An Isabelle/HOL Formalization of AProVE's Termination Method for LLVM IR

Max W. Haslbeck
René Thiemann
maximilian.haslbeck@uibk.ac.at
rene.thiemann@uibk.ac.at
University of Innsbruck
Innsbruck, Austria

Abstract

AProVE is a powerful termination prover for various programming languages, including a termination analysis method for imperative programs specified in the LLVM intermediate representation (IR). The method internally works in three steps: first, it transforms LLVM IR code into a symbolic execution graph; second, the graph is translated into an integer transition system; finally, termination of the transition system is proved by the back end of AProVE.

Since AProVE is unverified software, our aim is to increase its reliability by certifying the generated proofs. To this end, we require formal semantics of all program representations, i.e., for LLVM IR, for symbolic execution graphs and for integer transition systems. As the latter is already available, we define the former ones. We note that our semantics for LLVM IR use arithmetic with unbounded integers. We further verify the first and the second step of AProVE's termination method, including verified algorithms to check concrete proofs. Since the third step can already be certified, we obtain a complete formally verified method for certifying AProVE's termination proofs of LLVM IR programs. The whole formalization has been done in Isabelle/HOL and our certifier is available as a Haskell program via code generation.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Software verification**; **Automated static analysis**.

Keywords: formal verification, Isabelle/HOL, static program analysis, termination analysis

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01.
https://doi.org/10.1145/3437992.3439935

ACM Reference Format:
Max W. Haslbeck and René Thiemann. 2021. An Isabelle/HOL Formalization of AProVE's Termination Method for LLVM IR. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3437992.3439935

1 Introduction

Consider the C program in Figure 1, which simply counts down the value in variable x and then returns successfully. It terminates assuming that the external function call that assigns an arbitrary initial value to x terminates. Programs such as this one are part of the C_Integer category in the *Annual International Termination and Complexity Competition* (termCOMP) [8]. Termination provers need to show termination or non-termination of C programs in this category. As of now, the termination tools are only required to provide a yes/no-answer in combination with some human-readable output. So one has to trust that the analyzing tools do not contain any bugs and in case of conflicting answers of two tools, a human referee is necessary. Such problems occured early on in the history of the termination competition. It was decided to add special categories where termination provers need to generate machine-readable certificates which prove their answer and these where then checked by independent certifier tools. To have the highest trust possible in the certifiers, these where developed and verified within theorem provers like Isabelle/HOL and Coq.

The problem is that the verified certifiers so far are limited to core evaluation mechanisms such as term rewriting [5, 17] or integer transition systems [4]. Our goal is now to develop a highly trustworthy certifier for termination proofs of C programs. To this end, we use Isabelle/HOL to specify the foundational parts of the certification and to develop the accompanying algorithms.

Here we focus on certifying AProVE's [7] termination technique for C programs [15, 16]. Ströder et al. verified termination and memory safety of C programs by first compiling them to the LLVM Intermediate Representation (IR) [1, 12]. Compared to C, the LLVM IR has a simpler syntax and has clearly defined platform-independent semantics. From the LLVM IR code a so called *symbolic execution graph* (SEG) is

```
extern int __VERIFIER_nondet_int(void);

int main()
{
    int x;
    x = __VERIFIER_nondet_int();
    while (x >= 0) {
        x = x - 1;
    }
    return 0;
}
```

Figure 1. Example C code

created. The defining property of the SEG is that any (infinite) program run of the LLVM IR code is represented by an (infinite) path in the SEG. In the next step the SEG is converted to an integer transition system (ITS) such that the termination of the ITS implies the termination of the SEG and therefore the termination of the LLVM IR code.

In this work we verify this approach in the theorem prover Isabelle/HOL [13]. Our main contributions are

- a formal definition of the semantics of LLVM IR code,
- a formal definition of the SEG and its semantics,
- a formal verification of the fact that an SEG can simulate LLVM IR program runs and
- a formal verification of the correctness of the transformation of an SEG to an ITS.

Since there already exists a certifier for termination proofs of ITSs, we are in the end able to verify termination proofs for LLVM IR code. The resulting certifier is able to validate nearly 50 % of the termination proofs of AProVE for the C_integer programs from the latest termination competition.

Outline. In Section 2 we briefly explain the basics of Isabelle/HOL and LLVM needed for this paper. Section 3 describes our Isabelle definition of the LLVM semantics. In Section 4 we formally define the SEG, its semantics and how to verify that an SEG over-approximates an LLVM IR program. Section 5 explains the details of the implementation of the project in Isabelle and AProVE. It also features the experimental evaluation of our code. We conclude in Section 6 and also outline future work there.

Related Work. The Vellvm project formalized a significant subset of the semantics of the LLVM IR in the Coq theorem prover [18]. They cover a bigger part of the LLVM IR semantics than we do: for example, Vellvm contains a memory model and bit vector arithmetic, both of which are still future work for us. The Vellvm project was then able to verify optimization transformations typically done on LLVM IR. Although LLVM IR semantics were already available in Coq we decided to use Isabelle for our approach. The main reason for this decision is the availability of the IsaFoR library (which supports, for example, termination proofs of ITSs generated by AProVE) in Isabelle.

Lammich specified a subset of the LLVM IR semantics in Isabelle in order to generate verified LLVM IR functions [11]. His semantics contain proper bit vector operations and a memory model. We may try to integrate parts of these semantics into our work in the future.

The *Annual International Termination and Complexity Competition* (termCOMP) and the *Competition on Software Verification* (SV-COMP) [2] both feature categories on the termination of imperative programs, for example C or Java programs. Both competitions give a good overview of the latest research developments. In the SV-COMP competition, software verifiers have to provide a so called verification witness that is then used by independent validator software to check the verification result. The validators are unverified software, but with the usage of multiple independently developed validators a high degree of certainty is achieved. Our approach has the advantage that one only needs to trust our LLVM IR semantics definition and the Isabelle kernel to in turn be able to trust the results of our exported verified certifier.

Our project depends on the IsaFoR/CeTA libraries [17]. The IsaFoR/CeTA project was started with the aim of verifying output of termination provers for term rewrite systems in the termCOMP competition, but in the meantime CeTA can also certify termination proofs for integer transitions systems [4]. Our work is an extension to IsaFoR where we utilize its functionality to validate termination proofs of LLVM IR programs.

2 Preliminaries

2.1 Isabelle/HOL

Our formalization is developed in the proof assistant Isabelle/HOL [13]. When we use the name Isabelle throughout this work we mean, Isabelle/HOL, that is Isabelle extended with the HOL axioms. We state theorems and definitions in this paper in Isabelle syntax, with Isabelle keywords written in **bold**. Definitions and theorems presented below may slightly differ from the actual Isabelle source code. This was done to make them understandable to someone with a basic background in functional programming and logic but without the knowledge of special Isabelle intricacies. We make heavy use of Isabelle's `sum` type defined as **`datatype`** `('a, 'b) sum = Inl 'a | Inr 'b` (notation: `"'a + 'b"`) where `'a` and `'b` are type variables (see Figure 4 for an example of its usage). It is equivalent to Haskell's `Either` type and we adopt the same convention to encode error on the left (`Inl`) side and correct (or right) values on the right (`Inr`) side. We use it throughout our formalization as it has support for monadic notation and we can model basic error throwing and catching with it [14].

All lemmas and theorems stated here are proved in Isabelle. The theory files are available as part of the IsaFoR library.

```
define i32 @main() {
bb:
  %tmp = call i32 @__VERIFIER_nondet_int()
  br label %bb1

bb1:
  %.0 = phi i32 [ %tmp, %bb ], [ %tmp4, %bb3 ]
  %tmp2 = icmp sge i32 %.0, 0
  br i1 %tmp2, label %bb3, label %bb5

bb3:
  %tmp4 = sub nsw i32 %.0, 1
  br label %bb1

bb5:
  ret i32 0
}

declare i32 @__VERIFIER_nondet_int()
```

Figure 2. LLVM IR example

```
datatype basic_block = Basic_block
 (name : "name")
 (phis : "phi list")
 (instructions : "instruction list")
 (terminator : "terminator")

datatype llvm_fun = Function
 (fun_name: "name")
 (return_type: "llvm_type")
 (params: "parameter list")
 (blocks: "basic_block list")
| ExternalFunction
 (fun_name: "name")
 (return_type: "llvm_type")
 (params: "parameter list")

datatype llvm_prog = Llvm_prog
 (funs : "llvm_fun list")
```

Figure 3. `llvm_prog` definition in Isabelle

We also created a website for all the supplementary material[1] which includes links to HTML versions of the Isabelle theories. These can be viewed without installing Isabelle.

2.2 LLVM

The LLVM IR is defined in the LLVM Language Reference [1]. When we refer to LLVM throughout this paper we mean LLVM version 9.0.1 which we also used in our implementations. An LLVM IR program[2] consists of a list of functions, global variables and symbol table entries. We ignore global variables and symbol table entries for now and only allow function definitions and function declarations in our syntax. An LLVM function definition or declaration has a type, a name and list of parameters. A function definition also contains of a list of blocks. A block consists of a label, phi nodes, a list of instructions and a terminating instruction. Branching is only possible at a terminating instruction at the end of a block and only into another block inside the same function. Phi nodes are only allowed at the start of block and assign a new value to a variable based on the previous block. Phi nodes are a standard element of *single static assignment form* (SSA) [6]. In SSA form variables get only assigned once and it is often used in intermediate languages of compilers.

As an example, the C code within Figure 1 is compiled into LLVM code in Figure 2. When entering basic block `bb1` from block `bb`, the variable `%.0` gets assigned the value of `%tmp`; similarly it will be the value of `%tmp4` when entering from block `bb3`. The terminating instruction of block `bb1` is a conditional jump to block `bb3` or `bb5`, depending on the result `%tmp2` of the comparison in the previous line.

3 LLVM Semantics in Isabelle/HOL

3.1 LLVM Syntax

We adopt a straightforward approach to model the LLVM IR syntax in Isabelle/HOL. The abstract datatype definitions are similar to the ones used in the Vellvm project [18]. Figure 3 shows the definitions of basic blocks, functions and programs in Isabelle. Phi nodes, instructions and terminating instructions are separated in the `basic_block` definition even though they appear as one list in the LLVM IR source. The advantage is that then the type system will ensure certain properties on the input, e.g., that no terminator instruction appears in the middle of a basic block.

For function definitions we distinguish between internal and external functions. Only the former will have an implementation, given as a list of basic blocks. For example, the function `main` in the LLVM IR program of Figure 2 will become a `Function` in Isabelle, whereas `__VERIFIER_nondet_int` will become an `ExternalFunction`.

The formal syntax of LLVM IR does not enforce any restrictions regarding the SSA form. This is not a problem in our setting, since the property of being in SSA form is not exploited in AProVE's termination method.

3.2 Small Step Semantics

For modeling the semantics of LLVM IR programs, we first need a formal notion of LLVM constants, i.e., numbers.[3] In

[1] http://cl-informatik.uibk.ac.at/isafor/experiments/llvm/cpp2021

[2] The proper notion would be an LLVM IR *module*, but we prefer to speak of *programs*.

[3] As already mentioned, we do not have any memory model. Consequently, there is only one type that represents constants, namely numbers, which includes the Booleans that are represented by 0 and 1.

```
fun step :: "llvm_prog ⇒ ll_state ⇒ stuck + ll_state"
where
"step lf ls = case frames ls of
   (f#fs) ⇒
    case find_statement lf (pos f) of
        Inr (Instruction i) ⇒ run_instruction lf ls f fs i
      | Inr (Terminator t) ⇒ terminate_frame lf ls f fs t
      | _ ⇒ static_error ''Can't find next instruction''
 | [] ⇒ Inl Program_Termination"
```

Figure 4. Deterministic semantics: the `step` function

the formal setting constants are modeled as a pair of an unbounded integer together with a bit length. Here, the bit length is purely used for type checking, whereas the arithmetic is performed using mathematical integers ignoring any bit length. For example, consider the basic block `bb5` in Figure 2. The constant 0 of type `i32` would be modeled as pair (0,32). If one would replace the code of `bb5` by `ret i16 0`, this would result in an error during execution, since the returned constant would have a different type than that of the function `main`. By contrast, using `ret i32 (2^33)` would not be a problem, since internally our semantics use unbounded integer arithmetic. In future work, we may change this to a proper bit-vector semantics.

Next, we define the datatype `ll_state`, which is a list of frames, to model the machine state. A frame holds the current position (a triple of function name, block name and current line) and an assignment from program variables to LLVM constants, called stack.

Finally, we choose a small step operational semantics to model the LLVM IR semantics in Isabelle. This choice is motivated by the fact, that the termination of an LLVM IR program is then easily described as strong normalization of the small step relation.

We model the *deterministic* parts of our semantics by a function `step`, see Figure 4. Based on the current position in the first frame, it looks up the current instruction and returns a new `ll_state` or a value of type `stuck`. The `stuck` datatype encodes states where the program halts. This can be because of errors or if the program returns successfully. In order to execute an instruction, the `run_instruction` function is invoked, where a further case analysis on the kind of instruction is performed. At this point, `step` is incomplete, since only calls to internal functions are permitted (via the Isabelle function `call_function`), but calls to external functions – which might be non-deterministic – will result in an error state.

Our semantics are very similar to the one described in the Vellvm project. For instance, we use the same case analysis on the positions within a basic block as in Vellvm: the `step`-function in Figure 4 distinguishes between being in the main part of the basic block (`Instruction`), or being at the end of the block (`Terminator`); there is no case for phi instructions,

```
inductive assign_unknown_value for s f fs n s' where
"s' = update_frames_stack s f fs n (Inr c)
⟹ assign_unknown_value s f fs n s'"

definition call_arbitrary_function where
"call_arbitrary_function prog f fs n g ps s'=
  case map_of_funs prog fn of
      Some (ExternalFunction t fn ps') ⇒
        assign_unknown_value s f fs n s'
      | _ ⇒ s = call_function fn os"

definition step'
:: "llvm_prog ⇒ ll_state ⇒ stuck + ll_state ⇒ bool"
where
"step' prog s s' = case frames s of
   (f#fs) ⇒
    case find_statement prog (pos f) of
        Inr (Instruction (Named n (Call t g ps))) ⇒
          call_arbitrary_function prog f fs n g ps s'
      | _ ⇒ s'= step prog s)
 | [] ⇒ s'= step prog s)"
```

Figure 5. Full semantics: the `step'` predicate

since these are immediately executed after branching to a new basic block. [18, Section 4] gives a small overview of the details that one has to pay attention to when formally defining LLVM IR semantics in a theorem prover.

While `step` works for *internal* functions, we also need semantics that additionally cover *external* functions. To this end we introduce a non-deterministic relation in Figure 5, modeled by the predicate `step'`. We assume all external functions terminate and return an indeterminate integer value, i.e., one can choose an arbitrary constant `c` within the inductive predicate `assign_unknown_value`. Moreover, we reuse the functionality of `step` whenever possible. To be more concrete, `step'` only introduces a new case for function calls and otherwise refers to `step`; moreover, `call_arbitrary_function` also delegates internal function calls to `call_function` from the `step`-semantics.

Limitations. Right now our semantics do not correctly model the LLVM IR semantics. We do not have a memory model and we consider all integers as unbounded. Note that the latter deviation from the LLVM IR standard is also present in the annual termination competition (C_integer programs), as well as in the LLVM termination analysis method of AProVE [16]. Therefore undefined behavior in the case of invalid memory accesses or integer overflows cannot occur. Dividing by zero also leads to undefined behavior but our LLVM IR syntax and semantics do not yet implement the two division operations `sdiv` and `udiv`.

```
definition step'_relation :: "llvm_prog ⇒ ll_state rel"
where
"step'_relation prog = {(s, s') . step' prog s (Inr s')}"
```

Figure 6. Modeling Infinite Runs via the `step'_relation`

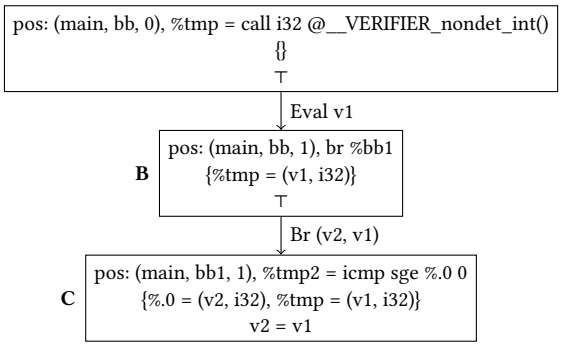

Figure 7. Start of SEG representing program in Figure 2

We did not implement *static* checks for errors such as violating SSA form, referencing undefined variables or typing errors. This is not a problem for a correct termination analysis, since these errors would result in programs that are rejected by the LLVM IR compiler and cannot be executed at all. However, our formal semantics still contain some of these checks, e.g., undefined variables and typing errors are detected during an execution of `step'` and result in an error-state at runtime. For readability reasons we did not include these checks in Figures 4 and 5, but they are visible in our Isabelle sources.

Termination. Note that the `step'` predicate relates an input state to the next state or to an error state. To study termination it suffices to concentrate on infinite runs, so in particular in runs where no error occurs. Therefore we define `step'_relation` as a relation between two (non-error) states, cf. Figure 6. Hence, a non-terminating LLVM program results in an infinite chain w.r.t. the `step'_relation`, i.e., termination of an LLVM program `prog` on initial states `init` is defined as

```
SN_on (step'_relation prog) init.
```

Here, `SN_on` is Isabelle's notion of strong normalization, i.e. the predicate is satisfied if and only if there exists no infinite chain using the relation `step'_relation prog` starting in `init`.

4 Symbolic Execution Graph

To prove termination of LLVM IR programs, AProVE uses *symbolic execution graphs* (SEG) as intermediate representations of program runs. Figure 7 and Figure 8 show parts of the SEG constructed from the LLVM IR code in Figure 2.

An SEG consists of a finite set of nodes, each node in an SEG being an abstract state that represents several concrete states. A node contains the current position, an assignment from program variables to logical variables and a formula over the logical variables. We call the formula in the abstract state the *knowledge base*. For instance, the position of node **C** in Figure 7 points to line 1 of building block `bb1` inside the function `main`, the values of program variables `%.0` and `%tmp` are stored in the logical variables v2 and v1, respectively, and the knowledge base expresses that both logical variables contain the same value. Hence, node **C** represents a frame of a concrete state that has the same position as node **C**, and where the assignment of program variables in the frame stores the same values for `%.0` and `%tmp`.

At this point it is interesting to observe that there is mismatch between abstract states and concrete states: a concrete state can have several frames – one for each (recursive) function call; by contrast, an abstract state describes a single frame. This mismatch is not present in [16], where an abstract state consists of a list of abstract frames. In that setting, function calls are handled by pushing and popping abstract frames. With this method one cannot handle recursive functions, whereas, the current implementation of AProVE can handle them. Furthermore, the current implementation of SEGs in AProVE does not correspond to the definition of SEGs in [16]. Instead it uses abstract states that encode a single frame as is described in this paper. The solution to the mismatch problem is to relate abstract states only to the first frame of any concrete state. In Section 4.3 we describe how we can then still handle function calls.

An SEG is constructed via several inference rules. The majority of the rules allow us to perform a symbolic evaluation step depending on the current instruction, e.g., assignment (Eval) or branching (Br). Moreover, there is a rule for case splitting (Refine) where a node gets outgoing edges to two different nodes. And finally, there is a rule for generalizations (Gen) where a start node can be connected to a target node and all abstract states represented by the start node are also represented by the the target node. For certain inference rules one needs to store a renaming of variables as we will show in the following example. Figure 7 and Figure 8 show parts of the SEG created from the LLVM program in Figure 2. Here, Figure 7 shows the first three nodes and Figure 8 shows the lower part of the SEG. The paths from the first nodes to the lower part all represent at least one run through the loop in the LLVM IR program. Therefore all program variables are assigned in node **D** and also in node **L**. They are both at the same position and node **D** is more general than node **L**, but one needs to rename the local variables accordingly, e.g., v12 to v18, since program variable `%.0` stores v12 in node **D**, but v18 in node **L**.

The soundness of the approach hinges on the property that whenever a concrete state can make a step via the LLVM semantics, there is a non-empty path in the SEG that represents that step. The correctness of the construction of an SEG has been shown "on paper" [16]. We now go one step further and show the correctness of the concepts surrounding an SEG in Isabelle. This includes the definition of an SEG

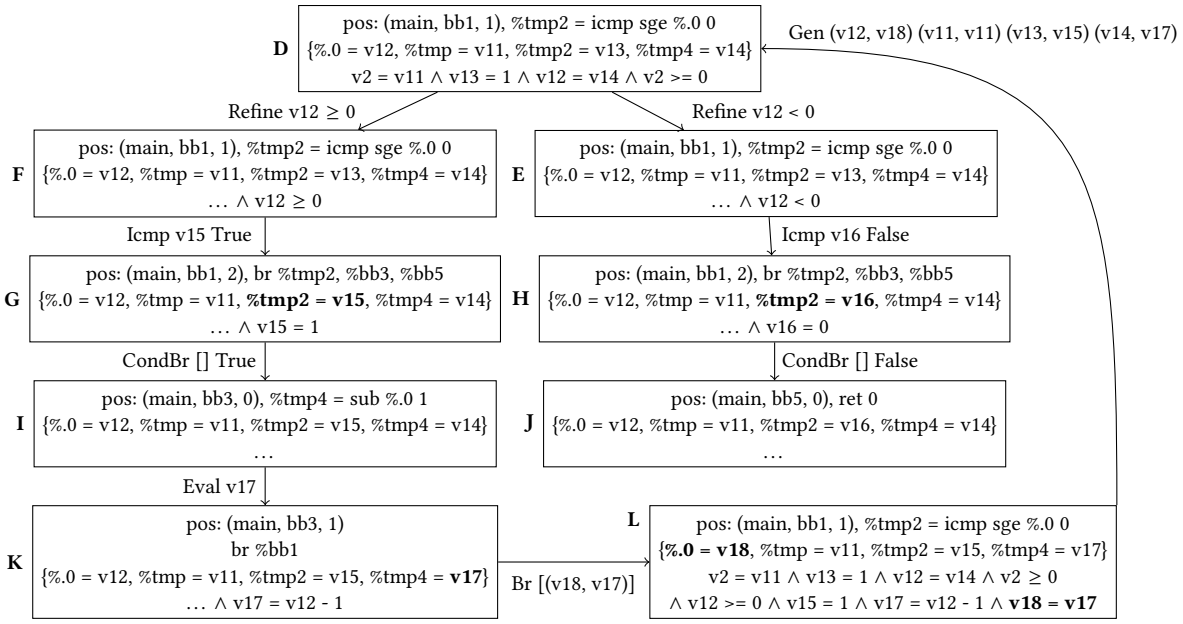

Figure 8. Lower part of SEG representing program in Figure 2,
relevant changes to the previous state are bold, … represent the knowledge base from the previous state

```
datatype ('pos,'pv,'lv) abstract_state =
  As (pos : "'pos")
     (stack : "'pv ⇒ 'lv option")
     (kb : "'lv IA.formula")

datatype ('pos,'n,'pv,'lv) graph' = Seg
  (edges :: "('n × 'n) set")
  (as_of_node :: "'n ⇒ ('pos,'pv,'lv) abstract_state")
  (renaming_of_edge :: "('n × 'n) ⇒ ('lv ⇒ 'lv)")

inductive represents_state where
"frame.pos f = abstract_state.pos as
∧ same_stack_keys (frame.stack f) (abstract_state.stack as)
∧ (∀n lv i. frame.stack f n = Some i
   ⟶ abstract_state.stack as n = Some lv ⟶ v lv = i)
∧ v ⊨IA kb as
⟹ represents_state (f : fs) as"
```

Figure 9. SEG and abstract state definition

itself, its semantics, how an SEG represents an LLVM IR program and the correctness of the inference rules for checking that an SEG represents an LLVM IR program. We then define executable code to check if an SEG exported from AProVE indeed was constructed by these inference rules. This gives us very strong guarantees that a termination analysis is correct.

4.1 Formal Definition of an SEG

The definition of an SEG in Isabelle is given in Figure 9. The formal definition of an abstract state is a direct translation of the previous textual description: we just store the position (of type `'p`), the mapping from program variables to logical variables and the knowledge base. For the latter we use IsaFoR's type `'lv IA.formula` for integer arithmetic formulas over logical variables of type `'lv`.

Next, an SEG (**`datatype`** `graph'`) consists of a set of edges, an assignment from nodes to abstract states and a function from edges to renamings.

The inductive definition `represents_state` encodes the relation between an `abstract_state` and an `ll_state`. An abstract state represents an `ll_state` if it represents the first frame in the state. This means that abstract state and frame share the same position and program variables, and there exists an assignment `v` of logical variables to integers which satisfies the knowledge base of the abstract state and is in sync with the assignment in the frame.

For example, let `s` be a concrete state where the first frame has position `(main, bb1, 1)` and where the frame assigns the integer 1 to both program variables `%.0` and `%tmp`. Then `s` is represented by the abstract state in node **C** in Figure 7.

4.2 Evaluation in SEGs

Recall that the key property of SEGs is to simulate infinite runs of LLVM IR programs via infinite paths in the graphs. Unfortunately, at this point the paper proofs in [16] do not specify how execution on SEGs works. As such, in Figure 10 we now provide a formal semantics for evaluation in SEGs.

The transitions of an SEG form a binary relation on states and are formally defined as inductive set `as_step`. Here, a state in the SEG is a pair of a node and an assignment which maps

```
inductive as_as_step where
  ∀n lv. stack as₁ pv = Some lv ⟶ σ pv = v lv
∧ ∀n lv. stack as₂ pv = Some lv ⟶ τ pv = v' lv
∧ v ⊨ kb as₁
∧ v' ⊨ kb as₂
∧ ∀lv. v (μ lv) = v' lv
⟹ as_as_step as₁ as₂ σ τ μ

inductive_set as_step where
  (n₁, n₂) ∈ edges seg ∧ renaming_of_edge (n₁, n₂) = μ
∧ as_as_step (as_of_node seg n₁) (as_of_node seg n₂) σ τ μ
⟹ ((n₁,σ), (n₂,τ)) ∈ as_step seg
```

Figure 10. Evaluation in SEGs

```
inductive evalInf where
"find_statement prog (pos as₁) = Inr (Instruction (Named x
(Binop binop o₁ o₂)))
∧ pos as₂ = inc_pos (pos as₁)
∧ operand_value as₁ o₁ = Some t₁
∧ operand_value as₁ o₂ = Some t₂
∧ φ = encode_binop binop vₙ t₁ t₂
∧ update_as as₁ as₂ x vₙ φ
∧ as_of_node seg n₁ = as₁
∧ as_of_node seg n₂ = as₂
∧ renaming_of_edge seg (n₁,n₂) = id
∧ (n₁,n₂) ∈ edges seg
⟹ evalInf prog seg n₁ n₂"
```

Figure 11. Formal version of inference rule `evalInf`

program variables to integers. In order for $((n_1,\sigma), (n_2,\tau))$ to become a transition in `as_step` the following conditions must be satisfied: There must be two assignments `v` and `v'` from logical variables to integers; if the stack in the source abstract state maps a logical variable `lv` to a program variable `pv`, then σ `pv = v lv` holds, i.e. the same integer value is assigned; the same holds for `v'`, τ and the target abstract state; furthermore `v` and `v'` have to satisfy the respective knowledge bases in the abstract states; finally, the assignment `v` is obtained from `v'` via the renaming μ that is stored for the edge (n_1, n_2) in the SEG.

4.3 SEGs Representing LLVM Programs

An SEG represents an LLVM program if for every step in the LLVM semantics there is a corresponding non-empty path in the SEG semantics. The inference rules in [16] are designed in a way that they only permit to construct SEGs, so that these always represent the input LLVM program. In this part we are going to formally verify this property. This task is non-trivial because of three problems: first, the abstract states in this paper deviate from the definition in [16]; second, there is no clear definition of the semantics of SEGs in [16]; and third, even simple inference rules get complex when spelling out all the details.

Except for the aforementioned problems, most inference rules can directly be verified formally. In total, we verify the rules `evalInf`, `genInf`, `refineInf`, `condBrInf`, `brInf`, `evalExternalInf`, `callInf`, `returnInf`, `icmpInf` and `refineInf`. Since we are only interested in verifying an already constructed SEG, we just refer to [16] regarding the problem of how to apply the inference rules for obtaining a suitable SEG. Here, we focus on showing soundness of the rules itself. While the formalization covers all rules in full detail, in this paper we concentrate on a few rules.

The first rule is `evalInf`. Its purpose is to perform a symbolic evaluation to simulate an assignment with a binary operation, i.e., `x = o₁ binop o₂`. The formal definition is given in Figure 11 and we will explain this definition in full detail for the purpose of giving an idea of the complexity of defining a simple inference rule with all formal details. In this inference rule, we assume that node n_1 is provided and that it contains an abstract state as_1 which itself points to an assignment with a binary operation. The two operands o_1 and o_2 (program variables or constants) are converted by `operand_value` into terms t_1 and t_2, respectively, by changing the program variables into logical variables. The computation of the binary operation is encoded via the Isabelle function `encode_binop`. It constructs a formula ϕ, an equality between a fresh logical variable v_n and the result of the binary operation. This formula is used to construct a new abstract state as_2; this state has to point to the next program position, the stack of as_2 must be exactly like as_1, except that now the program variable `x` points to the fresh logical variable v_n, and the knowledge base of as_2 must be an implication of the knowledge base of as_1 in combination with ϕ. Here, the latter two conditions are enforced via the Isabelle predicate `update_as`. Finally, a new node n_2 with abstract state as_2 can be added to the SEG and the resulting edge between n_1 and n_2 must not use any renaming of variables, which is expressed via the identity renaming `id`.

After its definition we then show soundness of the `evalInf` rule. To be more precise, we show that whenever a valid step in the LLVM semantics was made from concrete state cs_1 to cs_2, and cs_1 is represented by some node n_1, then the successor node n_2 of an `evalInf`-inference represents cs_2 and a transition in the SEG semantics is possible. The precise statement is given in Figure 12.

We omit the full definitions of the remaining inference rules and just briefly mention some of the remaining ones.

The rule `evalExternalInf` is quite similar to `evalInf`, except that it is used for assignments via external function calls. The main difference is that no formula ϕ is computed, so there will be no information about the result of the external call in the knowledge base of the new node. An example is given by the edge between node **A** and node **B** in Figure 7.

```
lemma evalInf_represents_step:
assumes "evalInf prog seg n₁ n₂"
  and "represents cs₁ (as_of_node seg n₁)"
  and "step prog cs₁ = Inr cs₂"
shows "represents cs₂ (as_of_node seg n₂)"
 and "((n₁,assig_of_state cs₁),(n₂,assig_of_state cs₂))
      ∈ as_step seg"
```

Figure 12. Soundness of `evalInf`

```
lemma SN_as_step_imp_SN_step'_relation:
assumes "seg_represents prog seg"
  and "represents c (as_of_node seg n)"
  and "SN_on (as_step seg) {(n, assig_of_state c)}"
shows "SN_on (step'_relation prog) {c}"
```

Figure 13. Soundness of the inference rules

The rules `condBrInf` and `brInf` encode jumping into another basic block. The atomic evaluation of the phi nodes is integrated into these rules.

The rule `genInf` describes how to reuse nodes by showing that one node generalizes another one. This rule is used to obtain finite SEGs, since without it an SEG would most often result in an infinite tree. It is the only rule where we actually use the renaming on the edge of an SEG. The renaming is used to reunite the logical variables used for the same program variables. Renaming is necessary because different logical variables may be used for the same program variables. As an example, consider nodes **L** and **D** in Figure 8 that are connected by the `genInf` rule. In node **L**, program variable `%.0` is assigned to v18 and in node **D** it is assigned to v12. Therefore the renaming needs to contain the assignment v12 $\mapsto$ v18. The conditions of the `genInf` rule further ensure that the knowledge base of the abstract state in the first node implies the knowledge base of the abstract state in the second node modulo the renaming, so that the second node does in fact correspond to more concrete states than the first node does. Note that `genInf` is the rule which we could not verify using the definition of SEGs as in [16].

The last rules we mention are the rules for function calls and returning from a function call. Here, the deviation from the definition of abstract states becomes problematic. To illustrate the problem, consider a concrete state with frames $f_1 : f_2 : f_3 : \ldots$ where f_1 is represented by some abstract state `as`. If now a return statement is executed, then the resulting state is roughly $f_2 : f_3 : \ldots$. But since no information about f_2 is stored in `as`, it will not be possible to simulate the execution of this return statement in the SEG.

The solution to this problem is to allow shortcuts and pruning in the simulation of an infinite LLVM program execution as follows: Whenever a function call is made in this infinite computation, two situations can occur. First, the function call will return after several execution steps; in this case we can treat this function call as if it would be the invocation of some external (terminating) function and we make a shortcut from the call to the state when the call has returned. Second, the function call does not return; in that case we don't have to simulate the return statement and just jump into the first block of the called function and thereby prune all frames below. In total, whenever there is an internal function call, two new nodes must be created in the SEG, one which jumps into the called function and one which jumps to the position after the call. The benefit of adding shortcuts is that there is no longer the demand to simulate any return statement. Consequently, the rule `returnInf` for return statements is now quite simple: the respective node then has no outgoing edge and the SEG ends in these nodes.

A nice side effect of our treatment of function calls is the fact that in this way we also support recursive LLVM IR programs. These programs were explicitly excluded in [16]. We note that the support for the rule `callInf` is not yet implemented in AProVE and the tested programs in Section 5.3 do not contain function calls.

After the discussion of some concrete inference rules, we now define a predicate `seg_represents` to express that for every node in the SEG a suitable inference rule has been applied, and that all successor nodes of the inference rules are also part of the SEG. With the accompanying lemmas from all the rules above we then show that if there is an infinite execution w.r.t. the `step'` semantics then there exists an infinite execution in the SEG semantics. Therefore if the SEG is strongly normalizing, then the `step'` relation is strongly normalizing, cf. Figure 13. We start by assuming that there is an infinite chain in the `step'` semantics. Since we have a SEG that represents an LLVM programm and we therefore know that for every transition in the `step'` semantics, there is at least one transition in the `as_step` semantics. This implies an infinite chain in `as_step seg` which is a contradiction to the assumption that `as_step seg` is strongly normalizing.

4.4 Certificate for the Fact That an SEG Represents an LLVM Program

We have previously described several inference rules that permit to construct SEGs. Now we show how to verify that a given SEG was really constructed w.r.t. the inference rules. To this end, we require that the certificate contains the graph structure of the SEG and additionally for every node we need to know which inference rule was applied, in combination with further information on how it was applied. Such an example certificate was already given in a graphical form in Figures 7 and 8. For example, at node **I** we see that `evalInf` was applied and the new logical variable was named v17. Another example is node **L**, which applies `genInf`, and here the renaming of variables is provided.

```
lemma check_eval_evalInf:
assumes "isOK (check_eval prog seg n₁ n₂ vₙ)"
shows "evalInf prog seg n₁ n₂"
```

Figure 14. Soundness of the checker for `evalInf`

```
lemma check_seg_represents:
assumes "isOK (check_seg_represents prog seg)"
shows "seg_represents prog seg"
```

Figure 15. Soundness of the checker of SEGs

Hence, we define executable functions in Isabelle which check for a given node with the help of the auxiliary information, whether indeed the annotated inference rule was correctly applied. The structure of the soundness statements of these checkers is always the same and an example is given in Figure 14 for checking an application of `evalInf`. The statements express that if the checker accepts, then the property is guaranteed to hold. Here, `isOK` just checks whether the result of the checker, a sum-type, is a right (`Inr`) value, and not an error message (`Inl`).

Note that several of these checkers internally rely upon the verified SMT solver of IsaFoR/CeTA [3]. The reason is that many inference rules require that a certain linear arithmetic formula implies another arithmetic formula. For example, for `evalInf` we must check validity of $\psi_1 \wedge \phi \longrightarrow \psi_2$ where ψ_1 and ψ_2 are the knowledge bases of nodes n_1 and n_2, respectively and ϕ is the formula of the inference rule.

Next, we define an executable function `check_seg_represents` which calls the corresponding check functions for every node in the graph. Figure 15 shows the overall soundness of the checker for SEGs.

4.5 Converting SEGs to ITSs

At this point, we already showed how to check if an SEG represents an LLVM IR program, and that in the positive case termination of the SEG implies termination of the program. So, it remains to prove termination of an SEG, which is done by converting it to an ITS, and then use the existing infrastructure to show termination of the ITS in a certified way [4]. The only missing step is hence the verification of the translation of SEGs to ITSs. At this point it becomes crucial that SEGs have an execution semantics on their own, so that a verification of the translation is possible without having to relate to the LLVM IR semantics or the inference rules on how to construct an SEG.

The ITS model in [4] consists of a graph with edges annotated with integer arithmetic formulas. Variables in these formulas are either so called pre-, post- or intermediate variables.

For conversion to an ITS we only need the knowledge bases in the abstract states of an SEG and the renamings on the edges. To be more precise, the conversion turns the logical variables of the source state into pre-variables, the target state variables become post variables and all other variables in the knowledge bases become intermediate variables. We define a function `lts_of_graph` in Isabelle to perform this conversion and prove its soundness in Figure 16. Note that in the formalization, ITSs are available in a more general form, namely as *labeled* transition systems (LTSs), so many of the names in this section have a prefix `lts` instead of `its`. Sometimes the prefix `IA.` is used to refer the particular instance of LTSs that are ITSs.

```
lemma lts_termination_SN_as_step:
assumes "IA.lts_termination
    (lts_of_graph seg {loc_of_node n})"
shows "SN_on (as_step seg) {(n, v)}"
```

Figure 16. Soundness of the translation of SEGs to ITSs

```
lemma lts_renaming_termination:
assumes "lts_termination L_A"
  and "renamed_lts r L_A L_I"
shows "lts_termination L_I"
```

Figure 17. Soundness of renaming elements within ITSs

There is a problem at this point, namely that our function `lts_of_graph` which converts an SEG to an ITS reuses the same logical variable names as the SEG, whereas AProVE renames variables to a standardized naming scheme when creating any ITS. Futhermore, the lemma `lts_termination_SN_as_step` depends on the same variable names in the SEG and ITS and AProVE's methods to prove termination of an ITS depend on the standardized naming scheme. We need to show that the termination of the AProVE ITS implies the termination of the Isabelle ITS and we solve this problem via a predicate on two ITSs called `renamed_lts`. It takes two ITSs and a variable renaming, and checks whether the one ITS is obtained from the other by the renaming and by testing on equivalent formulas. By checking for equivalence instead of equality of the formulas we also take care of the fact the order of clauses in the Isabelle ITS and the AProVE ITS can be different. We extended AProVE to generate a renaming when constructing the ITS from the SEG. It then exports the ITS L_A and the renaming and we then check if the ITS L_I constructed by `lts_of_graph seg` is a `renamed_lts`.

Figure 17 shows that if L_A terminates, then L_I terminates. Consequently, it suffices to prove termination of the ITS L_A that has been constructed by AProVE to ensure termination of `seg`.

```
lemma llvm_check_termination:
assumes "isOK (llvm_check_termination prog fn ll_proof)"
  and "initial_llvm_frame prog fn fr" and "frames s = [fr]"
shows "SN_on (step'_relation prog) {s}"
```

Figure 18. Soundness of Certifier

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="cpfHTML.xsl"?>
<certificationProblem
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="cpf.xsd">
  <input><llvm>
      <function><name>main</name></function>
      <llvmprog>
define i32 @main() {
...
      </llvmprog>
  </llvm></input>
  <cpfVersion>2.1</cpfVersion>
  <proof><llvmTerminationProof>
      <seg>...</seg>
      <lts>...</lts>
      <renamings>...</renamings>
      <ltsTerminationProof>...</ltsTerminationProof>
  </llvmTerminationProof></proof>
</certificationProblem>
```

Figure 19. CPF example

5 The Verified Certifier and Its Evaluation

5.1 Assembling the Certifier for LLVM IR Termination Proofs

A complete proof of termination for an LLVM IR function consists of an SEG, an ITS, a renaming of the ITS and a termination proof for the ITS, and these components are combined in a suitable datatype to represent LLVM termination proofs. Such a proof can be passed to our verified certifier, which is defined as the executable Isabelle function `llvm_check_termination`. In addition to the proof, the checker takes the input LLVM program `prog` and a function name `fn`, and checks that all components have been constructed correctly. Internally, it invokes the various certifiers for checking the correct application of each of the transformations, as well as the already existing checker for termination proofs of ITSs.

Soundness of the certifier is stated in Figure 18: if the certifier `llvm_check_termination` is successful, then the function `fn` in `prog` terminates for all inputs. More precisely, termination is ensured for all states `s` that correspond to exactly one function call of `fn` with arbitrary inputs. Here, the predicate `initial_llvm_frame prog fn fr` encodes that a frame `fr` is a proper initial frame of the function, i.e. the parameters are initialized and the position is at the start of the function. Moreover, the state `s` must contain exactly one frame `fr` – otherwise a lower frame could, for example, contain a call to a non-terminating function, so that it would be wrong to conclude termination of the `step'_relation` on such an input.

Our new certifier for termination of LLVM IR code has been integrated into the IsaFoR library. In this library, there are already several certifiers for other properties, and they all use the common CPF format for encoding proofs. Hence, we extend CPF to also cover LLVM termination proofs. As seen in Figure 19, a CPF proof is an XML file consisting of an input problem and a corresponding proof. The `<input>`-element contains the original LLVM IR code and the `<proof>`-element contains all necessary proof parts. Parsing XML is already part of the IsaFoR library and except for the LLVM IR input everything is parsed by exported code from Isabelle. For parsing the LLVM IR input we rely on the `llvm-hs` Haskell library.[4] This is, right now, the only part of our code which is not generated by Isabelle's code generator.

5.2 AProVE

AProVE is already able to analyze termination of LLVM programs by conversion to SEGs and then to ITSs. It also features the ability to create proofs in the CPF format, for example for term rewrite systems or ITSs. For this work, we extend AProVE to also produce certificates for LLVM IR termination proofs (like the one in Figure 19). This involves some adjustments in the structure of the proof objects that are internally stored by AProVE. The reason is that some of the information that is required in the certificates were originally discarded by AProVE. After inserting the additional information we add a CPF export for LLVM termination proofs. It is based on a newly written CPF output for the SEG, for the conversion to an ITS and the renaming, as well as on the existing CPF output for ITS termination proofs.

5.3 Experiments

We run AProVE and our exported Haskell code on examples from the C_Integer category in the Termination competition database.[5] The programs in this category are written in C and do not contain memory access commands. For the actual termination, analyzing tools must follow the C standard with the only difference that the `int` type is unbounded.

Our first step is to compile the programs of this category from C to LLVM. If we simply compile these C programs with `clang` and minimal optimizations (command line parameter `-O1`), all programs are simplified to a main function that consists of single command: `ret 0`. This is due to the fact that the C standard allows implementations to assume that while loops without side effects (like memory access or IO) terminate [10, Section 6.8.5].

For this reason we first run `clang` without any optimizations. We then need to run the optimizer with one optimization pass: `mem2reg`. The reason for this is if `clang` is run without any optimizations on C code, passing parameters to functions is, for example, implemented with passing pointers

[4] https://github.com/llvm-hs/llvm-hs
[5] http://termination-portal.org/wiki/TPDB

Configuration	AProVE timeout	AProVE fail	AProVE success	certifier timeout	certifier fail	certifier success
certified version	110	112	102	0	0	102
only ITS	0	23	311	17	24	270

Figure 20. Running AProVE and certifier on C_Integer programs

in memory. `mem2reg` then rewrites these memory accesses to register accesses, which results in programs that fall into our syntactic fragment of LLVM IR.

For our experiments we consider two configurations of AProVE. The *full version* is the version that has been used in the latest termination competition. The *certified version* of AProVE differs from the full version by restricting the search for LLVM termination proofs to the shape that we described in this paper – the full version also tries alternative approaches to conclude termination, e.g., by using other back ends, which are not supported by our certifier. We run both configurations of AProVE and the certifier with a 5 minute timeout respectively. As hardware we use a standard iMac with a 4.2 GHz Quad-Core Intel Core i7 processor and 32GB of RAM.

The full version is able to generate termination proofs for 216 out of the 334 programs in the C_Integer category, and the certified version finds 102 proofs. Although the certified version is clearly less powerful – it solves roughly 50 % of the examples in comparison to the full version – it has the advantage that all 102 proofs have been certified by our certifier.

More detailed experimental data on the certified version is given in Figure 20. There we additionally added a configuration *only ITS*. This configuration invokes AProVE in a way that it just has to convert a given LLVM IR program into an ITS without proving termination of the latter. Moreover, the certifier then just needs to check that the construction of the ITS has been sound.

Since AProVE never times out when just constructing an ITS, it is obvious that all time outs of the certified version appear during the attempt to find a termination proof for the ITS. The 23 failures during the construction of ITSs come from programs which use non-linear arithmetic expressions, for which there is no CPF export available. Note that AProVE is able to generate 311 ITSs, and in 270 cases our certifier can validate that their termination implies termination of the LLVM IR programs. However, for certain large input the certifier times out. The time outs always come from the validity checker for linear integer arithmetic. This means that all the used inferences may be valid but we cannot check them in time. A more efficient validity checker for linear integer arithmetic in Isabelle would be needed.[6] For 24 certificates our certifier finds errors within the certificate. It either finds a linear arithmetic formula that is not valid or it finds an error where the assignment of phi nodes when branching is not done correctly. We are not yet sure where these error arise exactly but we suspect that at one point AProVE does not properly keep track of the logical variables it newly creates and assigns. Finally note that the certified version of AProVE fails 112 times. In these cases AProVE can show termination, but fails during generation of the certification. The reason is that the internal ITS termination prover sometimes applies methods for which there is no CPF export available.

The full experimental data is available at our website for supplementary material.[7] This includes the C source files, the corresponding LLVM IR modules, the generated CPF files and the generated Haskell code.

6 Conclusion and Future Work

We have developed a verified certifier for checking termination proofs of LLVM IR code. Using Isabelle we have formalized results regarding the semantics of the LLVM IR, formally specified the SEG, linked the termination of LLVM IR code with the termination of the corresponding SEG and linked the termination of the SEG with the termination of the corresponding ITS. We developed and verified algorithms to check the correspondence between LLVM IR code, SEGs and ITSs. We used Isabelle's code export feature to generate an executable certifier in Haskell. We then extended AProVE to generate certificates that can be checked by the Haskell certifier. We successfully tested AProVE and our certifier on a test suite of programs used in a termination competition.

We plan to extend the LLVM IR semantics in Isabelle with a proper memory model and to extend our certifier to also handle proofs of memory safety. Ströder et al. already described how an SEG can be used to show memory safety [16] and Zhao and Lammich both define a memory model for LLVM IR in a theorem prover [11, 18]. To properly handle LLVM IR code, we also need to extend our semantics with bit vector arithmetic instead of unbounded integers. Hensel et al. showed how to use the SEG to handle termination of actual LLVM IR code with bit vector semantics [9]. There, the semantics of the SEG still use unbounded integers. Possible overflows are handled by branching in the SEG and adding constraints in the knowledge bases of the corresponding abstract states.

[6] Or, like in this paper, we use an external tool to generate a proof of the validity of an integer arithmetic formula and then only check that with a certifier.

[7] http://cl-informatik.uibk.ac.at/isafor/experiments/llvm/cpp2021/

Independently from work on LLVM IR semantics and the SEG, improvements to the ITS back end would be worthwhile. The ITS back end uses an incremental Simplex algorithm to check validity of linear integer arithmetic formulas [3]. Possible extensions would be to handle non-linear arithmetic, bit vector arithmetic or optimizations of the existing algorithm to speed up the certifier. Furthermore, not all techniques used by AProVE to show termination of an ITS are formalized in IsaFoR and can be certified. Properly implementing these termination techniques in AProVE and the IsaFoR library would automatically lead to more certified termination proofs for LLVM IR code, as the experiments in Section 5.3 have shown.

Acknowledgments

This research was supported by the Austrian Science Fund (FWF) project Y757. We thank the development team of AProVE at RWTH Aachen especially Jera Hensel for helping us integrate our code within AProVE. We thank Ralph Bottesch for proofreading the drafts of this paper. We also thank the reviewers for their helpful suggestions.

References

[1] 2019. LLVM Language Reference Manual (Version 9.0.0). https://releases.llvm.org/9.0.0/docs/LangRef.html (Archive link: https://web.archive.org/web/20191228071421/https://releases.llvm.org/9.0.0/docs/LangRef.html). Accessed: 2020-09-01.

[2] Dirk Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 347–367. https://doi.org/10.1007/978-3-030-45237-7_21

[3] Ralph Bottesch, Max W. Haslbeck, Alban Reynaud, and René Thiemann. 2020. Verifying a Solver for Linear Mixed Integer Arithmetic in Isabelle/HOL. In *NASA Formal Methods*, Ritchie Lee, Susmit Jha, and Anastasia Mavridou (Eds.). Springer International Publishing, Cham, 233–250. https://doi.org/10.1007/978-3-030-55754-6_14

[4] Marc Brockschmidt, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. 2017. Certifying Safety and Termination Proofs for Integer Transition Systems. In *Automated Deduction – CADE 26*, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 454–471. https://doi.org/10.1007/978-3-319-63046-5_28

[5] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. 2007. Certification of Automated Termination Proofs. In *Frontiers of Combining Systems*, Boris Konev and Frank Wolter (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 148–162. https://doi.org/10.1007/978-3-540-74621-8_10

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[7] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2016. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning* 58, 1 (Oct. 2016), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[8] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. 2019. The Termination and Complexity Competition. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 156–166. https://doi.org/10.1007/978-3-030-17502-3_10

[9] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. 2018. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming* 97 (2018), 105 – 130. https://doi.org/10.1016/j.jlamp.2018.02.004

[10] ISO/IEC 9899:2018 2018. *Information technology — Programming languages — C*. Standard. International Organization for Standardization, Geneva, CH.

[11] Peter Lammich. 2020. Efficient Verified Implementation of Introsort and Pdqsort. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 307–323. https://doi.org/10.1007/978-3-030-51054-1_18

[12] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[13] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. Vol. 2283. https://doi.org/10.1007/3-540-45949-9

[14] Christian Sternagel and René Thiemann. 2014. Certification Monads. *Archive of Formal Proofs* (Oct. 2014). http://isa-afp.org/entries/Certification_Monads.html, Formal proof development.

[15] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. 2014. Proving Termination and Memory Safety for Programs with Pointer Arithmetic. In *Automated Reasoning*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.). Springer International Publishing, Cham, 208–223. https://doi.org/10.1007/978-3-319-08587-6_15

[16] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. 2017. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning* 58, 1 (2017), 33–65. https://doi.org/10.1007/s10817-016-9389-x

[17] René Thiemann and Christian Sternagel. 2009. Certification of Termination Proofs Using CeTA. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 452–468. https://doi.org/10.1007/978-3-642-03359-9_31

[18] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.* 47, 1 (Jan. 2012), 427–440. https://doi.org/10.1145/2103621.2103709