Termination Analysis of C Programs Using Compiler Intermediate Languages

Stephan Falke[1], Deepak Kapur[2], and Carsten Sinz[1]

1 **Institute for Theoretical Computer Science**
Karlsruhe Institute of Technology (KIT), Germany
`{stephan.falke, carsten.sinz}@kit.edu`

2 **Department of Computer Science**
University of New Mexico, Albuquerque, NM, USA
`kapur@cs.unm.edu`

Abstract

Modeling the semantics of programming languages like C for the automated termination analysis of programs is a challenge if complete coverage of all language features should be achieved. On the other hand, low-level intermediate languages that occur during the compilation of C programs to machine code have a much simpler semantics since most of the intricacies of C are taken care of by the compiler frontend. It is thus a promising approach to use these intermediate languages for the automated termination analysis of C programs. In this paper we present the tool `KITTeL` based on this approach. For this, programs in the compiler intermediate language are translated into term rewrite systems (TRSs), and the termination proof itself is then performed on the automatically generated TRS. An evaluation on a large collection of C programs shows the effectiveness and practicality of `KITTeL` on "typical" examples.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.41

Category System Description

1 Introduction

Methods for automatically proving termination of imperative programs operating on integers have received increased attention recently. The most commonly used automatic method for this is based on linear ranking functions which linearly combine the values of the program variables in a given state [5, 6, 19, 20]. More recently, the combination of abstraction refinement and linear ranking functions has been considered [8, 9]. Based on this idea, the tool `Terminator` [10] has reportedly been used for showing termination of device drivers.

Developing a tool that can handle all intricacies of C is a challenge since C employs a complex syntax and semantics. It is not clear to what extent the implementations of the aforementioned methods can handle real-life C programs since the papers are typically based on idealized transition systems and the implementations are not publicly available.

We advocate to perform the termination analysis of C programs not on the source code level but rather on the level of a compiler intermediate representation (IR). This approach has the following advantages:

1. The IR is considerably simpler than C. This makes it relatively easy to accept *any* C program as an input. Features of the IR that are not (yet) supported by the termination analysis techniques can easily be abstracted automatically.

This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

2. The program that is analyzed is much closer to the program that is actually executed on the computer since ambiguities of C's semantics have already been resolved.
3. In producing the IR, compilers already use program optimizations that might simplify the termination analysis significantly.

For similar reasons, termination analysis of Java programs is often performed on the bytecode level and not on the source code [1, 22, 18, 4].

In this paper, we focus on the LLVM compiler framework and its intermediate language LLVM-IR [17]. The method itself is independent of the concrete IR, however. Since there are compilers for various programming languages built atop of LLVM, the methods presented in this paper can be used for the termination analysis of programs written in C, C++, Objective-C, and further programming languages.

Termination analysis of LLVM-IR programs is then performed by generating a term rewrite system (TRS) from the LLVM-IR program. Termination analysis of TRSs has been investigated extensively in the past (see [24] for a survey). In this paper, TRSs with constraints over the integers (`int`-*based TRSs*) are used, where the constraints are relations on the variables expressed as quantifier-free formulas from non-linear arithmetic. Similarly to what was proposed in [12, 15], well-known methods from the term rewriting literature can be adapted for the termination analysis of `int`-based TRSs.

▶ **Example 1.** Consider the C program on the left-hand side:

```
int power(int x, int y) {
    int r = 1;
    while (y > 0) {
        r = r * x;
        y = y − 1;
    }
    return r;
}
```

$$\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}) \to \mathsf{state}_{\mathsf{entry_{in}}}(v_x, v_y, v_{y.0}, v_{r.0})$$
$$\mathsf{state}_{\mathsf{entry_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \to \mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_y, 1)$$
$$\mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \to \mathsf{state}_{\mathsf{bb_{in}}}(v_x, v_y, v_{y.0}, v_{r.0})\ [\![v_{y.0} > 0]\!]$$
$$\mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \to \mathsf{state}_{\mathsf{return_{in}}}(v_x, v_y, v_{y.0}, v_{r.0})\ [\![v_{y.0} \leq 0]\!]$$
$$\mathsf{state}_{\mathsf{bb_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \to \mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_{y.0} - 1, v_{r.0} * v_x)$$
$$\mathsf{state}_{\mathsf{return_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0})$$

Using the methods presented in this paper, the `int`-based TRS shown on the right-hand side is automatically obtained from the LLVM-IR of the C program. Intuitively, the variables v_x and v_y represent the inputs to the function, whereas the variables $v_{y.0}$ and $v_{r.0}$ correspond to the (changing) program variables y and r used inside the loop of the function (why the program variable y gives rise to v_y and $v_{y.0}$ is explained in Section 3). The function symbols used in the `int`-based TRS intuitively correspond to a program counter. ◀

The approach has been implemented in the publicly available termination tool `KITTeL`. An empirical evaluation on a large collection of examples taken from various sources clearly shows the effectiveness and practicality of our method.

2 `int`-Based TRSs

In order to model integers, the function symbols from $\mathcal{F}_{\texttt{int}} = \mathcal{F}_{\mathbb{Z}} \cup \{+, *, -\}$ with $\mathcal{F}_{\mathbb{Z}} = \{\mathsf{n} \mid n \in \mathbb{Z}\}$ and types $+, * : \texttt{int} \times \texttt{int} \to \texttt{int}$, and $- : \texttt{int} \to \texttt{int}$ are used. Terms built from these function symbols and a disjoint set $\mathcal{V}$ of variables are called `int`-*terms*. We use a simplified notation for `int`-terms, e.g., the `int`-term $(x + (-(y * y))) + 3$ is written as $x - y^2 + 3$. A *linear* `int`-term is an `int`-term that does not contain any occurrence of "$*$".

$\mathcal{F}_{\texttt{int}}$ is extended by finitely many function symbols f with types $\texttt{int} \times \ldots \times \texttt{int} \to \texttt{univ}$, where `univ` is a type distinct from `int`. The set containing these additional function symbols is denoted by $\mathcal{F}$ and $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$ denotes the set of terms of the form $f(s_1, \ldots, s_n)$ where $f \in \mathcal{F}$ and $s_1, \ldots, s_n$ are `int`-terms. A *substitution* is a mapping from variables to `int`-terms.

`int`-constraints are quantifier-free formulas from (non-linear) integer arithmetic. An *atomic* `int`*-constraint* has the form $s \simeq t$, $s \geq t$, or $s > t$ for `int`-terms s, t and the set of `int`*-constraints* is the closure of atomic `int`-constraints under $\top$ (truth), $\neg$ (negation), and $\wedge$ (conjunction). The Boolean connectives $\bot$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ are defined as usual. `int`-constraints have the expected semantics regarding `int`*-validity* and `int`*-satisfiability*. These properties are in general only decidable for linear or variable-free `int`-constraints.

The rewrite rules of `int`-based TRSs are equipped with `int`-constraints. These constraints are used in order to restrict the applicability of the rewrite rules, see Definition 3. The rules generalize the $\mathcal{PA}$-based rewrite rules from [12]. Alternatively, they can be interpreted as a restricted form of the rewrite rules considered in [15] which allow nested function symbols.

▶ **Definition 2.** An `int`*-based rewrite rule* has the form $l \to r[\![\varphi]\!]$ such that $l = f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are pairwise distinct variables, $r \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$, and φ is an `int`-constraint.

Notice that r and φ may contain variables that are not occurring in l. The restriction that the arguments on the left-hand side are pairwise distinct variables simplifies the definition of the rewrite relation of an `int`-based TRS since matching becomes trivial. Notice that equality between the arguments x_i and x_j can be enforced by adding the `int`-constraint $x_i \simeq x_j$. The constraint $\top$ is omitted in an `int`-based rewrite rule $l \to r[\![\top]\!]$. An `int`*-based term rewrite system (*`int`*-based TRS)* $\mathcal{R}$ is a finite set of `int`-based rewrite rules.

▶ **Definition 3.** For an `int`-based TRS $\mathcal{R}$, the relation $s \to_{\texttt{int}\backslash\mathcal{R}} t$ for terms s, t of the form $f(\mathsf{n}_1, \ldots, \mathsf{n}_k)$ with $\mathsf{n}_1, \ldots, \mathsf{n}_k \in \mathcal{F}_{\mathbb{Z}}$ holds iff there exist $l \to r[\![\varphi]\!] \in \mathcal{R}$ and an $\mathcal{F}_{\mathbb{Z}}$-based substitution σ such that **1.** $s = l\sigma$, **2.** $\varphi\sigma$ is `int`-valid, and **3.** $t = \mathsf{norm}(r\sigma)$. Here, a substitution σ is $\mathcal{F}_{\mathbb{Z}}$*-based* iff $\sigma(x) \in \mathcal{F}_{\mathbb{Z}}$ for all variables x and $\mathsf{norm}(r\sigma)$ evaluates according to the usual semantics of "$+$", "$*$", and "$-$" on variable-free terms.

Termination of `int`-based TRSs can be shown by using an extension of the methods presented in [12] which are motivated by the dependency pair method [2, 16, 11] and are based on the notion of *chains*. For an `int`-based TRS $\mathcal{R}$, a (possibly infinite) sequence of `int`-based rewrite rules $l_1 \to r_1[\![\varphi_1]\!], l_2 \to r_2[\![\varphi_2]\!], \ldots$ from $\mathcal{R}$ is an $\mathcal{R}$*-chain* iff there exists an $\mathcal{F}_{\mathbb{Z}}$-based substitution σ such that $\mathsf{norm}(r_i\sigma) = l_{i+1}\sigma$ and $\varphi_i\sigma$ is `int`-valid for all $i \geq 1$.

Chains provide a precise characterization of termination in the sense that an `int`-based TRS $\mathcal{R}$ is terminating if and only if there are no infinite $\mathcal{R}$-chains. This characterization of termination is utilized by introducing sound *processors* which are used to transform an `int`-based TRS into a set of `int`-based TRSs such the input TRS is terminating if all output TRSs are terminating. The following are the two most important processors for `int`-based TRSs (details on these processors and their implementation can be found in [13]):

- SCC decomposition: Here, it is approximated which rules may follow each other in chains. Then, $\mathcal{R}$ is decomposed into the non-trivial SCCs of the thus obtained graph.
- Polynomial interpretations: A polynomial interpretation maps each n-ary $f \in \mathcal{F}$ to a polynomial $\mathcal{P}ol(f) \in \mathbb{Z}[x_1, \ldots, x_n]$. This mapping extends to terms from $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$ by letting $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = \mathcal{P}ol(f)(t_1, \ldots, t_n)$. Then, terms are compared (in the context of a constraint) by comparing polynomials, and all strictly decreasing rules may be deleted.

3 Translating LLVM-IR Programs into `int`-Based TRSs

Converting programs from a real-life programming language such as C into `int`-based TRSs is non-trivial. C has a complex syntax and semantics, resulting in many cases that need to be considered. An alternative to operating on the source code level is the use of compiler intermediate languages. These intermediate languages typically have a simple syntax and semantics, thus simplifying the translation into `int`-based TRSs significantly.

In this paper, we consider LLVM and its intermediate language LLVM-IR [17]. An LLVM-IR program is an assembly program for a register machine with an unbounded number of registers. A program consists of type definitions, global variable declarations, and the program itself, given in the form of one or more functions. Each function is represented as a graph of basic blocks (see Example 4 for an LLVM-IR program), where each basic block is a list of instructions, and execution of a function starts at the basic block named `entry`. For our purpose, LLVM-IR instructions can be categorized into six classes:

- *Three-address code (TAC)* instructions such as `%2 = mul i32 %r.0, %x`.
- *Control flow* instructions: *Branch* (`br`), *return* (`ret`), *phi* (`phi`).
- *Function calls* using `call` instructions.
- *Memory access* instructions, namely `load` and `store`.
- *Address calculations* using `getelementptr` instructions.
- *Auxiliary instructions* like type casts or bit-level instructions.

Branches and return instructions are only allowed as the last instruction of a basic block and each basic block is terminated by one of these instructions.

LLVM-IR programs are in *static single assignment (SSA)* form, i.e., each register (variable) is assigned exactly once in the static LLVM-IR program. Due to this, it becomes necessary to introduce the phi-instruction `phi`, which is used to select one of several values whenever the control flow in a program converges again (e.g., after an `if-then-else` statement). For example, the meaning of `%r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]` contained in the basic block `bb1` in Example 4 is that the register `%r.0` is assigned the value `1` if the control flow passed from `entry` to `bb1`. If control passed from `bb` to `bb1`, then `%r.0` is assigned the value contained in `%1`. Phi-instructions only occur at the beginning of basic blocks.

All variables in LLVM-IR are typed. Available types include a void type, integer types like `i32` (where the bit-width is given explicitly), floating-point types, and derived types (such as pointer, array and structure types). The integer type `i1` is used as a dedicated Boolean type. Aggregate types (structures and arrays) are accessed using memory load/store operations and offset calculations using the `getelementptr` instruction.

▶ Assumption 1. All LLVM-IR integer types `i`k with $k > 1$ are identified with $\mathbb{Z}$.

3.1 Single Non-Recursive Function Operating on Integers

First, it is assumed that the LLVM-IR program operates only on integer types. Furthermore, it is assumed that there is exactly one function, and that this function does not contain any `call` instruction. It thus only contains arithmetical instructions (`add`, `sub`, `mul`, signed and unsigned `div` and `rem`), comparison instructions (equality `eq`, disequality `neq`, (un)signed greater-than (`u`|`s`)`gt`, greater-or-equal (`u`|`s`)`ge`, less-than (`u`|`s`)`lt`, and less-or-equal (`u`|`s`)`le`), control flow instructions, and type cast instructions.

▶ **Example 4.** For the C program from Example 1, the LLVM-IR program shown in Figure 1 is obtained using the LLVM compiler frontend `llvm-gcc`. Here, the basic blocks `bb1` and `bb` correspond to the while-loop in the C program. ◀

An LLVM-IR program is now translated into an `int`-based TRS as follows. Each integer-typed function argument, each register defined by an integer-typed TAC instruction, and each register defined by an integer-typed phi-instruction is mapped to a variable in the TRS.

```
define i32 @power(i32 %x, i32 %y) {
entry:
  br label %bb1

bb1:
  %y.0 = phi i32 [ %y, %entry ], [ %2, %bb ]
  %r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]
  %0 = icmp sgt i32 %y.0, 0
  br i1 %0, label %bb, label %return
```

```
bb:
  %1 = mul i32 %r.0, %x
  %2 = sub i32 %y.0, 1
  br label %bb1

return:
  ret i32 %r.0
}
```

Figure 1 LLVM-IR program for the C program from Example 1.

Then, each integer-typed TAC instruction I is assigned two function symbols $\mathsf{state}_{I_{\mathsf{in}}}$ and $\mathsf{state}_{I_{\mathsf{out}}}$ and gives rise to a rewrite rule $\mathsf{state}_{I_{\mathsf{in}}}(\ldots) \to \mathsf{state}_{I_{\mathsf{out}}}(\ldots)[\![\varphi]\!]$ that mimics the effect of I. Here, division and remainder instructions are handled by introducing fresh variables on the right-hand side and adding appropriate constraints on that variable.

The control flow of the LLVM-IR program is mimicked as follows. The function symbols $\mathsf{state}_{\mathsf{start}}$ and $\mathsf{state}_{\mathsf{stop}}$ are introduced, denoting starting and stopping states, respectively. Next, each basic block bb is assigned two function symbols $\mathsf{state}_{bb_{\mathsf{in}}}$ and $\mathsf{state}_{bb_{\mathsf{out}}}$. These function symbols correspond to the points after the final phi-instruction in bb and before the branch or return instruction of bb, respectively. If bb contains the (possibly empty) sequence $\Omega = \langle I_1, \ldots, I_n \rangle$ of integer-typed TAC instructions, then, for two consecutive instructions I_k and I_{k+1}, the function symbols $\mathsf{state}_{I_{k_{\mathsf{out}}}}$ and $\mathsf{state}_{I_{k+1_{\mathsf{in}}}}$ are identified. Furthermore, rules $\mathsf{state}_{bb_{\mathsf{in}}}(\ldots) \to \mathsf{state}_{I_{1_{\mathsf{in}}}}(\ldots)$ and $\mathsf{state}_{I_{n_{\mathsf{out}}}}(\ldots) \to \mathsf{state}_{bb_{\mathsf{out}}}(\ldots)$ (if Ω is non-empty) or $\mathsf{state}_{bb_{\mathsf{in}}}(\ldots) \to \mathsf{state}_{bb_{\mathsf{out}}}(\ldots)$ (if Ω is empty) is added. If bb is terminated by a return instruction, then the rule $\mathsf{state}_{bb_{\mathsf{out}}}(\ldots) \to \mathsf{state}_{\mathsf{stop}}(\ldots)$ is added. Otherwise, bb is terminated by a branch instruction. For an unconditional branch to bb', a rule $\mathsf{state}_{bb_{\mathsf{out}}}(\ldots) \to \mathsf{state}_{bb'_{\mathsf{in}}}(\ldots)$ is added, where the variables on the right-hand side that correspond to phi-instructions are instantiated according to their value in the case where control flow passes from bb to bb'. A conditional branch is treated similarly, but now the rules are equipped with the (possibly negated) branch condition as a constraint.

▶ **Example 5.** Consider the C program from Example 1 and its LLVM-IR from Example 4. Using the translation outlined above, the `int`-based TRS

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_y, 1, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\ [\![v_{y.0} > \mathsf{0}]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\ [\![v_{y.0} \leq \mathsf{0}]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{1}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{1}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{2}(v_x, v_y, v_{y.0}, v_{r.0}, v_{r.0} * v_x, v_2)\\
\mathsf{state}_{2}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{3}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_{y.0} - 1)\\
\mathsf{state}_{3}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_2, v_1, v_1, v_2)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)
\end{aligned}$$

is obtained. Here, simplified names have been used for the function symbols. ◀

Now an LLVM-IR program is terminating if the `int`-based TRS $\mathcal{R}_P$ is terminating, but $\mathcal{R}_P$ might be non-terminating even if P is terminating (see Section 3.5 for a partial remedy).

3.2 Simplification of `int`-Based Rewrite Rules

The translation given above produces a large number of `int`-based rewrite rules since each integer-typed TAC instruction and each transition between basic blocks gives rise to one or more rules. In order to decrease the number of `int`-based rewrite rules, it is possible to combine several rules into a single one. Intuitively, this corresponds to the composition of the effect of several integer-typed TAC instructions into a single state change.

For `int`-based TRSs obtained from LLVM-IR, the set of *control points* C consists of the function symbols $\mathsf{state}_{\mathsf{start}}$, $\mathsf{state}_{\mathsf{stop}}$, and $\mathsf{state}_{bb_{\mathsf{in}}}$ for each basic block bb of the program. It is then possible to eliminate `int`-based rewrite rules that contain a function symbol not occurring in C by combining an `int`-based rewrite rule $\mathsf{state}_i(x_1, \ldots, x_n) \to \mathsf{state}_j(e_1, \ldots, e_n)[\![\varphi]\!]$, where $\mathsf{state}_i \in C$ and $\mathsf{state}_j \notin C$, with a rule $\mathsf{state}_j(x_1, \ldots, x_n) \to \mathsf{state}_k(e'_1, \ldots, e'_n)[\![\psi]\!]$, resulting in $\mathsf{state}_i(x_1, \ldots, x_n) \to \mathsf{state}_k(e'_1\omega, \ldots, e'_n\omega)[\![\varphi \wedge \psi\omega]\!]$ where $\omega = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. This *chaining* needs to be done for all possible rules that have state_j on their left-hand side. The old rules are replaced by the new rules and the process is iterated until all rules with a function symbol from C on the left-hand side also have a function symbol from C on their right-hand side.

▶ **Example 6.** Continuing Example 5, the control points are $\mathsf{state}_{\mathsf{start}}$, $\mathsf{state}_{\mathsf{stop}}$, $\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}$, $\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}$, $\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}$, and $\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}$. Combining rules w.r.t. these control points produces

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_y, 1, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\ [\![v_{y.0} > 0]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\ [\![v_{y.0} \le 0]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}-1, v_{r.0} * v_x, v_1, v_{y.0}-1)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)
\end{aligned}$$

as a new `int`-based TRS. ◀

After the combination of `int`-based rewrite rules, it is possible to remove some arguments from the function symbols. Notice that the effect of instructions that are only used in the same basic block where they are defined or in phi-instructions has been propagated by the combination of rules. Thus, the corresponding variables can be removed as arguments from the function symbols. On the syntactic level of rewrite rules, an argument position i is *unneeded* if, for all rewrite rules $l \to r[\![\varphi]\!]$, the variable occurring in position i of l does not occur in φ and only in argument position i of r.

▶ **Example 7.** After removing the unneeded arguments in Example 6,

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0})\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_y, 1)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0})\ [\![v_{y.0} > 0]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0})\ [\![v_{y.0} \le 0]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0} - 1, v_{r.0} * v_x)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0})
\end{aligned}$$

is obtained since arguments 5 and 6 are not needed. The methods outlined in Section 2 can easily prove termination of this TRS. ◀

3.3 Several Functions Operating on Integers

In this section it is discussed how the translation from LLVM-IR programs into `int`-based TRSs can be extended to the case of several functions. For this, the user first specifies which function should be the starting function for the termination analysis (often, this is the `main` function). It is then necessary to include all functions that are (transitively) called by this starting function in the termination analysis.

A given LLVM-IR program might not contain implementations of all functions being called. Instead, some functions may only be given as prototype declarations (e.g., library functions).

▶ Assumption 2. All functions that are only declared as prototypes are terminating. Furthermore, these functions do not call functions defined in the program.

If the user-defined functions have a function call hierarchy with arbitrary recursion, then it needs to be ensured that the sequence of recursive calls is terminating. For this, each call instruction to a function with non-void type gives rise to *two* rewrite rules. One rewrite rule introduces a fresh variable on the right-hand side which abstracts the return value of the called function.[1] This rule has the form $\mathsf{state}_i(\ldots) \to \mathsf{state}_j(\ldots, z, \ldots)$, where z is a fresh variable. The second rewrite rule has the form $\mathsf{state}_i(\ldots) \to \mathsf{state}^f_{\mathsf{start}}(\ldots)$, where $\mathsf{state}^f_{\mathsf{start}}$ is the called function's start symbol. A call to a function with void type is handled similarly, but no fresh variable is introduced on the right-hand side.

▶ **Example 8.** The C and LLVM-IR programs in Figure 2 compute Ackermann's function. Termination of the generated TRS can easily be shown using the methods from Section 2. ◀

3.4 Programs Containing Pointers and Floating Point Numbers

`int`-based TRSs (currently) do not support pointers or floating point numbers. Thus, all instructions of these types are ignored in the translation. In order to have a non-termination preserving translation, instructions that take a pointer or a floating point number and return an integer (such as `load` or `fptosi`) are abstracted to an unspecified value which corresponds to a fresh variable on the right-hand side of the generated rule. Similarly, comparisons of floating point numbers are abstracted to return arbitrary results.

3.5 Utilizing Static Analysis Methods

Notice that the translation from LLVM-IR programs into `int`-based TRSs does not propagate information about the initial state of the program. Thus, the `int`-based TRS $\mathcal{R}_P$ might be non-terminating even if the program P is terminating since reductions w.r.t. $\mathcal{R}_P$ are not restricted to reductions that are reachable from the initial state. It is thus desirable to make information about the initial state explicit throughout the program. Furthermore, a successful automatic termination proof might require simple invariants on the program variables (such as "a variable is always non-negative"). This kind of information can be obtained automatically using static analysis tools such as `Aspic/C2fsm` [14]. The obtained information can be added to the C program in the form of calls to an `assume` function with a built-in semantics. In the translation, these calls are converted into constraints that correspond to the invariants. We are planning to integrate a static analysis tool into the translation process so that these annotations do not need to be added manually.

[1] This simple abstraction is already sufficient in many cases. It is of course also possible to add constraints on the return value. Non-recursive functions can also be inlined on the LLVM-IR level, thus precisely tracking their return value.

```
int ack(int m, int n) {
    if (m <= 0)
        return n + 1;
    else if (n <= 0)
        return ack(m − 1, 1);
    else
        return ack(m − 1, ack(m, n − 1));
}
```

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_m, v_n)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_m, v_n) \ [\![v_m \leq 0]\!]\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_m, v_n) \ [\![v_m > 0]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{stop}}(v_m, v_n)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_m, v_n) \ [\![v_n \leq 0]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_m, v_n) \ [\![v_n > 0]\!]\\
\mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{start}}(v_m - 1, 1)\\
\mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{stop}}(v_m, v_n)\\
\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{start}}(v_m, v_n - 1)\\
\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{start}}(v_m - 1, z)\\
\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{stop}}(v_m, v_n)
\end{aligned}$$

```
define i32 @ack(i32 %m, i32 %n) {
entry:
  %0 = icmp sle i32 %m, 0
  br i1 %0, label %bb, label %bb1

bb:
  %1 = add nsw i32 %n, 1
  ret i32 %1

bb1:
  %2 = icmp sle i32 %n, 0
  br i1 %2, label %bb2, label %bb3

bb2:
  %3 = sub nsw i32 %m, 1
  %4 = call i32 @ack(i32 %3, i32 1)
  ret i32 %4

bb3:
  %5 = sub nsw i32 %n, 1
  %6 = call i32 @ack(i32 %m, i32 %5)
  %7 = sub nsw i32 %m, 1
  %8 = call i32 @ack(i32 %7, i32 %6)
  ret i32 %8
}
```

Figure 2 Ackermann's function in C, LLVM-IR, and as an `int`-based TRS.

4 Evaluation

In order to show the effectiveness and practicality of the proposed approach, it has been implemented in the tool `KITTeL` (KIT `int`-based TRS Termination Laboratory). Like its predecessor `pasta` [12], `KITTeL` consists of about 2400 lines of OCaml code. The input to `KITTeL` is an `int`-based TRS, the translation from LLVM-IR into `int`-based TRSs has been implemented in the separate tool `llvm2kittel` using about 3800 lines of C++ code.

The implementation in `KITTeL/llvm2kittel` has been evaluated on a collection of 174 examples that were taken from various places, including several recent papers on the termination of imperative programs [3, 5, 6, 8, 9, 19, 20], the textbook [21], and the `zlib` compression library. Furthermore, 31 examples were taken from TPDB's Java category [23] and converted to C. The collection of examples includes "classical" algorithms such as searching and sorting algorithms, cyclic redundancy check and hash code algorithms, encryption/decryption algorithms, image processing algorithms, and numerical algorithms. 14 out of these 174 examples require simple invariants on the program variables (such as "a variable is always non-negative") for a successful termination proof. This kind of information can be obtained automatically using static program analysis tools such as `Aspic/C2fsm` [14].

`KITTeL/llvm2kittel` has been able to show termination of all[2] examples fully automatically, on average taking less than 0.3 seconds (on a 2.4 GHz Intel® Core™2 Duo processor with 4 GB main memory) for each example, with the longest time being slightly more than 3 seconds. These times include the compilation from C into LLVM-IR, the translation from

[2] If the invariants are omitted from the aforementioned 14 examples, then termination cannot be shown.

LLVM-IR into a TRS, and the termination analysis of the obtained TRS. The following table contains details for some of the examples. Here, "LOC" gives the number of code lines in the C program and "RR" gives the number of rewrite rules that are generated.

C program	LOC	RR	Time / s	C program	LOC	RR	Time / s
`allroots`	200	77	0.861	`fft`	99	30	0.342
`almabench`	390	42	0.370	`hash`	241	80	0.566
`barr-crc16`	265	45	0.398	`jfdctint`	366	15	0.374
`barr-crc32`	265	45	0.402	`lpbench`	419	134	1.155
`barr-crc-ccitt`	265	35	0.318	`mergesort-recursive`	42	50	0.634
`bellman-ford`	75	39	0.369	`n-body`	141	35	0,287
`blit`	98	28	0.311	`prim`	83	44	0.487
`blowfish`	476	43	0.389	`sort`	138	90	0.757
`bmpfile`	749	254	3.050	`spectral-norm`	52	39	0.312
`bresenham`	36	9	0.106	`sphere`	157	68	0.617
`c-aes`	236	64	0.385	`spiral`	176	80	0.722
`c-des`	399	64	0.477	`zlib-adler32`	124	34	0.891
`cube`	146	68	0.616	`zlib-crc32-BYFOUR`	335	41	1.182
`dijkstra`	78	58	0.693	`zlib-crc32`	333	13	0.170

Notice that an empirical comparison with the methods from [5, 6, 8, 9, 19, 20] is not possible since implementations of these methods are not publicly available. The 31 Java programs from TPDB were also analyzed using the web interfaces of the Java termination tools `COSTA` [1] and `AProVE` [18, 4] using the default settings.

	Successful proofs	Unsuccessful attempts	Timeouts (60s)	Average time / s
`KITTeL`	31	–	–	0.133
`COSTA`	22	9	–	0.265
`AProVE`	28	–	3	13.265

Thus, `KITTeL` clearly shows the practicality and effectiveness of the proposed approach on a collection of "typical" examples. The examples, detailed results, an a link to a web interface of `KITTeL` are available at `http://baldur.iti.kit.edu/~falke/kittel/`.

5 Conclusions

We have presented a method for showing termination of C programs that is based on compiler intermediate languages and term rewriting techniques. For this, a C program is translated into an intermediate language by the compiler frontend and the obtained intermediate representation is then translated into a term rewrite system. Finally, termination of the obtained TRS is shown using term rewriting techniques.

In this paper, all integer types of the intermediate language are identified with $\mathbb{Z}$. Notice, however, that this abstraction might alter the termination behavior of the program under investigation. The methods from [5, 6, 8, 9, 19, 20] also exhibit this problem, and only [7] investigates the generation of ranking functions for bitvectors. In future work, we are planning to investigate how to model the bitvector behavior more precisely. While the translation into TRSs does not need to be modified substantially, proving termination of a TRS operating on bitvectors has not been investigated thus far. A further topic for future work is to suitably model the memory content (stack, heap, and global variables).

References

1 Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination analysis of Java bytecode. In *FMOODS '08*, pages 2–18, 2008.

2 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1–2):133–178, 2000.

3 Aaron Bradley, Zohar Manna, and Henny Sipma. Linear ranking with reachability. In *CAV '05*, pages 491–504, 2005.

4 Marc Brockschmidt, Carsten Otto, and Jürgen Giesl. Modular termination proofs of recursive Java bytecode programs by term rewriting. In *RTA '11*, 2011.

5 Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *TACAS '01*, pages 67–81, 2001.

6 Michael Colón and Henny Sipma. Practical methods for proving program termination. In *CAV '02*, pages 442–454, 2002.

7 Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *TACAS '10*, pages 236–250, 2010.

8 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS '05*, pages 87–101, 2005.

9 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06*, pages 415–426, 2006.

10 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV '06*, pages 415–418, 2006.

11 Stephan Falke and Deepak Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *RTA '08*, pages 94–109, 2008.

12 Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE '09*, pages 277–293, 2009.

13 Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. Technical Report 2011-6, Department of Informatics, Karlsruhe Institute of Technology, Germany, 2011. Available at `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000021789`.

14 Paul Feautrier and Laure Gonnord. Accelerated invariant generation for C programs with Aspic and C2fsm. *ENTCS*, 267(2):3–13, 2010.

15 Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *RTA '09*, pages 32–47, 2009.

16 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework. In *LPAR '04*, pages 301–331, 2005.

17 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88, 2004.

18 Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.

19 Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI '04*, pages 239–251, 2004.

20 Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS '04*, pages 32–41, 2004.

21 Robert Sedgewick. *Algorithms in C.* Addison-Wesley, 1990.

22 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3):8:1–8:70, 2010.

23 TPDB. Termination problem data base 7.0.2, 2010. Available from `http://termcomp.uibk.ac.at/2010/downloads/`.

24 Hans Zantema. Termination. In TeReSe, editor, *Term Rewriting Systems*, chapter 6. Cambridge University Press, 2003.