hipoteza je potrjena, saj s pomočjo sistema namigov dolžine lahko v vseh definiranih primerih izpeljem dolžino tipa.

8 Priloge

Vse priloge in izvorna koda so dosegljive na https://ass.si/git/adrian/raziskovalna2 in so licencirane z odprtokodno licenco. Program za generiranje vpoglednih tabel je dostopen na https://github.com/adimineman/gate-bf