Formal Languages project
Levels 1,2,3+

VILLANI Neven

1 Structure

— `prelude.h` : global typedefs and macros
— `lex.l`, `lang.y` : parsing, dispatch function calls from `main`, free up ressources
— `ast.h`, `ast.c` : structure of the parse result and builders to use inside the parser
— `argparse.h`, `argparse.c` : argument parsing
— `repr.h`, `repr.c` : internal representation of the program for better execution (see **Level 2**)
— `memreg.h`, `memreg.c` : generic facility to handle freeing ressources (see **Notes**)
— `printer.h`, `printer.c` : pretty-printers for all levels
— `hashset.h`, `hashset.c` : custom set and worklist implementation for **Level 3**

2 Usage

Flags are available to control the execution of the program.

Level 1 : `--ast` (a.k.a. `-a`) will print the syntax tree in a readable manner

Level 2 : `--rand` (`-R`) will randomly execute 100 steps on 100 instances of the program

Level 3 : `--all` (`-A`) will exhaustively explore all configurations

Levels 2&3 :
`--repr` (`-r`) will print the internal representation as text,
`--dot` (`-d`) will dump and render a `.dot` file for `graphviz`
`--trace` (`-t`) will print for each reachable configuration a sequence of steps that leads to it being satisfied

Misc :
`--help` (`-h`) will print a help message and exit,
`--no-color` (`-c`) will turn of ANSI color code formatting for all pretty-prints

Examples :

```
$ make
$ ./lang assets/sort.prog -ar --no-color
#     prints the ast and the representation of sort, without special
   characters
$ ./lang assets/sort.prog --dot
#     renders assets/sort.prog.png that represents the program
$ ./lang assets/sort.prog --all --rand -t
#     executes sort in both Monte-Carlo and exhaustive modes,
#     prints paths towards satisfied conditions
```

3 Level 1

Although a later email said to print a program that is semantically equivalent, I followed the original requirement to print "a representation of the structure".

The output is *not* a valid program.
This is also in order to make it easier to link the statements to their corresponding internal representation by

means of their unique identifier which is printed in the AST.

```
$ ./lang assets/lock.prog --ast  -c
================ AST ==================
||  GLOBAL [lock]
||  GLOBAL [fin]
||  PROC {p} {
||    LOCAL [todo]
||    <0> SET [lock] <- (2)
||    <1> SET [todo] <- (1)
||    <9> LOOP {
||      WHEN (Eq (todo) (0))
||        <2> BREAK
||      ELSE
||        <3> SET [todo] <- (0)
||        <4> SET [lock] <- (Add (lock) (1))
||        <8> CHOICE {
||          WHEN (1)
||            <5> SKIP
||          WHEN (1)
||            <6> SET [lock] <- (Sub (lock) (1))
||            <7> SET [todo] <- (1)
||        }
||    }
||    <10> SET [lock] <- (Sub (lock) (1))
||    <11> SET [fin] <- (1)
||  }
||  REACH? (Eq (lock) (1))
||  REACH? (Eq (lock) (2))
||  REACH? (Eq (lock) (3))
||  REACH? (Eq (lock) (4))
||  REACH? (And (fin) (Not (Eq (lock) (2))))
=======================================
```

Notice that the use of S-expressions makes the parsing of operators unambiguous, and that each statement has a `<n>` number that uniquely identifies it and that links it to the internal representation.

4 Level 2

The chosen representation is as a graph, with $u \to v$ if and only if there exists an environment in which v can be reached from u in one execution step (i.e. one assignment or one branching statement).

Each statement is represented internally as

```
typedef struct RStep {
    bool advance;
    RAssign* assign;
    unsigned nbguarded;
    struct RGuard* guarded;
    struct RStep* unguarded;
    uint id;
} RStep;
```

Where :
- `advance` is used to identify loops
- `assign` represents an assignment
- `guarded` indicates guarded clauses in the event of an `if` or `do`
- `unguarded` is either an `else` for an `if` or `do` clause, or the next step to execute for a normal statement.

The algorithm for calculating one step of the execution is as follows :

1. if `assign` is not `NULL`, perform assignment
2. if `nbguarded == 0` then jump to `unguarded` and terminate
3. else if one of `guarded` is satisfied, execute it
4. otherwise `unguarded` is an `else` branch, execute it

If at any point `NULL` is reached it means the procedure has terminated.

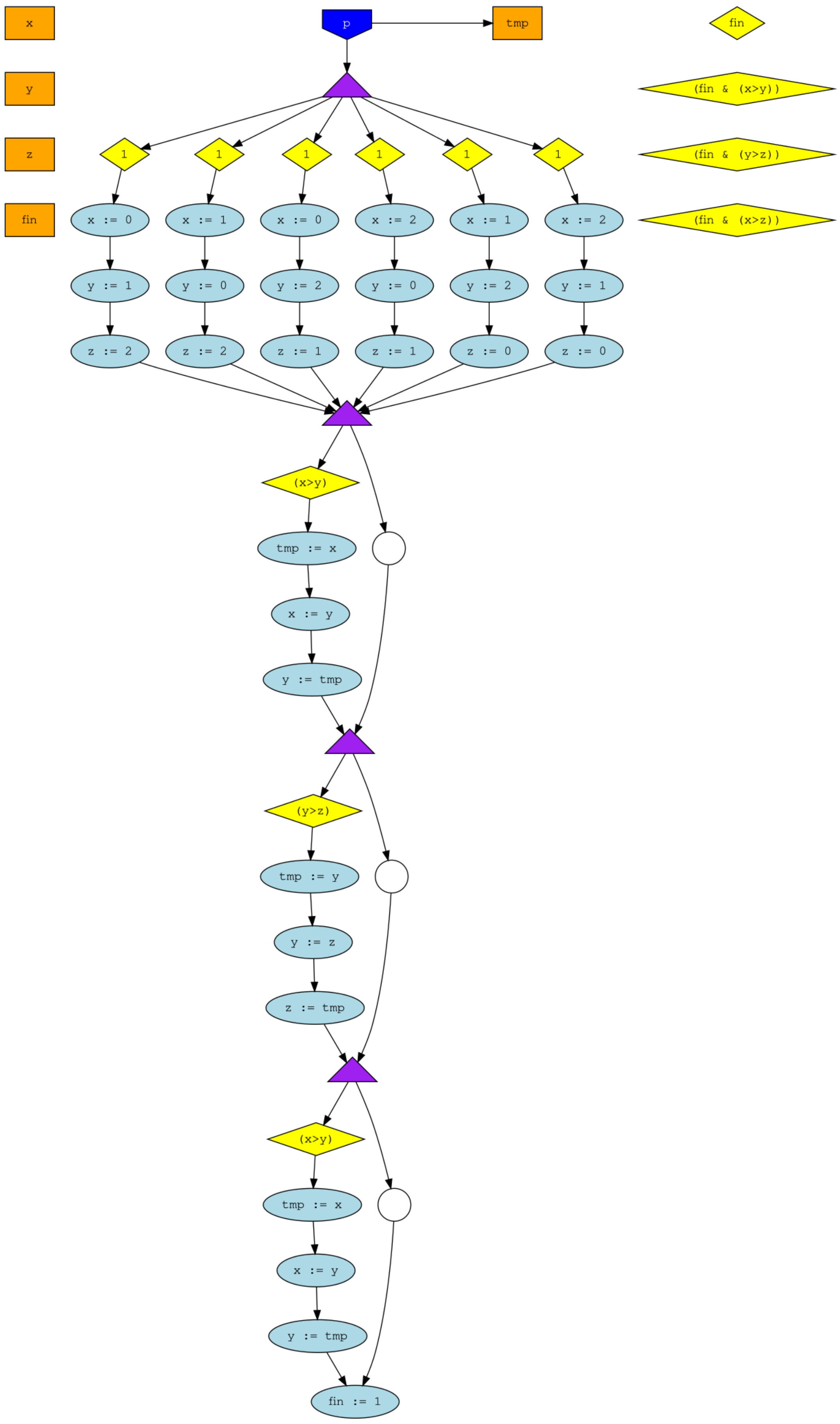
Expressions possess direct pointers to the variables they contain, which are set up during the translation step in `repr.c`.

This level also includes some pretty-printing facilities.

```
$ ./lang assets/lock.prog --repr  -c
================ REPR =================
||  ref {1. lock}
||  ref {0. fin}
||  thread 'p' entrypoint [0]
||    ref {2. todo}
||    <0> {1. lock} <- (2) then [1]
||    <1> {2. todo} <- (1) then [9]
||    <9> 1 guarded, default
||      when (Eq {2. todo} (0)) jump [2]
||        <2> skip [10]
||        <10> {1. lock} <- (Sub {1. lock} (1)) then [11]
||        <11> {0. fin} <- (1) <END>
||        <END>
||      else jump [3]
||        <3> {2. todo} <- (0) then [4]
||        <4> {1. lock} <- (Add {1. lock} (1)) then [8]
||        <8> 2 guarded
||          when (1) jump [5]
||            <5> skip [9] (loop)
||          when (1) jump [6]
||            <6> {1. lock} <- (Sub {1. lock} (1)) then [7]
||            <7> {2. todo} <- (1) then [9] (loop)
||        </>
||    </>
||  end
||  reach? (Eq {1. lock} (1))
||  reach? (Eq {1. lock} (2))
||  reach? (Eq {1. lock} (3))
||  reach? (Eq {1. lock} (4))
||  reach? (And {0. fin} (Not (Eq {1. lock} (2))))
=======================================
```

Note :
- — variables now have a unique identifier as well, ensuring that local variable shadows are implemented correctly
- — the process indicates its entry point which is where the execution will start
- — each statement has a `[n]` marker which indicates to jump to the corresponding `<n>`, and `<n>` is the same as in the pretty-printing for **Level 1**
- — the continuation of a loop is inlined inside it, and looping constructs are identified with `(loop)`

An alternative representation as a graph is also available,
for example `./lang assets/sort.prog --dot && xdg-open assets/sort.prog.png` :



Conditions are diamonds, assignments are ovals, branching statements (`do`,`if`) are triangles, variables are rectangles.

5 Level 3

I chose to implement my own hashset and worklist, in part because I had already started working on them when the hashset example was provided.
The hashset uses a congruential linear hash function on variable values and states of each process.

The worklist is a queue, which ensures that during exhaustive execution each path found that satisfies a condition is the shortest possible path.

With `--trace`, one can see a computation path that leads to each condition (this also applies to the `--rand` mode).

```
$ ./lang assets/lock.prog --trace -A -c

 {1} (Eq {1. lock} (1)) is not reachable

 {2} (Eq {1. lock} (2)) is reachable
  | p: [0]
  | * global [lock: 0] [fin: 0]
  | * local 'p': [todo: 0]
  | {1. lock} <- 2
  | * global [lock: 2] [fin: 0]
  | * local 'p': [todo: 0]

 {3} (Eq {1. lock} (3)) is reachable
  | p: [0]
  | * global [lock: 0] [fin: 0]
  | * local 'p': [todo: 0]
  | {1. lock} <- 2
  | * global [lock: 2] [fin: 0]
  | * local 'p': [todo: 0]
  | 'p' -> [1]
  | {2. todo} <- 1
  | * global [lock: 2] [fin: 0]
  | * local 'p': [todo: 1]
  | 'p' -> [9]
  | 'p' -> [3]
  | {2. todo} <- 0
  | * global [lock: 2] [fin: 0]
  | * local 'p': [todo: 0]
  | 'p' -> [4]
  | {1. lock} <- 3
  | * global [lock: 3] [fin: 0]
  | * local 'p': [todo: 0]

 {4} (Eq {1. lock} (4)) is not reachable

 {5} (And {0. fin} (Not (Eq {1. lock} (2)))) is not reachable
```

6 Semantics

Some parts of the semantics were left for us to clarify, here is what I have chosen.

Breaking without a loop terminates the process

```
var f;
proc p
    f := 1;
    break; // proc exits here
    f := 2
end
```

```
reach f == 1 // reachable
reach f == 2 // unreachable
```

Division by zero is considered false and blocking

```
var f, g;
proc p
    f := 1 / 0; proc is blocked here
    f := 2
end

proc q
    if
    :: 1 \% 0 -> g := 1 // guard is false
    :: else -> g := 2 // else branch is chosen
    fi
end

reach f == 2 // unreachable
reach g == 1 // unreachable
reach g == 2 // reachable

reach 1 / 0 // unreachable
```

Range
In addition, I had some time left to implement as a small extension a range operator. It is not available for exhaustive exploration, but the Monte-Carlo method supports it, although since it exponentially increases the number of configurations the results may not always be accurate if too few iterations are done.
It is best explained by the following example :

```
var f,g;
proc p
    f := 2 * {0..5} // f can take any even value between 0 and 10
end

proc q
    var x;
    x := 5;
    if
    :: 0 < {-x..x} -> g := 1 // ~50% chance guard is satisfied
    :: {1..-1} -> g := 2 // empty range is never satisfied
    fi
end

reach f == 0 // reachable
reach f == 1 // unreachable
reach f == 10 // reachable
reach f == 11 // unreachable

reach g == 1 // reachable

reach {0..1} // reachable
reach {1..0} // unreachable
```

7 Other remarks

— I made sure that no memory is leaked by implementing a registry that stores a linked list of blocks of memory to free.
More specific information about this is available in the code.

To facilitate the freeing of ressources I decided that only in `main` could `exit` be called, and all allocations inside a function must either be returned by the function, freed before the end, or added to one of the registries for `main` (or the caller) to free.
The only exceptions to this rule are `HashSet` and `WorkList`, since they keep track of all their contents and have their fields hidden.

The `make valgrind` target will validate the absence of possible leaks on a range of valid and invalid inputs.

— One suggested extension was to display the line number in case of a parsing error, so I just turned on a few options to get verbose error printing and line numbers.

— When several variables are declared with the same name, the duplicate declarations are ignored. When several procedures are declared with the same name it does not affect the computation since nothing relies on a procedure's name. It does however impede readability of the trace.