Generating Verified LLVM from Isabelle/HOL

Peter Lammich
The University of Manchester
peter.lammich@manchester.ac.uk

Abstract

We present a framework to generate verified LLVM programs from Isabelle/HOL. It is based on a code generator that generates LLVM text from a simplified fragment of LLVM, shallowly embedded into Isabelle/HOL. On top, we have developed a separation logic, a verification condition generator, and an LLVM backend to the Isabelle Refinement Framework.

As case studies, we have produced verified LLVM implementations of binary search and the Knuth-Morris-Pratt string search algorithm. These are one order of magnitude faster than the Standard-ML implementations produced with the original Refinement Framework, and on par with unverified C implementations. Adoption of the original correctness proofs to the new LLVM backend was straightforward.

The trusted code base of our approach is the shallow embedding of the LLVM fragment and the code generator, which is a pretty printer combined with some straightforward compilation steps.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Logic and verification; Theory of computation → Separation logic

Keywords and phrases Isabelle/HOL,LLVM,Separation Logic,Verification Condition Generator,Code Generation

Digital Object Identifier 10.4230/LIPIcs...

Supplement Material `http://www21.in.tum.de/~lammich/isabelle_llvm`

1 Introduction

The Isabelle Refinement Framework [33, 26, 27] features a stepwise refinement approach to verified algorithms, using the Isabelle/HOL theorem prover [42, 41]. It has been successfully applied to verify many algorithms and software systems, among them LTL and timed automata model checkers [15, 6, 48], network flow algorithms [32, 31], a SAT-solver certification tool [29, 30], and even a SAT solver [16]. Using Isabelle/HOL's code generator [18], the verified algorithms can be extracted to functional languages like Haskell or Standard ML. However, the code generator only provides partial correctness guarantees, i.e., termination of the generated code cannot be proved. Moreover, the generated code is typically slower than the same algorithms implemented in C or Java.

The original Refinement Framework [33, 26] could only generate purely functional code. The first remedy to the performance problem was to introduce array data structures that behave like functional lists on the surface, but are implemented by destructively updated arrays behind the scenes, similar to Haskell's now deprecated DiffArray. While this gained some performance, the array implementation itself was not verified, such that we had to trust its correctness. Moreover, an array access still required a significant amount of overhead compared to a simple pointer dereference in C.

The next step towards more efficient verified implementations was the Sepref tool [27]. It generates code for Imperative HOL [7], which provides a heap monad inside Isabelle/HOL, and a code generator extension to generate code that uses the stateful arrays provided by ML, or the heap monad of Haskell. The Sepref tool performs automatic data refinement from abstract data types like maps or sets to concrete implementations like hash tables, which can be placed on the heap and destructively updated. Moreover, it provides tools [28] to assist in the definition of new data structures, exploiting ‘free theorems’ [45] that it obtains from parametricity properties of the abstract data types. Using Imperative HOL as backend, we gained some additional performance: For example, the GRAT tool [29, 30] provides a verified checker for UNSAT certificates in the DRAT format [47]. It is faster than the unverified state-of-the art checker DRAT-TRIM [47], which is written in C. However, the GRAT tool spends most of its run time in an unverified certificate preprocessor. Nevertheless, optimizing the verified part of the code is important: The very same technique was also implemented in Coq, using purely functional data structures [12, 11]. There, the verified code was actually the bottleneck[1].

This paper presents a next step towards efficient verified algorithms: A refinement framework to generate verified code in LLVM intermediate representation [35] with total correctness guarantees. LLVM is an imperative intermediate language with a powerful and well-tested optimizing compiler. We first formalize the semantics of Isabelle-LLVM, a simple imperative language shallowly embedded into Isabelle/HOL, and designed to be easily translated to actual LLVM text (§2). On top of Isabelle-LLVM, we build a separation logic and a verification condition generator, which allows convenient reasoning about Isabelle-LLVM programs (§3). Finally, we modify the Sepref tool to target Isabelle-LLVM instead of Imperative/HOL (§4), connecting the Refinement Framework to our LLVM code generator. This only affects the last refinement step, such that most parts of existing verifications can be reused. As case studies (§5), we verify a binary search algorithm and adopt an existing formalization [19] of the Knuth-Morris-Pratt string search algorithm [24]. The resulting LLVM code is significantly faster than the corresponding Standard-ML code and on par with unverified C implementations. The paper ends with the discussion of future work (§6) and related work (§7). The Isabelle theories described in this paper are available as supplement material (URL displayed in paper header).

2 Isabelle-LLVM

2.1 State Monad

The basis of Isabelle-LLVM is a state-error monad, which we use to conveniently model the preconditions of instructions, their effect on memory, as well as arbitrary recursive programs. We define the algebraic data types:

$$('a,'s)\ M = M\ (run\colon 's \Rightarrow ('a,'s)\ mres) \qquad ('a,'s)\ mres = NTERM \mid FAIL \mid SUCC\ \ 'a\ \ 's$$

An entity of type ‹$('a,'s)\ M$› contains a function ‹run› that maps a start state of type ‹$'s$› to a *monad result* that indicates either nontermination, a failure, or a successful execution with a result of type ‹$'a$› and a new state. We define the standard monad combinators:

$$\begin{array}{ll} \texttt{return}\ x\ = M\ (\lambda s.\ SUCC\ x\ s) & get\ = M\ (\lambda s.\ SUCC\ s\ s) \\ \texttt{fail}\ = M\ (\lambda\_.\ FAIL) & set\ s = M\ (\lambda\_.\ SUCC\ ()\ s) \\ \texttt{bind}\ m\ f\ = M\ (\lambda s.\ \texttt{case}\ run\ m\ s\ \texttt{of}\ SUCC\ x\ s \Rightarrow run\ (f\ x)\ s \mid r \Rightarrow r) & \\ \texttt{assert}\ \Phi\ = \texttt{if}\ \Phi\ \texttt{then return}\ ()\ \texttt{else fail} & \end{array}$$

That is, ‹$\texttt{return}\ x$› returns result ‹x› without changing the state, ‹$\texttt{fail}$› aborts the computation, ‹get› returns the current state, and ‹$set\ s$› updates the current state. Finally, ‹$\texttt{bind}\ m\ f$› first executes ‹m›, and then ‹f› with the result of ‹m›. If ‹m› fails or does not

[1] Later, the checker was rewritten in ACL2, also using imperative data structures [11, 20]

terminate, the whole bind fails or does not terminate. The derived ‹`assert` Φ› combinator can be conveniently used to abort the computation if some precondition is violated, e.g., on division by zero.

We use do-notation, i.e. ‹`do` { $x{\leftarrow}m;\ f\ x$ }› is short for ‹`bind` $m\ (\lambda x.\ f\ x)$›. Moreover, we define a flat chain complete partial order [37] on ‹*mres*›, with ‹$\bot$:= *NTERM*›. For a monotonic function ‹$F :: ('a \Rightarrow ('b,'s)\ M) \Rightarrow 'a \Rightarrow ('b,'s)\ M$›, ‹*REC F*› is the least fixed point. As functions defined using the monad combinators are monotonic by construction [25], we can define arbitrary recursive computations. The partial function package [25] provides automation for monotonicity proofs and for defining simple recursive functions. Mutual recursion still requires some manual effort, though it could be automated, too.

2.2 Memory Model

We use a high-level memory model that does not directly expose the bit-level representation of values and assumes an infinite supply of memory. The memory is modeled as a list of blocks. Each block is either deallocated, or it is a list of values. A value is a pair of values, a pointer, or an integer. We model memory by the following data types[2]:

$$\begin{array}{ll} memory = MEMORY\ (block\ list) & block \quad = val\ list\ option \\ val = PAIR\ val\ val \mid PRIM\ primval & primval = PV\_INT\ lint \mid PV\_PTR\ rptr \end{array}$$

Here, the type ‹*lint*› is a fixed bit width word type with a two's complement semantics, as used by LLVM, and pair corresponds to a 2-element structure in LLVM. The type ‹*rptr*› is either null or an address. An address is a path through the memory structure to a value:

$$\begin{array}{ll} rptr \ = NULL \mid ADDR\ nat\ nat\ (va\_dir\ list) & va\_dir = PFST \mid PSND \end{array}$$

An address consists of a *block index*, a *value index*, and a *value address*, which is a list of directions to either descend into the first or the second value of a pair.

For the rest of this paper, we will use the state monad with a memory as state. Thus, we define the type ‹$'a\ llM = ('a{,}memory)\ M$›. It is straightforward to define functions ‹$load :: rptr \Rightarrow val\ llM$› and ‹$put :: val \Rightarrow rptr \Rightarrow unit\ llM$› to read/write a value from/to a pointer, or fail if the pointer is invalid. For the actual store function, we check that the structure of the value does not change, i.e. pairs remain pairs, pointers remain pointers, and words of width w remain words of width w:

$$\begin{array}{l} store\ x\ p = \texttt{do}\ \{\ y \leftarrow load\ p;\ \texttt{assert}\ (vstruct\ x = vstruct\ y);\ put\ x\ p\ \} \\ where \\ vstruct\ (PAIR\ a\ b)\ =\ VS\_PAIR\ (vstruct\ a)\ (vstruct\ b) \\ vstruct\ (PRIM\ (PV\_PTR\ \_))\ =\ VS\_PTR \\ vstruct\ (PRIM\ (PV\_INT\ w))\ =\ VS\_INT\ (width\ w) \end{array}$$

[2] We have slightly simplified the presentation. The actual implementation defines the concepts memory, block, and value in a modular fashion, in order to ease future extensions.

Similarly, we define an allocate and a free function:

```
allocn v n = do {
  blocks ← get;
  set (blocks@[Some (replicate n v)]);
  return (ADDR |blocks| 0 []) }
```

```
free (ADDR bi 0 []) = do {
  blocks ← get;
  assert (bi < |blocks| ∧ blocks!bi ≠ None);
  set (blocks[bi:=None]) }
free _ = fail
```

Here, ‹l_1 @l_2› concatenates two lists, ‹|l|› is the length of list ‹l›, ‹l!i› is the ith element of ‹l›, and ‹l[i:=x]› replaces the ith element of ‹l› by ‹x›. The allocate function takes an initial value and a block size, appends a new block to the memory, and returns a pointer to the start of the new block (value index 0, and value address []). The free function expects a pointer to the start of a block, checks that this block is not already deallocated, and then deallocates the block by setting it to ‹None›.

2.3 Towards a Shallow Embedding

While we explicitly model values in memory by the type ‹val›, we model values in registers in a more shallow fashion: We identify LLVM registers with Isabelle variables that have a type of shape ‹T = T × T | n word | T ptr›. Here, ‹×› is Isabelle's product type, ‹n word› is the n bit word type from Isabelle's word library[3], and ‹'a ptr› is a pointer with an attached phantom type for the value pointed to (‹'a ptr = PTR rptr›). For each type ‹'a› of shape ‹T›, we define the functions:

```
to_val    :: 'a ⇒ val             struct_of  :: 'a itself ⇒ vstruct
from_val :: val ⇒ 'a              init        :: 'a
```

such that

```
from_val o to_val = id                vstruct (to_val x) = (struct_of TYPE('a))
to_val init = zero_initializer (struct_of TYPE('a))
```

Here, ‹TYPE('a) :: 'a itself› reflects type ‹'a› into a term. The functions ‹to_val› and ‹from_val› inject a T-shaped type ‹'a› into a value with structure ‹struct_of TYPE('a)›. Moreover, ‹init::'a› corresponds to the all-zeroes value, i.e., the value where all pointers are null pointers, and all integers are 0.

2.4 Instructions

In a next step, we define the instructions of Isabelle-LLVM. Each instruction is identified with an Isabelle constant. For example, the load instruction is modeled by:

```
ll_load :: 'a ptr ⇒ 'a llM
ll_load (PTR p) = do {
  v ← load p;
  assert (vstruct v = struct_of TYPE('a));
  return (from_val v) }
```

[3] For convenient notation, we use the type ‹n word› as if it were a type depending on a variable n. Isabelle/HOL is not dependently typed. Instead, n is actually a type variable with type-class ‹len›, which provides a function ‹len_of :: 'a::len itself ⇒ nat› to extract the length as a term.

It loads a value from the specified pointer, checks that its structure matches the expected type ‹'a›, and then converts the value to ‹'a›.

For allocation and deallocation, we provide the instructions:

$ll\_malloc :: 'a\ itself \Rightarrow n\ word \Rightarrow 'a\ ptr\ llM \qquad ll\_free :: 'a\ ptr \Rightarrow unit\ llM$

Note that LLVM does not contain a heap manager. Instead, we assume that the generated code will be linked with the C standard library, and let the code generator produce calls to ‹*calloc*› and ‹*free*›. We also define instructions to access the elements of a pair, to offset a pointer, and to advance a pointer into a pair. The code generator maps these instructions to the corresponding LLVM instructions ‹*getelementptr*›, ‹*insertvalue*›, and ‹*extractvalue*›.

Integer instructions are defined on the ‹*n word*› type. For example, we define:

$ll\_udiv :: n\ word \Rightarrow n\ word \Rightarrow n\ word\ llM$
$ll\_udiv\ a\ b = \texttt{do}\ \{\ \texttt{assert}\ (b \neq 0);\ \texttt{return}\ (a\ div\ b)\ \}$

where ‹*div*› is the unsigned division from Isabelle's word library. Note the use of assertions to exclude undefined behavior, e.g., division by zero.

2.5 Modeling Control Flow

Next, we put together instructions to form procedure bodies. We only allow structured control flow via if-then-else, while, procedure calls, and sequential composition: The body of a procedure is modeled by an Isabelle term of type ‹*'a llM*› and shape ‹*block*›, where

$block = \texttt{do}\ \{\ var \leftarrow cmd;\ block\ \}\ |\ \texttt{return}\ var$
$cmd = ll\_{<}opcode{>}\ arg^*\ |\ proc\_name\ arg^*\ |\ llc\_if\ arg\ block\ block\ |\ llc\_while\ block\ block$
$arg = var\ |\ number\ |\ null\ |\ init$

with

$llc\_if :: 1\ word \Rightarrow 'a\ llM \Rightarrow 'a\ llM \Rightarrow 'a\ llM$
$llc\_if\ b\ t\ e = \texttt{if}\ b{=}1\ \texttt{then}\ t\ \texttt{else}\ e$

$llc\_while :: ('a \Rightarrow 1\ word\ llM) \Rightarrow ('a \Rightarrow 'a\ llM) \Rightarrow 'a \Rightarrow 'a\ llM$
$llc\_while\ b\ c\ s = \texttt{do}\ \{ctd \leftarrow b\ s;\ llc\_if\ ctd\ (\texttt{do}\ \{s \leftarrow c\ s;\ llc\_while\ b\ c\ s\})\ (\texttt{return}\ s)\}$

That is, a block is a list of commands whose results are bound to variables, terminated by a return instruction. A command is either an instruction, a procedure call, or an if-then-else or while statement. The arguments of instructions and procedure calls, as well as the condition of an if-then-else statement, must be variables or constants (i.e., numbers, the null pointer, or a zero-initialized value). The condition of a while statement is modeled as a block returning a ‹*1 word*›, such that it can be re-evaluated prior to each loop iteration. A program is represented by a set of (monomorphic) theorems of the shape ‹$proc_i\ x_1\ \dots\ x_n = block$›, where the ‹$proc_i$› are Isabelle functions, the ‹x_i› are variables, and all free variables on the right hand side are among the ‹x_i›.

▶ **Example 1.** Figure 1 shows the Isabelle specification of a procedure named ‹*fib*›, which takes a 64 bit word argument, and returns a 64 bit word. Our semantics can be directly executed inside Isabelle. The following Isabelle command evaluates ‹*fib*› on the first few natural numbers, and an empty memory:

value ‹$map\ (\lambda n.\ run\ (fib\ n)\ (MEMORY\ []))\ [0,1,2,3]$›
$(*\ output:\ [SUCC\ 0\ (MEMORY\ []),\ SUCC\ 1\ \dots,\ SUCC\ 1\ \dots,\ SUCC\ 2\ \dots]\ *)$

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t (return n) (do {
    n₁ ← ll_sub n 1;
    a  ← fib n₁;
    n₂ ← ll_sub n 2;
    b  ← fib n₂;
    c  ← ll_add a b;
    return c
  }) }
```

Figure 1 Isabelle-LLVM program

```
define i64 @fib(i64 %x) {
  start:
    %t = icmp ule i64 %x, 1
    br i1 %t, label %then, label %else
  then:
    br label %ctd_if
  else:
    %n_1 = sub i64 %x, 1
    %a   = call i64 @fib (i64 %n_1)
    %n_2 = sub i64 %x, 2
    %b   = call i64 @fib (i64 %n_2)
    %c   = add i64 %a, %b
    br label %ctd_if
  ctd_if:
    %x1a = phi i64 [ %x, %then ], [ %c, %else ]
    ret i64 %x1a }
```

Figure 2 Generated LLVM text

2.6 Code Generation

The LLVM intermediate representation [35] is a strongly typed control flow graph (CFG) based intermediate language that uses single static assignment (SSA) form [13]. A procedure is a list of basic blocks, the first block in the list being the entry point of the procedure. A basic block is a list of instructions, finished by a terminator instruction that determines the next basic block to execute (or to return from the current procedure). Each non-void instruction defines a fresh register containing its result. A register can only be accessed in the part of the CFG that is dominated by its definition. To transfer values from registers to other parts of the CFG, ϕ-instructions are used. A ϕ-instruction must be located at the start of a basic block. It lists, for each possible predecessor block, an accessible register in this predecessor block. The ϕ-instruction evaluates to the value of the register from those predecessor block from which execution was actually transferred. The result of the ϕ-instruction is bound to a fresh register, which can then be accessed from the current basic block.

It is straightforward to map an Isabelle-LLVM program to an actual LLVM program. Each equation of the form ‹*proc* x_1 .. x_n = *block*› is mapped to an LLVM function named ‹*proc*›. A block is mapped to a control flow graph. Instructions and procedure calls are directly mapped to LLVM instructions and calls. An ‹*x* ← *llc_if b t e*› is translated to conditional branching, using a ϕ-instruction to define the result register ‹*x*› when joining the control flow. An ‹*x* ← *llc_while b c s*› is translated similarly.

▶ **Example 2.** Figure 2 displays the output of our code generator for the ‹*fib*› constant displayed in Figure 1.

2.6.1 Mapping the Memory Model

Mapping the abstract memory model of Isabelle-LLVM to actual LLVM is slightly more involved. For example, recall the ‹*ll_malloc* :: *'a itself* ⇒ *n word* ⇒ *'a ptr llM*› instruction. It has to be mapped to the function ‹*void* calloc*(*size_t*, *size_t*)› from the C standard library.

For this, we have to parameterize the code generator with the architecture dependent size of the ‹*size_t*› type. Next, we have to obtain the size of type ‹*'a*› and cast the ‹*n word*› parameter to ‹*size_t*›. Here, our code generator will refuse downcast, as this might result in bits being dropped. Finally, we have to cast the returned ‹*void**› to the correct return type. Moreover, the ‹*calloc*› function returns ‹*null*› if not enough memory is available. In contrast, our semantics always returns a new block of memory. We insert code to terminate the program in a defined way if it runs out of memory. The relation between our semantics and the actual LLVM program then becomes: Either the program terminates with an out-of-memory condition, or it behaves as modeled by the semantics. Our current implementation prints an error message and terminates the process with exit code 1 if it runs out of memory.

A similar issue arises when comparing pointers: LLVM does not have instructions for pointer comparison. Instead, pointers have to be cast to integers, which can then be compared. However, this requires to know the bit-width of a pointer, which we cannot model in our semantics that admits unboundedly many different pointers. Instead, we model the instructions ‹*ll_ptrcmp_eq*› and ‹*ll_ptrcmp_ne*›, and let the code generator generate the cast to integers and the integer comparison.

2.7 Preprocessing

In the previous sections we have described the semantics of Isabelle-LLVM and its translation to actual LLVM. However, Isabelle-LLVM programs have to adhere to a very restrictive shape (cf. §2.5), which makes them easy to map to actual LLVM code, but tedious to write directly. Thus, we implement a preprocessor that tries to automatically transform user-specified equations to valid Isabelle-LLVM. While the preprocessing is highly incomplete, i.e., it cannot convert every equation to a well-shaped one, it works well in practice, allowing for concise specifications. Note that the preprocessor *proves* the new equations from the original ones. Thus, errors in the preprocessor cannot affect soundness: Either, it fails to prove the equations, or it produces ill-shaped equations, which the code generator will reject.

The user specifies an initial set of constants, which must be instantiated to monomorphic types, i.e., must not contain any type variables. For each constant, the preprocessor then gathers the defining equation, instantiates it to the actual monomorphic type of the constant, transforms it by inlining and fixed point unfolding, and then repeats the process for any new constant occurring on the right-hand side of the transformed equation. Note that a constant is identified by its name and type, such that a constant with the same name can occur multiple times in the final Isabelle-LLVM program. The code generator will disambiguate the names. At the end, we have a set of monomorphic equations that define all constants that occur in the final program, and can be passed to the actual code generator. We now describe the inlining and fixed point unfolding transformations.

2.7.1 Inlining

Inlining first applies user defined rewrite rules and then flattens nested expressions, converting function calls to the shape ‹$r \leftarrow f\, x_1\, \ldots\, x_n$› or ‹$r \leftarrow$ `return` $(f\, x_1\, \ldots\, x_n)$›, where the x_i are either constants, variables, or *monadic* arguments of type ‹$\ldots \Rightarrow$ _ *llM*›. Subterms of type ‹_ *llM*› are recursively flattened. We iterate the rewriting and flattening steps until a fixed point is reached.

▶ **Example 3.** Consider the following definition of the constant ‹*fib'*›:

$fib' :: m\ word \Rightarrow m\ word\ llM$

```
fib' n = if n ≤ 1 then return n
         else do { n₁ ← fib' (n − 1); n₂ ← fib' (n − 2); return (n₁ + n₂) }
```

When started with ‹fib' :: 64 word ⇒ 64 word llM›, the preprocessor automatically translates this equation to the equation displayed in Figure 1. During the translation, it uses the following inlining rules:

```
if b then c else t = llc_if (from_bool b) c t          return (a + b) = ll_add a b
return (from_bool (a≤b)) = ll_icmp_ule a b              return (a − b) = ll_sub a b
```

Our default setup contains similar rules for the other operations, as well as rules to map tuples and case-distinctions over tuples to ‹*insertvalue*› and ‹*extractvalue*› instructions.

2.7.2 Fixed-Point Unfolding

The preprocessor generates recursive functions from fixed-point combinators. It examines the right hand side of an equation for patterns ‹p› for which it has an unfold rule of the form ‹$p = F\ p$›. It then defines a new constant ‹$f\ x_1\ \ldots\ x_n = F\ (f\ x_1\ \ldots\ x_n)$›, where the ‹$x_i$› are the free variables in the pattern ‹p›. Finally, it replaces ‹p› by ‹$f\ x_1\ \ldots\ x_n$› in the equation. This way, specifications with fixed point combinators are automatically transformed to a set of recursive equations, as required by the code generator.

For example, the ‹*llc_while*› combinator is defined as a fixed point (cf. §2.5). Using its definition as an unfold rule, the preprocessor will automatically convert while loops into tail calls. This allows for using while-loops without trusting their translation in the code generator. A configuration option in our tool lets the user choose between direct while-loop translation or unfolding into a tail call.

▶ **Example 4.** Consider the following program:

```
euclid :: 64 word ⇒ 64 word ⇒ 64 word
euclid a b = do {
  (a,b) ← llc_while
    (λ(a,b) ⇒ ll_cmp (a ≠ b))
    (λ(a,b) ⇒ if (a≤b) then return (a,b−a) else return (a−b,b))
    (a,b);
  return a }
```

From this, the preprocessor proves the following two equations (before inlining):

```
euclid a b = do {
    (a, b) ← euclid₀ (a, b);
    return a }
euclid₀ s = do {
    ctd ← case s of (a, b) ⇒ ll_cmp (a ≠ b);
    llc_if ctd (do {
      s ← case s of (a, b) ⇒ if a ≤ b then return (a, b − a) else return (a − b, b);
      euclid₀ s
    }) (return s) }
```

That is, it defined a new constant ‹$euclid_0$› to replace the while loop by tail recursion.

3 Verification Condition Generator

The next step towards generating verified LLVM programs is to establish a reasoning infrastructure. In this section, we describe our separation logic [43] based verification condition generator. Note that, while applying complex operations on the proof state, at the end, our VCG conducts a proof that goes through Isabelle's inference kernel. Thus, bugs in the VCG cannot cause unsoundness.

3.1 Separation Algebra

The first step to obtain a separation logic is to define a separation algebra on a suitable abstraction of the memory. A separation algebra [8] is a structure with a zero, a disjointness predicate $a\#b$, and a disjoint union $a + b$. Intuitively, elements describe parts of the memory. Zero describes the empty memory, $a\#b$ means that a and b describe disjoint parts of the memory, and $a + b$ describes the memory described by the union of a and b. For the exact definition of a separation algebra, we refer to [8, 22]. We note that separation algebras naturally extend over functions, pairs, and option types.

We abstract a value by a partial function from value addresses (‹*va_dir list*›) to primitive values, such that the addresses in the domain of the function are independent, i.e., no address is the prefix of another address:

$$\textbf{typedef}\ aval = \{\ m :: vaddr \Rightarrow 'a\ option.\ \forall va, va' \in dom\ m.\ va \neq va' \longrightarrow indep\ va\ va'\ \}$$
$$val\_\alpha :: val \Rightarrow aval$$
$$val\_\alpha\ (PRIM\ x) = [[] \mapsto x]$$
$$val\_\alpha\ (PAIR\ x\ y) = PFST \cdot val\_\alpha\ x + PSND \cdot val\_\alpha\ y$$

Here, ‹$[k \mapsto v]$› is the partial function that maps ‹k› to ‹v›, and ‹$i \cdot a$› prepends the item ‹i› to all addresses in the domain of ‹a›. It is straightforward (though technically involved) to show that abstract values form a separation algebra, where the empty map is zero, maps are disjoint iff their domains are pairwise independent, and union merges two maps.

A natural abstraction of a block (‹*val list*›) would be a function from indexes to abstract values, mapping invalid indexes to 0. However, this abstraction does not contain enough information to reason about deallocation. In order to deallocate a block, we have to own the whole block. However, from the abstraction, we cannot infer the size of the block, and thus we cannot specify an assertion that ensures that we own the whole block. A remedy (which the author has seen in [1]) is to additionally abstract a block to its size. Thus, abstract blocks have the type ‹$ablock = (nat \Rightarrow aval) \times nat\ option$›. The option type is required to make the second elements of the tuples a separation algebra. We use the trivial separation algebra here, where two elements are only disjoint if at least one of them is ‹*None*›. Finally, we define ‹$amemory = nat \Rightarrow ablock$›, and a function ‹$\alpha :: memory \Rightarrow amemory$› that abstracts memory by a function from block indexes to abstract blocks, mapping deallocated or invalid indexes to zero.

3.2 Basic Reasoning Infrastructure

Predicates of type ‹$assn = amemory \Rightarrow bool$› are called *assertions*. The *weakest precondition* of a program ‹$c :: 'a\ llM$›, a *postcondition* ‹$Q :: 'a \Rightarrow assn$›, and a memory ‹s› is defined as:

$$wp\ c\ Q\ s = (\exists r\ s'.\ run\ c\ s = SUCC\ r\ s' \wedge Q\ r\ (\alpha\ s'))$$

Intuitively, ‹$wp\ c\ Q\ s$› states that program ‹c›, if run on memory ‹s›, terminates successfully with the result ‹r›, and the abstraction of the new state ‹s'› satisfies ‹Q›.

For assertions ‹P› and ‹Q›, the *separating conjunction* ‹$P{*}Q$› describes a memory that can be split into two disjoint parts described by ‹P› and ‹Q›, respectively:

$$(P * Q)\ s = \exists s_1\ s_2.\ s_1 \# s_2 \wedge s = s_1 + s_2 \wedge P\ s_1 \wedge Q\ s_2$$

Validity of a *Hoare triple* ‹$\{P\}\ c\ \{Q\}$› is defined as follows:

$$\models \{P\}\ c\ \{Q\} = \forall F\ s.\ (P{*}F)\ (\alpha\ s) \longrightarrow wp\ c\ (\lambda r\ s'.\ (Q\ r * F)\ s')\ s$$

That is, if the memory can be split into a part described by the *precondition* ‹P›, and a *frame* described by ‹F›, then command ‹c› will succeed, and the new memory consists of a part described by the postcondition ‹Q› and the unchanged frame. Our Hoare triples satisfy the frame rule: ‹$\models \{P\}\ c\ \{Q\} \implies \models \{P * F\}\ c\ \{\lambda r.\ Q\ r * F\}$› for all ‹$F$›.

3.3 Basic Rules

Once we have set up the separation algebra and the abstraction function, we can prove Hoare triples for the basic operations of our memory model. For example, we prove the following rules for ‹*allocn*› and ‹*free*›:

$$\models \{\square\}\ allocn\ v\ n\ \{\lambda p.\ malloc\_tag\ n\ p * range\ \{0..{<}n\}\ (\lambda\_.\ v)\ p\}$$
$$\models \{malloc\_tag\ n\ p * \exists blk.\ range\ \{0..{<}n\}\ blk\ p\}\ free\ p\ \{\lambda\_.\ \square\}$$

where ‹$\square = \lambda s.\ s{=}0$› describes the empty memory, ‹$malloc\_tag\ n\ p$› asserts that ‹p› points to the beginning of a block, and the size field of this block's abstraction is ‹n›, and ‹$range\ I\ f\ p$› describes that for all ‹$i \in I$›, ‹$p + i$› points to value ‹$f\ i$›. Intuitively, ‹*allocn*› creates a block of size ‹n›, initialized with values ‹v›, and a tag. If one possesses both, the whole block and the tag, it can be deallocated by free. For the Isabelle-LLVM memory instructions, we obtain the following rules:

$$\models \{n{\neq}0\}\ ll\_malloc\ TYPE('a)\ n\ \{\lambda p.\ range\ \{0..{<}n\}\ (\lambda\_.\ init)\ p * malloc\_tag\ n\ p\}$$
$$\models \{range\ \{0..{<}n\}\ blk\ p * malloc\_tag\ n\ p\}\ ll\_free\ p\ \{\lambda\_.\ \square\}$$
$$\models \{pto\ x\ p\}\ ll\_load\ p\ \{\lambda r.\ r{=}x * pto\ x\ p\}$$
$$\models \{pto\ xx\ p\}\ ll\_store\ x\ p\ \{\lambda\_.\ pto\ x\ p\}$$

Here, ‹$pto\ x\ p$› describes that p points to value x, and we write predicates as if they were assertions on the empty memory, e.g., ‹$n{\neq}0$› instead of ‹$\lambda s.\ s{=}0 \wedge n{\neq}0$›. We prove similar rules for the other instructions.

3.4 Automating the VCG

In order to efficiently prove Hoare triples, some automation is required. We provide a verification condition generator with a frame inference heuristics. The first step to prove a Hoare triple is to convert it to a proposition on weakest preconditions:

$$[\![\bigwedge F\ s.\ STATE\ (P{*}F)\ s \implies wp\ c\ (\lambda r\ s'.\ (Q\ r * F)\ s')\ s]\!] \implies \models \{P\}\ c\ \{Q\}$$

where ‹$STATE\ P\ s = P\ (\alpha\ s)$›. In general, the VCG operates on subgoals of the form ‹$STATE\ P\ s \implies wp\ c\ Q\ s$›. It then iteratively performs one of the following steps[4]:

[4] This is a simplified presentation. The actual VCG is an instantiation of a generic VCG framework that can be configured with various solvers, rules, and heuristics.

simplification Apply a rewrite rule to transform ‹$wp\ c\ Q\ s$› into some equivalent proposition. For example, binding is resolved by the rule:

$$wp\ (\texttt{do}\ \{x{\leftarrow}m;\ f\ x\})\ Q\ s = wp\ m\ (\lambda x.\ wp\ (f\ x)\ Q)\ s$$

rule If there is a Hoare triple of the form ‹$\models \{P'\}\ c\ \{Q'\}$›, the VCG tries to infer a frame ‹F› such that ‹$P \vdash P'{*}F$›, and replaces the goal by ‹$STATE\ (Q'{*}F)\ s' \implies Q\ s'$› for a fresh ‹$s'$›. Here, ‹$P \vdash Q = \forall s.\ P\ s \implies Q\ s$› denotes entailment.

final If the goal has the form ‹$STATE\ P\ s \implies Q\ s$› such that ‹$Q$› is not of the form ‹$wp\ \_\ \_\ \_$›, a heuristics is used to prove ‹$P \vdash Q$›.

The actual verification conditions are generated during frame inference and the final proof heuristics. For example, the rule for ‹ll_malloc› requires to prove that the size operand is not zero. The VCG will try to prove these goals by a default tactic, and leave them to the user if this tactic fails.

▶ **Example 5.** Recall the function ‹$euclid :: 64\ word \Rightarrow 64\ word \Rightarrow 64\ word\ llM$› from Example 4. We prove the following Hoare triple:

$$\models \{uint_{64}\ a\ a_\dagger * uint_{64}\ b\ b_\dagger * 0{<}a * 0{<}b\}\ euclid\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ uint_{64}\ (gcd\ a\ b)\ r_\dagger\}$$

Here, ‹$uint_{64}\ a\ a_\dagger$› states that ‹$a_\dagger{::}64\ word$› is an unsigned integer with value ‹$a{::}int$›, where ‹int› is the type of (mathematical) integers in Isabelle, and ‹gcd› is Isabelle's greatest common divisor function. After annotating a suitable loop invariant, the VCG generates the following two verification conditions:

$$[\![\ gcd\ x\ y = gcd\ a\ b;\ x \neq y;\ x \leq y;\ \ldots\ ]\!] \implies gcd\ x\ (y - x) = gcd\ a\ b$$
$$[\![\ gcd\ x\ y = gcd\ a\ b;\ \neg\ x \leq y;\ \ldots\ ]\!] \implies gcd\ (x - y)\ y = gcd\ a\ b$$

These are straightforward to prove in Isabelle, e.g., using sledgehammer [3].

3.5 Data Structures and Basic Refinement

Recall Example 5. The Hoare triple that is proved there first maps the 64 bit word arguments and results to mathematical integers, and then phrases the correctness statement in terms of mathematical integers. This approach is often more feasible than stating correctness on the concrete implementation directly. In our case, we would have to define the concept of greatest common divisor for 64 bit words. In general, an algorithm often computes some function on abstract mathematical concepts like integers or sets, but has to implement these by concrete data structures like 64 bit words or hash-tables. Thus, a concise way to specify the correctness statement is to first map the implementations back to the abstract concepts, and then state the actual correctness abstractly.

In separation logic based reasoning, a data structure provides a *refinement assertion* ‹$A\ x\ x_\dagger :: assn$›, which describes that the abstract value ‹x› is implemented by the concrete value ‹$x_\dagger$›. We define refinement assertions to implement integers and natural numbers by n bit words, and to implement lists by blocks of memory. On top of that, we define more complex data structures like dynamic arrays. Note that new data structures can easily be added. In general, an implementation does not completely implement an abstract mathematical concept. For example, n bit words can only represent the integers ‹$sints\ n = \{-2^{n-1} ..< 2^{n-1}\}$›, and hash-tables can only represent finite sets. Thus, the rules for the operations generally come with additional preconditions. For example, the rule to implement subtraction on integers by subtraction on n bit words is the following:

$\models \{sint_n\ a\ a_\dagger\ *\ sint_n\ b\ b_\dagger\ *\ a{-}b \in sints\ n\}\ ll\_sub\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ sint_n\ (a{-}b)\ r_\dagger\}$
for $a_\dagger\ b_\dagger :: n\ word$ **and** $a\ b :: int$

Here, ‹$sint_n$› implements mathematical integers by n-bit words. Note that the postcondition does not mention the operands ‹a,b› again, though they are still valid after the operation. As ‹$sint_n$› is *pure*, i.e., does not use the memory, our VCG will automatically add the corresponding assertions to the postcondition.

4 Automatic Refinement

Our basic VCG infrastructure can be used to verify simple algorithms like ‹*euclid*› from Example 5. However, many complex algorithms have already been verified using the Isabelle Refinement Framework [33]. It features a non-deterministic programming language with a refinement calculus and a VCG. It allows to express an algorithm using abstract mathematical concepts, and then refine it in multiple steps towards an efficient implementation. The last step of a refinement is typically performed by the Sepref tool [27], which translates a program from the non-deterministic monad of the Refinement Framework into the deterministic heap monad of Imperative HOL [7], replacing abstract data types by concrete implementations. We have modified the Sepref tool to translate to Isabelle-LLVM's monad instead. We only had to modify the translation phase. The preprocessing phases, which only work on the abstract program, remained unchanged.

The translation phase works by symbolically executing the abstract program, thereby synthesizing a structurally similar concrete program. During the symbolic execution, the relation between the abstract and concrete variables is modeled by refinement assertions. The predicate ‹$hnr\ \Gamma\ m_\dagger\ \Gamma'\ R\ m$› means that concrete program ‹$m_\dagger$› implements abstract program ‹m›, where ‹Γ› contains the refinements for the variables before the execution, ‹Γ'› contains the refinements after the execution, and ‹R› is the refinement assertion for the result of ‹m›. For example, a ‹`bind`› is processed by the following rule:

1 $[\![\ hnr\ \Gamma\ m_\dagger\ \Gamma'\ R_x\ m;$
2 $\bigwedge x\ x_\dagger.\ hnr\ (R_x\ x\ x_\dagger\ *\ \Gamma')\ (f_\dagger\ x_\dagger)\ (R'_x\ x\ x_\dagger\ *\ \Gamma'')\ R_y\ (f\ x);$
3 $MK\_FREE\ R'_x\ free;$
4 $]\!]\ \Longrightarrow\ hnr\ \Gamma\ (\texttt{do}\ \{x_\dagger{\leftarrow}m_\dagger; r_\dagger{\leftarrow}f_\dagger\ x_\dagger;\ free\ x_\dagger;\ \texttt{return}\ r_\dagger\})\ \Gamma''\ R_y\ (\texttt{do}\ \{x{\leftarrow}m;\ f\ x\})$

To refine ‹$x{\leftarrow}m;\ f\ x$›, we first execute ‹m›, synthesizing the concrete program ‹$m_\dagger$› (line 1). The state after ‹m› is ‹$R_x\ x\ x_\dagger\ *\ \Gamma'$›, where ‹$x$› is the result created by ‹m›. From this state, we execute ‹$f\ x$› (line 2). The new state is ‹$R'_x\ x\ x_\dagger\ *\ \Gamma''\ *\ R_y\ y\ y_\dagger$›, where ‹$y$› is the result of ‹$f\ x$›. Now, the variable ‹$x$› goes out of scope, such that it has to be deallocated. The predicate ‹$MK\_FREE\ R'_x\ free = \forall x\ x_\dagger.\ \models \{R'_x\ x\ x_\dagger\}\ free\ x_\dagger\ \{\lambda\_.\ \Box\}$› (line 3) states that ‹*free*› is a deallocator for data structures implemented by refinement assertion ‹R'_x›. Note that the refinement for variable ‹x› may change: If ‹$f_\dagger\ x_\dagger$› overwrites ‹$x_\dagger$›, the refinement assertion for ‹x› will be changed to the special assertion ‹*invalid*›. The deallocator for ‹*invalid*› is simply a no-op. Adding support for deallocators was the most substantial change we applied to the Sepref tool. Its original target language, Imperative HOL, is garbage collected, such that there is no need for explicit deallocation.

4.1 Data Structure Library

Once the basic Sepref tool is adapted, we can define data structures. Reusing the basic data structures from the original Sepref tool is not possible, as Imperative HOL uses arbitrary precision integers and algebraic data types, while we have only fixed width words and pairs. Up to now, we have added the implementation of integers and natural numbers by n bit words and some basic container data structures like dynamic arrays, bit-vectors, and min-heaps. Thereby, we could reuse the existing infrastructure of the Sepref tool: For example, there is support to automatically generate rules that also support refinement of the elements of a data structure, exploiting 'free theorems' [45] which stem from parametricity properties of the abstract types.

5 Case Studies

To assess the usability of our approach, we have verified a binary search algorithm and the Knuth-Morris-Pratt string search [24] algorithm. Both algorithms have also been verified with the original Sepref tool, such that we can compare the two approaches.

5.1 Binary Search

Binary search is a simple algorithm to find a value in a sorted array. Despite its simplicity, it has a history of flawed implementations[5], making it a natural example for formal verification.

We start with a high-level specification: For a list ‹*xs*› and a value ‹*x*›, find the index of the first element greater or equal to ‹*x*›. We define the following constant:

```
fi_spec xs x = spec i. i = find_index (λy. x≤y) xs
```

where ‹*find_index P xs*› is a standard list function that returns the index of the first element in ‹*xs*› that satisfies ‹*P*›, or ‹*length xs*› if there is no such element.

Next, we phrase the binary search algorithm in the Isabelle Refinement Framework:

```
bin_search xs x ≡ do {
  (l,h) ← while
    (λ(l,h). l<h)
    (λ(l,h). do {
      assert (l<length xs ∧ h≤length xs ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then return (m+1,h) else return (l,m)
    })
    (0,length xs);
  return l }
```

It is a standard exercise to prove that the algorithm adheres to its specification:

```
bs_correct: sorted xs ⟹ bin_search xs x ≤ fi_spec xs x
```

Finally, we invoke our adapted Sepref tool:

```
sepref_definition bs_impl [llvm_code] is bin_search
  :: (larray64 sint64)^k → sint64^k → snat64
  unfolding bin_search_def [...] by sepref
export_llvm bs_impl file bin_search.ll
lemmas bs_impl_correct = bs_impl.refine[FCOMP bs_correct]
```

[5] A buggy implementation in the Java Standard Library has gone undetected for nearly a decade [5].

This produces an Isabelle-LLVM program ‹*bs_impl*›, exports it to actual LLVM text, and proves the refinement theorem ‹*bs_impl_correct*›:

$$(bs\_impl,\ fi\_spec) : [\lambda(xs, \_).\ sorted\ xs]\ (larray_{64}\ sint_{64})^k \times sint_{64}^k \rightarrow snat_{64}$$

Here, $\langle snat_w \rangle$ implements natural numbers by signed w-bit words[6]. Moreover, $\langle larray_w\ A \rangle$ refines a list to an array and a w-bit length field, the elements of the list being refined by assertion $\langle A \rangle$. The notation $\langle [\Phi]\ A_1^{k|d} \times \ldots \times A_n^{k|d} \rightarrow R \rangle$ specifies a refinement with precondition $\langle \Phi \rangle$, such that the arguments are refined by $\langle A_1 \ \ldots\ A_n \rangle$ and the result is refined by $\langle R \rangle$. The $\cdot^{k|d}$ annotations specify whether an argument is overwritten (k for keep, d for destroy). While we use this notation a lot in the Refinement Framework, it is straightforward to prove a standard Hoare triple from it. By unfolding some definitions we get:

$$\begin{aligned}&\models \{larray_{64}\ sint_{64}\ xs\ xs_\dagger * sint_{64}\ x\ x_\dagger * sorted\ xs\ \}\\ &\quad bs\_impl\ xs_\dagger\ x_\dagger\\ &\quad \{\lambda i_\dagger.\ \exists i.\ larray_{64}\ sint_{64}\ xs\ xs_\dagger * snat_{64}\ i\ i_\dagger * i{=}find\_index\ (\lambda y.\ x{\le}y)\ xs\}\end{aligned}$$

That is, if we start with an array $\langle xs_\dagger \rangle$ representing the sorted list ‹*xs*›, and a 64-bit word $\langle x_\dagger \rangle$ representing the integer ‹*x*›, then the array still represents $\langle xs_\dagger \rangle$, and the result $\langle i_\dagger \rangle$ represents a natural number ‹*i*›, which is equal to the correct index.

The Sepref tool implements mathematical integers by 64-bit words, proving absence of overflows. This is only possible because the assertion in ‹*bin_search*› explicitly states that the indexes are in bounds. Moreover, note the expression $\langle l + (h{-}l)\ div\ 2 \rangle$ that we used to compute the midpoint index. On mathematical integers, it is equal to $\langle (l{+}h)\ div\ 2 \rangle$. However, on fixed-width words, the latter may overflow, while the former does not[7].

5.2 Knuth-Morris-Pratt String Search

Next, we regard the Knuth-Morris-Pratt (KMP) string search algorithm [24], a well-known linear time algorithm to find the index of the first occurrence of a string s in a string t:

$$\begin{aligned}&ss\_spec\ s\ t = \texttt{spec}\\ &\quad None \Rightarrow \nexists i.\ sublist\_at\ s\ t\ i\ |\\ &\quad Some\ i \Rightarrow sublist\_at\ s\ t\ i \wedge (\forall ii{<}i.\ \neg sublist\_at\ s\ t\ ii))\end{aligned}$$

where ‹*sublist_at s t i*› specifies that list ‹*s*› occurs in list ‹*t*› at index ‹*i*›:

$$sublist\_at\ s\ t\ i = \exists ps\ ss.\ t = ps@s@ss \wedge i = length\ ps$$

We have recently formalized KMP with the original Sepref tool [19]. The adaption of the existing formalization was straightforward: In the abstract part, we had to explicitly add a few in-bounds assertions. Most of them were already contained implicitly in the original proof. For the synthesis step, we only had to add setup for the fixed-width word types. The result of the automatic synthesis is an Isabelle-LLVM program ‹*kmp_impl*›, and the theorem:

$$\begin{aligned}&(kmp\_impl,\ ss\_spec)\\ &\quad : [\lambda s\ t.\ |s| + |t| < 2^{63}]\ (larray_{64}\ sint_{64})^k \times (larray_{64}\ sint_{64})^k \rightarrow snat\_option_{64}\end{aligned}$$

Here $\langle snat\_option_{64} \rangle$ implements the type ‹*nat option*› by signed 64-bit words, mapping ‹*None*› to -1.

[6] As LLVM's index operations are on signed words, it's convenient to always implement sizes and indexes by signed types, even if they are natural numbers.
[7] Exactly this overflow caused the infamous bug in the Java Standard Library [5].

$n/10^6$	C	LLVM	SML	SML*
1	121	100	1999	139
2	251	204	4209	289
3	379	304	6516	440
4	513	412	8843	600
5	635	514	11494	756
6	767	617	13646	917
7	908	726	16032	1076
8	1038	854	18421	1250
9	1162	945	20957	1409
10	1293	1045	23409	1564

Table 1 Time (ms) to search for the values $0, 2, \ldots < 5n$ in an array $[0, 5, \ldots < 5n]$.

a-*l*	C++	LLVM	SML	SML*
16-8	499	597	4616	918
16-64	511	598	4621	926
16-512	513	590	4573	909
32-8	453	551	4471	850
32-64	465	552	4523	857
32-512	463	544	4456	840
64-8	418	530	4433	803
64-64	420	531	4514	809
64-512	416	523	4411	799

Table 2 Time (ms) to run the *a*-*l* benchmark suite from StringBench [44]. Here *a* is the alphabet size, and *l* the pattern size. The sample size is $3 \cdot 2^{20}$ characters. The algorithm stops after finding the first match.

5.3 Runtime

We have compared our verified LLVM implementations to unverified C/C++ implementations of the same algorithms, as well as to the Standard ML (SML) implementations generated by the original Sepref tool. While we have implemented binary search in C ourselves, we used a publicly available code snippet [34] for KMP[8]. The programs were compiled with MLton-2018 [39] and clang-6.0 [10], and run on a standard laptop machine (2.8GHz Quadcore i7 with 16MiB RAM). Tables 1 and 2 display the results: The verified LLVM implementations are on par with the unverified C/C++ implementations, and an order of magnitude faster than the SML implementations.

Isabelle's code generator uses arbitrary precision integers, which tend to be significantly slower than fixed-width integers. The SML* column shows the results when we manually replace the arbitrary precision integers by 64-bit integers in the generated code. While this is unsound in general, it gives us a lower bound of what would be possible in SML with more elaborate code generator configurations[9]. SML* is significantly faster than the original SML, but still 1.5 times slower than LLVM.

6 Future Work

While our case studies only cover medium complex algorithms, we expect that our approach will scale to more complex algorithms, e.g. model checkers [48, 16] and SAT solvers [16], which have already been formalized with the original refinement framework. While these formalizations use a combination of functional and imperative data structures, the LLVM backend only supports imperative data structures. We expect the necessary changes to be manageable, but non-trivial. In particular, the current Sepref tool only supports pure data structures to be nested in containers. In the Imperative HOL setting, we simply use functional data structures inside containers. For LLVM, nested container data structures

[8] One easily finds many C implementations of KMP, mainly differing in the loop structure. We tried to choose one that is close to our implementation.

[9] Fleury et al. [16] have successfully experimented with such code generator tuning.

currently require ad-hoc proofs on the separation logic level. We leave the lifting of Sepref to support nested imperative data structures to future work.

Moreover, the refinement from arbitrary precision integers to fixed size integers was quite straightforward for our case studies, and we expect these refinements to be more complex in general. We leave it to future work to explore this issue more systematically, and to provide semi-automated tools, e.g. along the lines of AutoCorres [17].

Our code generator, as well as most standard code generators in theorem provers, translates from logic to target language code, implicitly identifying logical concepts with programming language concepts. This approach is simple, however, the translation algorithm and its implementation become part of the trusted code base. More recently, code generators that translate into a deeply embedded semantics of the target language have been developed [40, 21]. We leave a translation to a deep embedding of LLVM to future work, and note that a deep embedding will also enable more advanced control flow constructs like exceptions and breaking from loops, without significantly increasing the trusted code base.

Compared to actual LLVM, Isabelle-LLVM makes a few simplifying assumptions: We do not support floating point arithmetic, though this could be added, e.g. based on Lei Yu's floating point formalization [49]. Moreover, we only support two-element structures (pairs). This nicely fits Isabelle HOL's product datatype, and the nested structures resulting from longer tuples should not be a problem for LLVM's optimizer. Also, we do not support concepts that are handy for program optimization, but not required for code generation, like poison values. Isabelle-LLVM assumes an infinite supply of memory, and thus cannot assign a bit-size to pointers. This assumption helps us to retain a deterministic semantics, which is executable inside the theorem prover (cf. Example 1). We plan to use this feature for systematic testing of our code generator against the actual LLVM compiler. A similar assumption is implicitly made for the stack, as our semantics permits arbitrarily deep recursive procedure calls. We remedy this mismatch between semantics and reality by terminating the program in a defined way if it runs out of heap. To protect against stack overflows, LLVM provides mechanisms like stack probing or split stack, which, however, require some effort to enable. We leave that to future work, and note that our generated code allocates no large blocks of memory on the stack. Thus, stack overflows are likely to hit the guard pages inserted by most operating systems, which will cause defined termination of the process.

Currently, we interface our generated LLVM code from C programs compiled by clang. However, the ABIs of C and LLVM only partially match, and some LLVM constructs cannot be expressed in C at all. Currently, it is the user's responsibility to implement a correct header file. We plan to automatically generate header files and adapter functions to make the exported code accessible from C.

7 Related Work

This project would not have been possible without several independent Isabelle developments: We use the Separation Algebra library [23, 22] as basis for our separation logic. We substantially extended this library by a frame inference heuristics, and formalized the extension of separation algebras over functions, products, and options. Moreover, we use Isabelle's machine word library [2] to model the 2's complement arithmetic of LLVM. We slightly extended this library by adding a few lemmas. Finally, the Eisbach language [38] was a great help for prototyping the verification condition generator, although most of the final VCG is now implemented directly in the more low-level Isabelle/ML.

The Vellvm project [50, 51] verifies LLVM program transformations in Coq. To be useful, e.g. as backend for clang, they have to formalize a substantial fragment of LLVM. On the other hand, we can afford to formalize a simplified and abstract semantics that is just powerful enough to cover what Sepref generates.

We drew some of the ideas for our separation logic from the Verifiable C project [1], a Coq formalization of a separation logic on top of the CompCert C semantics [4].

There exists various formalizations of low-level imperative languages, eg [36, 46]. These are focused on specifying the semantics, and we are not aware of any complex algorithm verifications using these formalizations.

The DeepSpec project [14] aims at a completely verified computation environment, down to machine code, including the operating system. This is much more ambitious than the work presented here, which stops at a (simplified) LLVM semantics. For proving correct imperative programs, they have a separation logic based VCG for a fragment of C [1, 9], which they apply to several small C programs, mainly for cryptographic algorithms.

8 Conclusions

We have developed Isabelle-LLVM, a shallowly embedded imperative language designed to be easily translated to actual LLVM text. On top of this, we have built a verification infrastructure, and re-targeted the Sepref tool to connect the Refinement Framework to LLVM. As case studies, we have generated verified LLVM code for a binary search algorithm and the Knuth-Morris-Pratt string search algorithm. Both implementations are an order of magnitude faster than the ones generated with the original Sepref tool, and on par with unverified C implementations. The additional effort required to refine to LLVM instead of Standard ML was quite low.

Acknowledgement

We thank Maximilian P. L. Haslbeck and Simon Wimmer for proofreading and useful suggestions. We received funding from DFG grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases".

References

1 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

2 Joel Beeren, Matthew Fernandez, Xin Gao, Gerwin Klein, Rafal Kolanski, Japheth Lim, Corey Lewis, Daniel Matichuk, and Thomas Sewell. Finite machine word library. *Archive of Formal Proofs*, June 2016. `http://isa-afp.org/entries/Word_Lib.html`, Formal proof development.

3 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013. `doi:10.1007/s10817-013-9278-5`.

4 Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. URL: `http://xavierleroy.org/publi/Clight.pdf`.

5 Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. URL: `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

6 Julian Brunner and Peter Lammich. Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. `doi:10.1007/s10817-017-9418-4`.

7 Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.

8 C. Calcagno, P.W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378, July 2007.

9 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. Vst-floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61, 02 2018. `doi:10.1007/s10817-018-9457-5`.

10 Clang: a C language family frontend for LLVM. URL: `https://clang.llvm.org/`.

11 Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proc. of CADE*. Springer, 2017.

12 Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Proc. of TACAS*, pages 118–135. Springer, 2017.

13 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: `http://doi.acm.org/10.1145/115372.115320`, `doi:10.1145/115372.115320`.

14 Deep spec project web page. URL: `https://deepspec.org/`.

15 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

16 Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proc. of CPP*, pages 158–171, 2018.

17 David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *Proc. of PLDI '14*, pages 429–439, 2014. `doi:10.1145/2594291.2594296`.

18 Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement in Isabelle/HOL. In *Proc. of ITP*, pages 100–115. Springer, 2013.

19 Fabian Hellauer and Peter Lammich. The string search algorithm by Knuth, Morris and Pratt. *Archive of Formal Proofs*, December 2017. `http://isa-afp.org/entries/Knuth_Morris_Pratt.html`, Formal proof development.

20 Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *Proc. of ITP*. Springer, 2017.

21 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 999–1026, Cham, 2018. Springer International Publishing.

22 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *ITP*, pages 332–337. Springer, Aug 2012.

23 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. `http://isa-afp.org/entries/Separation_Algebra.html`, Formal proof development.

24 Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. `arXiv:https://doi.org/10.1137/0206024`, `doi:10.1137/0206024`.

25 Alexander Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.

26 Peter Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.

27 Peter Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

28 Peter Lammich. Refinement based verification of imperative data structures. In Jeremy Avigad and Adam Chlipala, editors, *CPP 2016*, pages 27–36. ACM, 2016.

29 Peter Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017.

30 Peter Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In *SAT*, pages 457–463, 2017.

31 Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of ITP*, pages 219–234, 2016.

32 Peter Lammich and S. Reza Sefidgar. Formalizing network flow algorithms: A refinement approach in Isabelle/HOL. *J. Autom. Reasoning*, 62(2):261–280, 2019. `doi:10.1007/s10817-017-9442-4`.

33 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In Lennart Beringer and Amy P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

34 Yong Li. Knuth-Morris-Pratt code snippet. URL: `https://gist.github.com/yongpitt/5704216`.

35 LLVM language reference manual. URL: `https://llvm.org/docs/LangRef.html`.

36 Andreas Lochbihler. Java and the Java Memory Model - A unified, machine-checked formalisation. In *Proc. of ESOP*, pages 497–517, 2012. `doi:10.1007/978-3-642-28869-2\_25`.

37 George Markowsky. Chain-complete posets and directed sets with applications. *algebra universalis*, 6(1):53–68, Dec 1976. `doi:10.1007/BF02485815`.

38 Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar 2016. `doi:10.1007/s10817-015-9360-2`.

39 MLton. URL: `http://mlton.org/`.

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

41 Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

42 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

43 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc of. Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

44 Stringbench benchmark suite. URL: `https://github.com/almondtools/stringbench`.

45 Philip Wadler. Theorems for free! In *Proc. of FPCA*, pages 347–359. ACM, 1989.

46 Conrad Watt. Mechanising and verifying the webassembly specification. In *Proc. of CPP*, pages 53–65, 2018. `doi:10.1145/3167082`.

47 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.

48 Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *TACAS 2018*, pages 61–78, 2018.

49 Lei Yu. A formal model of ieee floating point arithmetic. *Archive of Formal Proofs*, July 2013. `http://isa-afp.org/entries/IEEE_Floating_Point.html`, Formal proof development.

50 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. of POPL*, pages 427–440. ACM, 2012. URL: `http://doi.acm.org/10.1145/2103656.2103709`, `doi:10.1145/2103656.2103709`.

51 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. *SIGPLAN Not.*, 48(6):175–186, June 2013. URL: `http://doi.acm.org/10.1145/2499370.2462164`, `doi:10.1145/2499370.2462164`.