Verified VCG and Verified Compiler for Dafny

Downloaded from: https://research.chalmers.se, 2026-10-14 17:12 UTC

Citation for the original published paper (version of record):
Nezamabadi, D., Myreen, M., Tan, Y. (2026). Verified VCG and Verified Compiler for Dafny. Cpp 2026 Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs Co Located with Popl 2026: 171-186. http://dx.doi.org/10.1145/3779031.3779092

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3779031.3779092

RESEARCH-ARTICLE

Verified VCG and Verified Compiler for Dafny

DANIEL NEZAMABADI, Swiss Federal Institute of Technology, Zurich, Zurich, ZH, Switzerland

MAGNUS O. MYREEN, Chalmers University of Technology, Gothenburg, Vastra Gotaland, Sweden

YONG KIAM TAN, Agency for Science, Technology and Research, Singapore, Singapore City, Singapore

Open Access Support provided by:

Chalmers University of Technology

Swiss Federal Institute of Technology, Zurich

Agency for Science, Technology and Research, Singapore

PDF Download
3779031.3779092.pdf
28 January 2026
Total Citations: 0
Total Downloads: 38

Published: 08 January 2026

Citation in BibTeX format

CPP '26: 15th ACM SIGPLAN International Conference on Certified Programs and Proofs
January 12 - 13, 2026
Rennes, France

Conference Sponsors:
SIGPLAN

https://doi.org/10.1145/3779031.3779092
ISBN: 9798400723414

Verified VCG and Verified Compiler for Dafny

Daniel Nezamabadi
ETH Zurich
Switzerland

Magnus O. Myreen
Chalmers University of Technology
University of Gothenburg
Sweden

Yong Kiam Tan
Institute for Infocomm Research (I²R), A*STAR
Singapore
Nanyang Technological University
Singapore

Abstract

Dafny is a verification-aware programming language that comes with a compiler and static program verifier. However, neither the compiler nor the verifier is proved correct; in fact, soundness bugs have been found in both tools. This paper shows that the aforementioned Dafny tools can be developed with foundational correctness guarantees. We present a functional big-step semantics for an imperative subset of Dafny and, based on this semantics, a verified verification condition generator (VCG) and a verified compiler for Dafny. The subset of Dafny we have formalized includes mutually recursive method calls, while loops, and arrays—these language features are significant enough to cover challenging examples such as McCarthy's 91 function and array-based programs that are used when teaching Dafny. The verified VCG allows one to prove functional correctness of annotated Dafny programs, while the verified compiler can be used to compile verified Dafny programs to CakeML programs. From there, one can obtain executable machine code via the (already verified) CakeML compiler, all while provably maintaining the functional correctness guarantees that were proved for the source-level Dafny programs. Our work has been mechanized in the HOL4 theorem prover.

CCS Concepts: • **Theory of computation** → Higher order logic; **Program verification**; • **Software and its engineering** → *Compilers*; **Software verification**;

Keywords: verified compilation, verified verification condition generator, CakeML, Dafny, interactive theorem proving

ACM Reference Format:
Daniel Nezamabadi, Magnus O. Myreen, and Yong Kiam Tan. 2026. Verified VCG and Verified Compiler for Dafny. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779031.3779092

This work is licensed under a Creative Commons Attribution 4.0 International License.
CPP '26, Rennes, France
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779092

```
method M(n: int) returns (r: int)
  ensures r == if n <= 100 then 91 else n - 10
  decreases 111 - n
{
  if n <= 100 {
    var tmp := M(n + 11);
    r := M(tmp);
  } else {
    r := n - 10;
  }
}
```

Figure 1. "91 function" in Dafny.[1]

1 Introduction

Dafny [41] is a verification-aware programming language with built-in support for specifications and supporting tools: a compiler and static program verifier. Recently, Dafny has been successfully used by Amazon Web Services to develop a new and improved version of their authorization engine while provably maintaining the functional behavior of the original implementation [13].

To build an intuition on how Dafny works, consider McCarthy's 91 function [43] shown in Figure 1, a nested recursive function that is used as a basic test for automated verification. It is interesting from an automation perspective because to show that the value of the `decreases` clause actually decreases in the second recursive call, the prover must simultaneously establish the postcondition for `tmp`. To prove that the implementation satisfies the specifications given in the `ensures` and `decreases` clauses, the verifier translates Dafny to Boogie [6, 40], an intermediate verification language and verification condition generator (VCG). The generated verification conditions are then checked using a satisfiability modulo theories (SMT) solver, such as Z3 [17]. In order to run the program, the Dafny compiler translates it to a language, e.g., C#, whose toolchain(s) can then be used to generate a binary.

Both the Dafny compiler and VCG pipeline are substantial pieces of software that must produce trustworthy results. Unfortunately, past work has identified issues—particularly soundness bugs—in both the implementation of the compiler

[1] Adapted from the Dafny GitHub repository.

and the verifier [18, 34]. We aim to address these limitations by showing that it is possible for Dafny to be done in a foundational, end-to-end manner. In this work, we make the following contributions:

- Define functional big-step semantics [49] for an imperative subset of Dafny, including mutually recursive method calls, while loops, and arrays (Section 2).
- Implement a verified compiler from our subset of Dafny to CakeML [38], which, in turn, provides a verified compiler down to machine code (Section 3).
- Define a weakest precondition (wp) calculus for the Dafny subset, prove its soundness, and use it to implement a verified VCG (Section 4).

Our subset is expressive enough to support examples such as the previously introduced 91 function, and array-based programs that are used when teaching Dafny. We demonstrate our contributions in Section 5, where we apply both the compiler and VCG to the 91 function and a method that swaps two elements in an array (Swap), followed by proving the resulting verification conditions to be valid. This yields functional correctness guarantees at the level of compiled CakeML functions.

We have mechanized this work in the HOL4 theorem prover. Below, we present and motivate key design choices in the formalization that enabled each of our contributions.

2 Semantics

Before we can write a verified compiler or verified VCG, we must first formalize the meaning of Dafny programs. We opted to define an untyped, *functional big-step semantics* [49] for Dafny, i.e., a definitional interpreter function with a clock (also called fuel) to ensure its termination and thereby definability within the HOL4 logic. This style of semantics is also used for CakeML and its compiler's intermediate languages because it is particularly well-suited for mechanizing compiler correctness results in HOL4 [49].

The main syntactic objects are expressions (dfy_exp), statements (dfy_stmt), and programs (program); Dafny programs are a list of declarations used to look up functions and methods. Our interpreters for Dafny expressions and statements have the following type signatures, respectively:

```
evaluate_exp  : state → dfy_env → dfy_exp
                → state × value exp_result
evaluate_stmt : state → dfy_env → dfy_stmt
                → state × stmt_result
```

In this section, we will first introduce the types representing the semantic primitives such as value, α exp_result, stmt_result, dfy_env, and state. Afterward, we will define interpreter semantics for Dafny expressions and statements, followed by the semantics of Dafny programs.

2.1 Semantic Primitives

Values. We support integers, booleans, strings, and arrays, and define the value type as follows:

```
value =
  IntV int
| BoolV bool
| StrV string
| ArrV num num type
```

Array values hold their length, location on the heap, and the type of their elements.[2] Note that our language supports nested arrays, e.g., an array of arrays of integers, which is not to be confused with multi-dimensional arrays. The heap itself is a list of heap values; thus, heap locations are indices into that list. We currently only support arrays but plan to add objects in the future.

```
heap_value = HArr (value list) type
```

Results. The interpreter can fail due to undefined behavior (Rfail), such as trying to read an undeclared or uninitialized variable, or timing out (a feature of functional big-step semantics):

```
err_result = Rfail | Rtimeout
```

The result type is different for expressions and statements. Evaluating an expression can either return a value or fail due to the reasons above:

```
α exp_result = Rval α | Rerr err_result
```

By parameterizing the type of the value being returned, we can use the same data type when returning a single value or a list of values, with the latter occurring when we evaluate a list of expressions.

When evaluating a statement, the result can be to either continue or stop the current evaluation:

```
stmt_result = Rcont | Rstop stop
```

The current evaluation can be stopped due to evaluating a return statement or an error:

```
stop = Sret | Serr err_result
```

By factoring out the reason for stopping into a separate type, we make it easier in the future to add support for more statements that cause changes in the control flow, such as `continue` and labeled break statements.

Environments. The semantic environment record contains the entire Dafny program, which is searched when calling functions or methods.

```
dfy_env = ⟪ prog : program ⟫
```

[2] While it is not necessary for array values to hold the length of the array in the one-dimensional case, it is necessary for multi-dimensional arrays (for which we plan to add support in the future), as in some cases we cannot determine the length of inner arrays by indexing to them first if they follow zero-length dimensions.

States. We define the state of the interpreters with the following record:

```
state = ⟪
  clock : num;
  locals : (string × value option) list;
  heap : heap_value list;
  locals_old : (string × value option) list;
  heap_old : heap_value list;
  locals_prev : (string × value option) list;
  heap_prev : heap_value list
⟫
```

To ensure that the functional big-step interpreters are well-defined, terminating functions, the clock keeps track of the number of clock ticks that are still available. Each time we evaluate an expression or statement that could potentially diverge, such as calls or while loops, we first check whether there are ticks still available. If no more ticks are available, the interpreter returns the timeout error as its result. Otherwise, the clock is decremented, and evaluation continues.

The locals field is an association list that keeps track of all declared variables and their values. By using the first match when looking up a variable, we can support shadowing by simply prepending elements. Since Dafny does not require local variables to be initialized immediately when they are declared, we use an option to indicate their initialization status. Recall that trying to read an undeclared or uninitialized variable causes the interpreter to return the Rfail error. As mentioned earlier, the heap field models a list of heap values, where elements are looked up via their (list) index.

The locals_old and heap_old fields hold the values of locals and heap right before executing the body of a method. These are necessary to support the semantics of Dafny's `old`-expressions, which are discussed in the next section.

Finally, the locals_prev and heap_prev fields are used internally by the generated verification conditions, allowing them to refer to a previously set state as needed.

2.2 Expressions and old-expressions

The semantics of Dafny expressions is straightforward. For example, we define the semantics for a unary operation as follows by first evaluating the operand to a value, then applying the unary operator:

```
evaluate_exp st0 env (UnOp uop e) ≝
  case evaluate_exp st0 env e of
    (st1,Rval v) ⇒
      (case do_uop uop v of
         None ⇒ (st1,Rerr Rfail)
       | Some res ⇒ (st1,Rval res))
  | (st1,Rerr err) ⇒ (st1,Rerr err)
```

Evaluating an expression only changes the state by decrementing the clock when function calls are made, which we have proven as a theorem about the semantics. Evaluation order for expressions follows Dafny's short-circuit semantics. In the absence of short-circuiting, we choose to evaluate expressions from left to right, which feels more natural in the context of Dafny.

As a verification-oriented programming language, Dafny provides expressions that are only for verification. These can be used to give hints to the automated verifier (with the `assert` statement) or annotate methods with a pre- and postcondition (with keywords `requires` and `ensures` respectively). In particular, the expressions in these constructs are required to be Dafny expressions that evaluate to boolean values. The postcondition expression for many methods will often need to relate the state of the heaps before and after the execution of the method body. Accordingly, Dafny has special support for `old`-expressions, which allow referring to the state before the body was executed within the postcondition.

To illustrate this, consider the following method that swaps two elements in an array:

```
method Swap(a: array<int>, i: int, j: int)
  requires 0 <= i < a.Length && 0 <= j < a.Length
  ensures a[i] == old(a[j]) && a[j] == old(a[i])
  modifies a
{
  // <- old refers to the state here
  var temp := a[i];
  a[i] := a[j];
  a[j] := temp;
  // <- ensures refers to state here
}
```

As `ensures` refers to the state after executing the method body, the user must use `old` to refer to the array before it was modified.

We define the semantics for `old` as

```
evaluate_exp st env (OldHeap e) ≝
  case evaluate_exp (use_old_heap st) env e of
    (st1,r) ⇒ (unuse_old_heap st1 st,r)

use_old_heap st ≝ st with heap := st.heap_old
unuse_old_heap cur prev ≝ cur with heap := prev.heap
```

In other words, a user-annotated expression wrapped within `old` is evaluated in the heap state with respect to the old heap, which the semantics sets when it enters a callee.

We also track the old locals because the state does not separate track parameters, i.e., they are indistinguishable from other locals, and because Dafny allows shadowing parameters. To illustrate this point, suppose that after swapping the elements in the array, the user writes `var a := 0;` shadowing `a` to be an integer. If we were to take the `ensures` as is, `a` would mistakenly refer to the newly declared integer variable. We avoid this by interpreting references to parameters in `ensures` annotations to be implicitly wrapped in an Old

expression, which, analogous to OldHeap above, evaluates an expression in the old heap and the old locals.

2.3 Statements

As many Dafny programs are written in an imperative style, we also define semantics for statements, which may modify the state of the program.

Skip, Return, and Then. We can easily define the semantics of Skip, Return, and Then:

```
evaluate_stmt st env Skip ≝ (st,Rcont)
evaluate_stmt st env Return ≝ (st,Rstop Sret)
evaluate_stmt st₀ env (Then stmt₁ stmt₂) ≝
  case evaluate_stmt st₀ env stmt₁ of
    (st₁,Rcont) ⇒ evaluate_stmt st₁ env stmt₂
  | (st₁,Rstop stp) ⇒ (st₁,Rstop stp)
```

Recall that the clock is used to prove termination of the interpreter. As the recursive call in the Then case takes a strictly smaller statement as an argument, we do not need to decrement the clock to prove termination.

Assign. Dafny supports parallel assignment, meaning that it is possible to swap the value of two variables by writing, e.g.,

```
x, y := y, x;
```

which we support by first evaluating the expressions on the right-hand side of an assignment, followed by assigning to the left-hand sides from left to right.

Array Allocation. To allocate a (possibly nested) one-dimensional array in Dafny, we can write

```
a := new T[len];
```

where `T` is the type of the array and `len` its length. Note that the allocation on the right-hand side of the assignment does not specify an initial value. According to the Dafny reference, the initial array elements are arbitrary values of type `T`. This is in contrast to our semantics, which supplies a default value for the array elements. For the types supported by our subset, defining a default value is straightforward. The advantage of this tweak is that it avoids the non-deterministic choice of an arbitrary value in the semantics, which otherwise would have to be accounted for, e.g., by making the semantics non-deterministic via an oracle.

Formally, array allocation is defined as appending a heap value HArr xs t to the heap, where xs is a list of len copies of the default value, and returning a value ArrV which, as described in Section 2.1, records the length, location on the heap, and the type of the array. Note that Dafny does not support deallocation.

```
method Find(a: array<int>, key: int)
  returns (i: int)
  ensures 0 <= i ==> i < a.Length && a[i] == key
  ensures i < 0 ==>
    forall k :: 0 <= k < a.Length ==> a[k] != key
{
  i := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length
    invariant
      forall k :: 0 <= k < i ==> a[k] != key
  {
    if a[i] == key { return; }
    i := i + 1;
  }
  i := -1;
}
```

Figure 2. Linear search in Dafny.[4]

While. We define the semantics for While as follows:[3]

```
evaluate_stmt st₀ env
  (While guard invs decrs mods body) ≝
  if st₀.clock = 0 then (st₀,Rstop (Serr Rtimeout))
  else
    case evaluate_exp (dec_clock st₀) env guard of
      (st₁,Rval (BoolV F)) ⇒ (st₁,Rcont)
    | (st₁,Rval (BoolV T)) ⇒
      (case evaluate_stmt st₁ env body of
        (st₂,Rcont) ⇒
          evaluate_stmt st₂ env
            (While guard invs decrs mods body)
      | (st₂,Rstop stp) ⇒ (st₂,Rstop stp))
    | (st₁,Rval guard_v) ⇒ (st₁,Rstop (Serr Rfail))
    | (st₁,Rerr err) ⇒ (st₁,Rstop (Serr err))
```

Note that, unlike Then, the recursive call takes the same while loop as an argument, meaning that the size of the statement being evaluated stays the same. Thus, every time we enter the loop, we need to check and decrement the clock.

It is not strictly necessary to check and decrement the clock immediately on entry; instead, it is possible to check and decrement the clock right before the recursive call. We decided to use the former, as it simplifies the correctness proof for the compiler by more closely matching the semantics of function calls, which is what loops compile to.

Method Call. We avoid showing the full semantics of a method call due to its length. Instead, we use an example.

[3]Note that we do not check that the loop invariants `invs` hold here. As we will see in Section 4, they are considered in the wp-calculus instead. This suggests that our semantics for Dafny is actually a combination of the functional big-step semantics presented here and the sound wp-calculus.
[4]Adapted from https://dafny.org/dafny/OnlineTutorial/guide.

Suppose we want to call a method `Find`, which implements linear search (Fig. 2). We write

```
idx := Find(a, key)
```

as the method call. Note that the caller must assign all out-parameters. The semantics of the method call are as follows:

1. Look up "Find" in the prog field of the environment.
2. Evaluate the parameters `a` and `key` in the caller.
3. Check if the in- and out-parameters `a`, `key`, and `i` are distinct.
4. Change locals to only contain the parameters `a` and `key` initialized with their values, and the uninitialized out-parameter `i`.
5. Copy the changed locals and the current heap to locals_old and heap_old, respectively.
6. Check the clock, and time out if no more ticks remain.
7. Decrement the clock and evaluate the method body, expecting Rstop Sret as the return value.
8. Evaluate the out-parameter `i` in the callee.
9. Restore locals, locals_old, and heap_old to the original values of the caller.
10. Assign the value of the out-parameter `i` to `idx`.

There are two noteworthy points regarding the evaluation of the method body in step 6. First, evaluating the method body requires a recursive call to evaluate_stmt. As the method body can be arbitrarily large, we must check and decrement the clock to be able to prove termination. Second, we require the recursive call to return Rstop Sret; that is, the method body must end with a return statement. Note that if a source program does not explicitly end with return (such as in Fig. 2), the return statement is added automatically by our frontend.

2.4 Programs

Matching Dafny's notion of executable programs, we define the semantics of a program as first checking whether methods and functions have unique names, followed by calling "Main" from an initial state where locals, heap, and their old counterpart are initialized with the empty list.

```
evaluate_program ck (Program members) ≝
  if ¬distinct (map member_name members) then
    (init_state ck,Rstop (Serr Rfail))
  else
    evaluate_stmt (init_state ck)
      (mk_env (Program members))
      (MetCall [] «Main» [])
```

3 Verified Compiler

We now turn to describing our formally verified compiler from Dafny to CakeML, which, together with CakeML's formally verified compiler [59], yields a verified compilation pipeline for Dafny down to machine code. Our overall Dafny compilation pipeline is as follows:

1. Generate an S-expression from an extended version of Dafny's existing frontend.
2. Parse the S-expression into a Dafny AST in HOL4.
3. Remove `assert` statements by replacing them with Skip.
4. Freshen the AST, which updates all variable names to be unique and start with "v", simplifying proofs.
5. Compile the freshened Dafny AST into a CakeML AST, discarding annotations.
6. Compile the CakeML AST to machine code in a verified manner using the CakeML compiler [59].

This pipeline is engineered to maintain compatibility with both Dafny tooling and the CakeML codebase and proofs. We have successfully compiled and run machine code for 18 code examples, including Fibonacci, the 91 function, swapping two elements in an array, and binary search.

In the rest of this section, we focus on describing the Dafny-to-CakeML transformation, namely steps 3 and 4. The compiler for these steps is defined as the function compile

```
compile dfy ≝
  from_program (freshen_program (remove_assert dfy))
```

The technical challenges in implementing and verifying compile include reconciling our choice of left-to-right evaluation order with CakeML's right-to-left order and handling the possibility of early returns from Dafny method calls. We first explain our implementation, then present the compiler correctness statement and proof.

3.1 Compiler Implementation

Let us return to the linear search implementation shown in Figure 2 to describe our compiler. This example showcases many key features of Dafny; in order to compile it, we need to map Dafny's variables, arrays, while loops, (early) returns, and methods to CakeML. Note that neither `ensures` nor `invariant` clauses, nor their contents such as `forall`, are compiled, as they are used only for verification purposes.

Variables. We compile Dafny variables to CakeML references, which are mutable variables and thus semantically equivalent. For example, `i := i + 1` in Dafny is compiled to `i := !i + 1`, where `!` is the dereference operator in CakeML. We postpone the discussion of variable declarations to the compilation of the method signature later in this section.

Arrays. Dafny arrays are compiled to tuples that hold the array and its length for the reason mentioned in Footnote 2. For example, in Figure 2, the array `a` is represented by a corresponding CakeML value of type `(int * int array)`. Thus, `a[i]` is compiled to `Array.sub (snd (!a)) (!i)` and `a.Length` is compiled to `fst (!a)`.

While Loops. We compile while loops as tail-recursive functions, matching the definition of `while` in Standard

ML [44]. The CakeML compiler performs tail-call optimization, i.e., tail-recursive functions do not require allocating a new stack frame with every call, meaning that there is no risk of our compilation of loops causing excessive stack allocation. The loop in Figure 2 is compiled to:

```
let fun loop () =
  if !i < fst (!a) then (
    if Array.sub (snd (!a), !i) = !key
    then raise Return else ();
    i := !i + 1;
    loop ()
  ) else ()
in loop (); ... end
```

Note that we call `loop` at least once. After entering the loop, we check whether the loop condition holds. If it does, we execute the body and recursively tail-call the loop. If the condition does not hold, the expression evaluates to `()`.

Additionally, observe that the Dafny code returns immediately once the key is found. In CakeML, we compile this as raising the `Return` exception. To ensure the existence of this exception, the compiler inserts

```
exception Return;
```

at the beginning of every program. We postpone the discussion of how we handle the `Return` exception from the perspective of the caller until later in the text.

The rest of the body after the loop is compiled as

```
i := ~1;
raise Return
```

Recall that the final (non-early) return line is implicitly inserted by the Dafny frontend at the source level and thereby compiled the same way as all return calls to a CakeML `Return` exception. This insertion ensures that every method call explicitly returns, which is consistent with our semantics.

Methods. Before discussing the compilation of the rest of the method outside the body, it is useful to first discuss the compilation of method calls. Suppose that `idx` is a local variable, and `a` and `key` are bound to some array and integer, respectively. Then, the method call

```
idx := Find(a, key)
```

is compiled to

```
let val t0 = dfy_Find key a
in idx := t0 end
```

Note how we assign to the left-hand side in a separate expression, prepend "dfy_" to the method name, and reverse the order of arguments. While separating the assignment is not necessary in this case, it is necessary when a method has multiple out-parameters, in which case, the method returns a tuple. For example,

```
min, max := MinMax(a)
```

is compiled to

```
let val (t0, t1) = dfy_Find key a
in min := t0; max := t1 end
```

If a method does not have any out-parameters, its return value and the assignment become `()`.

By prepending "dfy_" to method names, combined with the fact that after the freshen pass all user-generated variable names start with "v", the compiler can generate additional names, such as `t0` and `t1` in the preceding examples, which (provably) do not change the semantics of the program.

We reverse the order of arguments to account for the difference in evaluation order between CakeML and Dafny. Instead of reversing the order of arguments, we could have enforced a left-to-right evaluation order using let-bindings. The disadvantage of this approach is that we need to generate as many internal variables as there are arguments and prove that they do not change the semantics. While this is possible and we do similar proofs in other parts of the compiler, they are tedious.

Having a picture of how method calls are compiled, we can now discuss the compilation of the rest of the method outside the body. In particular, the compiled method must return the values of the out-parameters at the end of the method. As a method always explicitly returns, we can simply wrap the compiled body in an exception handler, which reads the out-parameters and returns their value:

```
<compiled body>
  handle Return => !i
```

The last remaining step is to compile the method signature:

```
fun dfy_Find key a =
let
  val key = ref key
  val a = ref a
  val i = ref 0
in ... end
```

As the body refers to the parameters as variables, and we compile variables as references, we have to allocate references for `key`, `a`, and `i`. Note that `key` and `a` are initialized with the respective arguments, whereas the out-parameter `i` (uninitialized in Dafny) is initialized to be `0`.

In general, we compile variable declarations, including out-parameters, as references that are initialized with `0` regardless of their type. This can produce CakeML code that does not pass CakeML's type checker but allows us to avoid an extra layer of boxing, e.g., by storing references to options. However, in some cases, this compilation scheme still introduces unnecessary assignments. In the future, we plan to implement a compiler pass that eliminates such redundant assignments where possible, which would allow further optimizations such as removing references in cases where a variable is only read.

Note that generating potentially type-incorrect CakeML is not necessarily a problem in terms of correctness for two

```
exception Return;

(* fst and snd are inlined in the compiler *)
fun fst (x, y) = x;
fun snd (x, y) = y;

fun dfy_Find (key: int) (a: int * (int array)) =
let
  val key = ref key
  val a = ref a
  val i = ref 0
in
  (i := 0;
   let fun loop () =
     if !i < fst (!a) then (
       if Array.sub (snd (!a), !i) = !key
       then raise Return else ();
       i := !i + 1;
       loop ()
     ) else ()
   in
     loop ();
     i := ~1;
     raise Return
   end)
  handle Return => !i
end;
```

Figure 3. Result of compiling `Find` to CakeML

reasons: first, the correctness theorem of the CakeML compiler applies to all programs with well-defined semantics, regardless of their type correctness. Second, as a valid Dafny program will always assign variables before reading them, the generated CakeML program will also have well-defined semantics. We will formalize a more general version of this argument with the correctness of our compiler in Section 3.2.

Figure 3 shows the final compilation result for `Find`.

3.2 Compiler Correctness

Informally, our correctness theorem states that evaluating a Dafny program using Dafny's functional big-step semantics is equivalent to evaluating the compiled Dafny program using CakeML's functional big-step semantics, according to a notion of equivalence we define.

3.2.1 Equivalence. Recall from Section 2 that the functional big-step semantics returns a state and a result. Step-by-step, we will build up our notion of equivalence for values, results, states, and environment, which we will use for our correctness theorems.

Value Relation. We define equivalence between Dafny (left) and CakeML (right) values using the inductive val_rel, whose definition is straightforward for integers, booleans, and strings. For example, it is defined as follows for integers:

$$\vdash \mathsf{val_rel}\ m\ (\mathsf{IntV}\ i)\ (\mathsf{Litv}\ (\mathsf{IntLit}\ i))$$

Array values in Dafny are equivalent to a CakeML tuple containing the length of an array and its location in CakeML's store. We require that the lengths match, converting between `num` and `int` as necessary, and that the locations are related through the map `m`.

$$\begin{aligned}&\vdash \mathsf{lookup}\ m\ loc = \mathsf{Some}\ loc' \Rightarrow\\ &\quad \mathsf{val_rel}\ m\ (\mathsf{ArrV}\ len\ loc\ ty)\\ &\qquad (\mathsf{Tuplev}\ [\mathsf{Litv}\ (\mathsf{IntLit}\ (\&len));\ \mathsf{Loc}\ \mathsf{T}\ loc'])\end{aligned}$$

Result Relation. The equivalence between Dafny and CakeML expression results is built on val_rel:

$$\begin{aligned}&\mathsf{exp_res_rel}\ m\ (\mathsf{Rval}\ dfy_v)\ (\mathsf{Rval}\ [cml_v]) \stackrel{\text{def}}{=}\\ &\ \mathsf{val_rel}\ m\ dfy_v\ cml_v\end{aligned}$$

For statements that execute normally in Dafny, we only require that the corresponding CakeML expression return some value. In the case of return, we check that the correct `Return` exception has been raised in CakeML:

$$\begin{aligned}&\mathsf{stmt_res_rel}\ \mathsf{Rcont}\ (\mathsf{Rval}\ v_0) \stackrel{\text{def}}{=} \mathsf{T}\\ &\mathsf{stmt_res_rel}\ (\mathsf{Rstop}\ \mathsf{Sret})\ (\mathsf{Rerr}\ (\mathsf{Rraise}\ val)) \stackrel{\text{def}}{=}\\ &\ \mathsf{is_ret_exn}\ val\end{aligned}$$

State Relation. We define the equivalence between a Dafny and CakeML state as follows:

$$\begin{aligned}&\mathsf{state_rel}\ m\ l\ s\ t\ cml_env \stackrel{\text{def}}{=}\\ &\ s.\mathsf{clock} = t.\mathsf{clock} \wedge \mathsf{array_rel}\ m\ s.\mathsf{heap}\ t.\mathsf{refs} \wedge\\ &\ \mathsf{locals_rel}\ m\ l\ s.\mathsf{locals}\ t.\mathsf{refs}\ cml_env.\mathsf{v}\end{aligned}$$

Here, s and t are the Dafny and CakeML states, respectively. By requiring the clocks to be equal, we can prove that a Dafny program diverges if and only if the corresponding CakeML program diverges.

The array_rel relation requires that for every array on Dafny's heap, the mapping m provides the location of the corresponding array on CakeML's store and additionally requires that their contents satisfy val_rel.

The locals_rel relation requires that for every local variable in Dafny, the mapping l provides the corresponding reference in CakeML, which must be bound to the same name in the CakeML environment (cml_env). If the local variable is initialized in Dafny, the values must be equivalent with respect to val_rel.

Environment Relation. We define env_rel to specify both a notion of equivalence between Dafny's and CakeML's environments and additional well-formedness conditions on the environments. First, we require the presence of basic functions and constructors such as `True`, `False`, and `::` (cons) in CakeML. Additionally, we require that the program in Dafny's environment be well-formed, and for each

of its members, the corresponding function must exist in CakeML's environment.

3.2.2 Compiler Correctness Statement. We formalize the correctness statement described informally at the beginning of this section:

```
⊢ evaluate_program dfy_ck prog = (s,Rcont) ∧
  compile prog = inr cml_decs ⇒
   ∃ ck t' m' r_cml.
    evaluate_decs (cml_init_state ffi (dfy_ck + ck))
     (cml_init_env ffi) cml_decs =
     (t',r_cml) ∧
    state_rel m' empty s t' (cml_init_env ffi) ∧
    stmt_res_rel Rcont r_cml
```

This is a forward simulation result, which suffices to show that the (deterministic) semantics of Dafny programs are preserved in the output CakeML programs. Note that the correctness statement only talks about Dafny programs that terminate successfully (Rcont). We can use the VCG to be sure that a given program successfully terminates, which is implied by the correctness of the VCG and the soundness of the weakest precondition calculus (Section 4.2). In Section 5, we will combine this fact with the compiler theorem in the context of the 91 function. As indicated by the sum value, the Dafny-to-CakeML compiler can fail, for example, if the compiler is asked to compile a `forall`-expression; compiler correctness only applies when it succeeded.

To understand why we existentially quantify over additional clock ticks *ck* in the CakeML initial state, recall that state_rel requires Dafny's and CakeML's clocks to be the same. However, in some cases it may be necessary to compile Dafny code into CakeML code that requires (finitely) more ticks in the semantics. For example, a Dafny method with *n* parameters is compiled to a curried function, meaning that a call to that method requires at least one tick in Dafny's semantics, whereas it requires at least *n* ticks in CakeML's semantics.

Correctness of Statement Compilation. A major component of the top-level correctness proof is the correctness of the compiler for statements. Omitting some technical details, it is stated as

```
⊢ evaluate_stmt s env_dfy stmt_dfy = (s',r_dfy) ∧
  r_dfy ≠ Rstop (Serr Rfail) ∧
  from_stmt stmt_dfy lvl = inr e_cml ∧
  state_rel m l s t env_cml ∧
  env_rel env_dfy env_cml ∧ is_fresh_stmt stmt_dfy ∧
  no_shadow (set (map fst s.locals)) stmt_dfy ⇒
   ∃ ck t' m' r_cml.
    evaluate (t with clock := t.clock + ck) env_cml
      [e_cml] = (t',r_cml) ∧
    state_rel m' l s' t' env_cml ∧ m ⊑ m' ∧
    stmt_res_rel r_dfy r_cml
```

Note the similarities to the top-level correctness statement. Here, we prove that evaluating a Dafny statement and its compiled counterpart results in a state and result that are equivalent under their respective relations; we also permit the CakeML semantics to begin with more clock ticks.

The main difference is that the assumptions of this latter statement are more general. In particular, we universally quantify over all states and environments that satisfy state_rel and env_rel, respectively. Additionally, we assume the properties that are established by the freshen pass, namely that variable names are unique and begin with "v", which is captured by is_fresh_stmt and no_shadow. This generality allows us to prove compiler correctness using the induction principle arising from the termination proof of the functional big-step semantics for Dafny.

4 Verified VCG

This section presents our formally verified verification condition generator (VCG), which, given a Dafny program, produces a list of verification conditions as Dafny expressions. Verifying the output expressions shows functional correctness of Dafny programs and, moreover, guarantees well-definedness as assumed by our compiler. A key challenge is to design the VCG to support sound analysis for loop framing and termination, which, in particular, enables verification for subtle examples like the 91 function (Fig. 1).

We approached the VCG formalization in two phases:

1. Define a weakest precondition calculus for Dafny as an inductive relation in HOL4 and prove it sound with respect to the Dafny semantics.
2. Define a concrete VCG implementation function that checks the program and emits verification conditions as Dafny expressions; this step is proved correct with respect to the wp-calculus.

This two-phase methodology is also applied when extending the wp-calculus with new rules or refining existing ones. By splitting the VCG into a calculus and a function, we can focus on clearly and concisely specifying the wp-rules in the calculus while leaving potential performance considerations to the implementation of the VCG function. Mirroring our development process, we will first describe our calculus and its soundness, followed by the VCG and its correctness.

4.1 Weakest Precondition Calculus

We implement the wp-calculus as an 8-place inductive relation for Dafny statements:

stmt_wp *m reqs stmt post ens decs mods ls*

In order of appearance, the parameters are: the set of available methods *m*, the preconditions *reqs*, a Dafny statement *stmt*, the postconditions *post* and *ens*, a termination measure *decs*, a list of locations that may be modified *mods*, and a list of defined locals and their types *ls*. The parameters *reqs*, *post*, *ens*, *decs*, and *mods* are lists of expressions, matching

the type of user annotations in the input program. The rules of our calculus are more easily understood if one considers the parameter *reqs* as the output and all the other parameters as inputs, which, as we will see later, is mirrored by the implementation of the VCG.

We distinguish two kinds of postconditions, where the parameter *ens* determines the conditions that must hold upon statement return (i.e., Rstop), while *post* determines the conditions that must hold assuming we continue executing normally (i.e., Rcont). This is necessary to support early-return semantics, and the difference is easily illustrated by the definition of stmt_wp for the Return and Skip cases, which requires their preconditions to be ens and post, respectively:

$$\vdash \text{stmt_wp } m \; ens \; \text{Return} \; post \; ens \; decs \; mods \; ls$$
$$\vdash \text{stmt_wp } m \; post \; \text{Skip} \; post \; ens \; decs \; mods \; ls$$

We can also easily express the relation in the case of statement composition:

$$\begin{aligned}\vdash\ & \text{stmt_wp } m \; pre_1 \; s_1 \; pre_2 \; ens \; decs \; mods \; ls \wedge \\ & \text{stmt_wp } m \; pre_2 \; s_2 \; post \; ens \; decs \; mods \; ls \Rightarrow \\ & \text{stmt_wp } m \; pre_1 \; (\text{Then } s_1 \; s_2) \; post \; ens \; decs \; mods \; ls\end{aligned}$$

The verification condition lists are interpreted conjunctively, so to add a verification condition, we can simply prepend it to the precondition, as in the case of user assertions:

$$\vdash \text{stmt_wp } m \; (e{::}post) \; (\text{Assert } e) \; post \; ens \; decs \; mods \; ls$$

Note that there is an implicit well-formedness check for verification conditions: since they are expressions, to prove their validity we must show that they evaluate to True in all states, which is impossible if an expression is not well-formed, as its evaluation would fail in some state.

Assign. A classical wp-rule for assignment would perform variable substitution, which can be tricky to get right, in particular in the presence of parallel assignment. Additionally, if variables are substituted for large expressions, the generated conditions can explode in size [23]. We avoid both of these issues by using a let-expression instead, for which we have defined functional big-step semantics similar to the assignment statement.

To illustrate this, suppose we want to determine the weakest precondition for the statement

```
b := a + a + a;
x, y := b + b + b, c;
```

and the postcondition `x < y`. In the case of variable substitution, the intermediate result stemming from Then is

```
b + b + b < c
```

with the final result being

```
a + a + a + a + a + a + a + a + a < c
```

With our approach of using `let`, the intermediate result is

```
let x = b + b + b, y = c in
  x < y
```

with the final result being

```
let b = a + a + a in
let x = b + b + b, y = c in
  x < y
```

Formally, the complete rule for assignment is as follows:

$$\begin{aligned}\vdash\ & \text{map fst } l = \text{map VarLhs } ret_names \wedge \\ & \text{map snd } l = \text{map ExpRhs } exps \wedge \text{distinct } ret_names \wedge \\ & \text{every } (\lambda\, v.\ \neg\text{mem } v \; mods) \; ret_names \wedge \\ & \text{length } exps = \text{length } ret_names \wedge \\ & \text{set } ret_names \subseteq \text{set (map fst } ls) \wedge \\ & \text{get_types } ls \; exps = \text{inr } rhs_tys \wedge \\ & \text{get_types } ls \; (\text{map Var } ret_names) = \text{inr } lhs_tys \wedge \\ & lhs_tys = rhs_tys \Rightarrow \\ & \quad \text{stmt_wp } m \; [\text{Let (ZIP } (ret_names, exps)) \; (\text{conj } post)] \\ & \quad\ (\text{Assign } l) \; post \; ens \; decs \; mods \; ls\end{aligned}$$

The precondition is now a Let expression that binds the names being assigned to their respective expressions in the scope of the postcondition. The rule also performs several syntactic well-formedness and type correctness[5] checks on assignments, which is necessary for soundness; these checks are usually added/modified as we discovered their necessity when formalizing soundness of the calculus. We omit further discussion of these checks for brevity.

Array Update. We also support array updates such as

```
a[idx] := e;
```

Similar to the assignment rule, the rule for array updates performs several syntactic well-formedness and type correctness checks. In particular, it syntactically checks that `a` is part of the `modifies` clause. As a consequence, only variables can be included in the `modifies`, and not expressions such as `a[idx]`; we aim to lift this restriction in the future.

Omitting some well-formedness checks, the weakest precondition for the example above is:

```
0 <= idx /\ idx < a.Length /\
SetPrev (ForallHeap [a]
  (a[Prev idx] = (Prev e) /\
   forall i: int ::
     (i != (Prev idx) /\
      0 <= i /\ i < a.Length ==>
        a[i] = PrevHeap (a[i]))
   ==> post))
```

The expressions SetPrev, Prev, PrevHeap, and ForallHeap are our extensions to provide sufficient expressivity for verification conditions. The expression ForallHeap quantifies over all heaps where previously allocated locations outside `a` remain unchanged, including heaps with newly allocated locations. More specifically, the argument to ForallHeap is a list of locations that are havoced and thus must be a subset of what is provided in the `modifies` clause. By using Prev

[5]While VCG and type checking are typically performed as separate passes, we have combined them into a single phase for this work.

and PrevHeap within SetPrev, expressions are evaluated at the point of SetPrev, with *locals* and *heap* appropriately set. The index must be wrapped in Prev to correctly handle cases such as `a[a[idx]] := a[idx]`. Together, the verification condition frames the unmodified state around the update.

Array Allocation. To understand the wp-rule for array allocation, consider

```
a := new T[len]; <next statement>
```

Observe that the example above is a sequence of statements, with the first statement assigning the newly allocated array to `a`. This is because the wp-rule for array allocation is a special case of the wp-rule for statement composition: subsequent statements may modify `a`, so it must be added to the list of locations that the following statement may modify. Note that requiring allocation to be followed by another statement does not limit expressivity: statements only occur in methods, which, as previously mentioned, must explicitly return using a Return statement, meaning that allocation never occurs on its own.

Again, omitting some well-formedness checks, the weakest precondition for the example above is:

```
0 <= len /\
SetPrev (ForallHeap [] (forall a: array<T> ::
  (a.Length = (Prev len) /\
   let x = (Prev default_e) in
     (forall i: int ::
        (0 <= i /\ i < a.Length ==> a[i] = x))
   ==> wp_next_stmt)))
```

Note that it uses the constructs Prev and ForallHeap, which were introduced in the context of array updates. However, in contrast to array updates, the list passed to ForallHeap is empty, meaning that, except for potentially newly allocated locations, the heap is unchanged. Recall from Section 2.3 that we define arrays to be initialized with some default value. This fact is expressed in the verification condition using the let-expression. Finally, wp_next_stmt is the weakest precondition of the statement following the allocation. It is the same as in the general rule for statement composition, except that *a* has been added to its *mods*.

While. Fundamentally, our definition of the wp-rule for while loops is the same as the standard Hoare Logic rule with annotated invariants. However, our definition must additionally account for termination and framing to ensure that the loop preserves properties of variables not modified by its body. To illustrate these points, consider SumToN (Fig. 4), which includes the `decreases` clause automatically guessed by the Dafny frontend. Keeping it high-level, the weakest precondition for the while loop is:

```
invariant (holds on entry) /\
invariant is maintained /\
forall sum, i ::
  !(i <= n) /\ invariant ==> ensures
```

```
method SumToN(n: int) returns (sum: int)
  requires n >= 0
  ensures sum == n * (n + 1) / 2
  ensures n >= 0
{
  var i := 1;
  sum := 0;

  while i <= n
    invariant 1 <= i <= n + 1
    invariant sum == (i - 1) * i / 2
    decreases n - i
  {
    sum := sum + i;
    i := i + 1;
  }
}
```

Figure 4. Summing to n in Dafny.

Note that in the final conjunct, where we prove the postcondition, we only quantify over `sum` and `i`, which we have determined by syntactically checking which locals are assigned to. In particular, we do not quantify over `n`, effectively framing the property `n >= 0`. Since the loop does not include a `modifies` clause, the heap is unchanged.

To ensure that the loop terminates, we need to check whether the expression in `decreases` actually decreases. For this, we need to be able to refer to the value of `decreases` at the beginning of an iteration, which we achieve by storing that value in fresh variables.

If we were to explicitly apply this transformation to the loop body in Figure 4, we would get

```
var prev_n_i := n - i;
sum := sum + i;
i := i + 1;
```

It should be noted that we do not explicitly apply this transformation. Instead, the wp-calculus quantifies over a list of variables that are sufficiently unique.

Knowing this, we can now discuss where the `decreases` check happens, namely as part of checking that the invariant is maintained

```
i <= n /\ invariant ==> body_wp
```

where `body_wp` represents the weakest precondition of the loop body and the postcondition

```
invariant /\
n - i < prev_n_i /\
0 <= prev_n_i /\ 0 <= n - i
```

Method Call. The weakest precondition for method calls must make sure that the precondition of the call is satisfied, recursive calls decrease the termination measure in `decreases`, and that the postcondition of the method call

implies the weakest precondition of the rest of the statements. An (optional) `modifies` clause indicates locations on the heap that may have changed.

For example, the weakest precondition of the first call in the 91 function (Fig. 1) is

```
// precondition
let n = n + 11 in True /\
// decreases
(let n = n + 11 in 111 - n) < old(111 - n) /\
0 <= (let n = n + 11 in 111 - n) /\
0 <= old(111 - n) /\
// postcondition implies wp for the rest
forall tmp: int ::
  (let n = n + 11, r = tmp in
     r == if n <= 100 then 91 else n - 10)
   ==> ...
```

We can simplify the condition by noticing that, at the method call, `n <= 100` holds, and that because the method does not update `n`, we know that `old(n)` is equal to `n`.

Applying substitution and basic simplification rules to the last conjunct, we get:

```
forall tmp: int ::
  tmp == if n <= 89 then 91 else n + 1
  ==> ...
```

By considering the fact that `n <= 100`, we know that `tmp` is a number that is between 91 and 101. Hence, it is less than 111, meaning that the value of `111 - tmp` is non-negative. Considering what we know about `tmp` from the postcondition of the call, we also know that it is larger than `n`. Thus, the next recursive call will pass the `decreases` check. From the postcondition and the range of `tmp`, we know that `r` will be exactly 91, proving the postcondition of the method for `n <= 100`.

4.2 Weakest Precondition Soundness Statement

The soundness of our weakest precondition calculus consists of a top-level soundness theorem for methods, which uses a soundness result for stmt_wp.

Informally, the soundness theorem for methods states that if for every method its precondition (`requires`) implies the weakest precondition (stmt_wp) of its method body, then each method will adhere to its specification whenever started from a state satisfying its precondition.

This top-level method soundness theorem is proved by induction on a well-founded order in such a way that we get an inductive hypothesis stating that any method call using an input state that is smaller, according to the well-founded order, behaves according to its specification. The order we use is a lexicographic order on the "level" of a method and the values of its `decreases` clause. The level of a method is determined using a topological sort on the call graph of a program. Thus, if methods are (mutually) recursive, they have the same level and may only call each other if the values of the `decreases` clause actually decrease at calls. Otherwise, the callee has a lower level than the caller, and no `decreases` clause is required.

The soundness proof of methods uses the following soundness theorem for the weakest precondition for statements. One can read this soundness theorem as saying: if *reqs* holds in the initial state *st*, and all methods in *m* can be found in the environment *env*, then evaluating *stmt* will terminate and result in a state where *post* or *ens* is satisfied, depending on whether the statement returns.

$$
\begin{aligned}
&\vdash \text{stmt_wp } m\ reqs\ stmt\ post\ ens\ decs\ ls \Rightarrow \\
&\quad \text{conditions_hold } st\ env\ reqs \wedge \\
&\quad \text{compatible_env } env\ m \wedge \ldots \Rightarrow \\
&\quad \exists\, st'\ ret. \\
&\qquad \text{eval_stmt } st\ env\ stmt\ st'\ ret \wedge \\
&\qquad \text{case } ret \text{ of} \\
&\qquad\quad \text{Rcont} \Rightarrow \text{conditions_hold } st'\ env\ post \\
&\qquad \mid \text{Rstop Sret} \Rightarrow \text{conditions_hold } st'\ env\ ens \\
&\qquad \mid \text{Rstop (Serr } v_3\text{)} \Rightarrow \text{F} \wedge \\
&\qquad \ldots
\end{aligned}
$$

Note that this is a total correctness result—the fact that evaluating *stmt* will terminate is hidden in the eval_stmt relation (more details below), which requires there to exist clocks such that evaluation does not time out.

The soundness theorem above has some of its assumptions omitted (…) because the assumptions include the lengthy inductive hypothesis regarding method calls: specifically, that we can assume that every method that we might call adheres to its specification if the call is made in a state where the well-founded order used in the method soundness proof has decreased. The fact that the measure decreases is something we get to know from the verification conditions we generate for each call site.

To prove the soundness of stmt_wp, we proceed by induction over the definition of stmt_wp, showing that each wp-rule is locally sound. For its proof, it is more convenient to work with a relational-style semantics, as opposed to the functional big-step style used in the compiler proof.

To bridge the relational and functional styles, we defined the relations eval_exp and eval_stmt (shown below), which effectively quotient out the functional big-step clocks in evaluate_exp and evaluate_stmt, respectively, by existential quantification.

$$
\begin{aligned}
&\text{eval_exp } st\ env\ e\ v \stackrel{\text{def}}{=} \\
&\quad \exists\, ck_1\ ck_2. \\
&\qquad \text{evaluate_exp } (st \text{ with clock } := ck_1)\ env\ e = \\
&\qquad\quad (st \text{ with clock } := ck_2, \text{Rval } v) \\
&\text{eval_stmt } st\ env\ body\ st'\ ret \stackrel{\text{def}}{=} \\
&\quad \exists\, ck_1\ ck_2. \\
&\qquad \text{evaluate_stmt } (st \text{ with clock } := ck_1)\ env\ body = \\
&\qquad\quad (st' \text{ with clock } := ck_2, ret) \wedge \\
&\qquad ret \neq \text{Rstop (Serr Rtimeout)}
\end{aligned}
$$

This allows us to make further relational big-step-style definitions and also to prove lemmas about these relations. For example, given an expression *e*, eval_true asserts that there exist clocks such that *e* evaluates to true; in fact, conditions_hold (used in the statement of the soundness theorem) is also formally defined with respect to this relation.

$$\text{eval_true } st\ env\ e \stackrel{\text{def}}{=} \text{eval_exp } st\ env\ e\ (\text{BoolV T})$$
$$\text{conditions_hold } st\ env \stackrel{\text{def}}{=} \text{every } (\text{eval_true } st\ env)$$

We can state the relational big-step-style semantics for the "then" branch of an if-statement as

$$\vdash \text{eval_true } st\ env\ grd \wedge \text{eval_stmt } st\ env\ thn\ st_1\ ret \Rightarrow \text{eval_stmt } st\ env\ (\text{If } grd\ thn\ els)\ st_1\ ret$$

Similar relational big-step style lemmas about the definitions eval_exp and eval_stmt are used throughout the soundness proof.

4.3 Verified VCG Implementation

The final step in our development of a verified VCG for Dafny is to define a function that generates verification conditions. The implementation follows naturally from the definition of the wp-calculus and is written in a result monad. For example, several clauses of the generator are shown below, corresponding to the wp-calculus snippets defined earlier.

```
stmt_vcg m Return post ens decs mods ls ≝ inr ens
stmt_vcg m Skip post ens decs mods ls ≝ inr post
stmt_vcg m (Then s1 s2) post ens decs mods ls ≝
 case dest_ArrayAlloc s1 of
  None ⇒
   do
    pre' ← stmt_vcg m s2 post ens decs mods ls;
    stmt_vcg m s1 pre' ens decs mods ls
   od
 | Some ... ⇒ ...
stmt_vcg m (Assert e) post ens decs mods ls ≝
 inr (e::post)
stmt_vcg m (Assign ass) post ens decs mods ls ≝
 let (lhss,rhss) = UNZIP ass
 in
  do
   vars ← result_mmap dest_VarLhs lhss;
   es ← result_mmap dest_ExpRhs rhss;
   assert (distinct vars)
    «stmt_vcg:Assign: variables not distinct»;
   assert (list_disjoint vars mods)
    «stmt_vcg:Assign: assigning to mods»;
   ...
   inr [Let (ZIP (vars,es)) (conj post)]
  od
```

Note that in the case of Then, we split on whether the first statement allocates an array, matching our description of the wp-rule for array allocation as a special case of statement composition.

We proved the correctness of stmt_vcg by showing that it only produces verification conditions that can be derived using the wp-calculus:

$$\vdash \text{stmt_vcg } m\ stmt\ post\ ens\ decs\ mods\ ls = \text{inr } res \Rightarrow \text{stmt_wp } (\text{set } m)\ res\ stmt\ post\ ens\ decs\ mods\ ls$$

While we have focused on the VCG for statements in this section, our mechanization also includes a formalization of the VCG for methods. In particular, the generated verification condition for methods requires that the `requires` clause of a method imply the weakest precondition of the body.

5 Putting It All Together

To demonstrate that our formally verified tools work in unison, we have used them to compile and verify the 91 function in an end-to-end manner. As a brief reminder, the complete workflow is as follows.

1. We use an (untrusted) Dafny frontend to produce an S-expression for the 91 function.
2. The S-expression is parsed in HOL4, and we use HOL4's in-logic evaluation to generate verification conditions from stmt_vcg.
3. Since the verification condition is a Dafny expression, we prove its validity by showing that the expression evaluates to True in all states. We have mechanized this proof, which involves expanding the semantics and using HOL4's decision procedure for integers.
4. Finally, we combine the validity of the verification condition, the soundness of the wp-calculus, the correctness of the VCG, and the correctness of the compiler to prove that the generated CakeML function satisfies the specification of the 91 function:

$$\vdash \text{compile_member mccarthy} = \text{inr } mccarthy_cml \wedge \text{clos_env_ok } clos_env \Rightarrow \text{AppReturns } (\text{INT } n)\ (\text{Recclosure } clos_env\ [mccarthy_cml]\ \text{"dfy_M"})\ (\text{INT } (\text{if } n \leq 100 \text{ then } 91 \text{ else } n - 10))$$

The assumption clos_env_ok ensures that basic functions and constructors are available to the method, while the predicate AppReturns [14] is defined in the context of the CakeML translator [47] and best understood as a Hoare triple. Here, it says that applying the (recursive) closure of the CakeML 91 function to an input CakeML integer n returns a CakeML integer that is either 91 if $n \leq 100$ or $n - 10$ otherwise. Notice that this result is verified entirely for CakeML program semantics and does not involve the Dafny toolchain(s).

We can apply both the compiler and the VCG to other methods, including the previously presented methods `Swap`, `Find`, and `SumToN`. Out of these methods, we have also proved the validity of the verification condition for `Swap`. We have not attempted to prove the verification conditions of the

other methods, as the manual approach does not scale well. Our aim is to improve this in future work by adding a path to SMT, allowing the use of SMT solvers to automatically prove the validity of verification conditions.

Similarly, we hope to minimize the proof effort needed to carry the functional correctness guarantees of verified Dafny methods to their CakeML counterparts, as we have manually done for the 91 function.

6 Related Work

We start with an overview of related work at the intersection of formalization and verification-aware programming platforms like Dafny, F*, and Why/Why3. This is followed by a discussion of the broader literature on verified compilation and verified verification condition generation/checking.

6.1 Verified Verification-Aware Programming

Dafny. To the best of our knowledge, verification condition generation and compilation for Dafny has not been done in a foundational way before. Nezamabadi and Myreen [48] implemented a Dafny-to-CakeML compiler for a subset of Dafny that is larger than the one we present here, but only proved correctness for the compilation of some binary expressions. There have also been efforts to implement Dafny's compilers in Dafny [16], with the initial target being a purely functional subset of Dafny.

Dafny targets the intermediate verification language Boogie [6, 40] in order to generate verification conditions; there has been work by Parthasarathy et al. [50] on validating Boogie's outputs using Isabelle/HOL.

Gladshtein et al. [27] present a framework for foundational, multi-modal program verifiers, which they instantiate, among other things, to reason about Dafny-style programs. In contrast to the deep embedding presented in this work, they use a monadic shallow embedding in Lean.

F*. F* [57] is a proof-oriented programming language that, like Dafny, supports automated reasoning about programs using SMT solvers. However, while Dafny mostly follows an imperative style, F* encourages a higher-order functional programming with effects style and has a dependent type system. Strub et al. [56] have implemented a type checker for F* in F* that returns type derivations. In particular, it can return the type derivation for typechecking itself, which in turn can be checked in Rocq.

F* is also a popular target for embedding domain-specific programming languages: Low* [53] is a low-level programming language that is shallowly embedded into F* and has a paper formalization of extracting it to Clight [9], which can be compiled by the CompCert [42] compiler, a verified C compiler that has been formalized in Rocq. Steel [26] is a concurrent programming language that is shallowly embedded into F*. It is based on the concurrent separation logic SteelCore [58] and has verified verification condition generation mechanized in F*. There has also been work on embedding a subset of the x64 assembly language in F* [25], which includes an embedding of Vale [10], an automated verifier for high-performance assembly code that builds on top of existing verification languages like Dafny and F*, and a verified and efficient VCG. To our knowledge, these embeddings have not been verified in a foundational proof assistant.

Why3 and Why. Why3 [21] is a platform for deductive program verification. It provides an ML dialect called WhyML for which it can generate verification conditions. Unlike Dafny, it does not automatically discharge the generated verification conditions. Instead, users can choose between automated and interactive provers. To generate simpler verification conditions, it statically controls aliasing using its type system [22]. Why is a verification condition generator for a "WHILE" language [20]. Similar to Why3, Why can output its verification condition to automated and interactive provers. It has inspired work by Herms et al. [31], where they have implemented a VCG that can produce verification conditions for multiple provers and proven it sound in Rocq.

6.2 Verified Compilation and Verified Verification Condition Generation

Combining Compilation and Verification. The Verified Software Toolchain (VST) [4] project combines CompCert with static analysis tools for invariant generation and verified invariant checking for an end-to-end verified toolchain for C programs. It has been used, for example, to verify the OpenSSL implementation of SHA-256 [5] and HMAC with SHA-256 [8].

CakeML provides a verified implementation of Characteristic Formulae for ML [15, 19, 30, 51], which allows users to state specifications in the style of separation logic and prove them correct using tactics within HOL4.

seL4 [36] is a verified microkernel written in C that has been formalized in Isabelle/HOL. Its compilation down to machine code has been proven correct using a translation validation approach [55]. More specifically, it makes use of a proof-producing decompilation of the binary [45, 46], a formalization of the machine architecture [24], and SMT solvers to show that the binary is a refinement of its C semantics. It has also made extensive use of the VCG provided by Simpl to prove refinement between its executable, monad specification, and high-performance C implementation [63].

Verification Condition Generation. Over the years, there have been multiple works on verification condition generation that were built as tactics in interactive theorem provers such as HOL or Isabelle: Gordon [28] presented a shallow embedding of a simple imperative language, for which he mechanically derived a Hoare logic and developed tactics to generate verification conditions.

Agerholm [2] presented a shallow embedding of an imperative language with non-determinism and loops, for which

he formalized weakest precondition semantics and implemented a verification condition generator using tactics.

Homeier and Martin [32] presented a deep embedding of a standard while loop language including expressions with side effects, for which they proved sound axiomatic semantics and used the axiomatic semantics to define a verified verification condition generator.

Huisman and Jacobs [33] formalized the semantics of an imperative language including loops and abrupt termination (e.g., return and continue) in higher-order logic, provided an extension of Hoare logic with abrupt termination, and used it in the context of Java with the proof tool PVS. Its approach is reminiscent of our combination of functional big-step semantics and a weakest precondition calculus.

Schirmer [54] presented Simpl, a model for sequential imperative programming languages, including a proven sound and complete Hoare logic for both partial and total correctness and a verification condition generator embedded as an Isabelle tactic.

There has also been work on verifying optimized verification condition generation: Vogels et al. [62] used Rocq to formalize an efficient VC generation algorithm [23, 39] that avoids an exponential blowup. Grégoire and Sacchini [29] formalized in Rocq the use of static analysis to simplify generated verification conditions for Java bytecode.

Verified End-to-End Compilers. PureCake [35] is a verified compiler for a Haskell-like language, which, similar to our Dafny compiler, is built on top of CakeML. Pancake [52] is a systems programming language with a verified compiler, which reuses the lower parts of the CakeML backend.

CertiCoq [3] compiles Gallina, the specification language of Rocq, to C, which can be compiled using CompCert to form an end-to-end verified compiler. Note that while CertiCoq's components have been verified, proving a composed end-to-end correctness theorem is still ongoing work [37].

Vélus [11] is a verified compiler for synchronous dataflow languages like Lustre [12] and is built on top of CompCert.

In the context of the CompCert project, the translation validation approach has been employed for an SSA-based middle-end [7], instruction scheduling [60], and software pipelining [61].

7 Conclusion

We have presented a verified compiler and verified verification condition generator based on functional big-step semantics for an imperative subset of Dafny. Our subset includes mutually recursive method calls, while loops, and arrays, which is expressive enough to support interesting examples such as McCarthy's 91 function and array-based programs that are used when teaching Dafny. Together, our formalization shows that it is possible to obtain foundational correctness guarantees across the entire toolchain for verification-aware programming languages.

Future Work. Beyond the contributions described in this paper, we believe that our work provides a scaffolding that can be expanded in different directions: feature support, automation, and integration with the CakeML ecosystem.

By improving feature support and automation, we can bring our implementation closer to a drop-in replacement for the current Dafny toolchain. Initial steps in this regard could be to support Dafny's compiler litmus test suite and to implement and verify a connection to SMT, allowing the use of automatic SMT solvers. Supporting Dafny's compiler litmus test suite will most likely involve building support for Dafny's object-oriented features (traits and classes) as well as functional features (first-class function values and algebraic data types). The latter features have straightforward compilation targets in CakeML, and ideas to compile traits and classes have been very briefly sketched in previous work [48].

Furthermore, a deeper integration of our work with the CakeML ecosystem could allow CakeML code to call Dafny code, and vice versa. Thus, it may enable the input and output of Dafny programs to be placed on solid foundations. More generally, a program could simultaneously make use of Dafny's verification automation, CakeML's HOL4 frontend [1, 47], and Hoare-style reasoning using separation logic [19, 30, 51], without compromising on trustworthiness.

Acknowledgments

We thank Clément Pit-Claudel for early discussions of this work, Fabio Madge for answering our questions about Dafny, Rustan Leino for discussions about Dafny at LICS 2025, and the anonymous reviewers for their comments. This work was supported by an Amazon Research Award and NTU Singapore's Global Connect Fellowship. The second author was partially funded by Swedish Research Council grant 2021-05165.

References

[1] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. 2020. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *J. Autom. Reason.* 64, 7 (2020), 1287–1306. doi:10.1007/S10817-020-09559-8

[2] Sten Agerholm. 1991. Mechanizing Program Verification in HOL. In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley (Eds.). IEEE Computer Society, 208–222.

[3] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*.

[4] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 1–17. doi:10.1007/978-3-642-19718-5_1

[5] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31. doi:10.1145/2701415

[6] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 364–387. doi:10.1007/11804192_17

[7] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014), 4:1–4:35. doi:10.1145/2579080

[8] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *USENIX Security*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 207–221.

[9] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.* 43, 3 (2009), 263–288. doi:10.1007/S10817-009-9148-3

[10] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 917–934.

[11] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *PLDI*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 586–601. doi:10.1145/3062341.3062358

[12] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *POPL*. ACM Press, 178–188. doi:10.1145/41625.41641

[13] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clément Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas G. Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua M. Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. 2025. Formally Verified Cloud-Scale Authorization. In *ICSE*. IEEE, 2508–2521. doi:10.1109/ICSE55347.2025.00166

[14] Arthur Charguéraud. 2010. *Characteristic formulae for mechanized program verification.* Ph. D. Dissertation. Université Paris Diderot.

[15] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *ICFP*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 418–430. doi:10.1145/2034773.2034828

[16] Dafny. [n. d.]. compiler-bootstrap. https://github.com/dafny-lang/compiler-bootstrap.

[17] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[18] Alastair F. Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. 2024. Randomised Testing of the Compiler for a Verification-Aware Programming Language. In *ICST*. IEEE, 407–418. doi:10.1109/ICST60714.2024.00044

[19] Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. 2018. Program Verification in the Presence of I/O - Semantics, Verified Library Routines, and Verified Applications. In *VSTTE (LNCS, Vol. 11294)*, Ruzica Piskac and Philipp Rümmer (Eds.). Springer, 88–111. doi:10.1007/978-3-030-03592-1_6

[20] Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV (LNCS, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 173–177. doi:10.1007/978-3-540-73368-3_21

[21] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (LNCS, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. doi:10.1007/978-3-642-37036-6_8

[22] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. A Pragmatic Type System for Deductive Verification. (2016). https://inria.hal.science/hal-01256434

[23] Cormac Flanagan and James B. Saxe. 2001. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, Chris Hankin and Dave Schmidt (Eds.). ACM, 193–205. doi:10.1145/360204.360220

[24] Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP (LNCS, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 243–258. doi:10.1007/978-3-642-14052-5_18

[25] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.* 3, POPL (2019), 63:1–63:30. doi:10.1145/3290376

[26] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. doi:10.1145/3473590

[27] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. *Proc. ACM Program. Lang.* POPL. To appear.

[28] Michael J. C. Gordon. 1989. *Mechanizing Programming Logics in Higher Order Logic.* Springer, New York, NY, 387–439. doi:10.1007/978-1-4612-3658-0_10

[29] Benjamin Grégoire and Jorge Luis Sacchini. 2007. Combining a Verification Condition Generator for a Bytecode Language with Static Analyses. In *TGC (LNCS, Vol. 4912)*, Gilles Barthe and Cédric Fournet (Eds.). Springer, 23–40. doi:10.1007/978-3-540-78663-4_4

[30] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *ESOP (LNCS, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 584–610. doi:10.1007/978-3-662-54434-1_22

[31] Paolo Herms, Claude Marché, and Benjamin Monate. 2012. A Certified Multi-prover Verification Condition Generator. In *VSTTE (LNCS, Vol. 7152)*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer, 2–17. doi:10.1007/978-3-642-27705-4_2

[32] Peter V. Homeier and David F. Martin. 1994. Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator. In *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop, Proceedings (LNCS, Vol. 859)*, Thomas F. Melham and Juanito Camilleri (Eds.). Springer, 269–284. doi:10.1007/3-540-58450-1_48

[33] Marieke Huisman and Bart Jacobs. 2000. Java Program Verification via a Hoare Logic with Abrupt Termination. In *FASE (LNCS, Vol. 1783)*, T. S. E. Maibaum (Ed.). Springer, 284–303. doi:10.1007/3-540-46428-X_20

[34] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamaric, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *ISSTA*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 556–567. doi:10.1145/3533767.3534382

[35] Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. PureCake: A Verified Compiler for a Lazy Functional Language. *Proc. ACM Program. Lang.* 7, PLDI, 952–976. doi:10.1145/3591259

[36] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and

Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. doi:10.1145/1743546.1743574

[37] Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface between Coq and C. *Proc. ACM Program. Lang.* 9, POPL (2025), 687–717. doi:10.1145/3704860

[38] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. doi:10.1145/2535838.2535841

[39] K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inf. Process. Lett.* 93, 6 (2005), 281–288. doi:10.1016/J.IPL.2004.10.015

[40] K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[41] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[42] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[43] Zohar Manna and John McCarthy. 1969. *PROPERTIES OF PROGRAMS AND PARTIAL FUNCTION LOGIC*.

[44] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[45] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2008. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *FMCAD*, Alessandro Cimatti and Robert B. Jones (Eds.). IEEE, 1–8. doi:10.1109/FMCAD.2008.ECP.24

[46] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2012. Decompilation into logic - Improved. In *FMCAD*, Gianpiero Cabodi and Satnam Singh (Eds.). IEEE, 78–81.

[47] Magnus O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.* 24, 2-3 (2014), 284–315. doi:10.1017/S0956796813000282

[48] Daniel Nezamabadi and Magnus Myreen. 2025. Baking for Dafny: A CakeML Backend for Dafny. *CoRR* abs/2501.05111 (2025). arXiv:2501.05111 doi:10.48550/ARXIV.2501.05111 Accepted to the Dafny workshop at POPL 2025.

[49] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *ESOP (LNCS, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 589–615. doi:10.1007/978-3-662-49498-1_23

[50] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 2021. Formally Validating a Practical Verification Condition Generator. In *CAV (LNCS, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 704–727. doi:10.1007/978-3-030-81688-9_33

[51] Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. 2019. Characteristic Formulae for Liveness Properties of Non-Terminating CakeML Programs. In *ITP (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. doi:10.4230/LIPICS.ITP.2019.32

[52] Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana J. Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. 2023. Pancake: Verified Systems Programming Made Sweeter. In *PLOS*. ACM, 1–9. doi:10.1145/3623759.3624544

[53] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.* 1, ICFP (2017), 17:1–17:29. doi:10.1145/3110261

[54] Norbert Schirmer. 2006. *Verification of sequential imperative programs in Isabelle-HOL*. Ph. D. Dissertation. Technical University Munich, Germany. http://mediatum.ub.tum.de/mediatum/servlets/TUMDistributionServlet?id=mediaTUM_derivate_000000000002972

[55] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *PLDI*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 471–482. doi:10.1145/2491956.2462183

[56] Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. 2012. Self-certification: bootstrapping certified typecheckers in F* with Coq. In *POPL*, John Field and Michael Hicks (Eds.). ACM, 571–584. doi:10.1145/2103656.2103723

[57] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F. In *POPL*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. doi:10.1145/2837614.2837655

[58] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. doi:10.1145/3409003

[59] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *J. Funct. Program.* 29 (2019), e2. doi:10.1017/S0956796818000229

[60] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL*, George C. Necula and Philip Wadler (Eds.). ACM, 17–27. doi:10.1145/1328438.1328444

[61] Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *POPL*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 83–92. doi:10.1145/1706299.1706311

[62] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2010. A machine-checked soundness proof for an efficient verification condition generator. In *SAC*, Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 2517–2522. doi:10.1145/1774088.1774610

[63] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David A. Cock, and Michael Norrish. 2009. Mind the Gap. In *TPHOLs (LNCS, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 500–515. doi:10.1007/978-3-642-03359-9_34

Received 2025-09-13; accepted 2025-11-13