To `Goto` Where No Statement Has Gone Before

Mike Barnett and K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA, {mbarnett,leino}@microsoft.com

Abstract. This paper presents a method for deriving an expression from the low-level code compiled from an expression in a high-level language. The input is a low-level control flow graph (CFG) and the derived expression is in a form that can be used as input to an automatic theorem prover. The method is useful for program verification systems that take as input both programs and specifications after they have been compiled from a high-level language. This is the case for systems that encode specifications in an existing programming language and do not have a special compiler. The method always produces an expression, unlike the heuristics for decompilation which may fail. It is efficient: the resulting expression is linear in the size of the CFG by maintaining all sharing of subgraphs.

0 Introduction

A program verifier checks that a given program satisfies its specifications. Some programming languages such as Eiffel [15], Java with JML [11], or Spec# [3] provide the programmer a nice syntax for writing the specifications in the source text. This has many advantages, *e.g.*, that programmers are immediately aware of the relationship between their code and its specification. However, in a multi-language platform like .NET, one would like to have one program verifier that works for any language, regardless of what special syntax, if any, each language may provide. In this paper, we consider one issue that arises in such a multi-language setting.

Code Contracts for .NET [1] is a library-based framework for writing specifications in .NET code. Programmers use the methods from the contract library to write specifications within their program (written in any .NET language, like C#, Visual Basic, or F#) as stylized method calls at the beginning of a method's body. For example, Figure 0 shows a method with a postcondition, expressed as a call to to `Contract.Ensures`. The regular .NET compiler for the source program is invoked to produce bytecode. Code Contracts then has several tools which operate on the resulting bytecode, for example the runtime checker rewrites the bytecode to move the evaluation of postconditions to all of the method body's exit points.

We are connecting an existing program verifier to the Code Contracts framework by translating the compiled bytecode into an intermediate verification language, Boogie 2 [0,14, 12], and then generating verification conditions for a theorem prover (we primarily use the SMT solver Z3 [6]). Source-program uses of Code Contracts show up in the bytecode as calls to the contract methods, preceded by a snippet of code that evaluates the arguments. For the example in Figure 0, the bytecode computes the postcondition and then passes that boolean value as the argument to `Contract.Ensures`.

Therefore, *expressions* in the source language become *code*. In general, the code is a linearized form of a DAG, with a high degree of sharing.

The problem is that the verification conditions needed by the theorem prover must be first-order formulas. While there are various contexts in which this can be avoided, the body of a quantifier *must* be a genuine expression, not code.

We propose to convert the code representing a boolean expression back into a genuine expression in two steps. First, our program verifier identifies the code snippets in the bytecode and converts them into *code expressions* of the form

$$\{\{ \quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ e \quad \}\}$$

where S denotes some code in the intermediate verification language, e denotes the value returned by the code expression, and b denotes a list of local variables that may be used in S and e. Defining code expressions in the intermediate verification language has the advantage that we can make use of facilities in the intermediate verification language that expect expressions, like pre- and postconditions and bodies of logical quantifiers.

Second, we define the meaning of a code expression in terms of a first-order formula. We show how to construct this formula from the code expression. The resulting formula is "efficient": it maintains the sharing in the DAG, and is thus linear in the size of the control-flow graph of the code expression.

In this paper, we also give some healthiness conditions for what it means to interpret code as a genuine expression.

1 The Starting Point

An example program in the C# programming language using Code Contracts is shown in Figure 0. The example shows a simple method that has a postcondition (encoded

```
using System.Diagnostics.Contracts;
public class C {

  public int[] M(int[] A, int k) {
    Contract.Ensures(
      Contract.Result<int[]>().Length == A.Length &&
      Contract.ForAll(
        0, Contract.Result<int[]>().Length,
        i => A[i] == 0 || Contract.Result<int[]>()[i] == k/A[i]
      )
    );
    ...
  }
}
```

Fig. 0. A portion of a C# program using Code Contracts

using the method `Contract.Ensures`). It states that the return value (encoded with `Contract.Result`) has the same length as the parameter `A` and that each element is the division of `k` by the corresponding element of `A`, except in the case that the element is zero [0]. In order to state that, it uses a quantifier: the method `Contract.Forall` is given three arguments, an inclusive lower bound, an exclusive upper bound, and an *anonymous delegate*. The latter is the .NET form for a *lambda expression*, *i.e.*, a functional value. The type of `Forall` restricts the function to take a single argument of type **`int`** and return a boolean[1]. In the example, the function's parameter is named `i`. In traditional notation, the function would be written as $(\lambda\ i : int\ .\ A[i] \neq 0 \Rightarrow result[i] = k/A[i])$. Anonymous delegates are lexically scoped and "capture" references, such as to the method's parameter `A`.

The source-language compiler (in this case the C# compiler) is used to compile the program to MSIL. Since we do not have control over the C# compiler, the specifications are compiled into MSIL just as the "real" program is. In particular, short-circuit boolean expressions are compiled into *code expressions*. These are a linearized DAG of basic blocks with assignment statements and goto statements where the value of the boolean expression is left on the stack. For the current example, Figure 1 shows the MSIL that the anonymous delegate in Figure 0 compiles into. A more readable form written in C# is:

```
bool Anonymous(int i) {
  bool b;
  if (A[i] == 0) goto L_0024;
  if (result[i] == k/A[i]) goto L_0024;
  b := false;
  goto L_0028;
  L_0024: b := true;
  L_0028: return b;
}
```

2 The Midpoint: Boogie

An intermediate verification language serves a purpose analogous to that of an intermediate representation in a compiler: it separates the concerns of defining source-language semantics from the concerns of generating formulas for a theorem prover. Many program verifiers are built around an architecture that uses an intermediate verification language (*e.g.*, [0,9, 4]).

[0] The method `Contract.Result` is generic and must be instantiated since its type cannot be inferred from its arguments because it is a nullary method (hence the open-close parentheses). Type instantiation is indicated by referring to the return type of the method, **`int`**`[]`, within angled brackets. This shows why it is so nice to have a language provide surface syntax for specifications!

[1] There is another version of `Forall` that allows a more general predicate.

```
.method public hidebysig instance bool <M>b__0(int32 i) cil managed
{
  .maxstack 4
  .locals init (
    [0] bool CS$1$0000)
  L_0000: ldarg.0
  L_0001: ldfld int32[] C/<>c__DisplayClass1::A
  L_0006: ldarg.1
  L_0007: ldelem.i4
  L_0008: brfalse.s L_0024
  L_000a: call !!0 [Microsoft.Contracts]System.Diagnostics.Contracts.Contract::Result<int32[]>()
  L_000f: ldarg.1
  L_0010: ldelem.i4
  L_0011: ldarg.0
  L_0012: ldfld int32 C/<>c__DisplayClass1::k
  L_0017: ldarg.0
  L_0018: ldfld int32[] C/<>c__DisplayClass1::A
  L_001d: ldarg.1
  L_001e: ldelem.i4
  L_001f: div
  L_0020: ceq
  L_0022: br.s L_0025
  L_0024: ldc.i4.1
  L_0025: stloc.0
  L_0026: br.s L_0028
  L_0028: ldloc.0
  L_0029: ret
}
```

Fig. 1. The bytecode compiled from the body of the anonymous delegate in Figure 0. The labels on each line are the byte offsets of the instructions. The code from offset 0x0 to 0x7 represents the left disjunct $A[i] == 0$. The right disjunct, `Contract.Result<int[]>()[i] == k/A[i]`, is computed in the code from offset 0x0a to 0x20. "arg 0" refers to **`this`**, the implicit receiver and "arg 1" refers to the parameter `i`. There is an implicit receiver because the captured variables in an anonymous delegate become fields on a compiler-generated class in order to retain the necessary state in between invocations. In this case, there are fields for `A` and `k`.

2.0 Previously...

We reiterate the language from [2], which forms the core of the Boogie intermediate verification language:

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{Block}^+ \\
\mathit{Block} & ::= & \mathit{BlockId}\ :\ \mathit{Stmt}\ ;\ \mathit{Goto} \\
\mathit{Stmt} & ::= & \mathit{VarId} := \mathit{Expr}\ \mid\ \mathbf{havoc}\ \mathit{VarId} \\
 & & \mid\ \mathit{Stmt}\ ;\ \mathit{Stmt}\ \mid\ \mathbf{skip} \\
 & & \mid\ \mathbf{assert}\ \mathit{Expr}\ \mid\ \mathbf{assume}\ \mathit{Expr} \\
\mathit{Goto} & ::= & \mathbf{goto}\ \mathit{BlockId}^*
\end{array}
$$

In our core language, a program consists of a set of basic blocks, where the unstructured control flow between blocks is given by goto statements. A goto with multiple target

labels gives rise to a non-deterministic choice; a goto with no target labels gives rise to normal termination. The $BlockId$'s listed in a goto statement are the *successors* of the block.

The semantics of the core language is defined over *traces*, *i.e.*, sequences of program states. Each finite trace either terminates normally or ends in an error. There are two assignment statements: $x := e$ sets variable x to the value of expression e, and $\mathbf{havoc}\ x$ sets x to an arbitrary value. Semi-colon is the usual sequential composition of statements, and $\mathbf{skip}$, which is the unit element of semi-colon, terminates normally without changing the state. The assert statement $\mathbf{assert}\ e$ behaves as $\mathbf{skip}$ if e evaluates to $true$; otherwise, it causes the trace to end in an error (we say the trace *goes wrong*). The assume statement $\mathbf{assume}\ e$ is a *partial command* [16]: it behaves as $\mathbf{skip}$ if e evaluates to $true$; otherwise, it leads to no traces at all. The assume statement is thus used to describe which traces are feasible.

The normally terminating traces of a block are extended with the traces of the block's successors.

Note that the core language does not have a method call as a primitive statement; a method call is encoded as a sequence of statements that assert the method's precondition, use havoc statements to set the locations that the method may modify to an arbitrary value, and then assume the method's postcondition.

Verification condition generation proceeds by first converting the program into *passive form*, where loops are cut (see [2]) and where all assignment statements are replaced by assumptions expressed over a single-assignment form of the program variables [10]. For example, a statement

$$x := y\ ;\ x := x + y\ ;\ \mathbf{assert}\ y < x$$

is converted into a passive form like

$$\mathbf{assume}\ x_1 = y_0\ ;\ \mathbf{assume}\ x_2 = x_1 + y_0\ ;\ \mathbf{assert}\ y_0 < x_2$$

where y_0, x_1, and x_2 are fresh variables.

The passive program is turned into a formula via *weakest preconditions* [8]. For any passive statement S and any predicate Q characterizing a set of post-states of S, $wp[\![S, Q]\!]$ is a predicate that characterizes those pre-states from which execution of S will not go wrong and will end in a state described by Q. The weakest-precondition equations for passive statements are as follows:

$$\begin{array}{lcl} wp[\![\mathbf{skip}, Q]\!] & = & Q \\ wp[\![S\ ;\ T,\ Q]\!] & = & wp[\![S, wp[\![T, Q]\!]]\!] \\ wp[\![\mathbf{assert}\ e, Q]\!] & = & e \wedge Q \\ wp[\![\mathbf{assume}\ e, Q]\!] & = & e \Rightarrow Q \end{array}$$

In each of the last two equations, the occurrence of e on the left-hand side is an expression in Boogie, whereas its occurrence on the right-hand side must be an expression in the input language of the theorem prover. These expressions are usually so similar that we do not mind glossing over this difference; however, for code expressions this makes an important difference.

To deal with (unstructured) control flow, we introduce a variable A_{ok} for every block labeled A, and we define A_{ok} to be $true$ iff no execution from A goes wrong [2]. In particular, for any block A with body S and successors $Succ(A)$, we define

$$A_{ok} \quad = \quad wp[\![S, \bigwedge_{B \in Succ(A)} B_{ok}]\!]$$

2.1 Adding Code Expressions to Boogie

We extend the core language to include code expressions. Previously [2], we left implicit the definition of *Expr* (and its implementation did not allow code expressions). Now, we explicitly extend the definition of *Expr* to include them:

$$\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Expr}\ \mathit{op}\ \mathit{Expr} \mid \mathit{MethodCall} \mid \mathit{CodeExpr} \\
\mathit{CodeExpr} & ::= & \{\!\{\ \mathit{LocalDecl}^*\ \mathit{CodeBlock}^+\ \}\!\} \\
\mathit{LocalDecl} & ::= & \mathit{VarId}\ :\ \mathit{Type} \\
\mathit{CodeBlock} & ::= & \mathit{BlockId}\ :\ \mathit{Stmt}\ ;\ \mathit{Transfer} \\
\mathit{Transfer} & ::= & \mathit{Goto} \mid \mathit{Return} \\
\mathit{Return} & ::= & \mathbf{return}\ \mathit{Expr}
\end{array}$$

We need each code expression to be a self-contained unit. In order to achieve that, we assume that each code expression is *well-formed* by meeting the following conditions:

- A *transfer command* comprising a goto statement has at least one successor.
- All successors are other blocks within the code expression.
- No block in a code expression is a successor of any block not in the code expression.
- The graph induced on the blocks by the successor relation is acyclic.
- All paths within the code expression end with a block whose transfer command comprises a return statement.

We also assume each code expression has a first block labeled "*Start*", which is the entry point to the code expression.

2.2 When Is Code an Expression?

It is one thing to syntactically allow code expressions in Boogie, but we still must consider when a code expression really does represent a genuine expression, *i.e.*, when we are justified in using the same semantics for them as for genuine expressions. Thus the question of this section: when can we look at a chunk of code and consider it a genuine expression?

It must meet four requirements:

0. It must be deterministic. (All branches are mutually exclusive.)
1. It must be total in terms of not being a partial command.
2. It must be total, in the "expression sense". That is, its execution does not go wrong (*i.e.*, failing an assertion, like dereferencing *null* or dividing by zero).

3. It must not have any side effects (on variables other than the local variables it introduces).

The first two requirements are not enforced (yet), but instead are left as the programmer's responsibility. Code expressions originating in compiled .NET code meet both requirements. In case a code expression contains a partial command, our scheme will derive the value $true$ in states where the partiality comes into play.

We enforce the third requirement by ignoring all assertions within a code expression. Many verifiers enforce such *definedness checks* [13] for expressions separately from the expressions themselves by inserting extra checks which guarantee that the expression is total.

The fourth requirement is enforced by making sure that all assignment statements within a code expression are to its local variables and that all method calls are to *pure methods*, *i.e.*, methods whose Boogie encoding do not have any modifies clauses.

3 The Endpoint: Deriving an Expression from Code

But now we have a mismatch: we have code expressions in places where they need to be translated to expressions in the prover's language. We either need a new definition for the weakest precondition when an expression is a code expression or we need a translation scheme that produces a genuine expression from a code expression.

We take the latter approach and, for now, restrict ourselves to *boolean* code expressions, *i.e.*, the value they return is a boolean. For boolean code expressions that meet the requirements in Section 2.2, we compute an equivalent boolean expression (that does not contain any code expressions). For the code expression

$$\{\{ \quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ e \quad \}\}$$

the equivalent boolean expression is

$$(\ \forall\, b \bullet\ \mathit{wp}[\![S, e]\!]\ ) \tag{0}$$

This presumes that S is a structured command, *i.e.*, control flows from S to the return statement. When the code expression is unstructured, then we form the block equations as in Section 2.0. The only difference is that for any block A whose transfer statement comprises a return statement $\mathbf{return}\ e$, we define the block equation as:

$$A_{ok} \quad = \quad \mathit{wp}[\![S, e]\!]$$

Because code expressions are acyclic, we can avoid having to quantify over the block variables by defining them via let-expressions. (Z3 supports the SMT-LIB format [17], which allows let-expressions in the verification condition.)

So the body of the anonymous delegate can be represented in Boogie as:

$\{\{\ \ b\colon \mathbf{bool};$
$\mathit{Start} : \mathbf{skip}\ ;\ \mathbf{goto}\ L0, L1;$
$L0 : \mathbf{assume}\ A[i] = 0\ ;\ \mathbf{goto}\ L2;$
$L1 : \mathbf{assume}\ A[i] \neq 0\ ;\ \mathbf{goto}\ L3, L4;$
$L2 : b := \mathit{true}\ ;\ \mathbf{goto}\ L5;$
$L3 : \mathbf{assume}\ \mathit{result}[i] = k/A[i]\ ;\ \mathbf{goto}\ L2;$
$L4 : \mathbf{assume}\ \mathit{result}[i] \neq k/A[i]\ ;\ b := \mathit{false}\ ;\ \mathbf{goto}\ L5;$
$L5 : \mathbf{skip}\ ;\ \mathbf{return}\ b;\ \ \}\}$

We first convert it into passive form by introducing a new *incarnation* of a variable each time it is assigned. Join points (*e.g.*, $L5$) also produce a new incarnation with equations pushed into each predecessor relating the value of the variable in that branch with that of the join point's incarnation.

$\{\{\ \ b\colon \mathbf{bool};$
$\mathit{Start} : \mathbf{skip}\ ;\ \mathbf{goto}\ L0, L1;$
$L0 : \mathbf{assume}\ A[i] = 0\ ;\ \mathbf{goto}\ L2;$
$L1 : \mathbf{assume}\ A[i] \neq 0\ ;\ \mathbf{goto}\ L3, L4;$
$L2 : \mathbf{assume}\ b_0 = \mathit{true}\ ;\ \mathbf{assume}\ b_2 = b_0\ ;\ \mathbf{goto}\ L5;$
$L3 : \mathbf{assume}\ \mathit{result}[i] = k/A[i]\ ;\ \mathbf{goto}\ L2;$
$L4 : \mathbf{assume}\ \mathit{result}[i] \neq k/A[i]\ ;\ \mathbf{assume}\ b_1 = \mathit{false}\ ;\ \mathbf{assume}\ b_2 = b_1\ ;\ \mathbf{goto}\ L5;$
$L5 : \mathbf{skip}\ ;\ \mathbf{return}\ b_2;\ \ \}\}$

Then the block equations, written as let-expressions, are:

$$
\begin{array}{lllll}
\mathbf{let} & L5_{ok} & = & wp[\![\mathbf{skip}, b_2]\!] & \mathbf{in} \\
\mathbf{let} & L2_{ok} & = & wp[\![\mathbf{assume}\ b_0 = \mathit{true}\ ;\ \mathbf{assume}\ b_2 = b_0, L5_{ok}]\!] & \mathbf{in} \\
\mathbf{let} & L3_{ok} & = & wp[\![\mathbf{assume}\ \mathit{result}[i] = k/A[i], L2_{ok}]\!] & \mathbf{in} \\
\mathbf{let} & L4_{ok} & = & wp[\![\mathbf{assume}\ \mathit{result}[i] \neq k/A[i]\ ;\ \mathbf{assume}\ b_1 = \mathit{false}\ ; & \\
 & & & \mathbf{assume}\ b_2 = b_1, L5_{ok}]\!] & \mathbf{in} \\
\mathbf{let} & L1_{ok} & = & wp[\![\mathbf{assume}\ A[i] \neq 0, L3_{ok} \wedge L4_{ok}]\!] & \mathbf{in} \\
\mathbf{let} & L0_{ok} & = & wp[\![\mathbf{assume}\ A[i] = 0, L2_{ok}]\!] & \mathbf{in} \\
\mathbf{let} & \mathit{Start}_{ok} & = & wp[\![\mathbf{skip}, L0_{ok} \wedge L1_{ok}]\!] & \mathbf{in} \\
 & \mathit{Start}_{ok} & & &
\end{array}
$$

After simplifying[2] the expression is equivalent to:

$$
\begin{array}{lllll}
\mathbf{let} & L5_{ok} & = & b_2 & \mathbf{in} \\
\mathbf{let} & L2_{ok} & = & b_0 = \mathit{true} \Rightarrow b_2 = b_0 \Rightarrow L5_{ok} & \mathbf{in} \\
\mathbf{let} & L3_{ok} & = & \mathbf{result}[i] = k/A[i] \Rightarrow L2_{ok} & \mathbf{in} \\
\mathbf{let} & L4_{ok} & = & \mathbf{result}[i] \neq k/A[i] \Rightarrow b_1 = \mathit{false} \Rightarrow b_2 = b_1 \Rightarrow L5_{ok} & \mathbf{in} \\
\mathbf{let} & L1_{ok} & = & A[i] \neq 0 \Rightarrow L3_{ok} \wedge L4_{ok} & \mathbf{in} \\
\mathbf{let} & L0_{ok} & = & A[i] = 0 \Rightarrow L2_{ok} & \mathbf{in} \\
\mathbf{let} & \mathit{Start}_{ok} & = & L0_{ok} \wedge L1_{ok} & \mathbf{in} \\
 & \mathit{Start}_{ok} & & &
\end{array}
$$

[2] Yes, we realize it doesn't look particularly simple. We mean that we have applied the definition of the weakest-precondition.

If we denote that entire expression by R, then the genuine expression which is equivalent to the body of the anonymous delegate is:

$$(\forall\, b_0, b_1, b_2 \bullet\ R\ )$$

and the entire postcondition of the method in Figure 0 is:

$$\begin{array}{l} \mathbf{result}.Length = A.Length\ \ \wedge \\ (\forall\, i \bullet\ 0 \leq i < \mathbf{result}.Length \Rightarrow (\forall\, b_0, b_1, b_2 \bullet\ R\ )) \end{array}$$

Looking closely[3], one can see that the truth value of this expression is equivalent to the original postcondition.

We perform this translation in a depth-first traversal of the program, replacing each code expression from innermost to outermost.

4 Non-Boolean Code Expressions

In this section, we extend our translation of boolean code expressions to code expressions of any type. The basic idea is to distribute the non-boolean code expression to a context where its value can be stated as a boolean antecedent.

Let $G[\cdot]$ denote an expression context with a "hole". That is, if we place an expression e in the hole, written $G[e]$, we get an expression with an occurrence of e as a subexpression. We assume bound variables in G are suitably renamed so as to always avoid name capture of the free variables of e.

Now, let e be a code expression of an arbitrary type (that is, not necessarily boolean), and let $VC[e]$ be the verification condition (in other words, the verification condition contains an occurrence of e). We now show how to transform expression $VC[e]$ to an equivalent expression that does not contain this occurrence of e but instead contains a boolean code expression. First, for any variable x occurring free in e and introduced in the verification condition by a let binding $\mathbf{let}\ x = t\ \mathbf{in}\ u$, replace x by t in e. Then, consider any context G such that $G[e]$ is a boolean subexpression of $VC[e]$; that is, $G[e]$ is some subexpression of $VC[e]$ such that the free variables of e are also free variables of $G[e]$. Specifically, if e is contained in a quantifier, then $G[e]$ can be the body of the innermost such quantifier; if e is not contained in any quantifier, then $G[e]$ can simply be $VC[e]$.

Since $G[e]$ is boolean, it is equivalent to the expression

$$(\forall\, k \bullet\ k = e \Rightarrow G[k]\ )$$

where k is a fresh variable. Considering that e is a code expression, we have:

$$\begin{array}{ll} & (\forall\, k \bullet\ k = \{\!\{\quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ d\quad \}\!\} \Rightarrow G[k]\ ) \\ = & \quad \{\ \text{distribute "}k =\text{" over the code expression}\ \} \\ & (\forall\, k \bullet\ \{\!\{\quad \mathbf{var}\ b;\ \ S\ ;\ \mathbf{return}\ k = d\quad \}\!\} \Rightarrow G[k]\ ) \end{array}$$

The transformation we have just showed can thus be used to replace non-boolean code expressions with boolean ones, after which the semantics that we have defined for boolean code expressions earlier in the paper can be used.

[3] Squinting helps too.

5 Related Work

An alternative means for recovering boolean expressions would be to *decompile* the MSIL back into a high-level expressions [5]. For the trivial example with which we have demonstrated our scheme, this clearly would be quite easy.

However, we believe all decompilers are heuristic and so may not always be able to successfully decompile an expression, certainly not without perhaps introducing the same redundancy as a tree-encoding of the DAG, compared to the linear size of our derived expression. Also, a decompiler's goal is to produce an expression which is "close" to a boolean expression that a programmer would write. We are not concerned with making the expression "readable", but instead just need to be able to communicate it to a theorem prover.

6 Conclusions

In Section 0, we noted that there are contexts in which code expressions do not need to be converted back into a genuine expression. For instance, Boogie encodes preconditions (respectively, postconditions) as assume (respectively, assert) statements in the Boogie program itself. Instead of forming the verification condition $P \Rightarrow wp[\![S, Q]\!]$ for a program S, precondition P, and postcondition Q, it computes the weakest precondition with respect to $true$ of the program:

$$\begin{array}{l}\textbf{assume}\ \ P\ ;\\ S;\\ \textbf{assert}\ \ Q\end{array}$$

This means that if P or Q are code expressions, they can be *in-lined* and the assume (assert) "distributed" so that any return statement in the code expression, $\textbf{return}\ e$, becomes an assertion (assumption) on e. Then, the definitions of wp in Section 2 will produce a first-order formula that is accepted by theorem provers.

But this cannot be done for quantifiers: instead they must be translated into an equivalent quantifier in the input language of the theorem prover, which does not include code expressions. Therefore, we need to perform our technique only for code expressions occurring within a quantifier. As we progress with the implementation of this scheme in Boogie, we will need to see if the introduction of the quantifier in Equation 0 leads to problems with triggering. (A trigger is the pattern a Simplify-like SMT solver requires before it instantiates a quantifier [7].)

In summary, in this paper, we have adapted our previous work on verification condition generation [2] to provide a scheme for turning code that represents an expression back into an expression in order for it to be easily translated into input for an automatic theorem prover. The scheme avoids decompilation and is efficient. We also outlined four healthiness conditions for ensuring that a code expression can be treated as a genuine expression.

Acknowledgements

We would like to thank Manuel Fähndrich, Francesco Logozzo, and Michał Moskal for valuable help and insight.

References

0. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
1. Mike Barnett, Manuel Fähndrich, and Francesco Logozzo. Embedded contract languages. In *ACM SAC - OOPS*. ACM, March 2010.
2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM Press.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
4. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS 2007*, pages 19–33, 2007.
5. Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software — Practice and Experience*, 25(7):811–829, July 1995.
6. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.
7. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
8. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
9. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.
10. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.
11. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
12. K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at http://research.microsoft.com/ leino/papers.html.
13. K. Rustan M. Leino. Specification and verification of object-oriented software. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *Engineering Methods and Tools for Software*

Safety and Security, volume 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. Summer School Marktoberdorf 2008 lecture notes.

14. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, March 2010.
15. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
16. Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.
17. Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.