Project 3
Program Verification 2013

Bert Massop
Tom Tervoort

April 4, 2013

Contents

1 Introduction

In this report we present our approach to symbolic verification of programs in $Lang_0$ by means of a weakest precondition calculus. We assume programs are well-typed and do not contain references to undeclared variables, allowing us to focus on the weakest precondition calculus.

2 Approach

To prove whether a program X in $Lang_0$ adheres to a specification, represented by pre-condition $\mathcal{P}$ and post-condition $\mathcal{Q}$, we need to find the weakest pre-condition $\mathbf{wp}\ X\ \mathcal{Q}$ of the program given the $\mathcal{Q}$. We can then check whether $\mathcal{P} \Rightarrow \mathbf{wp}\ X\ \mathcal{Q}$. If the condition holds, we can conclude the program adheres to the specification.

2.1 Weakest precondition rules

The weakest precondition rules we use are listed below. Note that the interpretation of the substitution operations on $\mathcal{Q}$ should be from left to right, i.e. first apply the leftmost substitution. T denotes the stack pointer, $\mathcal{S}_T$ denotes the value on the stack at position T. Most rules follow pretty straightforward from the definition of the instruction.

$$
\begin{array}{ll}
\mathbf{wp}\ (\texttt{PUSHLITERAL}\ lit) & \mathcal{Q} = \mathcal{Q}[lit/\mathcal{S}_T][T+1/T] \\
\mathbf{wp}\ \texttt{POP} & \mathcal{Q} = T \geq 0 \\
 & \quad \wedge\ \mathcal{Q}[T-1/T] \\
 & \\
\mathbf{wp}\ (\texttt{SETLOCAL}\ k\ x) & \mathcal{Q} = \mathcal{Q}[x/loc_k] \\
\mathbf{wp}\ (\texttt{LOADLOCAL}\ k) & \mathcal{Q} = \mathcal{Q}[loc_k/\mathcal{S}_T][T+1/T] \\
\mathbf{wp}\ (\texttt{STORELOCAL}\ k) & \mathcal{Q} = T \geq 0 \\
 & \quad \wedge\ \ \mathcal{Q}[T-1/T][\mathcal{S}_T/loc_k] \\
 & \\
\mathbf{wp}\ (\texttt{LOADPARAM}\ k) & \mathcal{Q} = \mathcal{Q}[p_k/\mathcal{S}_T][T+1/T] \\
\mathbf{wp}\ (\texttt{STOREPARAM}\ k) & \mathcal{Q} = T \geq 0 \\
 & \quad \wedge\ \ \mathcal{Q}[T-1/T][\mathcal{S}_T/p_k] \\
 & \\
\mathbf{wp}\ (\texttt{IFTRUE}\ \mathcal{A}\ \texttt{ELSE}\ \mathcal{B}) & \mathcal{Q} = T \geq 0 \\
 & \quad \wedge\ \ \mathcal{S}_T \neq \texttt{false} \Rightarrow (\mathbf{wp}\ \mathcal{A}\ \mathcal{Q})[T-1/T] \\
 & \quad \wedge\ \ \mathcal{S}_T = \texttt{false} \Rightarrow (\mathbf{wp}\ \mathcal{B}\ \mathcal{Q})[T-1/T] \\
 & \\
\mathbf{wp}\ \mathcal{F}_a & \mathcal{Q} = T \geq 1 \\
 & \quad \wedge\ \mathcal{Q}[T-1/T][\mathcal{F}_a(\mathcal{S}_{T-1}, \mathcal{S}_T)/\mathcal{S}_{T-1}] \\
\mathbf{wp}\ \mathcal{F}_b & \mathcal{Q} = T \geq 1 \\
 & \quad \wedge\ \mathcal{F}_b(\mathcal{S}_{T-1}, \mathcal{S}_T) \Rightarrow \mathcal{Q}[T-1/T][\texttt{true}/\mathcal{S}_{T-1}] \\
 & \quad \wedge\ \neg\mathcal{F}_b(\mathcal{S}_{T-1}, \mathcal{S}_T) \Rightarrow \mathcal{Q}[T-1/T][\texttt{false}/\mathcal{S}_{T-1}] \\
 & \\
\mathbf{wp}\ (\texttt{START}\ n) & \mathcal{Q} = \mathcal{Q}[-1/T][\forall n : a_n/p_n] \\
\mathbf{wp}\ \texttt{RETURN} & \mathcal{Q} = T \geq 0 \\
 & \quad \wedge\ \mathcal{Q}[\mathcal{S}_T/\mathbf{return}] \\
\mathbf{wp}\ (\mathcal{A}; \mathcal{B}) & \mathcal{Q} = \mathbf{wp}\ \mathcal{A}\ (\mathbf{wp}\ \mathcal{B}\ \mathcal{Q})
\end{array}
$$

In the above expressions, F_a and F_b represent arithmetic and Boolean-yielding operators, such that $\mathcal{F}_a \in \{\texttt{ADD}, \texttt{SUB}, \texttt{MUL}\}$ and $\mathcal{F}_b \in \{\texttt{LT}, \texttt{GT}, \texttt{EQ}, \ldots\}$. Note that we have substituted the `MIN` command with `SUB` for increased clarity and have added a `START` command, which is roughly equivalent to the **prog** declaration in $Lang_0$.

2.2 Extensions

We plan to implement the extensions 'return from anywhere' and 'bounded verification'. To implement these extensions, we need slightly different weakest precondition rules.

When we encounter a sequence of a `RETURN` followed by other instructions, we just ignore the rest of the instructions and apply the `RETURN` rule immediately. The rule for `WHILETRUE` is slightly more complicated. For some given *bound*, we essentially unroll the evaluation of the while-loop for $\{1, \ldots, bound\}$ iterations. If we exceed the maximum number of iterations, we blindly assume the program execution meets the post-condition.

$$
\begin{aligned}
&\textbf{wp}\ \texttt{RETURN} && \mathcal{Q} = T \geq 0 \\
&&& \quad \wedge\, \mathcal{Q}_{post}[\mathcal{S}_T/\textbf{return}] \\
&\textbf{wp}\ (\texttt{RETURN}; \mathcal{B}) && \mathcal{Q} = \textbf{wp}\ \texttt{RETURN}\ \mathcal{Q} \\
\\
&\textbf{wp}\ (\texttt{WHILETRUE}\ \mathcal{A}) && \mathcal{Q} = w_{bound} \\
&&& w_0 = T \geq 0 \\
&&& \quad \wedge\, \mathcal{S}_T = \texttt{false} \Rightarrow Q[T-1/T] \\
&&& w_k = T \geq 0 \\
&&& \quad \wedge\, \mathcal{S}_T \neq \texttt{false} \Rightarrow \textbf{wp}\ (\texttt{POP}; \mathcal{A})\ w_{k-1} \\
&&& \quad \wedge\, w_{k-1}
\end{aligned}
$$

Note that the `IFTRUE` and `WHILETRUE` rules would normally break the 'return from anywhere' behaviour, as `RETURN` would enforce the weakest precondition of the continuation of the `IFTRUE` / `WHILETRUE` instead of the actual post-condition. Hence the variable $\mathcal{Q}_{post}$ in the `RETURN` rule, which we let contain the actual post-conditions from the program specification.

3 Implementation

Implementation of a calculus with substitution rules can easily go wrong. Therefore, we try to implement our weakest precondition calculus in such a way that it closely adheres to the weakest precondition rules we specified before. This way we try to ensure, although we cannot directly proof it, that our implementation is an actual representation of the rules. The following code fragment illustrates this:

```
SETLOCAL  k x → with q [ Literal x // Local k      ]
LOADLOCAL k   → with q [ Var (Local k) // stack 0
                       , Var T + 1 // T            ]
```

Also, we try to make use of datatype-generic code wherever possible by means of the `syb` package. This greatly decreases the complexity of our implementation.

There are however a few peculiarities in the implementation of our weakest precondition calculus. First, we need to represent both Boolean values as well as integer values on the stack. Since we assume the program to be correctly typed, we can use any integer representation for Booleans. We do so by applying the age-old rule from C: `false` is represented by 0, any other value represents `true`. Note that our weakest precondition rules already did not check for something explicitly being `true`: they checked inequality with `false` instead, making this work out of the box.

We represent Boolean conditions (such as the $\mathcal{P}$ and $\mathcal{Q}$), arithmetic expressions and variables in their own data type.

```
data Condition = GT Expr Expr  | GTE Expr Expr | LT Expr Expr
               | LTE Expr Expr | EQ Expr Expr  | NEQ Expr Expr
               | Forall Name Condition   | Exists Name Condition
               | And Condition Condition | Or Condition Condition
               | Not Condition
               | True | False
  deriving (Data, Typeable, Show, Eq, Read)

data Expr = Add Expr Expr | Sub Expr Expr | Mul Expr Expr
          | Literal Literal
          | Var Var
  deriving (Data, Typeable, Show, Eq, Read)

data Var = Local Local     -- Local variables.
         | Param Param     -- Program arguments that may have been touched
                           -- by the program.
         | Argument Param -- Untouched program arguments.
         | Stack Expr      -- Stack location, with an integer expression
                           -- indicating position relative to the top of
                           -- the stack.
         | Scoped Name     -- Any scoped / named variable (for quantifiers).
         | Return          -- The return variable.
         | T               -- The stack pointer.
  deriving (Data, Typeable, Show, Eq, Read)
```

3.1 Substitution

Our weakest precondition rules consist of Boolean expressions and substitutions of variables. Substituting variables can most easily be done by traversing the entire expression, replacing any variable where necessary. We neatly express this as a generic map over the data structure, keeping us from having to implement rules for every constructor and therefore keeping us from making mistakes.

```
(//) :: Expr → Var → Condition → Condition
(//) to from = simplify ∘ everywhere (mkT subVar)
  where
    subVar e@(Var v) | v == from = to
                     | otherwise = e
    subVar e = e
```

Note the occurrence of *simplify* in the equation. Not only is it nice to simplify the expression for printing purposes, it is also necessary to successfully perform substitutions. Note that the **Stack** constructor takes an **Expr** as its argument – this arithmetic expression represents the offset from the top of the stack, materialized by the stack pointer **T**. If we were to replace (**Stack T**) by, for instance, (**Literal** 1), we would not replace (**Stack** (**T** + 1 − 1)) by default. This is of course an error, since **T** + 1 − 1 = **T**. For this reason, we somehow need to simplify the expressions in the **Stack** variables.

3.2 Simplification

In order to correctly match expressions involving the stack pointer T, we need to simplify expressions to some reduced form. Since expressions involving T (that we need to proof equality on) only contain additions and subtractions, we reduce all such expressions to `ADD`$(T, \mathit{literal})$ or `SUB`$(T, \mathit{literal})$ (in such a way that the sign of the literal is always positive, favouring `ADD` over `SUB` when the literal is zero). More generally, we reduce any arithmetic expression A containing multiple sequential additions and subtractions to `ADD`$(A', \mathit{literal})$ (or alternatively, `SUB`$(\ldots)$) wherever it is safe to do so, accumulating the literals in the expression.

To improve performance and legibility of the output, we also rewrite (in)equality and ordering constraints on two literals to their resulting Boolean value, and use the following rules to rewrite the Boolean expressions in the weakest precondition calculus.

$$\begin{aligned}
\texttt{false} \wedge A &\longrightarrow \texttt{false} \\
\texttt{true} \wedge A &\longrightarrow A \\
\texttt{false} \vee A &\longrightarrow A \\
\texttt{true} \vee A &\longrightarrow \texttt{true} \\
\neg\neg A &\longrightarrow A \\
\neg\texttt{true} &\longrightarrow \texttt{false} \\
\neg\texttt{false} &\longrightarrow \texttt{true}
\end{aligned}$$

$$\begin{aligned}
A \neq B &\longrightarrow \neg(A = B) \\
A \vee A &\longrightarrow A \\
\neg A \vee A &\longrightarrow \texttt{true} \\
A \wedge A &\longrightarrow A \\
\neg A \wedge A &\longrightarrow \texttt{false}
\end{aligned}$$

All this rewriting greatly reduces the complexity of the weakest precondition expressions, especially when there is branching involved. Simple programs without preconditions can often already be proven by the simplifier, resulting in a weakest precondition `true` (or alternatively, `false`). Once again, the rewriting function is a generic map over the condition data structure.

3.3 Proving correctness

Since proving any given Boolean formula is an NP-hard problem, we do not bother implementing a solver ourselves. Instead, we use Z3 to solve the Boolean propositions we generate. Fortunately, there exists a package `sbv` mapping Haskell data types to structures Z3 can understand. To be able to generate our proof, we have to translate our own data structures into something `sbv` understands. This is a rather trivial procedure, with some minor exceptions. `sbv` readily supports universal and existential quantification, as well as various kinds of integer types.

All variables that our simplifier was not able to factor out need to be mapped to either `sbv`-supported universal or existential integer variables. The quantification depends on what we are planning to do: if we are checking satisfiability (more on that later), we need existential quantification. For our proof, we need universal quantification. When assigning variables we make sure to name them properly: Z3 can provide counterexamples, which `sbv` labels with the appropriate variable name. It should be clear that this can be very useful in interpreting the problem.

We do not need any special representation of a stack: given our internal representation, we can only encounter concrete stack positions after substituting $\mathbf{T}$ by -1 in `START`. Most of the times, our simplifier will already have removed any of such references. We can treat all remaining stack references just like any other variable, and universally or existentially quantify an integer variable for it.

While we need to prove that the program adheres to its specification, the results of a proof can be misleading. If we were to accidentally specify a non-satisfiable pre-condition, our program would always proof correct, no matter the post-condition or the contents of the program. To warn the user in such a case, we first run a satisfiability check on the pre-condition specified. To check for satisfiability of the pre-condition, we existentially quantify over all variables and run the satisfiability checker in `sbv`.

The bound for bounded verification is specified in module WP and defaults to 5. The behaviour of the bounded verification can also be changed to a strict mode, that guarantees the while-loop terminate within the bound number of iterations. This can be very useful for testing or implementing generated test-cases.

4 Results / Examples

To verify the working of our symbolic prover, we created some example programs. For more details and the program code, please refer to the `examples/` directory (for $Lang_0$ pseudocode) or refer to the code in module Test (for executable Haskell code). The `.stdin` files[1] can be `cat` to standard input of the built Main module, which will in turn try to proof the specification.

4.1 Maximum of 4

This program returns the maximal value of its four arguments. It does this by maintaining a local variable for the current maximum and comparing each parameter to it, updating it when a larger parameter is encountered. The program, as well as its post-condition, are rather straightforward.

```
Prelude Test Main> doProve max4Example
Given precondition: 'TRUE'.
Weakest precondition of program: [removed long precondition].

Q.E.D.
```

4.2 Square root

Features: while loops.
This program calculates the floor of the square root of a nonnegative integer. It does this by plainly incrementing a variable from 0 until its square becomes higher than parameter x. Once this happens, you know that the value of this variable minus one is the rounded down square root of x.

If $y = \lfloor\sqrt{x}\rfloor$, then it should hold that y^2 is higher than $x - 1$, smaller than $x + 1$, and either equal to or smaller than x. This is checked in the post-condition.

```
Prelude Test Main> doProve sqrtExample
Given precondition: '(ARGUMENT_0 >= 0)'.
Weakest precondition of program: [removed long precondition].

Q.E.D.
```

4.3 Checking for even number

Features: existential quantification in post-condition, while loops, return from anywhere.
This program checks whether a natural number is even. Returns 1 if this is the case and 0 otherwise. The program simply keeps subtracting 2 from the parameter x until it becomes either 0 (meaning an even number) or 1 (meaning an odd number).

Because modulo or divisions are not supported in our logic language, the post-condition uses existential quantification to check whether the answer is correct: x being even implies that there exists some n for which $2n = x$; if x is odd there must be an n for which $2n + 1 = x$.

```
Prelude Test Main> doProve isEvenExample
Given precondition: '(ARGUMENT_0 >= 0)'.
Weakest precondition of program: [removed long precondition].

Q.E.D.
```

5 Discussion

While very simple programs can often quite easily be eliminated to either `true` or `false` by the simplifier, the weakest pre-conditions for programs incorporating while-loops usually tend to

[1] The content for `.stdin` files is generated by a simple call to *show <exampleName>* in the module Test.

explode. While we already have quite a powerful simplifier [2], having a more elaborate rewriter might partially solve this problem.

Still, we expect while-loops to always explode exponentially in the worst case. With a low enough bound on the number of loop iterations our approach stays usable, but when higher bounds are required, symbolic verification is not the best choice. Still, bounded verification can be performed in strict mode, which can be useful for automatically generating test-cases. We did however not implement this feature at this point.

It might be possible to reduce the running time complexity of the substitution / simplification stage by higher integration of the simplification with the substitution: we should not have to traverse the entire condition structure each time something minor changes. Also, we might want to encode more of the calculus logic in the language of the prover (Z3 or `sbv`), that may already have a more efficient way of solving these problems. All of such would however add to the complexity of the implementation, leaving more room for mistakes.

6 Conclusions

We have been able to successfully implement a symbolic verification system for the $Lang_0$ language, with extensions allowing for returning from the program from any location, and bounded verification for simple while loops.

Future work may include automatic test-case generation, additional instructions such as division and exception handling. Yet, care has to be taken not to make the implementation too complex, as it turned out to be very easy to mess up weakest precondition inference rules without it being immediately obvious.

[2] Try disabling the Boolean simplification on a non-trivial program – the program does not even have to be too complex before the calculated weakest precondition takes hundreds of megabytes of memory. For instance, the weakest precondition for the 'is even' example is about 6000 characters when pretty printed with the simplifier on (calculated in 400 ms), versus about 1.6 million with the simplifier off, taking 16 seconds to calculate. Times are measured compiled with `-O`.