

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



© Peter Lammich;

licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

46 in the definition of new data structures, exploiting ‘free theorems’ [45] that it obtains from
 47 parametricity properties of the abstract data types. Using Imperative HOL as backend, we
 48 gained some additional performance: For example, the GRAT tool [29, 30] provides a verified
 49 checker for UNSAT certificates in the DRAT format [47]. It is faster than the unverified
 50 state-of-the-art checker DRAT-TRIM [47], which is written in C. However, the GRAT tool
 51 spends most of its run time in an unverified certificate preprocessor. Nevertheless, optimizing
 52 the verified part of the code is important: The very same technique was also implemented in
 53 Coq, using purely functional data structures [12, 11]. There, the verified code was actually
 54 the bottleneck¹.

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

71 2 Isabelle-LLVM

72 2.1 State Monad

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

76
$$\langle 'a, 's \rangle M = M (\text{run}: 's \Rightarrow \langle 'a, 's \rangle \text{mres}) \quad \langle 'a, 's \rangle \text{mres} = \text{NTERM} \mid \text{FAIL} \mid \text{SUCC } 'a \ 's$$

79 An entity of type $\langle 'a, 's \rangle M$ contains a function $\langle \text{run} \rangle$ that maps a start state of type $\langle 's \rangle$ to
 80 a *monad result* that indicates either nontermination, a failure, or a successful execution with
 81 a result of type $\langle 'a \rangle$ and a new state. We define the standard monad combinators:

82
$$\begin{aligned} \text{return } x &= M (\lambda s. \text{SUCC } x \ s) & \text{get} &= M (\lambda s. \text{SUCC } s \ s) \\ \text{fail} &= M (\lambda _. \text{FAIL}) & \text{set } s &= M (\lambda _. \text{SUCC } () \ s) \\ \text{bind } m \ f &= M (\lambda s. \text{case run } m \ s \text{ of } \text{SUCC } x \ s \Rightarrow \text{run } (f \ x) \ s \mid r \Rightarrow r) \\ \text{assert } \Phi &= \text{if } \Phi \text{ then return } () \text{ else fail} \end{aligned}$$

88 That is, $\langle \text{return } x \rangle$ returns result $\langle x \rangle$ without changing the state, $\langle \text{fail} \rangle$ aborts the com-
 89 putation, $\langle \text{get} \rangle$ returns the current state, and $\langle \text{set } s \rangle$ updates the current state. Finally,
 90 $\langle \text{bind } m \ f \rangle$ first executes $\langle m \rangle$, and then $\langle f \rangle$ with the result of $\langle m \rangle$. If $\langle m \rangle$ fails or does not

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

91 terminate, the whole bind fails or does not terminate. The derived $\langle \text{assert } \Phi \rangle$ combinator
 92 can be conveniently used to abort the computation if some precondition is violated, e.g., on
 93 division by zero.

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

101 2.2 Memory Model

102 We use a high-level memory model that does not directly expose the bit-level representation
 103 of values and assumes an infinite supply of memory. The memory is modeled as a list of
 104 blocks. Each block is either deallocated, or it is a list of values. A value is a pair of values, a
 105 pointer, or an integer. We model memory by the following data types²:

```
106 memory = MEMORY (block list)           block = val list option
107 val = PAIR val val | PRIM primval      primval = PV_INT lint | PV_PTR rptr
108
109
```

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

```
113 rptr = NULL | ADDR nat nat (va_dir list)   va_dir = PFST | PSND
114
115
```

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

118 For the rest of this paper, we will use the state monad with a memory as state. Thus,
 119 we define the type $\langle 'a llm = ('a, memory) M \rangle$. It is straightforward to define functions
 120 $\langle load :: rptr \Rightarrow val llm \rangle$ and $\langle put :: val \Rightarrow rptr \Rightarrow unit llm \rangle$ to read/write a value from/to a
 121 pointer, or fail if the pointer is invalid. For the actual store function, we check that the
 122 structure of the value does not change, i.e. pairs remain pairs, pointers remain pointers, and
 123 words of width w remain words of width w :

```
124 store x p = do { y ← load p; assert (vstruct x = vstruct y); put x p }
125 where
126 vstruct (PAIR a b) = VS_PAIR (vstruct a) (vstruct b)
127 vstruct (PRIM (PV_PTR _)) = VS_PTR
128 vstruct (PRIM (PV_INT w)) = VS_INT (width w)
129
130
```

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.

XX:4 Generating Verified LLVM from Isabelle/HOL

131 Similarly, we define an allocate and a free function:

132

```
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
```

133 Here, $\langle l_1 @ l_2 \rangle$ concatenates two lists, $\langle |l| \rangle$ is the length of list $\langle l \rangle$, $\langle l!i \rangle$ is the i th element of
134 $\langle l \rangle$, and $\langle l[i:=x] \rangle$ replaces the i th element of $\langle l \rangle$ by $\langle x \rangle$. The allocate function takes an initial
135 value and a block size, appends a new block to the memory, and returns a pointer to the
136 start of the new block (value index 0, and value address []). The free function expects a
137 pointer to the start of a block, checks that this block is not already deallocated, and then
138 deallocates the block by setting it to $\langle None \rangle$.

139 2.3 Towards a Shallow Embedding

140 While we explicitly model values in memory by the type $\langle val \rangle$, we model values in registers
141 in a more shallow fashion: We identify LLVM registers with Isabelle variables that have a
142 type of shape $\langle T = T \times T \mid n \text{ word} \mid T \text{ ptr} \rangle$. Here, $\langle \times \rangle$ is Isabelle's product type, $\langle n \text{ word} \rangle$
143 is the n bit word type from Isabelle's word library³, and $\langle 'a \text{ ptr} \rangle$ is a pointer with an attached
144 phantom type for the value pointed to ($\langle 'a \text{ ptr} = PTR rptr \rangle$). For each type $\langle 'a \rangle$ of shape
145 $\langle T \rangle$, we define the functions:

146

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

150 such that

151

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

155 Here, $\langle TYPE('a) :: 'a \text{ itself} \rangle$ reflects type $\langle 'a \rangle$ into a term. The functions $\langle to_val \rangle$ and
156 $\langle from_val \rangle$ inject a T-shaped type $\langle 'a \rangle$ into a value with structure $\langle struct_of \ TYPE('a) \rangle$.
157 Moreover, $\langle init :: 'a \rangle$ corresponds to the all-zeroes value, i.e., the value where all pointers are
158 null pointers, and all integers are 0.

159 2.4 Instructions

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

162

```
163 ll_load :: 'a ptr ⇒ 'a llm  
164 ll_load (PTR p) = do {  
165   v ← load p;  
166   assert (vstruct v = struct_of TYPE('a));  
167   return (from_val v) }
```

168

³ For convenient notation, we use the type $\langle n \text{ word} \rangle$ 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 $\langle len \rangle$, which provides a function $\langle len_of :: 'a :: len \text{ itself} ⇒ nat \rangle$ to extract the length as a term.

169 It loads a value from the specified pointer, checks that its structure matches the expected
170 type $\langle a \rangle$, and then converts the value to $\langle a \rangle$.

171 For allocation and deallocation, we provide the instructions:

```
172  $ll\_malloc :: 'a \textit{ itself} \Rightarrow n \textit{ word} \Rightarrow 'a \textit{ ptr} \textit{ ULM}$        $ll\_free :: 'a \textit{ ptr} \Rightarrow \textit{ unit} \textit{ ULM}$   
173  
174
```

175 Note that LLVM does not contain a heap manager. Instead, we assume that the generated
176 code will be linked with the C standard library, and let the code generator produce calls to
177 $\langle alloc \rangle$ and $\langle free \rangle$. We also define instructions to access the elements of a pair, to offset a
178 pointer, and to advance a pointer into a pair. The code generator maps these instructions to
179 the corresponding LLVM instructions $\langle getelementptr \rangle$, $\langle insertvalue \rangle$, and $\langle extractvalue \rangle$.

180 Integer instructions are defined on the $\langle n \textit{ word} \rangle$ type. For example, we define:

```
181  $ll\_udiv :: n \textit{ word} \Rightarrow n \textit{ word} \Rightarrow n \textit{ word} \textit{ ULM}$   
182  $ll\_udiv \ a \ b = \textit{ do } \{ \textit{ assert } (b \neq 0); \textit{ return } (a \textit{ div } b) \}$   
183  
184
```

185 where $\langle div \rangle$ is the unsigned division from Isabelle's word library. Note the use of assertions
186 to exclude undefined behavior, e.g., division by zero.

187 2.5 Modeling Control Flow

188 Next, we put together instructions to form procedure bodies. We only allow structured
189 control flow via if-then-else, while, procedure calls, and sequential composition: The body of
190 a procedure is modeled by an Isabelle term of type $\langle a \textit{ ULM} \rangle$ and shape $\langle block \rangle$, where

```
191  $block = \textit{ do } \{ \textit{ var} \leftarrow \textit{ cmd}; \textit{ block} \} \mid \textit{ return } \textit{ var}$   
192  $\textit{ cmd} = ll\_<opcode> \ \textit{ arg}^* \mid \textit{ proc\_name} \ \textit{ arg}^* \mid ll\_if \ \textit{ arg} \ \textit{ block} \ \textit{ block} \mid ll\_while \ \textit{ block} \ \textit{ block}$   
193  $\textit{ arg} = \textit{ var} \mid \textit{ number} \mid \textit{ null} \mid \textit{ init}$   
194  
195
```

196 with

```
197  $ll\_if :: 1 \textit{ word} \Rightarrow 'a \textit{ ULM} \Rightarrow 'a \textit{ ULM} \Rightarrow 'a \textit{ ULM}$   
198  $ll\_if \ b \ t \ e = \textit{ if } b=1 \ \textit{ then } t \ \textit{ else } e$   
199  
200  
201  $ll\_while :: ('a \Rightarrow 1 \textit{ word} \textit{ ULM}) \Rightarrow ('a \Rightarrow 'a \textit{ ULM}) \Rightarrow 'a \Rightarrow 'a \textit{ ULM}$   
202  $ll\_while \ b \ c \ s = \textit{ do } \{ \textit{ ctd} \leftarrow b \ s; ll\_if \ \textit{ ctd} \ (\textit{ do } \{ s \leftarrow c \ s; ll\_while \ b \ c \ s \}) \ (\textit{ return } s) \}$   
203
```

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

213 ► **Example 1.** Figure 1 shows the Isabelle specification of a procedure named $\langle fib \rangle$, which
214 takes a 64 bit word argument, and returns a 64 bit word. Our semantics can be directly
215 executed inside Isabelle. The following Isabelle command evaluates $\langle fib \rangle$ on the first few
216 natural numbers, and an empty memory:

```
217 value  $\langle map \ (\lambda n. \textit{ run } (fib \ n) \ (\textit{ MEMORY } [])) \ [0,1,2,3] \rangle$   
218  $(\ast \ \textit{ output}: [\textit{ SUCC } 0 \ (\textit{ MEMORY } []), \ \textit{ SUCC } 1 \ \dots, \ \textit{ SUCC } 1 \ \dots, \ \textit{ SUCC } 2 \ \dots] \ \ast)$   
219  
220
```

```

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 $\langle proc\ x_1 .. x_n = block \rangle$ is mapped to an LLVM function named $\langle proc \rangle$. A block is mapped to a control flow graph. Instructions and procedure calls are directly mapped to LLVM instructions and calls. An $\langle x \leftarrow ll_if\ b\ t\ e \rangle$ is translated to conditional branching, using a ϕ -instruction to define the result register $\langle x \rangle$ when joining the control flow. An $\langle x \leftarrow ll_while\ b\ c\ s \rangle$ is translated similarly.

► **Example 2.** Figure 2 displays the output of our code generator for the $\langle fib \rangle$ 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 $\langle ll_malloc :: 'a\ itself \Rightarrow n\ word \Rightarrow 'a\ ptr\ lLM \rangle$ instruction. It has to be mapped to the function $\langle void*\ calloc(size_t, size_t) \rangle$ from the C standard library.

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

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

263 2.7 Preprocessing

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

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

283 2.7.1 Inlining

284 Inlining first applies user defined rewrite rules and then flattens nested expressions, converting
 285 function calls to the shape $\langle r \leftarrow f\ x_1 \dots x_n \rangle$ or $\langle r \leftarrow \mathbf{return}\ (f\ x_1 \dots x_n) \rangle$, where the x_i
 286 are either constants, variables, or *monadic* arguments of type $\langle \dots \Rightarrow _ U M \rangle$. Subterms of
 287 type $\langle _ U M \rangle$ are recursively flattened. We iterate the rewriting and flattening steps until a
 288 fixed point is reached.

289 ► **Example 3.** Consider the following definition of the constant $\langle fib' \rangle$:

290 $fib' :: m\ word \Rightarrow m\ word\ U M$
 291

```

292 fib' n = if n ≤ 1 then return n
293         else do { n1 ← fib' (n - 1); n2 ← fib' (n - 2); return (n1 + n2) }
294

```

When started with $\langle \text{fib}' :: 64 \text{ word} \Rightarrow 64 \text{ word llm} \rangle$, the preprocessor automatically translates this equation to the equation displayed in Figure 1. During the translation, it uses the following inlining rules:

```

298 if b then c else t = llc_if (from_bool b) c t           return (a + b) = ll_add a b
299 return (from_bool (a ≤ b)) = ll_icmp_ule a b           return (a - b) = ll_sub a b
300
301

```

Our default setup contains similar rules for the other operations, as well as rules to map tuples and case-distinctions over tuples to $\langle \text{insertvalue} \rangle$ and $\langle \text{extractvalue} \rangle$ 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 $\langle p \rangle$ for which it has an unfold rule of the form $\langle p = F p \rangle$. It then defines a new constant $\langle f x_1 \dots x_n = F (f x_1 \dots x_n) \rangle$, where the $\langle x_i \rangle$ are the free variables in the pattern $\langle p \rangle$. Finally, it replaces $\langle p \rangle$ by $\langle f x_1 \dots x_n \rangle$ 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 $\langle \text{llc_while} \rangle$ 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:

```

317 euclid :: 64 word ⇒ 64 word ⇒ 64 word
318 euclid a b = do {
319   (a,b) ← llc_while
320     (λ(a,b) ⇒ ll_cmp (a ≠ b))
321     (λ(a,b) ⇒ if (a ≤ b) then return (a,b-a) else return (a-b,b))
322     (a,b);
323   return a }
324
325

```

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

```

327 euclid a b = do {
328   (a, b) ← euclid0 (a, b);
329   return a }
330
331 euclid0 s = do {
332   ctd ← case s of (a, b) ⇒ ll_cmp (a ≠ b);
333   llc_if ctd (do {
334     s ← case s of (a, b) ⇒ if a ≤ b then return (a, b - a) else return (a - b, b);
335     euclid0 s
336   }) (return s) }
337

```

That is, it defined a new constant $\langle \text{euclid}_0 \rangle$ 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 ($\langle va_dir\ list \rangle$) 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:

```

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, $\langle [k \mapsto v] \rangle$ is the partial function that maps $\langle k \rangle$ to $\langle v \rangle$, and $\langle i \cdot a \rangle$ prepends the item $\langle i \rangle$ to all addresses in the domain of $\langle a \rangle$. 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 ($\langle val\ list \rangle$) 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 $\langle ablock = (nat \Rightarrow aval) \times nat\ option \rangle$. 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 $\langle None \rangle$. Finally, we define $\langle amemory = nat \Rightarrow ablock \rangle$, and a function $\langle \alpha :: memory \Rightarrow amemory \rangle$ 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 $\langle asn = amemory \Rightarrow bool \rangle$ are called *assertions*. The *weakest precondition* of a program $\langle c :: 'a\ llm \rangle$, a *postcondition* $\langle Q :: 'a \Rightarrow asn \rangle$, and a memory $\langle s \rangle$ is defined as:

```

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

```

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

386 For assertions $\langle P \rangle$ and $\langle Q \rangle$, the *separating conjunction* $\langle P * Q \rangle$ describes a memory that
 387 can be split into two disjoint parts described by $\langle P \rangle$ and $\langle Q \rangle$, respectively:

$$388 \quad (P * Q)\ s = \exists s_1\ s_2. s_1 \# s_2 \wedge s = s_1 + s_2 \wedge P\ s_1 \wedge Q\ s_2$$

391 Validity of a *Hoare triple* $\langle \{P\}\ c\ \{Q\} \rangle$ is defined as follows:

$$392 \quad \models \{P\}\ c\ \{Q\} = \forall F\ s. (P * F)\ (\alpha\ s) \longrightarrow wp\ c\ (\lambda r\ s'. (Q\ r * F)\ s')\ s$$

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

399 3.3 Basic Rules

400 Once we have set up the separation algebra and the abstraction function, we can prove Hoare
 401 triples for the basic operations of our memory model. For example, we prove the following
 402 rules for $\langle allocn \rangle$ and $\langle free \rangle$:

$$403 \quad \models \{\square\}\ allocn\ v\ n\ \{\lambda p. malloc_tag\ n\ p * range\ \{0..<n\}\ (\lambda_. v)\ p\}$$

$$404 \quad \models \{malloc_tag\ n\ p * \exists blk. range\ \{0..<n\}\ blk\ p\}\ free\ p\ \{\lambda_. \square\}$$

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

$$413 \quad \models \{n \neq 0\}\ ll_malloc\ TYPE('a)\ n\ \{\lambda p. range\ \{0..<n\}\ (\lambda_. init)\ p * malloc_tag\ n\ p\}$$

$$414 \quad \models \{range\ \{0..<n\}\ blk\ p * malloc_tag\ n\ p\}\ ll_free\ p\ \{\lambda_. \square\}$$

$$415 \quad \models \{pto\ x\ p\}\ ll_load\ p\ \{\lambda r. r=x * pto\ x\ p\}$$

$$416 \quad \models \{pto\ xx\ p\}\ ll_store\ x\ p\ \{\lambda_. pto\ x\ p\}$$

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

422 3.4 Automating the VCG

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

$$426 \quad \llbracket \bigwedge F\ s. STATE\ (P * F)\ s \implies wp\ c\ (\lambda r\ s'. (Q\ r * F)\ s')\ s \rrbracket \implies \models \{P\}\ c\ \{Q\}$$

429 where $\langle STATE\ P\ s = P\ (\alpha\ s) \rangle$. In general, the VCG operates on subgoals of the form
 430 $\langle STATE\ P\ s \implies wp\ c\ Q\ s \rangle$. It then iteratively performs one of the following steps⁴:

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.

431 **simplification** Apply a rewrite rule to transform $\langle wp\ c\ Q\ s \rangle$ into some equivalent proposition.

432 For example, binding is resolved by the rule:

$$433 \quad wp\ (\text{do}\ \{x \leftarrow m; f\ x\})\ Q\ s = wp\ m\ (\lambda x. wp\ (f\ x)\ Q)\ s$$

436 **rule** If there is a Hoare triple of the form $\langle \models \{P\}\ c\ \{Q\} \rangle$, the VCG tries to infer a frame $\langle F \rangle$
 437 such that $\langle P \vdash P' * F \rangle$, and replaces the goal by $\langle STATE\ (Q' * F)\ s' \implies Q\ s' \rangle$ for a fresh
 438 $\langle s' \rangle$. Here, $\langle P \vdash Q = \forall s. P\ s \implies Q\ s \rangle$ denotes entailment.

439 **final** If the goal has the form $\langle STATE\ P\ s \implies Q\ s \rangle$ such that $\langle Q \rangle$ is not of the form
 440 $\langle wp\ _ _ _ \rangle$, a heuristics is used to prove $\langle P \vdash Q \rangle$.

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

445 **► Example 5.** Recall the function $\langle euclid :: 64\ word \Rightarrow 64\ word \Rightarrow 64\ word\ llm \rangle$ from Ex-
 446 ample 4. We prove the following Hoare triple:

$$447 \quad \models \{uint_{64}\ a\ a_{\dagger} * uint_{64}\ b\ b_{\dagger} * 0 < a * 0 < b\}\ euclid\ a_{\dagger}\ b_{\dagger}\ \{\lambda r_{\dagger}. uint_{64}\ (gcd\ a\ b)\ r_{\dagger}\}$$

450 Here, $\langle uint_{64}\ a\ a_{\dagger} \rangle$ states that $\langle a_{\dagger} :: 64\ word \rangle$ is an unsigned integer with value $\langle a :: int \rangle$, where
 451 $\langle int \rangle$ is the type of (mathematical) integers in Isabelle, and $\langle gcd \rangle$ is Isabelle's greatest common
 452 divisor function. After annotating a suitable loop invariant, the VCG generates the following
 453 two verification conditions:

$$454 \quad \llbracket gcd\ x\ y = gcd\ a\ b; x \neq y; x \leq y; \dots \rrbracket \implies gcd\ x\ (y - x) = gcd\ a\ b$$

$$455 \quad \llbracket gcd\ x\ y = gcd\ a\ b; \neg x \leq y; \dots \rrbracket \implies gcd\ (x - y)\ y = gcd\ a\ b$$

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

459 3.5 Data Structures and Basic Refinement

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

469 In separation logic based reasoning, a data structure provides a *refinement assertion*
 470 $\langle A\ x\ x_{\dagger} :: assn \rangle$, which describes that the abstract value $\langle x \rangle$ is implemented by the concrete
 471 value $\langle x_{\dagger} \rangle$. We define refinement assertions to implement integers and natural numbers by n
 472 bit words, and to implement lists by blocks of memory. On top of that, we define more complex
 473 data structures like dynamic arrays. Note that new data structures can easily be added. In
 474 general, an implementation does not completely implement an abstract mathematical concept.
 475 For example, n bit words can only represent the integers $\langle sints\ n = \{-2^{n-1}.. < 2^{n-1}\} \rangle$, and
 476 hash-tables can only represent finite sets. Thus, the rules for the operations generally come
 477 with additional preconditions. For example, the rule to implement subtraction on integers by
 478 subtraction on n bit words is the following:

```

479  $\models \{ \text{shint}_n \ a \ a_{\dagger} * \text{shint}_n \ b \ b_{\dagger} * a - b \in \text{sints } n \} \ \text{ll\_sub} \ a_{\dagger} \ b_{\dagger} \ \{ \lambda r_{\dagger}. \text{shint}_n \ (a - b) \ r_{\dagger} \}$ 
480
481   for  $a_{\dagger} \ b_{\dagger} :: n \ \text{word}$  and  $a \ b :: \text{int}$ 
482

```

Here, $\langle \text{shint}_n \rangle$ implements mathematical integers by n -bit words. Note that the postcondition does not mention the operands $\langle a, b \rangle$ again, though they are still valid after the operation. As $\langle \text{shint}_n \rangle$ 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 $\langle \text{euclid} \rangle$ 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 $\langle \text{hnr } \Gamma \ m_{\dagger} \ \Gamma' \ R \ m \rangle$ means that concrete program $\langle m_{\dagger} \rangle$ implements abstract program $\langle m \rangle$, where $\langle \Gamma \rangle$ contains the refinements for the variables before the execution, $\langle \Gamma' \rangle$ contains the refinements after the execution, and $\langle R \rangle$ is the refinement assertion for the result of $\langle m \rangle$. For example, a $\langle \text{bind} \rangle$ is processed by the following rule:

```

506  $\llbracket \text{hnr } \Gamma \ m_{\dagger} \ \Gamma' \ R_x \ m;$ 
507  $\ \wedge x \ x_{\dagger}. \text{hnr} \ (R_x \ x \ x_{\dagger} * \Gamma') \ (f_{\dagger} \ x_{\dagger}) \ (R'_x \ x \ x_{\dagger} * \Gamma'') \ R_y \ (f \ x);$ 
508  $\ \text{MK\_FREE } R'_x \ \text{free};$ 
509  $\rrbracket \implies \text{hnr } \Gamma \ (\text{do } \{ x_{\dagger} \leftarrow m_{\dagger}; r_{\dagger} \leftarrow f_{\dagger} \ x_{\dagger}; \text{free } x_{\dagger}; \text{return } r_{\dagger} \}) \ \Gamma'' \ R_y \ (\text{do } \{ x \leftarrow m; f \ x \})$ 
510
511

```

To refine $\langle x \leftarrow m; f \ x \rangle$, we first execute $\langle m \rangle$, synthesizing the concrete program $\langle m_{\dagger} \rangle$ (line 1). The state after $\langle m \rangle$ is $\langle R_x \ x \ x_{\dagger} * \Gamma' \rangle$, where $\langle x \rangle$ is the result created by $\langle m \rangle$. From this state, we execute $\langle f \ x \rangle$ (line 2). The new state is $\langle R'_x \ x \ x_{\dagger} * \Gamma'' * R_y \ y \ y_{\dagger} \rangle$, where $\langle y \rangle$ is the result of $\langle f \ x \rangle$. Now, the variable $\langle x \rangle$ goes out of scope, such that it has to be deallocated. The predicate $\langle \text{MK_FREE } R'_x \ \text{free} = \forall x \ x_{\dagger}. \models \{ R'_x \ x \ x_{\dagger} \} \ \text{free } x_{\dagger} \ \{ \lambda _. \square \} \rangle$ (line 3) states that $\langle \text{free} \rangle$ is a deallocator for data structures implemented by refinement assertion $\langle R'_x \rangle$. Note that the refinement for variable $\langle x \rangle$ may change: If $\langle f_{\dagger} \ x_{\dagger} \rangle$ overwrites $\langle x_{\dagger} \rangle$, the refinement assertion for $\langle x \rangle$ will be changed to the special assertion $\langle \text{invalid} \rangle$. The deallocator for $\langle \text{invalid} \rangle$ 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

526 precision integers and algebraic data types, while we have only fixed width words and pairs.
 527 Up to now, we have added the implementation of integers and natural numbers by n bit words
 528 and some basic container data structures like dynamic arrays, bit-vectors, and min-heaps.
 529 Thereby, we could reuse the existing infrastructure of the Sepref tool: For example, there is
 530 support to automatically generate rules that also support refinement of the elements of a
 531 data structure, exploiting ‘free theorems’ [45] which stem from parametricity properties of
 532 the abstract types.

533 5 Case Studies

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

537 5.1 Binary Search

538 Binary search is a simple algorithm to find a value in a sorted array. Despite its simplicity, it
 539 has a history of flawed implementations⁵, making it a natural example for formal verification.

540 We start with a high-level specification: For a list $\langle xs \rangle$ and a value $\langle x \rangle$, find the index of
 541 the first element greater or equal to $\langle x \rangle$. We define the following constant:

```
542 fi_spec xs x = spec i. i = find_index ( $\lambda y$ .  $x \leq y$ ) xs
```

545 where $\langle find_index\ P\ xs \rangle$ is a standard list function that returns the index of the first element
 546 in $\langle xs \rangle$ that satisfies $\langle P \rangle$, or $\langle length\ xs \rangle$ if there is no such element.

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

```
548 bin_search xs x  $\equiv$  do {  

  549   (l,h)  $\leftarrow$  while  

  550   ( $\lambda(l,h)$ .  $l < h$ )  

  551   ( $\lambda(l,h)$ . do {  

  552     assert ( $l < length\ xs \wedge h \leq length\ xs \wedge l \leq h$ );  

  553     let  $m = l + (h - l) \text{ div } 2$ ;  

  554     if  $xs!m < x$  then return ( $m+1,h$ ) else return ( $l,m$ )  

  555   })  

  556   ( $0, length\ xs$ );  

  557   return l }
```

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

```
561 bs_correct: sorted xs  $\implies$  bin_search xs x  $\leq$  fi_spec xs x
```

564 Finally, we invoke our adapted Sepref tool:

```
565 sepref_definition bs_impl [llvm_code] is bin_search  

  566  $:: (larray_{64}\ sint_{64})^k \rightarrow sint_{64}^k \rightarrow snat_{64}$   

  567 unfolding bin_search_def [...] by sepref  

  568 export_llvm bs_impl file bin_search.ll  

  569 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].

572 This produces an Isabelle-LLVM program $\langle bs_impl \rangle$, exports it to actual LLVM text, and
 573 proves the refinement theorem $\langle bs_impl_correct \rangle$:

$$574 \quad (bs_impl, fi_spec) : [\lambda(xs, _). sorted\ xs] (larray_{64}\ sint_{64})^k \times sint_{64}^k \rightarrow snat_{64}$$

577 Here, $\langle snat_w \rangle$ implements natural numbers by signed w -bit words⁶. Moreover, $\langle larray_w\ A \rangle$
 578 refines a list to an array and a w -bit length field, the elements of the list being refined
 579 by assertion $\langle A \rangle$. The notation $\langle [\Phi] A_1^{k|d} \times \dots \times A_n^{k|d} \rightarrow R \rangle$ specifies a refinement with
 580 precondition $\langle \Phi \rangle$, such that the arguments are refined by $\langle A_1 \dots A_n \rangle$ and the result is refined
 581 by $\langle R \rangle$. The $\cdot^{k|d}$ annotations specify whether an argument is overwritten (k for keep, d for
 582 destroy). While we use this notation a lot in the Refinement Framework, it is straightforward
 583 to prove a standard Hoare triple from it. By unfolding some definitions we get:

$$584 \quad \models \{ larray_{64}\ sint_{64}\ xs\ xs_{\dagger} * sint_{64}\ x\ x_{\dagger} * sorted\ xs \}$$

$$585 \quad bs_impl\ xs_{\dagger}\ x_{\dagger}$$

$$586 \quad \{ \lambda i_{\dagger}. \exists i. larray_{64}\ sint_{64}\ xs\ xs_{\dagger} * snat_{64}\ i\ i_{\dagger} * i = find_index\ (\lambda y. x \leq y)\ xs \}$$

589 That is, if we start with an array $\langle xs_{\dagger} \rangle$ representing the sorted list $\langle xs \rangle$, and a 64-bit word
 590 $\langle x_{\dagger} \rangle$ representing the integer $\langle x \rangle$, then the array still represents $\langle xs_{\dagger} \rangle$, and the result $\langle i_{\dagger} \rangle$
 591 represents a natural number $\langle i \rangle$, which is equal to the correct index.

592 The Sepref tool implements mathematical integers by 64-bit words, proving absence of
 593 overflows. This is only possible because the assertion in $\langle bin_search \rangle$ explicitly states that
 594 the indexes are in bounds. Moreover, note the expression $\langle l + (h-l) \text{ div } 2 \rangle$ that we used to
 595 compute the midpoint index. On mathematical integers, it is equal to $\langle (l+h) \text{ div } 2 \rangle$. However,
 596 on fixed-width words, the latter may overflow, while the former does not⁷.

597 5.2 Knuth-Morris-Pratt String Search

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

$$600 \quad ss_spec\ s\ t = spec$$

$$601 \quad None \Rightarrow \nexists i. sublist_at\ s\ t\ i \mid$$

$$602 \quad Some\ i \Rightarrow sublist_at\ s\ t\ i \wedge (\forall ii < i. \neg sublist_at\ s\ t\ ii)$$

605 where $\langle sublist_at\ s\ t\ i \rangle$ specifies that list $\langle s \rangle$ occurs in list $\langle t \rangle$ at index $\langle i \rangle$:

$$606 \quad sublist_at\ s\ t\ i = \exists ps\ ss. t = ps@s@ss \wedge i = length\ ps$$

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

$$614 \quad (kmp_impl, ss_spec)$$

$$615 \quad : [\lambda s\ t. |s| + |t| < 2^{63}] (larray_{64}\ sint_{64})^k \times (larray_{64}\ sint_{64})^k \rightarrow snat_option_{64}$$

618 Here $\langle snat_option_{64} \rangle$ implements the type $\langle nat\ option \rangle$ by signed 64-bit words, mapping
 619 $\langle None \rangle$ 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 $0, 2, \dots < 5n$ in an array $[0, 5, \dots < 5n]$.

$a-l$	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 KMP⁸. 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 configurations⁹. 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.

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

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

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

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

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

680 **7 Related Work**

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

689 The Vellvm project [50, 51] verifies LLVM program transformations in Coq. To be useful,

690 e.g. as backend for clang, they have to formalize a substantial fragment of LLVM. On the
691 other hand, we can afford to formalize a simplified and abstract semantics that is just
692 powerful enough to cover what Sepref generates.

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

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

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

703 **8 Conclusions**

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

712 **Acknowledgement**

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

- 717 **1** Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon
718 Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge
719 University Press, New York, NY, USA, 2014.
- 720 **2** Joel Beeren, Matthew Fernandez, Xin Gao, Gerwin Klein, Rafal Kolanski, Japheth Lim,
721 Corey Lewis, Daniel Matichuk, and Thomas Sewell. Finite machine word library. *Archive*
722 *of Formal Proofs*, June 2016. http://isa-afp.org/entries/Word_Lib.html, Formal proof
723 development.
- 724 **3** Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledge-
725 hammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013. doi:10.1007/
726 s10817-013-9278-5.
- 727 **4** Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C
728 language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. URL: <http://xavierleroy.org/publi/Clight.pdf>.
- 730 **5** Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches
731 and mergesorts are broken. URL: [http://googleresearch.blogspot.com/2006/06/
732 extra-extra-read-all-about-it-nearly.html](http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html).
- 733 **6** Julian Brunner and Peter Lammich. Formal verification of an executable LTL model
734 checker with partial order reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. doi:
735 10.1007/s10817-017-9418-4.
- 736 **7** Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews.
737 Imperative functional programming with Isabelle/HOL. In Otmane Aït Mohamed, César A.
738 Muñoz, and Sofène Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 134–149.
739 Springer, 2008.
- 740 **8** C. Calcagno, P.W. O’Hearn, and Hongseok Yang. Local action and abstract separation logic.
741 In *LICS 2007*, pages 366–378, July 2007.
- 742 **9** Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel.
743 Vst-floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated*
744 *Reasoning*, 61, 02 2018. doi:10.1007/s10817-018-9457-5.
- 745 **10** Clang: a C language family frontend for LLVM. URL: <https://clang.llvm.org/>.
- 746 **11** Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp.
747 Efficient certified RAT verification. In *Proc. of CADE*. Springer, 2017.
- 748 **12** Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution
749 proof checking. In *Proc. of TACAS*, pages 118–135. Springer, 2017.
- 750 **13** Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck.
751 Efficiently computing static single assignment form and the control dependence graph. *ACM*
752 *Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: [http://doi.acm.org/10.
753 1145/115372.115320](http://doi.acm.org/10.1145/115372.115320), doi:10.1145/115372.115320.
- 754 **14** Deep spec project web page. URL: <https://deepspec.org/>.
- 755 **15** Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and
756 Jan-Georg Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of
757 *LNCS*, pages 463–478. Springer, 2013.
- 758 **16** Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with
759 watched literals using Imperative HOL. In *Proc. of CPP*, pages 158–171, 2018.
- 760 **17** David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don’t sweat the small
761 stuff: formal verification of C code without the pain. In *Proc. of PLDI ’14*, pages 429–439,
762 2014. doi:10.1145/2594291.2594296.
- 763 **18** Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement
764 in Isabelle/HOL. In *Proc. of ITP*, pages 100–115. Springer, 2013.
- 765 **19** Fabian Hellauer and Peter Lammich. The string search algorithm by Knuth, Morris and Pratt.
766 *Archive of Formal Proofs*, December 2017. [http://isa-afp.org/entries/Knuth_Morris_
767 Pratt.html](http://isa-afp.org/entries/Knuth_Morris_Pratt.html), Formal proof development.

- 768 20 Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking
769 of propositional proofs. In *Proc. of ITP*. Springer, 2017.
- 770 21 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal
771 Ahmed, editor, *Programming Languages and Systems*, pages 999–1026, Cham, 2018. Springer
772 International Publishing.
- 773 22 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *ITP*,
774 pages 332–337. Springer, Aug 2012.
- 775 23 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Separation algebra. *Archive of Formal*
776 *Proofs*, May 2012. http://isa-afp.org/entries/Separation_Algebra.html, Formal proof
777 development.
- 778 24 Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings.
779 *SIAM Journal on Computing*, 6(2):323–350, 1977. arXiv:<https://doi.org/10.1137/0206024>,
780 doi:10.1137/0206024.
- 781 25 Alexander Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43,
782 pages 1–13, 2010.
- 783 26 Peter Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99.
784 Springer, 2013.
- 785 27 Peter Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269.
786 Springer, 2015.
- 787 28 Peter Lammich. Refinement based verification of imperative data structures. In Jeremy Avigad
788 and Adam Chlipala, editors, *CPP 2016*, pages 27–36. ACM, 2016.
- 789 29 Peter Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer,
790 2017.
- 791 30 Peter Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal
792 correctness guarantees. In *SAT*, pages 457–463, 2017.
- 793 31 Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of*
794 *ITP*, pages 219–234, 2016.
- 795 32 Peter Lammich and S. Reza Sefidgar. Formalizing network flow algorithms: A refine-
796 ment approach in Isabelle/HOL. *J. Autom. Reasoning*, 62(2):261–280, 2019. doi:10.1007/
797 s10817-017-9442-4.
- 798 33 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to
799 Hopcroft’s algorithm. In Lennart Beringer and Amy P. Felty, editors, *ITP 2012*, volume 7406
800 of *LNCS*, pages 166–182. Springer, 2012.
- 801 34 Yong Li. Knuth-Morris-Pratt code snippet. URL: [https://gist.github.com/yongpitt/
802 5704216](https://gist.github.com/yongpitt/5704216).
- 803 35 LLVM language reference manual. URL: <https://llvm.org/docs/LangRef.html>.
- 804 36 Andreas Lochbihler. Java and the Java Memory Model - A unified, machine-checked formal-
805 isation. In *Proc. of ESOP*, pages 497–517, 2012. doi:10.1007/978-3-642-28869-2_25.
- 806 37 George Markowsky. Chain-complete posets and directed sets with applications. *algebra*
807 *universalis*, 6(1):53–68, Dec 1976. doi:10.1007/BF02485815.
- 808 38 Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language
809 for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar 2016. doi:10.1007/
810 s10817-015-9360-2.
- 811 39 MLton. URL: <http://mlton.org/>.
- 812 40 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic
813 into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014. doi:10.1017/
814 S0956796813000282.
- 815 41 Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publish-
816 ing Company, Incorporated, 2014.
- 817 42 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant*
818 *for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

- 819 **43** John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of*
820 *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
- 821 **44** Stringbench benchmark suite. URL: <https://github.com/almondtools/stringbench>.
- 822 **45** Philip Wadler. Theorems for free! In *Proc. of FPCA*, pages 347–359. ACM, 1989.
- 823 **46** Conrad Watt. Mechanising and verifying the webassembly specification. In *Proc. of CPP*,
824 pages 53–65, 2018. doi:10.1145/3167082.
- 825 **47** Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking
826 and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *SAT*
827 *2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.
- 828 **48** Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *TACAS*
829 *2018*, pages 61–78, 2018.
- 830 **49** Lei Yu. A formal model of iee floating point arithmetic. *Archive of Formal Proofs*, July 2013.
831 http://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.
- 832 **50** Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing
833 the LLVM intermediate representation for verified program transformations. In *Proc. of*
834 *POPL*, pages 427–440. ACM, 2012. URL: <http://doi.acm.org/10.1145/2103656.2103709>,
835 doi:10.1145/2103656.2103709.
- 836 **51** Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal
837 verification of SSA-based optimizations for LLVM. *SIGPLAN Not.*, 48(6):175–186, June 2013.
838 URL: <http://doi.acm.org/10.1145/2499370.2462164>, doi:10.1145/2499370.2462164.