Program Semantics and Analysis: Lab2

Thibault Raffaillac, 871109-A416

Introduction

As predicted by the examiners during the previous Lab1, implementing a compiler instead of an interpreter required it to be rewritten for this Lab. One would now need an interpreter for abstract lattice values, to enable optimizations based on constant values. However, implementing interpretation in the Abstract Machine (AM) code as instructed in this Lab would have made optimizations difficult to perform. For example, deleting a branch that is unreachable would require to recompose the whole BRANCH instruction then rewrite it in the instructions stack.

Instead, interpretation was introduced on the While code itself. To minimize the amount of code to maintain, it was even directly done while compiling to AM code. The previous version also had a Hashtable of the variables in use, initialized by parsing the AM code beforehand. It has now been moved to the interpretation section, where it stores the lattice values in addition to the properties necessary for further assembly compilation.

To allow for interpretation while generating AM code, two classes were introduced: ACode and BCode. Each simply contain a lattice value (respectively SignExc and TTExc) along with the Code that generates them. The classes in the whilesyntax package were modified so that the compilation functions return ACode or BCode where Code was previously returned.

Key operations

The lattice values TT, FF and ZERO can lead to optimization of the associated code. Thus, the values returned by the $\neg_{TE}$, $\wedge_{TE}$, $\vee_{TE}$, $+_{SE}$, $-_{SE}$, $*_{SE}$ and $/_{SE}$ operators are systematically checked, and if matched the associated code is replaced respectively by TRUE, FALSE and PUSH-0.

The TryCatch statement is associated with a stack of boolean error states, indicating whether an exception can be triggered in the various nested try statements. After evaluating the statements inside the try block, the catch Code is appended if and only if the topmost error state reports a possible exception raiser.

The Conditional, Whyle and TryCatch statements are associated with a stack of Hashtables of the variables in use. In the case of Conditional and TryCatch , when a new branch is parsed, a copy of the topmost Hashtable is pushed on the stack. When it returns, the copy is popped and all the variables it contains are merged in the previous Hashtable. When the same variable has two different lattice values in the old and new states, the least upper bound is used. In the case of the Whyle statement, interpreting while compiling imposes that only one loop is executed. Thus, an empty Hashtable is actually pushed on the stack, and when the loop returns, all variables found inside it are set to the lattice value Z.

The Conditional and Whyle statements will also skip branches when the corresponding TTExc lattice value evaluates to FF or TT.

When compiling an Assignment statement, the right hand side will first be compiled, returning the Code and its associated lattice value. If this value is a possible error, the current error state for TryCatch is set. If the error stack was empty, then the compiler can trigger an exception, as the possible exception raiser was not included in a try block. Also, when the assignment is done while the current error state is set, the assignment may of may actually not happen, thus the Variable stores the least upper bound of the lattice value received and the previous value stored.

Limitations

The SignExc lattice is quite limited regarding the information it provides. The value Z could be replaced by the storage of the integer itself. Also, following a possible exception the value ANY_A loses the value the expression could have if no exception had actually been triggered. The error state could instead be implemented as an other separate flag in the lattice.

Regarding the TryCatch statement, a THROW instruction could be added. Indeed, currently an assignment can only be flagged as possible exception raiser. If the exception is instead certain, all code should be skipped until the next catch label.

The current implementation loops only once in while loops. It could be improved to loop several times until the state of all variables remains unchanged or a maximum number of loops is reached.

At last, some forward optimizations could be added if the interpreter did more than one pass, like skipping a STORE if the target variable is not fetched later.

Modified operational semantics for AM

$$
\begin{array}{ll}
\langle \mathrm{ADD} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 +_{SE} v_2 : e, ps \rangle \\
\langle \mathrm{AND} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 \wedge_{TE} v_2 : e, ps \rangle \\
\langle \mathrm{BRANCH}(c_1, c_2) : c, v : e, ps \rangle & \triangleright \begin{cases} \langle c_1 : c, e, ps \rangle & \text{if } \mathrm{TT} \sqsubseteq_{TE} v \\ \langle c_2 : c, e, ps \rangle & \text{if } \mathrm{FF} \sqsubseteq_{TE} v \\ \langle c, e, \hat{ps} \rangle & \text{if } \mathrm{ERR}_B \sqsubseteq_{TE} v \end{cases} \\
\langle \mathrm{DIV} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 /_{SE} v_2 : e, ps \rangle \\
\langle \mathrm{EQ} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 =_{SE} v_2 : e, ps \rangle \\
\langle \mathrm{FALSE} : c, e, ps \rangle & \triangleright \langle c, \mathrm{FF} : e, ps \rangle \\
\langle \mathrm{FETCH\text{-}x} : c, e, ps \rangle & \triangleright \langle c, ps(x) : e, ps \rangle \\
\langle \mathrm{LE} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 \leq_{SE} v_2 : e, ps \rangle \\
\langle \mathrm{LOOP}(c_1, c_2) : c, e, ps \rangle & \triangleright \langle c_1 : \mathrm{BRANCH}(c_2 : \mathrm{LOOP}(c_1, c_2), \mathrm{NOOP}) : c, e, ps \rangle \\
\langle \mathrm{MULT} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 *_{SE} v_2 : e, ps \rangle \\
\langle \mathrm{NEG} : c, v_1 : e, ps \rangle & \triangleright \langle c, \neg_{TE} v_1 : e, ps \rangle \\
\langle \mathrm{NOOP} : c, e, ps \rangle & \triangleright \langle c, e, ps \rangle \\
\langle \mathrm{PUSH\text{-}n} : c, e, ps \rangle & \triangleright \langle c, \mathrm{abs}_Z(N(n)) : e, ps \rangle \\
\langle \mathrm{STORE\text{-}x} : c, v : e, ps \rangle & \triangleright \langle c, e, ps[x \to v] \rangle \\
\langle \mathrm{SUB} : c, v_1 : v_2 : e, ps \rangle & \triangleright \langle c, v_1 -_{SE} v_2 : e, ps \rangle \\
\langle \mathrm{TRUE} : c, e, ps \rangle & \triangleright \langle c, \mathrm{TT} : e, ps \rangle \\
\langle \mathrm{TRY}(c_1, c_2) : c, e, ps \rangle & \triangleright \langle c_1 : \mathrm{CATCH}(c_2) : c, e, ps \rangle \\
\langle \mathrm{CATCH}(c_1) : c, e, ps \rangle & \triangleright \langle c, e, ps \rangle \\
\langle \mathrm{CATCH}(c_1) : c, e, \hat{ps} \rangle & \triangleright \langle c_1 : c, e, ps \rangle
\end{array}
$$

Testing

The test programs from the previous Lab fortunately worked for this current Lab (variables start at 0). The program divtest.while was also helpful to further test the Whyle and TryCatch statements.

- Test 1: Assignment

```
i := plusOne;
i := i + 1;
plusOne := i
```

- Test 2: Safe division

```
i := minusOne;
i := i - 1;
minusOne := i;
i := timesTwo;
i := i * 2;
timesTwo := i;
try
      i := divByThree;
      i := i / 3;
      divByThree := i
catch
      skip
```

- Test 3: Branch skipping

```
i := plusOne;
(if true then
      i := i + 1;
      (while false do
            i := i + 2)
else
      skip);
plusOne := i
```

- Test 4: Reduction of boolean conditions

```
i := plusFifteen;
(if 255 = 256 then
      skip
else
      i := i + 1);
(if !(0 = 1) then
      i := i + 2
else
      skip);
(if 255 <= 256 then
      i := i + 4
else
      skip);
(if !(1 <= 0) & 1 <= 2 then
      i := i + 8
else
      skip);
plusFifteen := i
```