1 Semantics

1.1 MiniImp

The semantic of the MiniImp language is implemented in the Semantics.mli and Semantics.ml file. A `reduce` function is provided that transforms an AST into the evaluated value or an error. The AST type is defined in Types.mli and in Types.ml.

A program `p` is defined as follows:

$\langle p \rangle$:= '`def main with input`' $\langle x \rangle$ '`output`' $\langle y \rangle$ as $\langle c \rangle$

$\langle c \rangle$:= skip
| $\langle x \rangle$ '`:=`' $\langle a \rangle$
| $\langle c \rangle$ '`;`' $\langle c \rangle$
| '`if`' $\langle b \rangle$ '`then`' $\langle c \rangle$ '`else`' $\langle c \rangle$
| '`while`' $\langle b \rangle$ '`do`' $\langle c \rangle$
| '`for`' '`(`' $\langle c \rangle$ '`,`' $\langle b \rangle$ '`,`' $\langle c \rangle$ '`)`' '`do`' $\langle c \rangle$

$\langle b \rangle$:= $\langle v \rangle$ | $\langle b \rangle$ '`&&`' $\langle b \rangle$ | $\langle b \rangle$ '`||`' $\langle b \rangle$ | '`not`' $\langle b \rangle$
| $\langle a \rangle$ '`<`' $\langle a \rangle$ | $\langle a \rangle$ '`<=`' $\langle a \rangle$ | $\langle a \rangle$ '`>`' $\langle a \rangle$ | $\langle a \rangle$ '`>=`' $\langle a \rangle$
| $\langle a \rangle$ '`==`' $\langle a \rangle$

$\langle a \rangle$:= $\langle x \rangle$ | $\langle n \rangle$ | $\langle a \rangle$ '`+`' $\langle a \rangle$ | $\langle a \rangle$ '`-`' $\langle a \rangle$ | $\langle a \rangle$ '`*`' $\langle a \rangle$ | $\langle a \rangle$ '`/`' a
| $\langle a \rangle$ '`%`' $\langle a \rangle$ | $\langle a \rangle$ '`^`' $\langle a \rangle$ | '`powmod`' '`(`' $\langle a \rangle$ '`,`' $\langle a \rangle$ '`,`' $\langle a \rangle$ '`)`' | '`rand`' '`(`' $\langle a \rangle$ '`)`'

Where `%` is the modulo operator and the powmod operator is equivalent to `a ^ a % a`; the variables are all integers, `n` is an integer and `v` is a boolean literal.

The additional arithmetic expressions' semantics are implemented in a similar manner as with the other.

The sematic of `for` is as follows:

$$\texttt{for}\frac{\langle \sigma, c_1 \rangle \to_c \sigma_1 \quad \langle \sigma_1, \texttt{while } b \texttt{ do } c_3\texttt{; } c_2 \rangle \to_c \sigma_2}{\langle \sigma, \texttt{for(}c_1\texttt{, } b\texttt{, } c_2\texttt{) do } c_3 \rangle \to_c \sigma_2}$$

but the implementation exploits the structure and doesn't simply rewrite the for loop as a while loop.

1.2 MiniFun Semantics

The semantic of the MiniFun language is implemented in the Semantics.mli and Semantics.ml file. A `reduce` function is provided that transforms the AST into the avluated value or an error. The AST type is defined in Types.mli and in Types.ml.

A program `t` is defined as follows:

$\langle t \rangle$:= $\langle n \rangle$ | $\langle v \rangle$ | $\langle x \rangle$ | '`(`' $\langle t \rangle$ '`,`' $\langle t \rangle$ '`)`'
| '`fun`' $\langle x \rangle$ '`:`' $\langle type \rangle$ '`=>`' $\langle t \rangle$ | $\langle t \rangle$ $\langle t \rangle$
| $\langle op_1 \rangle$ $\langle t \rangle$ | $\langle t \rangle$ $\langle op_2 \rangle$ $\langle t \rangle$
| '`powmod`' '`(`' $\langle t \rangle$ '`,`' $\langle t \rangle$ '`,`' $\langle t \rangle$ '`)`'
| '`rand`' '`(`' $\langle t \rangle$ '`)`' |
| '`if`' $\langle t \rangle$ '`then`' $\langle t \rangle$ '`else`' $\langle t \rangle$
| '`let`' $\langle x \rangle$ '`=`' $\langle t \rangle$ '`in`' $\langle t \rangle$
| '`let`' '`rec`' $\langle x \rangle$ $\langle y \rangle$ '`: `' $\langle type \rangle$ '`=`' $\langle t \rangle$ '`in`' $\langle t \rangle$

$\langle op_1 \rangle$:= ‘not’ | ‘fst’ | ‘scn’

$\langle op_2 \rangle$:= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘^’ | ‘&&’ | ‘||’ | ‘==’
| ‘<’ | ‘<=’ | ‘>’ | ‘>=’

As reflected in the grammar, tuples have been implemented and the unary functions fst and scn return respectively the first element of the tuple and the second.

2 Types for MiniFun

A type τ is defined as either *int*, *bool*, a touple or a function.

$$\tau := int \mid bool \mid (\tau, \tau) \mid \tau \to \tau$$

The deduction rules regarding tuples are similar to those for functions:

$$\texttt{Tuple}\frac{\Gamma \vdash t_1 \triangleright \tau_1 \qquad \Gamma \vdash t_2 \triangleright \tau_2}{\Gamma \vdash (t_1, t_2) \triangleright \tau_1 * \tau_2}$$

$$\texttt{Fst}\frac{\Gamma \vdash t_1 \triangleright \tau_1}{\Gamma \vdash \texttt{fst}(t_1, t_2) \triangleright \tau_1}$$

$$\texttt{Snd}\frac{\Gamma \vdash t_2 \triangleright \tau_2}{\Gamma \vdash \texttt{snd}(t_1, t_2) \triangleright \tau_2}$$

The rules for function declaration with type annotations are thus:

$$\texttt{Fun}\frac{\Gamma[x \mapsto \tau] \vdash t \triangleright \tau'}{\Gamma \vdash \texttt{fun}x\texttt{:}\tau \to \tau'\texttt{=>}t \triangleright \tau \to \tau'}$$

$$\texttt{FunRec}\frac{\Gamma[f \mapsto \tau \to \tau'; x \mapsto \tau] \vdash t_1 \triangleright \tau' \qquad \Gamma[f \mapsto \tau \to \tau'] \vdash t_2 \triangleright \tau''}{\Gamma \vdash \texttt{let rec}f\texttt{:}\tau \to \tau' \texttt{ = } t_1 \texttt{ in } t_2 \triangleright \tau''}$$

In the files TypeChecker.mli and TypeChecker.ml there is the implementation of the deduction rules, but returns either the valid type of the expression or an error instead of simply the required option type of the valid type.

3 Parsing

3.1 MiniImp

Operators listed in order of precedence from highest to lowest:

Operator	Associativity
while	left
^	right
/ mod	left
not	-
+ - || &&	left
if	left
;	left

The expressions c_1`;`c_2 and c_1`;` are both recognized and give respectively `SEQUENCE(`c_1`,`c_2`)` and c_1, such that semicolons can be placed always at the end of a command.

Integers with a preceding minus sign can be interpreted as the opposite integer, with obviously lower precedence than the binary operator minus.

3.2 MiniFun

A decision was made to interpret \, `lambda` and `fun` all as the start of the definition of a function just for ease of typing. They are associated to the same token `LAMBDA`.

Operators listed in order of precedence from highest to lowest:

Operator	Associativity
function application	right
let	left
fun	left
fst snd	left
not rand	-
^	right
/ mod	left
+ −	left
== < ≤ > ≥	left
|| &&	left
powmod	left
λ if let letrec	left

Tuples require parentesis in their definition, but the tuple type does not since there is no ambiguity. The symbol `->` that defines the function type is right associative and has lowest precedence.

3.3 Interpreters

Both MiniImp and MiniFun have each an interpreter (miniFunInterpreter.ml and miniFunInterpreter.ml) that uses the package Clap to parse command line arguments and generate help pages.

The input to the program can be supplied both in stdin or as a command parameter after `-v`. The MiniFun interpreter also check the types before computing the output of the program and returns an error in case the types mismatch.

4 Control Flow Graph

The control flow graph data structure is implemented in the analysis library in the files Cfg.ml and Cfg.mli.

Each node contains only an id to distinguish from others.

The control flow structure is composed of a flag to know if it is empty or contains nodes and the set of all contained nodes. Since each node can only have at maximum 2 nodes as next nodes, the data structure contains a map from each node to a tuple of the two nodes or to a node. The structure also contains the back edges of each node implemented as a map from

each node to a list of nodes, the input value, the variables that are the input and ouput, the initial node and the terminal node. Finally there is a map from each node to a list of generic elements that in our case are simple statements.

4.1 MiniImp Simple Statement

MiniImp Simple Statements t is defined as follows:

$\langle t \rangle$:= skip | $\langle x \rangle$ '`:=`' $\langle a \rangle$ | $\langle b \rangle$ '`{?}`'

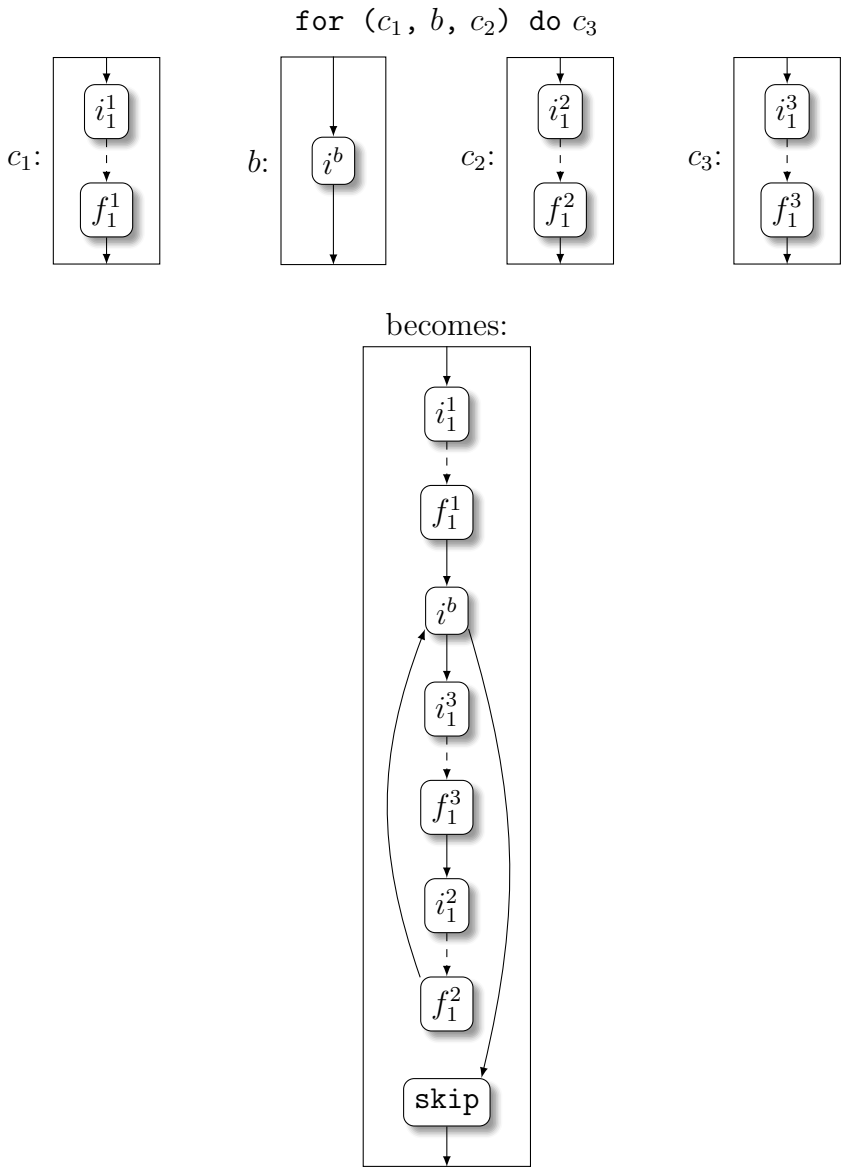
$\langle b \rangle$:= $\langle v \rangle$ | $\langle b \rangle$ '`&&`' $\langle b \rangle$ | $\langle b \rangle$ '`||`' $\langle b \rangle$ | '`not`' $\langle b \rangle$
| $\langle a \rangle$ '`==`' $\langle a \rangle$ | $\langle a \rangle$ '`<`' $\langle a \rangle$ | $\langle a \rangle$ '`<=`' $\langle a \rangle$ | $\langle a \rangle$ '`>`' $\langle a \rangle$ | $\langle a \rangle$ '`>=`' $\langle a \rangle$

$\langle a \rangle$:= $\langle n \rangle$ | $\langle x \rangle$ | $\langle a \rangle$ '`+`' $\langle a \rangle$ | $\langle a \rangle$ '`-`' $\langle a \rangle$ | $\langle a \rangle$ '`*`' $\langle a \rangle$ | $\langle a \rangle$ '`/`' $\langle a \rangle$
| $\langle a \rangle$ '`mod`' $\langle a \rangle$ | $\langle a \rangle$ '`^`' $\langle a \rangle$ | '`rand`' $\langle a \rangle$

The implemented cfg is neither minimal nor maximal, but can be either or both for some programs. In particular each node as associated a list of statements and sequence of statements in the AST is put, if possible, in the same node.

`?` is only allowed as the last element of the list of statemets associated with a node and a node has associated a `?` if and only if they have two next nodes.

The for loop is translated as:



We highlight the fact that the operation powermod is absent in the grammar of simple statements. In fact all powermod are replaced in the AST before translating into CFG with the function `rewrite_instructions` in replacePowerMod.ml and replacePowerMod.mli.

`powmod`(a_1, a_2, a_3) is translated into:

```
1  pow := a1;
2  exp := a2;
3  mod := a3;
4  res := 1;
5  if exp < 0 then
6      exp := 0 - exp;
7  else
8      skip;
9  while exp > 0 do (
10     if 1 = exp % 2 then
11         res := (res * pow) % mod;
12     else
13         skip;
14
15     pow := (pow * pow) % mod;
16     exp := exp / 2;
17 )
```

The variables `pow`, `exp`, `mod` and `res` are all fresh and the value of res is then substituted into powmod place. This might need some more scope than only the expression since `powmod` may be included in a `if` guard, thus it is placed before the `if`; in case it is in the guard of a `while` or a `for` loop it is also updated at the end of the body.

The reason for substituting `powmod` in the AST is that we would need to add nodes to form the `if` and `while` and it would prove more difficult.

5 Intermediate Code Generation

5.1 MiniRISC CFG

In the files CfgRISC.ml and CfgRISC.mli the CFG generated from the AST gets translated into intermediate code with the following MiniRISC simple statements:

$\langle t \rangle$:= Nop
| BRegOp $\langle brop \rangle$ $\langle r \rangle$ $\langle r \rangle \Rightarrow \langle r \rangle$
| BImmOp $\langle biop \rangle$ $\langle r \rangle$ $\langle n \rangle \Rightarrow \langle r \rangle$
| URegOp $\langle urop \rangle$ $\langle r \rangle \Rightarrow \langle r \rangle$
| Load $\langle r \rangle \Rightarrow \langle r \rangle$
| LoadI $\langle n \rangle \Rightarrow \langle r \rangle$
| Store $\langle r \rangle \Rightarrow \langle r \rangle$

$\langle brop \rangle$:= Add | Sub | Mult | Div | Mod | Pow | And | Or
| Eq | Less | LessEq | More | MoreEq

$\langle biop \rangle$:= AddI | SubI | MultI | DivI | ModI | PowI | AndI | OrI
| EqI | LessI | LessEqI | MoreI | MoreEqI

$\langle urop \rangle$:= Not | Copy | Rand

Since we stride towards shorter code and less instructions, we would prefer to use the `biop` version of each operation whenever possible. So for some operations that are commutative if the first term is the immediate value we swap the terms and use the `biop` variant instead of loading the value into a register and using the register for the calculation. Also some operations like `>` and `<` are opposite, so to invert the order we need to use the other `biop` version. The input variable and the output variable are also mapped to `in` and `out` registers, while all other variables are given fresh registers.

5.2 MiniRISC

The MiniRISC CFG is finally tranlated into MiniRISC intermediate code by the function `convert` in the files RISC.ml and RISC.mli. The grammar of MiniRISC is analogous to the one for MiniRISC Simple Statements:

$\langle t \rangle$:= Nop
| BRegOp $\langle brop \rangle$ $\langle r \rangle$ $\langle r \rangle \Rightarrow \langle r \rangle$
| BImmOp $\langle biop \rangle$ $\langle r \rangle$ $\langle n \rangle \Rightarrow \langle r \rangle$
| URegOp $\langle urop \rangle$ $\langle r \rangle \Rightarrow \langle r \rangle$
| Load $\langle r \rangle \Rightarrow \langle r \rangle$
| LoadI $\langle n \rangle \Rightarrow \langle r \rangle$
| Store $\langle r \rangle \Rightarrow \langle r \rangle$

| Jump ⟨*l*⟩
| CJump ⟨*r*⟩ ⟨*l*⟩ ⟨*l*⟩
| Label ⟨*l*⟩

⟨*brop*⟩ := Add | Sub | Mult | Div | Mod | Pow | And | Or
| Eq | Less | LessEq | More | MoreEq

⟨*biop*⟩ := AddI | SubI | MultI | DivI | ModI | PowI | AndI | OrI
| EqI | LessI | LessEqI | MoreI | MoreEqI

⟨*urop*⟩ := Not | Copy | Rand

where `l` is a string that uniquely identifies a label.

5.3 RISC Semantics

It is also implemented in the files RISCSemantics.ml and RISCSemantics.mli a reduce function, that evaluates MiniRISC code. The labels are used as is and not replaced by offsets, so the code is translated into a map from labels to code blocks for ease of computation.

6 Dataflow Analysis

A refined CFG structure used for analysis is defined in Dataflow.ml and Dataflow.mli. The CFG is supplemented with a map from each node to the support structure that stores the list of defined variables or live variables. Since the CFG is not minimal, there is also a list for each simple statement. A fixed point function then applies the input fuction until the map does not change. Simple structural equality is not appropriate since order in the lists should not matter; an internal function for equality is used.

6.1 Defined Variables

In the files definedVariables.ml and definedVariables.mli three functions are defined: `compute_defined_variables`, `compute_cfg` and `check_undefined_variables`.

`compute_defined_variables` creates the appropriate structure for the analysis and runs it. It returns the whole analysis structure. `compute_cfg` returns the CFG from the analysis data structure; in the case of defined variables analysis the CFG returned is the same as the one in input of `compute_defined_variables`. `check_undefined_variables` returns all variables that might be undefined at time of use.

Since the greatest fixed point is computed, first all variables are retrived from all code, then assigned to each input and ouput list of variables for each line of code.

Since it is an approximation some behaviour might not be intuitive. For example:

```
1 for (x := 0, x < 10, x := x + 1) do (
2     y := rand(x);
3 );
4 output := y;
```

will return the register associated with `y` as undefined since the guard of the for cycle might never be true.

6.2 Live Variables

In the files liveVariables.ml and liveVariables.mli three functions are defined: `compute_live_variables`, `compute_cfg` and `optimize_cfg`.

`compute_live_variables` creates the appropriate structure for the analysis and runs it. It returns the whole analysis structure. `compute_cfg` returns the CFG from the analysis data structure. `optimize_cfg` applies liveness analysis to reduce the number of registers used; returns the analysis structure (not the RISC CFG).

7 Target Code Generation

In the files reduceRegisters.ml and reduceRegisters.mli the function `reduceregisters` reduces the number of used registers by counting the syntactic occurrence of each variable and partitioning the set keeping the most used as registers. All registers are either renamed or put into memory. It is allowed for the input or output registers to be put in memory, in the latter case some code is added at the end of the program to retrive the value and put into a register (register `2`).

7.1 MiniImp to MiniRISC compiler

The file miniImpInterpreterReg.ml compiles from MiniImp to MiniRISC or execute the MiniRISC code. It uses the package Clap to parse command line arguments and generate help pages.

The input to the program can be supplied both in stdin or as a command parameter after `-v`. The flags for disabling the check for undefined variables or liveness analysis optimization are `-u` and `-l` respectively.

8 Running the code

The project uses the following packages: Dune, Menhir and Clap. They can be installed via Opam with the command `opam install dune menhir clap`. To compile the project simply run `dune build`. To run the test run `dune runtest`. In order to execute one of the interpreters run `dune exec <interpreter> -- <flags and options>`. To see a list of all options run `dune exec <interpreter> -- -h`. A binary version of the executables can also be found in ./_build/default/bin/.