Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,6
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Termination Analysis of C Programs Using Compiler Intermediate Languages

Stephan Falke, Deepak Kapur, and Carsten Sinz

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

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 a termination analysis method 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 the proposed method.

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 [7, 8, 31, 32, 4]. More recently, the combination of abstraction refinement and linear ranking functions has been considered [11, 12, 6]. Based on this idea, the tool `Terminator` [13], developed at Microsoft Research, 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 presentation in the papers is typically based on idealized transition systems and the implementations themselves 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 support most of C's features.
2. The program whose termination behavior 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, 35, 30].

In this paper, we focus on the LLVM compiler framework and its intermediate language LLVM-IR [28]. 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 for TRSs has been investigated extensively in the past (see [37] for a survey). In this paper, TRSs with constraints over the integers (`int`-based TRS) are used, where the constraints are relations on the variables expressed as quantifier-free formulas. Similarly to what was proposed in [18, 20], we adapt well-known methods from the term rewriting literature for the termination analysis of `int`-based TRSs.

▶ **Example 1.** Consider the following simple C program:

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

Using the methods developed in this paper, the following `int`-based TRS is obtained from the LLVM-IR of the C program:

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

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.[1] 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 collection of examples taken from recent papers on the termination analysis of imperative programs [4, 5, 6, 7, 8, 11, 12, 31, 32], from the textbook [34], from the Java category of TPDB [36] and converted to C, and from various online sources clearly shows the effectiveness and practicality of our method.

The approach advocated in this paper is similar to the approach presented in [18]. There are, however, the following important differences:

[1] Why the program variable y gives rise to v_y and $y_{y.0}$ is explained in Section 4.

1. In contrast to [18], we now consider the real-life programming language C. In order to support all intricacies of C, we use an existing compiler frontend and operate on the compiler intermediate representation.
2. While [18] was restricted to (linear) Presburger arithmetic, we now support non-linear arithmetic.

This paper is organized as follows. Section 2 introduces `int`-based TRSs. Before discussing the translation of LLVM-IR programs into `int`-based TRSs in Section 4, the translation of Simple programs into `int`-based TRSs is discussed in Section 3. Simple is the while-language used by the `Interproc` static analysis tool [26], and the translation of Simple programs into `int`-based TRSs presents the main ideas that are used for the translation of LLVM-IR into `int`-based TRSs in a simpler context. Section 5 discusses the role of static analysis methods for the termination analysis of programs. Next, Sections 6–10 discuss the termination analysis of `int`-based TRSs. Section 11 outlines the implementation of these methods in `KITTeL`. Finally, Section 12 presents an empirical evaluation of `KITTeL` and Section 13 concludes.

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*. This paper uses a simplified, more natural notation for `int`-terms, i.e., the `int`-term $(x + (-(y * y))) + \mathsf{3}$ will be written as $x - y^2 + \mathsf{3}$. A *linear* `int`-term is an `int`-term that does not contain any occurrence of the function symbol "$*$". Notice that `int`-terms correspond to polynomial expressions and that linear `int`-terms correspond to linear functions.

We extend $\mathcal{F}_{\texttt{int}}$ 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`. These additional function symbols are used to model program behavior, and the set containing them is denoted by $\mathcal{F}$. Then, $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$ 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. Notice that nesting of function symbols from $\mathcal{F}$ is *not permitted*, thus resulting in a very simple term structure. This simple structure is nonetheless sufficient for modeling programs. In the following, s^* denotes a tuple of `int`-terms, and notions from terms are extended to tuples of terms component-wise. A *substitution* is a mapping from variables to `int`-terms.

`int`-constraints are quantifier-free formulas from (non-linear) integer arithmetic. This extends the $\mathcal{PA}$-constraints used in [18] which were limited to (linear) Presburger arithmetic.

▶ **Definition 2** (Syntax). An *atomic* `int`-*constraint* has the form $s \simeq t$, $s \geq t$, or $s > t$ for `int`-terms s, t. The set of `int`-*constraints* is inductively defined as follows:
1. $\top$ is an `int`-constraint.
2. Every atomic `int`-constraint is an `int`-constraint.
3. If φ is an `int`-constraint, then $\neg\varphi$ is an `int`-constraint.
4. If φ_1, φ_2 are `int`-constraints, then $\varphi_1 \wedge \varphi_2$ is an `int`-constraint.

The Boolean connectives $\bot$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ are defined as usual. Furthermore, `int`-constraints of the form $s < t$ and $s \leq t$ denote the `int`-constraints $t > s$ and $t \geq s$, respectively. Also, $s \not\simeq t$ abbreviates $\neg(s \simeq t)$, and similarly for the other predicate symbols.

`int`-constraints have the expected semantics. In the next definition, $\overline{n}$ denotes the integer corresponding to the variable-free `int`-term n (i.e., $\overline{n}$ is the *evaluation* of n according to the standard semantics of "$+$", "$*$", and "$-$").

▶ **Definition 3** (Semantics). A variable-free `int`-constraint φ is `int`-*valid* iff
1. φ has the form $\top$, or
2. φ has the form $s \simeq t$ and $\overline{s} = \overline{t}$ in $\mathbb{Z}$, or
3. φ has the form $s \geq t$ and $\overline{s} \geq \overline{t}$ in $\mathbb{Z}$, or
4. φ has the form $s > t$ and $\overline{s} > \overline{t}$ in $\mathbb{Z}$, or
5. φ has the form $\neg\varphi_1$ and φ_1 is not `int`-valid, or
6. φ has the form $\varphi_1 \wedge \varphi_2$ and both φ_1 and φ_2 are `int`-valid.

An `int`-constraint φ with variables is `int`-*valid* iff $\varphi\sigma$ is `int`-valid for all ground substitutions $\sigma : \mathcal{V}(\varphi) \to \mathcal{T}(\mathcal{F}_{\texttt{int}})$. An `int`-constraint φ is `int`-*satisfiable* iff there exists a ground substitution $\sigma : \mathcal{V}(\varphi) \to \mathcal{T}(\mathcal{F}_{\texttt{int}})$ such that $\varphi\sigma$ is `int`-valid. Otherwise, φ is `int`-*unsatisfiable*.

`int`-validity and `int`-satisfiability are decidable for linear `int`-constraints [33].

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 6. The rules generalize the $\mathcal{PA}$-based rewrite rules from [18]. Alternatively, they can be interpreted as a restricted form of the rewrite rules considered in [20] which support nested function symbols.

▶ **Definition 4** (`int`-Based Rewrite Rules). An `int`-*based rewrite rule* has the form $l \to r[\![\varphi]\!]$ where $l = f(x_1, \ldots, x_n)$ for pairwise distinct variables $x_1, \ldots, x_n$, $r \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$, and φ is an `int`-constraint.

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. Notice that r and φ may use 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 $x_i \simeq x_j$.

`int`-based TRSs give rise to the following rewrite relation. It requires that the constraint of the `int`-based rewrite rule is `int`-valid after being instantiated by the matching substitution. This is in general only decidable if the constraint and the matching substitution are linear. An easy way to achieve decidability is to define the rewrite relation only on terms whose arguments are from $\mathcal{F}_{\mathbb{Z}}$. Then, the substitutions used for matching instantiate variables by constant symbols from $\mathcal{F}_{\mathbb{Z}}$.

▶ **Definition 5** ($\mathcal{F}_{\mathbb{Z}}$-Based Substitutions). A substitution σ is $\mathcal{F}_{\mathbb{Z}}$-*based* iff $\sigma(x) \in \mathcal{F}_{\mathbb{Z}}$ for all variables x.

▶ **Definition 6** (Rewrite Relation). For an `int`-based TRS $\mathcal{R}$, the relation $s \to_{\texttt{int}\backslash\mathcal{R}} t$ for terms s, t of the form $f(\mathsf{n}_1, \ldots, \mathsf{n}_k)$ 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, $\mathsf{norm}(r\sigma)$ evaluates the arguments according to the usual semantics of "+", "∗", and "−" on ground terms.

▶ **Example 7.** For $\mathcal{R}$ from Example 1, $\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(2, 2, 2, 1) \to_{\texttt{int}\backslash\mathcal{R}} \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(2, 2, 2, 1)$ using the third rewrite rule. To see this, notice that for $\sigma = \{v_x \mapsto 2, v_y \mapsto 2, v_{y.0} \mapsto 2, v_{r.0} \mapsto 1\}$, $(v_{y.0} > 0)\sigma = (2 > 0)$ is `int`-valid. Next, $\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(2, 2, 2, 1) \to_{\texttt{int}\backslash\mathcal{R}} \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(2, 2, 1, 2)$ since $\mathsf{norm}(\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(2, 2, 2 - 1, 1 * 2)) = \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(2, 2, 1, 2)$. ◀

3 Translating Simple Programs into `int`-Based TRSs

Before considering the translation from LLVM-IR programs into `int`-based TRSs, this section first considers a simple imperative programming language where programs are formed according to the grammar in Figure 1. Most of the ideas used for LLVM-IR programs in Section 4 are more intuitive when considered at the level of this toy programming language. The language defined in Figure 1 is the function-free[2] fragment of the Simple language that is also used as the input language of the `Interproc` static analysis tool [26].

```
  <program>   ::=  var <vars-decl>; begin <statement>+ end
  <vars-decl> ::=  id: int (, id: int)*
  <statement> ::=  skip;
               |   halt;
               |   assume (<bexpr>);
               |   id = random;
               |   id = <nexpr>;
               |   if (<bexpr>) then <statement>+ else <statement>+ endif;
               |   while (<bexpr>) do <statement>+ done;
  <bexpr>     ::=  brandom
               |   "int-constraints"
  <nexpr>     ::=  "int-terms"
```

Figure 1 Grammar for Simple programs.

Most constructs in this programming language have the expected meaning, e.g., `skip` is a do-nothing statement and `halt` halts the program execution. For the `int`-constraints in <bexpr>, conjunction is written as `and`, disjunction is written as `or`, and negation is written as `not`. Furthermore, the predicates are written `==`, `>=`, `>`, `<=`, and `<`.

The statement `assume (`*bexpr*`)` is equivalent to `if (`*bexpr*`) then skip; else halt; endif;`. Its effect is to consider only program runs that satisfy the given Boolean expression.

The `brandom`-construct can be used to abstract aspects of a program that cannot or do not need to be modeled precisely. For this, a nondeterministic choice is encoded as

```
if (brandom) then
   ...
else
   ...
endif;
```

Similarly, an assignment `x = random;` assigns an undetermined value to the variable `x`. Assumptions on this value can be modeled with a subsequent `assume`, e.g., the effect of `x = random; assume (x >= 0 and x <= 2);` is that the value of `x` is between 0 and 2 in the remaining program. An important use of `random` is to simulate division operations. For instance, the "statement" `y = x / 2;` is equivalent to

```
y = random;
assume (x - 2*y >= 0 and x - 2*y <= 1);
```

[2] Using the same ideas that are used for LLVM-IR in Section 4, it would be possible to support functions.

Similarly, the `if`-"statement" `if (x % 2 == 0) then ... else ... endif;` which tests whether the variable `x` is even can be simulated by

```
y = random;
if (brandom) then
  assume (x = 2*y);
  ...
else
  assume (x = 2*y + 1);
  ...
endif;
```

3.1 The Translation

The translation now proceeds as follows, where we assume that the program uses the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$. In a first step, each statement ω of the program is assigned two function symbols, $\mathsf{state}_{\mathsf{in}}^{\omega}$ and $\mathsf{state}_{\mathsf{out}}^{\omega}$. Furthermore, special function symbols, $\mathsf{state}_{\mathsf{start}}$ and $\mathsf{state}_{\mathsf{stop}}$, denoting starting and stopping states, are introduced. For a non-empty sequence $\Omega = \omega_1; \ldots; \omega_m;$ of statements, let $\mathsf{state}_{\mathsf{in}}^{\Omega} = \mathsf{state}_{\mathsf{in}}^{\omega_1}$ and $\mathsf{state}_{\mathsf{out}}^{\Omega} = \mathsf{state}_{\mathsf{out}}^{\omega_m}$. Furthermore, for all $1 \leq i < m$, the function symbols $\mathsf{state}_{\mathsf{out}}^{\omega_i}$ and $\mathsf{state}_{\mathsf{in}}^{\omega_{i+1}}$ are identified. A mapping from statements to `int`-based rewrite rules is now defined by the case distinction in Figure 2. In the translation of <bexpr>, both `brandom` and $\neg$`brandom` become $\top$ in the constraints of the rewrite rules. For a Simple program P with the sequence of statements Ω, the `int`-based TRS $\mathcal{R}_P$ consists of the `int`-based rewrite rules obtained for all statements occurring in the program and the additional `int`-based rewrite rules $\mathsf{state}_{\mathsf{start}}(x_1, \ldots, x_n) \to \mathsf{state}_{\mathsf{in}}^{\Omega}(x_1, \ldots, x_n)$ and $\mathsf{state}_{\mathsf{out}}^{\Omega}(x_1, \ldots, x_n) \to \mathsf{state}_{\mathsf{stop}}(x_1, \ldots, x_n)$.

Statement ω	`int`-based rewrite rules			
`skip;`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, x_n)$	
`halt;`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{stop}}(x_1, \ldots, x_n)$	
`assume (`bexpr`);`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, x_n)$	$[\![\mathit{bexpr}]\!]$
	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{stop}}(x_1, \ldots, x_n)$	$[\![\neg\mathit{bexpr}]\!]$
$\mathtt{x}_i$ `= random;`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, x_i', \ldots, x_n)$	
	where x_i' is a fresh variable			
$\mathtt{x}_i$ `=` nexpr`;`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, \mathit{nexpr}, \ldots, x_n)$	
`if (`bexpr`) then`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{in}}^{\Omega_1}(x_1, \ldots, x_n)$	$[\![\mathit{bexpr}]\!]$
Ω_1	$\mathsf{state}_{\mathsf{out}}^{\Omega_1}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, x_n)$	
`else`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{in}}^{\Omega_2}(x_1, \ldots, x_n)$	$[\![\neg\mathit{bexpr}]\!]$
Ω_2	$\mathsf{state}_{\mathsf{out}}^{\Omega_2}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, x_n)$	
`endif;`				
`while (`bexpr`) do`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{in}}^{\Omega}(x_1, \ldots, x_n)$	$[\![\mathit{bexpr}]\!]$
Ω	$\mathsf{state}_{\mathsf{out}}^{\Omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	
`done;`	$\mathsf{state}_{\mathsf{in}}^{\omega}(x_1, \ldots, x_n)$	$\to$	$\mathsf{state}_{\mathsf{out}}^{\omega}(x_1, \ldots, x_n)$	$[\![\neg\mathit{bexpr}]\!]$

Figure 2 Mapping from statements to `int`-based rewrite rules.

▶ **Example 8.** Using the translation given above, the Simple program

```
1 var x: int, y: int, r: int;
2 begin
3   r = 1;
4   while (y > 0) do
5     r = r * x;
6     y = y - 1;
7   done;
8 end
```

is translated into the `int`-based rewrite rules

$$
\begin{aligned}
\mathsf{state}_{\mathsf{start}}(x, y, r) &\to \mathsf{state}_3(x, y, r) \\
\mathsf{state}_3(x, y, r) &\to \mathsf{state}_4(x, y, 1) \\
\mathsf{state}_4(x, y, r) &\to \mathsf{state}_5(x, y, r) & [\![y > 0]\!] \\
\mathsf{state}_4(x, y, r) &\to \mathsf{state}_8(x, y, r) & [\![\neg(y > 0)]\!] \\
\mathsf{state}_5(x, y, r) &\to \mathsf{state}_6(x, y, r * x) \\
\mathsf{state}_6(x, y, r) &\to \mathsf{state}_7(x, y - 1, r) \\
\mathsf{state}_7(x, y, r) &\to \mathsf{state}_4(x, y, r) \\
\mathsf{state}_8(x, y, r) &\to \mathsf{state}_{\mathsf{stop}}(x, y, r)
\end{aligned}
$$

Here, the line numbers have been used as subscripts for the function symbols $\mathsf{state}^{\omega}_{\mathsf{in}}$ and $\mathsf{state}^{\omega}_{\mathsf{out}}$ in order to improve readability. ◀

The following theorem is based on the observation that any state transition of the Simple program P can be mimicked by a rewrite sequence w.r.t. $\mathcal{R}_P$. This is relatively straightforward, since the `int`-based TRS $\mathcal{R}_P$ that is generated is essentially a transition system.

▶ **Theorem 9.** *Let P be a* Simple *program. Then the above translation produces an* `int`*-based TRS $\mathcal{R}_P$ such that P is terminating if $\mathcal{R}_P$ is terminating.*

Proof idea. That the translation produces an `int`-based TRS is immediate by inspection. For the second statement, it can be shown that the `int`-based rewrite rules correspond to the operational semantics of Simple. Then, each infinite computation of P immediately gives rise to an infinite reduction w.r.t. $\mathcal{R}_P$. ◀

Notice that $\mathcal{R}_P$ might be non-terminating even if P is terminating since information about the starting state of the program is not propagated in the `int`-based TRS (but see Section 5).

3.2 Combination of `int`-Based Rewrite Rules

The translation given above produces a large number of `int`-based rewrite rules since each statement in the Simple program 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. On the level of the Simple program, this corresponds to the composition of several statements into a single statement. This is particularly useful for combining rules from a straight-line code segment (i.e., a code segment consisting of assignments, `skip`-, and `halt`-statements only) and can increase the performance of the subsequent termination analysis considerably.

For a Simple program P with the sequence of statements Ω, the *control points* of P are the function symbols $\mathsf{state}_{\mathsf{start}}$, $\mathsf{state}_{\mathsf{stop}}$, and $\mathsf{state}^{\omega}_{\mathsf{in}}$ for each `assume`-, `if`-, and `while`-statement occurring in P. In the following, let C be the set of control points of P. 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 $\mathsf{state}_j(x_1, \ldots, x_n) \to \mathsf{state}_k(e'_1, \ldots, e'_n)$ (notice that the `int`-based rewrite rules for function symbols not in C always have the `int`-constraint $\top$), resulting in

$$\mathsf{state}_i(x_1, \ldots, x_n) \to \mathsf{state}_k(e'_1\omega, \ldots, e'_n\omega)[\![\varphi]\!]$$

where $\omega = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. The old rules are replaced by the new rule 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. Finally, rules with a function symbol that is not in C are deleted.

▶ **Example 10.** Applying the combination of `int`-based rewrite rules to the `int`-based TRS from Example 8 produces the `int`-based TRS

$$\begin{aligned} \mathsf{state}_{\mathsf{start}}(x, y, r) &\to \mathsf{state}_4(x, y, 1) \\ \mathsf{state}_4(x, y, r) &\to \mathsf{state}_4(x, y - 1, r * x)[\![y > 0]\!] \\ \mathsf{state}_4(x, y, r) &\to \mathsf{state}_{\mathsf{stop}}(x, y, r) \qquad [\![\neg(y > 0)]\!] \end{aligned}$$

Here, the set of control points is $C = \{\mathsf{state}_{\mathsf{start}}, \mathsf{state}_{\mathsf{stop}}, \mathsf{state}_4\}$. ◀

Notice that the translation with subsequent combination of `int`-based rewrite rules according to control points is similar to the translation proposed in [18], but recall that the translation in [18] is restricted to (linear) Presburger arithmetic.

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

Lifting the method presented in Section 3 to a real programming language such as C 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 languages typically have a simple syntax and semantics, thus simplifying the translation into `int`-based TRSs significantly (for similar reasons, termination analysis for Java programs is often performed on the bytecode level and not on the source code [1, 35, 30]).

In this paper, we consider LLVM and its intermediate language LLVM-IR [28]. 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 11 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 working on registers or constants, 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 (type casts do not change the bit-level representation of the data) 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 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 11 is that the register `%r.0` is assigned the value `1` if the control flow passed from `entry` to `bb1`. If the control flow passed from `bb` to `bb1`, then `%r.0` is assigned the value contained in `%1`. These 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.[3]

4.1 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` instructions. 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 11.** For the C program from Example 1, the following LLVM-IR program is obtained using the LLVM compiler frontend `llvm-gcc`:

```
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
}
```

Here, the basic blocks `bb1` and `bb` correspond to the loop in the C program. ◀

[3] Details on `getelementptr` can be found at `http://llvm.org/docs/GetElementPtr.html`.

An LLVM-IR program is now translated into an `int`-based TRS as follows. Each integer-typed (i.e., of a type different from `i1`) 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. Similar to Section 3, each TAC instruction gives rise to a rewrite rule that mimics the effect of that instruction. Here, division instructions are handled as in Section 3 by introducing a fresh variable on the right-hand side and adding appropriate constraints on that variable. Remainder instructions are handled similarly by introducing fresh variables on the right-hand side.

Since `int`-based TRSs operate on mathematical integers, all integer types different from `i1` are identified with the mathematical integers in the following.

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

Integer type cast instructions thus do not have any effect.

The control flow of the LLVM-IR program is mimicked as follows. As in Section 3, 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 Ω of integer-typed TAC instructions, then a rule $\mathsf{state}_{bb_{\mathsf{in}}}(\ldots) \to \mathsf{state}^{\Omega}_{\mathsf{in}}(\ldots)$ (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 constraint that corresponds to the (negated) branch condition.

▶ **Example 12.** Consider the C program from Example 1 and its LLVM-IR from Example 11. 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, \mathsf{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) \quad [\![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) \quad [\![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} - \mathsf{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 corresponding to instructions in order to improve readability. ◀

The statement of Theorem 9 holds for LLVM-IR programs as well, i.e., an LLVM-IR program is terminating if the `int`-based TRS produced by the translation is terminating.

▶ **Theorem 13.** *Let P be an* LLVM-IR *program. Then the above translation produces an* `int`*-based TRS $\mathcal{R}_P$ such that P is terminating if $\mathcal{R}_P$ is terminating.*

Again, $\mathcal{R}_P$ might be non-terminating even if P is terminating (but see Section 5).

4.2 Simplification of `int`-Based Rewrite Rules

A combination of the `int`-based rewrite rules obtained by the translation can be done as in Section 3. For `int`-based TRSs obtained from LLVM-IR, the set of *control points* 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.

▶ **Example 14.** Continuing Example 12, 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

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

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 and 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 15.** After removing the unneeded arguments in Example 14,

$$
\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) && && (1)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_y, \mathsf{1}) && && (2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) && [\![v_{y.0} > \mathsf{0}]\!] && (3)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) && [\![v_{y.0} \le \mathsf{0}]\!] && (4)\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0} - \mathsf{1}, v_{r.0} * v_x) && && (5)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0}) && && (6)
\end{aligned}
$$

is obtained since arguments 5 and 6 are not needed. ◀

4.3 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 available as prototype declarations (library functions are a prime example).

▶ Assumption 2. It is assumed that all functions that are only declared as prototypes are terminating. Furthermore, these functions are assumed to not call functions defined in the program.

If the user-defined functions have function call hierarchies 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. This rule has the form $\mathsf{state}_i(\ldots) \to \mathsf{state}_{i+1}(\ldots, z, \ldots)$, 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.[4][5] A call to a function with void type is handled similarly, but no fresh variable is introduced on the right-hand side.

▶ **Example 16.** The following C program computes the Ackermann function:

```
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));
    }
}
```

The C program is compiled into the following LLVM-IR program:

```
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
```

[4] Intuitively, a non-terminating program run starting at the call instruction is either a non-terminating run starting in the called function f, or a run where the call to f terminates and the infinite run continues with the next instruction after the call.

[5] Another way to look at this is that the recursive call to f intuitively corresponds to the single rewrite rule $\mathsf{state}_i(\ldots) \to \mathsf{state}_{i+1}(\ldots, \mathsf{state}^f_{\mathsf{start}}(\ldots), \ldots)$. The rewrite rules that are generated by the proposed translation correspond the the dependency pairs [2] of that rewrite rule, where the nested function call has been replaced by a fresh variable.

```
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
}
```

Using the approach outlined above, the following `int`-based TRS is generated:

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

Termination of this TRS is easily shown using the methods presented in Sections 6–10. ◀

4.4 Programs Containing Pointers and Floating Point Numbers

`int`-based TRSs 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 rewrite rule. Pointer or floating point comparisons are handled the same way that `brandom` was handled in Section 3.

▶ **Example 17.** The following C program computes the maximal element in the range [`low`..`high`] of the array pointed to by `a`:

```
int max(int a[], int low, int high) {
    if (low >= high) {
        return a[low];
    } else {
        int mid = (low + high) / 2;
        int leftmax = max(a, low, mid);
        int rightmax = max(a, mid + 1, high);
        if (leftmax > rightmax) {
            return leftmax;
        } else {
            return rightmax;
        }
    }
}
```

This program is translated into the following LLVM-IR program (here, the `select` instruction chooses between `%5` and `%7`, depending on the truth value of `%8`):

```
define i32 @max(i32* %a, i32 %low, i32 %high) {
entry:
  %0 = icmp sge i32 %low, %high
  br i1 %0, label %bb, label %bb1

bb:
  %1 = getelementptr i32* %a, i32 %low
  %2 = load i32* %1
  ret i32 %2

bb1:
  %3 = add i32 %low, %high
  %4 = sdiv i32 %3, 2
  %5 = call i32 @max(i32* %a, i32 %low, i32 %4)
  %6 = add i32 %4, 1
  %7 = call i32 @max(i32* %a, i32 %6, i32 %high)
  %8 = icmp sgt i32 %5, %7
  %retval = select i1 %8, i32 %5, i32 %7
  ret i32 %retval
}
```

Termination of the generated `int`-based TRS

$$
\begin{array}{rcll}
\mathsf{state}_{\mathsf{start}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{entry_{in}}}(v_{low}, v_{high}) & \\
\mathsf{state}_{\mathsf{entry_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{bb_{in}}}(v_{low}, v_{high}) & [\![v_{low} \geq v_{high}]\!] \\
\mathsf{state}_{\mathsf{entry_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{bb1_{in}}}(v_{low}, v_{high}) & [\![v_{low} < v_{high}]\!] \\
\mathsf{state}_{\mathsf{bb_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{stop}}(v_{low}, v_{high}) & \\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{start}}(v_{low}, z_1) & [\![v_{low} + v_{high} - 2 * z_1 \geq 0 \land \\
 & & & \;\; v_{low} + v_{high} - 2 * z_1 < 2 \quad]\!] \\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{start}}(z_1 + 1, v_{high}) & [\![v_{low} + v_{high} - 2 * z_1 \geq 0 \land \\
 & & & \;\; v_{low} + v_{high} - 2 * z_1 < 2 \quad]\!] \\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{stop}}(v_{low}, v_{high}) & [\![z_2 > z_3]\!] \\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_{low}, v_{high}) & \to & \mathsf{state}_{\mathsf{stop}}(v_{low}, v_{high}) & [\![z_2 \leq z_3]\!]
\end{array}
$$

can easily be established using the methods developed in this paper (here, z_1 corresponds to the division and z_2 and z_3 correspond to the return values of the recursive calls). ◀

5 Utilizing Static Analysis Methods

Notice that the translations from Simple programs and LLVM-IR programs into `int`-based TRSs do 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.

▶ **Example 18.** The Simple program

```
var x: int;
begin
  x = 1;
  while (x < 217) do
    x = 2*x;
  done;
end
```

is clearly terminating since the `while`-loop is executed exactly 8 times. The `int`-based TRS

$$
\begin{aligned}
\mathsf{state}_{\mathsf{start}}(x) &\to \mathsf{state}_4(1) \\
\mathsf{state}_4(x) &\to \mathsf{state}_4(2 * x) [\![x < \mathsf{217}]\!] \\
\mathsf{state}_4(x) &\to \mathsf{state}_{\mathsf{stop}}(x) \quad [\![x \geq \mathsf{217}]\!]
\end{aligned}
$$

obtained by the translation and the combination of `int`-based rewrite rules according to control points is, however, non-terminating since $\mathsf{state}_4(\mathsf{0}) \to_{\mathtt{int}\backslash\mathcal{R}} \mathsf{state}_4(\mathsf{0}) \to_{\mathtt{int}\backslash\mathcal{R}} \ldots$. ◀

It is thus desirable to make information about the initial state explicit throughout the program. Furthermore, a successful automatic termination proof requires simple invariants on the program variables (such as "a variable is always non-negative") in some cases.

For Simple programs, this kind of information can be obtained automatically using the static analysis tool `Interproc` [26], which is based on the abstract interpretation framework [14] in combination with the interval [14], polyhedra [15], or octagon [29] domain. The translation from Section 3 can utilize this information if the invariants computed by the tool are added to the Simple program in the form of `assume`-statements. Notice that this does not alter the program behavior since the invariants imply that the added `assume`-statements are equivalent to a `skip`-statement.

▶ **Example 19.** Using the `Interproc` static analysis tool on the Simple program from Example 18, one invariant is obtained:

```
var x: int;
begin
  x = 1;
  while (x < 217) do
    assume (x >= 1);
    x = 2*x;
  done;
end
```

Now, the `int`-based TRS

$$
\begin{aligned}
\mathsf{state}_{\mathsf{start}}(x) &\to \mathsf{state}_4(1) \\
\mathsf{state}_4(x) &\to \mathsf{state}_5(x) \qquad [\![x < \mathsf{217}]\!] \\
\mathsf{state}_5(x) &\to \mathsf{state}_4(2 * x) [\![x \geq \mathsf{1}]\!] \\
\mathsf{state}_5(x) &\to \mathsf{state}_{\mathsf{stop}}(x) \quad [\![x < \mathsf{1}]\!] \\
\mathsf{state}_4(x) &\to \mathsf{state}_{\mathsf{stop}}(x) \quad [\![x \geq \mathsf{217}]\!]
\end{aligned}
$$

is the result of the translation and combination of `int`-based rewrite rules according to control points. This `int`-based TRS is terminating and the methods developed in this paper can easily prove this. ◀

Similarly, for C programs, the static analysis tool `Aspic/C2fsm` [19] can be used to automatically compute invariants. These invariants can then be added to the C program as calls to a (prototype only) `assume` function with a built-in semantics. These calls are then handled the same way that the `assume`-statements from Simple were handled (see Section 3).

6 Characterizing Termination of `int`-Based TRSs

The remainder of this paper is concerned with methods for showing termination of `int`-based TRSs. In order to verify termination of `int`-based TRSs, the notion of *chains* is used. Intuitively, a chain represents a possible sequence of rule applications in a reduction w.r.t. $\to_{\mathtt{int}\backslash\mathcal{R}}$. In the following, it is always assumed that different (occurrences of) `int`-based rewrite rules are variable-disjoint, and the domain of substitutions may be infinite. This allows for a single substitution in the following definition. Recall that $\to_{\mathtt{int}\backslash\mathcal{R}}$ is only applied at the root position of a term.

▶ **Definition 20** ($\mathcal{R}$-Chains)**.** Let $\mathcal{R}$ be an `int`-based TRS. 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 $i \geq 1$.

▶ **Example 21.** Continuing Example 15, the $\mathcal{R}$-chain

$$\begin{array}{rcll}
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) & \to & \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) & [\![v_{y.0} > 0]\!] \\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v'_x, v'_y, v'_{y.0}, v'_{r.0}) & \to & \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v'_x, v'_y, v'_{y.0} - 1, v'_{r.0} * v'_x) & \\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v''_x, v''_y, v''_{y.0}, v''_{r.0}) & \to & \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v''_x, v''_y, v''_{y.0}, v''_{r.0}) & [\![v''_{y.0} > 0]\!]
\end{array}$$

can be built by considering the substitution $\sigma = \{v_x \mapsto 2, v'_x \mapsto 2, v''_x \mapsto 2, v_y \mapsto 2, v'_y \mapsto 2, v''_y \mapsto 2, v_{y.0} \mapsto 2, v'_{y.0} \mapsto 2, v''_{y.0} \mapsto 1, v_{r.0} \mapsto 1, v'_{r.0} \mapsto 1, v''_{r.0} \mapsto 2\}$ since

$$\begin{array}{rcl}
\mathsf{norm}(\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0})\sigma) & = & \mathsf{norm}(\mathsf{state}(2, 2, 2, 1)) \\
& = & \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(2, 2, 2, 1) \\
& = & \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v'_x, v'_y, v'_{y.0}, v'_{r.0})\sigma
\end{array}$$

and

$$\begin{array}{rcl}
\mathsf{norm}(\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v'_x, v'_y, v'_{y.0} - 1, v'_{r.0} * v'_x)\sigma) & = & \mathsf{norm}(\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(2, 2, 2 - 1, 1 * 2)) \\
& = & \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(2, 2, 1, 2) \\
& = & \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v''_x, v''_y, v''_{y.0}, v''_{r.0})\sigma
\end{array}$$

where additionally $(v_{y.0} > 0)\sigma = (2 > 0)$ and $(v''_{y.0} > 0)\sigma = (1 > 0)$ are `int`-valid. ◀

Using the notion of $\mathcal{R}$-chains, the following characterization of termination of an `int`-based TRS $\mathcal{R}$ is easily obtained.

▶ **Theorem 22.** *Let $\mathcal{R}$ be an* `int`*-based TRS. Then $\mathcal{R}$ is terminating if and only if there are no infinite $\mathcal{R}$-chains.*

Proof. Let $\mathcal{R}$ be an `int`-based TRS.

"$\Leftarrow$" Assume that there exists a term s which starts an infinite $\to_{\mathtt{int}\backslash\mathcal{R}}$-reduction and consider an infinite reduction starting with s. According to the definition of $\to_{\mathtt{int}\backslash\mathcal{R}}$, there exist an `int`-based rewrite rule $l_1 \to r_1[\![\varphi_1]\!] \in \mathcal{R}$ and an $\mathcal{F}_{\mathbb{Z}}$-based substitution σ_1 such that $s = l_1\sigma_1$ and $\varphi_1\sigma_1$ is `int`-valid. The reduction then yields $\mathsf{norm}(r_1\sigma_1)$ and the infinite $\to_{\mathtt{int}\backslash\mathcal{R}}$-reduction continues with $\mathsf{norm}(r_1\sigma_1)$, i.e., the term $\mathsf{norm}(r_1\sigma_1)$ starts an infinite $\to_{\mathtt{int}\backslash\mathcal{R}}$-reduction as well. The first `int`-based rewrite rule in the infinite $\mathcal{R}$-chain that is being constructed is $l_1 \to r_1[\![\varphi_1]\!]$. The other `int`-based rewrite rules of the infinite $\mathcal{R}$-chain are determined in the same way: let $l_i \to r_i[\![\varphi_i]\!]$ be an `int`-based rewrite rule such that $\mathsf{norm}(r_i\sigma_i)$ starts an infinite $\to_{\mathtt{int}\backslash\mathcal{R}}$-reduction. Again, an `int`-based rewrite rule $l_{i+1} \to r_{i+1}[\![\varphi_{i+1}]\!]$ is applied to $\mathsf{norm}(r_i\sigma_i)$ using a substitution σ_{i+1} and the term $\mathsf{norm}(r_{i+1}\sigma_{i+1})$ starts an infinite $\to_{\mathtt{int}\backslash\mathcal{R}}$-reduction. This produces the next `int`-based rewrite rule in the infinite $\mathcal{R}$-chain. In this way, the infinite sequence

$$l_1 \to r_1[\![\varphi_1]\!],\ l_2 \to r_2[\![\varphi_2]\!],\ l_3 \to r_3[\![\varphi_3]\!],\ \ldots$$

is obtained. Since it is assumed that different (occurrences of) `int`-based rewrite rules are variable-disjoint, the substitution $\sigma = \sigma_1 \cup \sigma_2 \cup \ldots$ gives $\mathsf{norm}(r_i\sigma) = l_{i+1}\sigma$ and `int`-validity of the instantiated `int`-constraint $\varphi_i\sigma$ for all $i \geq 1$. Thus, the above infinite sequence is indeed an infinite $\mathcal{R}$-chain.

"⇒" Assume there exists an infinite $\mathcal{R}$-chain

$$l_1 \to r_1[\![\varphi_1]\!],\ l_2 \to r_2[\![\varphi_2]\!],\ l_3 \to r_3[\![\varphi_3]\!],\ \ldots$$

Hence, there exists a substitution σ such that

$$\begin{aligned}\mathsf{norm}(r_1\sigma) &= l_2\sigma,\\ \mathsf{norm}(r_2\sigma) &= l_3\sigma,\\ &\ \vdots\end{aligned}$$

and the instantiated `int`-constraints $\varphi_1\sigma, \varphi_2\sigma, \ldots$ are `int`-valid.

From this, the infinite $\to_{\mathtt{int}\backslash\mathcal{R}}$-reduction

$$\mathsf{norm}(r_1\sigma) \to_{\mathtt{int}\backslash\mathcal{R}} \mathsf{norm}(r_2\sigma) \to_{\mathtt{int}\backslash\mathcal{R}} \mathsf{norm}(r_3\sigma) \ldots$$

is obtained, and $\mathcal{R}$ is thus not terminating. ◀

In the next sections, various techniques for showing termination of `int`-based TRSs are developed. These techniques are stated independently of each other in the form of *termination processors*, following the dependency pair framework for ordinary term rewriting [21] and for term rewriting with built-in numbers [17]. The main motivation for this approach is that it allows to combine different termination techniques in a flexible manner since it typically does not suffice to just use a single technique in a successful termination proof.

Termination processors are used to transform an `int`-based TRS into a (finite) set of `int`-based TRSs for which termination is (hopefully) easier to show. A termination processor Proc is *sound* iff for all `int`-based TRSs $\mathcal{R}$, $\mathcal{R}$ is terminating whenever all `int`-based TRSs in $\mathsf{Proc}(\mathcal{R})$ are terminating. Notice that $\mathsf{Proc}(\mathcal{R}) = \{\mathcal{R}\}$ is possible. This can be interpreted as a failure of Proc and indicates that a different termination processor should be applied.

Using sound termination processors, a termination proof of $\mathcal{R}$ then consists of the repeated application of these processors. If all `int`-based TRSs obtained in this process are transformed into $\emptyset$, then $\mathcal{R}$ is terminating.

7 Splitting into Dual Clauses

Often, it is convenient to only consider `int`-based rewrite rules with a restricted kind of `int`-constraints. In particular, the restriction to `int`-constraints that are conjunctions of (negated) atomic `int`-constraints may be convenient. This can be achieved by a conversion into disjunctive normal form (DNF) and the introduction of one rewrite rule for each dual clause in this DNF.[6]

▶ **Theorem 23** (Processor Based on DNF). *The termination processor with* $\mathsf{Proc}(\mathcal{R}) = \bigcup_{l \to r[\![\varphi]\!] \in \mathcal{R}} \mathsf{dnf}(l \to r[\![\varphi]\!])$ *where*

$$\mathsf{dnf}(l \to r[\![\varphi]\!]) = \{l \to r[\![\psi]\!] \mid \psi \text{ is a dual clause in the DNF of } \varphi\}$$

is sound.

Proof. It needs to be shown that every occurrence of (a variable-renamed version of) $l \to r[\![\varphi]\!]$ in an infinite chain can be replaced by some `int`-based rewrite rule from $\mathsf{Proc}(\mathcal{R})$. Thus, assume that some infinite chain contains $\ldots, l \to r[\![\varphi]\!], \ldots$. Let the infinite chain be based on the substitution σ, i.e., $\varphi\sigma$ is `int`-valid. Thus, the DNF of $\varphi\sigma$ is `int`-valid as well, which means that (at least) one dual clause in the DNF of $\varphi\sigma$ is `int`-valid. Since there exists an `int`-based rewrite rule $l \to r[\![\psi]\!] \in \mathsf{Proc}(\mathcal{R})$ that corresponds to this dual clause, $l \to r[\![\varphi]\!]$ can be replaced by $l \to r[\![\psi]\!]$ and there exists an infinite $\mathsf{Proc}(\mathcal{R})$-chain as well. ◀

8 Termination Graphs

Notice that an `int`-based TRS $\mathcal{R}$ may give rise to infinitely many different $\mathcal{R}$-chains. This section introduces a method that represents these infinitely many chains in a finite graph. Then, each $\mathcal{R}$-chain (and thus each computation path in the imperative program) corresponds to a path in this graph. By considering the strongly connected components of this graph, it then becomes possible to decompose an `int`-based TRS into several independent `int`-based TRSs by determining which `int`-based rewrite rules may follow each other in a chain.

The termination processor for this idea uses *termination graphs*, which are motivated by the dependency graphs used in the dependency pair framework for ordinary term rewriting [2] and rewriting with built-in numbers [17].

▶ **Definition 24** (Termination Graphs). Let $\mathcal{R}$ be an `int`-based TRS. The nodes of the $\mathcal{R}$-*termination graph* $\mathsf{TG}(\mathcal{R})$ are the `int`-based rewrite rules from $\mathcal{R}$ and there is an arc from $l_1 \to r_1[\![\varphi_1]\!]$ to $l_2 \to r_2[\![\varphi_2]\!]$ iff $l_1 \to r_1[\![\varphi_1]\!],\ l_2 \to r_2[\![\varphi_2]\!]$ is an $\mathcal{R}$-chain.

A set $\mathcal{R}' \subseteq \mathcal{R}$ of `int`-based rewrite rules is a *strongly connected subgraph* of $\mathsf{TG}(\mathcal{R})$ iff for all `int`-based rewrite rules $l_1 \to r_1[\![\varphi_1]\!]$ and $l_2 \to r_2[\![\varphi_2]\!]$ from $\mathcal{R}'$ there exists a path from $l_1 \to r_1[\![\varphi_1]\!]$ to $l_2 \to r_2[\![\varphi_2]\!]$ that only traverses `int`-based rewrite rules from $\mathcal{R}'$. A strongly connected subgraph is a *strongly connected component* (SCC) if it is not a proper subset of any other strongly connected subgraph. Now, every infinite $\mathcal{R}$-chain contains an infinite tail that stays within an SCC of $\mathsf{TG}(\mathcal{R})$, and it is thus sufficient to prove the absence of infinite chains for each SCC separately.

[6] Here, it can be assumed that negated atomic constraints of the form $\neg(s \simeq t)$ are replaced by $s > t \vee t > s$.

▶ **Theorem 25** (Processor Based on Termination Graphs)**.** *The termination processor with* $\mathsf{Proc}(\mathcal{R}) = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$*, where* $\mathcal{R}_1, \ldots, \mathcal{R}_n$ *are the non-trivial*[7] *SCCs of* $\mathsf{TG}(\mathcal{R})$*, is sound.*[8]

Proof. After a finite number of `int`-based rewrite rules in the beginning, any infinite $\mathcal{R}$-chain only contains `int`-based rewrite rules from some non-trivial SCC. Hence, every infinite $\mathcal{R}$-chain gives rise to an infinite $\mathcal{R}_i$-chain for some $1 \leq i \leq n$ and Proc is thus sound. ◀

It is in general unclear whether $\mathsf{TG}(\mathcal{R})$ is computable. The following procedure can be used to approximate $\mathsf{TG}(\mathcal{R})$, where it is assumed that the processor from Section 7 that splits an `int`-based rewrite rule into several `int`-based rewrite rules according to the DNF of the `int`-constraint has already been applied. Then, in order to determine whether there is an arc from $l_1 \to r_1[\![\varphi_1]\!]$ to $l_2 \to r_2[\![\varphi_2]\!]$ if $r_1 = f(e_1, \ldots, e_n)$ and $l_2 = f(x_1, \ldots, x_n)$, it is determined whether the `int`-constraint $\mathsf{drop}(\varphi_1 \wedge \varphi_2\sigma)$ is `int`-satisfiable, where $\sigma = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ and drop drops all (negated) atomic `int`-constraints that contain "$*$" from the conjunction $\varphi_1 \wedge \varphi_2\sigma$. Alternatively, sound but incomplete methods can be used in order to determine whether $\varphi_1 \wedge \varphi_2\sigma$ is satisfiable in the integers.

▶ **Example 26.** Continuing Example 21, the `int`-based TRS generated there gives rise to the following termination graph:

$$\begin{array}{ccc} & \nearrow & (4) \to (6) \\ (1) \to (2) & & \qquad \nwarrow \\ & \searrow & (3) \leftrightarrow (5) \end{array}$$

There is an arc from (3) to (5) since the `int`-constraint $\mathsf{drop}(v_{y.0} > \mathsf{0}) = (v_{y.0} > \mathsf{0})$ is `int`-satisfiable. Similar reasoning is applied in order to determine the existence of the remaining arcs. The termination graph contains one non-trivial SCC and the termination processor of Theorem 25 returns the `int`-based TRS {(3), (5)}. ◀

9 `int`-Polynomial Interpretations

In this section, well-founded relations on terms are considered and it is shown that `int`-based rewrite rules may be deleted from an `int`-based TRS if their left-hand side is strictly "bigger" than their right-hand side. A promising way for the generation of such well-founded relations is the use of polynomial interpretations [27]. In contrast to [27], `int`-based TRSs allow for the use of polynomial interpretations with coefficients from $\mathbb{Z}$. In the term rewriting literature, polynomial interpretations with coefficients from $\mathbb{Z}$ have been utilized in [23, 22, 17, 18, 20].

An *`int`-polynomial interpretation* maps each symbol $f \in \mathcal{F}$ to a polynomial over $\mathbb{Z}$ such that $\mathcal{P}ol(f) \in \mathbb{Z}[x_1, \ldots, x_n]$ if f has n arguments. The mapping $\mathcal{P}ol$ is then extended to terms from $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathtt{int}}, \mathcal{V})$ by letting $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = \mathcal{P}ol(f)(t_1, \ldots, t_n)$ for all $f \in \mathcal{F}$. Now `int`-polynomial interpretations generate relations on terms as follows. Here, the requirement $[s]_{\mathcal{P}ol} \geq 0$ is needed for well-foundedness of $\succ_{\mathcal{P}ol}$.

▶ **Definition 27** ($\succ_{\mathcal{P}ol}$ and $\succsim_{\mathcal{P}ol}$)**.** For an `int`-polynomial interpretation $\mathcal{P}ol$, terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathtt{int}}, \mathcal{V})$, and an `int`-constraint φ, let:

---

[7] An SCC is trivial if it has size 1 and there is no arc from its element to itself.

[8] Notice, in particular, that $\mathsf{Proc}(\emptyset) = \emptyset$. Also, notice that `int`-based rewrite rules with unsatisfiable constraints are not connected to any `int`-based rewrite rule and do thus not occur in any non-trivial SCC.

- $s \succ^{\varphi}_{\mathcal{P}ol} t$ iff $\varphi \Rightarrow [s]_{\mathcal{P}ol} \geq 0$ and $\varphi \Rightarrow [s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ are `int`-valid.
- $s \gtrsim^{\varphi}_{\mathcal{P}ol} t$ iff $\varphi \Rightarrow [s]_{\mathcal{P}ol} \geq [t]_{\mathcal{P}ol}$ is `int`-valid.

Notice that $\succ_{\mathcal{P}ol}$ and $\gtrsim_{\mathcal{P}ol}$ are in general undecidable since $[s]_{\mathcal{P}ol}$ and $[t]_{\mathcal{P}ol}$ may be non-linear. A heuristic for the automatic generation of `int`-polynomial interpretations that ensure $s \succ^{\varphi}_{\mathcal{P}ol} t$ or $s \gtrsim^{\varphi}_{\mathcal{P}ol} t$ is presented in Section 11.1.

Using `int`-polynomial interpretations, `int`-based rewrite rules $l \to r[\![\varphi]\!]$ with $l \succ^{\varphi}_{\mathcal{P}ol} r$ can be removed from an `int`-based TRS if all remaining `int`-based rewrite rules $l' \to r'[\![\varphi']\!]$ satisfy $l' \gtrsim^{\varphi'}_{\mathcal{P}ol} r'$.

▶ **Theorem 28** (Processor Based on `int`-Polynomial Interpretations)**.** *Let $\mathcal{P}ol$ be an `int`-polynomial interpretation and let* Proc *be the termination processor with* $\mathsf{Proc}(\mathcal{R}) =$

- $\{\mathcal{R} - \mathcal{R}'\}$*, if* $\mathcal{R}' \subseteq \mathcal{R}$ *such that*
  - $l \succ^{\varphi}_{\mathcal{P}ol} r$ *for all* $l \to r[\![\varphi]\!] \in \mathcal{R}'$*, and*
  - $l \gtrsim^{\varphi}_{\mathcal{P}ol} r$ *for all* $l \to r[\![\varphi]\!] \in \mathcal{R} - \mathcal{R}'$*.*
- $\{\mathcal{R}\}$*, otherwise.*

Then Proc *is sound.*

Proof. This is a special case of Theorem 34 from Section 9.1. ◀

▶ **Example 29.** Recall the `int`-based TRS {(3), (5)} from Example 26. For this `int`-based TRS, an `int`-polynomial interpretation such that $\mathcal{P}ol(\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}) = x_3$ and $\mathcal{P}ol(\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}) = x_3 - 1$ can be used (this polynomial interpretation can be generated automatically by the heuristic presented in Section 11.1).

For (3), $\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \succ^{v_{y.0}>0}_{\mathcal{P}ol} \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0})$ since $v_{y.0} > 0 \Rightarrow v_{y.0} \geq 0$ and $v_{y.0} > 0 \Rightarrow v_{y.0} > v_{y.0} - 1$ are `int`-valid. For (5), $\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \gtrsim_{\mathcal{P}ol} \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0} - 1, v_{r.0} * v_x)$ since $v_{y.0} - 1 \geq v_{y.0} - 1$ is trivially `int`-valid. ◀

9.1 `int`-Reduction Pairs

For the proof of Theorem 28, it is convenient to give an abstract characterization of well-founded relations on terms that may be used for termination proofs of `int`-based TRSs. The relations that can be used should not make a distinction between terms that are `int`-equivalent, i.e., they need to satisfy the following requirement.

▶ **Definition 30** (`int`-Compatible Relations)**.** A relation $\bowtie$ on $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathtt{int}}, \mathcal{V})$ is `int`-*compatible* iff $s \bowtie t$ implies $s' \bowtie t'$ for all s', t' such that $s \simeq s'$ and $t \simeq t'$ are `int`-valid.[9]

The notion of `int`-reduction pairs is motivated by the notion of reduction pairs [25]. An `int`-reduction pair consists of two relations $\gtrsim$ and $\succ$, where it is *not* required that $\succ$ is the strict part of $\gtrsim$.

▶ **Definition 31** (`int`-Reduction Pairs)**.** An `int`-*reduction pair* $(\gtrsim, \succ)$ consists of two relations on $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathtt{int}}, \mathcal{V})$ such that $\succ$ is well-founded, $\gtrsim$ and $\succ$ are `int`-compatible, and $\succ$ is compatible with $\gtrsim$, i.e., $\gtrsim \circ \succ \subseteq \succ$ or $\succ \circ \gtrsim \subseteq \succ$.

Relations on $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathtt{int}}, \mathcal{V})$ are extended to operate on terms with constraints as follows. Intuitively, it suffices to consider all instantiations that make the constraint `int`-valid.

[9] Strictly speaking, $s \simeq s'$ needs to be valid in the theory of integers and uninterpreted functions since s and s' have a function symbol from $\mathcal{F}$ as their root symbol. But since s and s' do not contain nested function symbols this is equivalent to the validity of pairwise equality of arguments.

▶ **Definition 32** (Relations on Constrained Terms). Let $\bowtie$ be a relation on $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$. Let s, t be terms and let φ be an `int`-constraint. Then $s \bowtie^{\varphi} t$ iff $s\sigma \bowtie t\sigma$ for all $\mathcal{F}_{\mathbb{Z}}$-based substitutions σ such that $\varphi\sigma$ is `int`-valid.

▶ **Example 33.** Consider the relation $>_{\texttt{int}}$ on `int`-terms over $\mathcal{V}$, defined by $s >_{\texttt{int}} t$ iff $s > t$ is `int`-valid. Then $x + y \not>_{\texttt{int}} x$ since $x + y > x$ is not `int`-valid. On the other hand, $x + y >^{y>0}_{\texttt{int}} x$. ◀

Using `int`-reduction pairs, `int`-based rewrite rules $l \to r[\![\varphi]\!]$ such that $l \succ^{\varphi} r$ can be removed from an `int`-based TRS if all remaining `int`-based rewrite rules $l' \to r'[\![\varphi']\!]$ satisfy $l' \succsim^{\varphi'} r'$. This generalizes Theorem 28.

▶ **Theorem 34** (Processor Based on `int`-Reduction Pairs). *Let $(\succsim, \succ)$ be an `int`-reduction pair and let* Proc *be the termination processor with* $\mathsf{Proc}(\mathcal{R}) =$
- $\{\mathcal{R} - \mathcal{R}'\}$, *if $\mathcal{R}' \subseteq \mathcal{R}$ such that*
  - $l \succ^{\varphi} r$ *for all* $l \to r[\![\varphi]\!] \in \mathcal{R}'$, *and*
  - $l \succsim^{\varphi} r$ *for all* $l \to r[\![\varphi]\!] \in \mathcal{R} - \mathcal{R}'$.
- $\{\mathcal{R}\}$, *otherwise.*

Then Proc *is sound.*

Proof. In the second case soundness is obvious. Otherwise, it needs to be shown that every infinite $\mathcal{R}$-chain contains only finitely many `int`-based rewrite rules from $\mathcal{R}'$. Thus, assume that $l_1 \to r_1[\![\varphi_1]\!], l_2 \to r_2[\![\varphi_2]\!], \ldots$ is an infinite $\mathcal{R}$-chain using the substitution σ. Hence, $\mathsf{norm}(r_i\sigma) = l_{i+1}\sigma$ and $\varphi_i\sigma$ is `int`-valid for all $i \geq 1$.

Since $l_i \succsim^{\varphi_i} r_i$ for all $l_i \to r_i[\![\varphi_i]\!] \in \mathcal{R} - \mathcal{R}'$ and $l_i \succ^{\varphi_i} r_i$ for all $l_i \to r_i[\![\varphi_i]\!] \in \mathcal{R}'$, this implies $l_i\sigma \succsim r_i\sigma$ or $l_i\sigma \succ r_i\sigma$ for all $i \geq 1$. Furthermore, notice that $r_i\sigma \simeq \mathsf{norm}(r_i\sigma)$ is `int`-valid. Hence, using the `int`-compatibility of $\succsim$ and $\succ$, the infinite $\mathcal{R}$-chain gives rise to

$$l_1\sigma \bowtie_1 \mathsf{norm}(r_1\sigma) = l_2\sigma \bowtie_2 \mathsf{norm}(r_2\sigma) = l_3\sigma \ldots$$

where $\bowtie_i \in \{\succsim, \succ\}$. Therefore,

$$l_1\sigma \bowtie_1 l_2\sigma \bowtie_2 l_3\sigma \ldots$$

If the infinite $\mathcal{R}$-chain contains infinitely many `int`-based rewrite rules from $\mathcal{R}'$, then $\bowtie_i = \succ$ for infinitely many i. In this case, the compatibility of $\succ$ with $\succsim$ produces an infinite $\succ$-chain, contradicting the well-foundedness of $\succ$. Thus, only finitely many `int`-based rewrite rules from $\mathcal{R}'$ occur in the infinite $\mathcal{R}$-chain and there exists an infinite $(\mathcal{R} - \mathcal{R}')$-chain as well. ◀

For the proof of Theorem 28, it thus suffices to show that `int`-polynomial interpretations yield `int`-reduction pairs. For this, $s \succ_{\mathcal{P}ol} t$ iff $s \succ^{\top}_{\mathcal{P}ol} t$, and similarly for $\succsim_{\mathcal{P}ol}$.

▶ **Theorem 35.** *Let $\mathcal{P}ol$ be an `int`-polynomial interpretation. Then $(\succsim_{\mathcal{P}ol}, \succ_{\mathcal{P}ol})$ is an `int`-reduction pair.*

Proof. It needs to be shown that $\succ_{\mathcal{P}ol}$ is well-founded, that $\succsim_{\mathcal{P}ol}$ and $\succ_{\mathcal{P}ol}$ are `int`-compatible, and that $\succ_{\mathcal{P}ol}$ is compatible with $\succsim_{\mathcal{P}ol}$.

$\succ_{\mathcal{P}ol}$ **is well-founded:** For a contradiction, assume that $s_1 \succ_{\mathcal{P}ol} s_2 \succ_{\mathcal{P}ol} \ldots$ is an infinite descending sequence of terms. This means that $[s_i]_{\mathcal{P}ol} > [s_{i+1}]_{\mathcal{P}ol}$ and $[s_i]_{\mathcal{P}ol} \geq 0$ for all $i \geq 1$ and all instantiations of the variables by integers. By fixing an arbitrary instantiation, integers $d_1, d_2, \ldots \geq 0$ are obtained such that $d_1 > d_2 > \ldots$, which is clearly impossible.

$\gtrsim_{\mathcal{P}ol}$ and $\succ_{\mathcal{P}ol}$ are `int`-compatible. Let $s \gtrsim_{\mathcal{P}ol} t$ and assume that $s' \simeq s$ and $t \simeq t'$ are `int`-valid. Then $s = f(s^*), s' = f(s'^*), t = g(t^*)$, and $t' = g(t'^*)$, where $s^* \simeq s'^*$ and $t^* \simeq t'^*$ are `int`-valid. Clearly, $s \simeq s'$ implies that s and s' are equal for all instantiations of the variables by integers. Thus, $[s']_{\mathcal{P}ol} = \mathcal{P}ol(f)(s'_1, \ldots, s'_n) = \mathcal{P}ol(f)(s_1, \ldots, s_n) \geq \mathcal{P}ol(g)(t_1, \ldots, t_m) = \mathcal{P}ol(g)(t'_1, \ldots, t'_n) = [t']_{\mathcal{P}ol}$ for all instantiations of the variables by integers since $s \gtrsim_{\mathcal{P}ol} t$. But then $s' \gtrsim_{\mathcal{P}ol} t'$. `int`-compatibility of $\succ_{\mathcal{P}ol}$ is shown the same way.

$\succ_{\mathcal{P}ol}$ is compatible with $\gtrsim_{\mathcal{P}ol}$: For showing that $\succ_{\mathcal{P}ol} \circ \gtrsim_{\mathcal{P}ol} \subseteq \succ_{\mathcal{P}ol}$, let $s \succ_{\mathcal{P}ol} t \gtrsim_{\mathcal{P}ol} u$, i.e., $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol} \geq [u]_{\mathcal{P}ol}$ and $[s]_{\mathcal{P}ol} \geq 0$ for all instantiations of the variables by integers. But then $[s]_{\mathcal{P}ol} > [u]_{\mathcal{P}ol}$ for all instantiations of the variables as well and therefore $s \succ_{\mathcal{P}ol} u$.

Also, $\gtrsim_{\mathcal{P}ol} \circ \succ_{\mathcal{P}ol} \subseteq \succ_{\mathcal{P}ol}$. To see this, let $s \gtrsim_{\mathcal{P}ol} t \succ_{\mathcal{P}ol} u$. Then $[s]_{\mathcal{P}ol} \geq [t]_{\mathcal{P}ol} > [u]_{\mathcal{P}ol}$ and $[t]_{\mathcal{P}ol} \geq 0$ for all instantiations of the variables by integers. But then also $[s]_{\mathcal{P}ol} \geq 0$ and $[s]_{\mathcal{P}ol} > [u]_{\mathcal{P}ol}$ for all instantiations of the variables, i.e., $s \succ_{\mathcal{P}ol} u$. ◀

10 Chaining

It is possible to replace an `int`-based rewrite rule $l \to r[\![\varphi]\!]$ by a set of new `int`-based rewrite rules that are formed by chaining $l \to r[\![\varphi]\!]$ to the `int`-based rewrite rules that may follow it in an infinite chain[10]. This way, further information about the possible substitutions used for a chain can be obtained. Chaining of `int`-based rewrite rules corresponds to executing bigger parts of the imperative program at once, spanning several control points. This is of course similar to the idea of combining `int`-based rewrite rules as used in Sections 3 and 4.

▶ **Example 36.** Consider the following C program and its LLVM-IR program:

```
void f(int x) {
    while (x != 0) {
        if (x > 0) {
            x = −x + 1;
        } else {
            x = −x − 1;
        }
    }
}
```

```
define void @f(i32 %x) {
entry:
  br label %bb3

bb3:
  %x.0 = phi i32 [ %x, %entry ], [ %2, %bb1 ], [ %3, %bb2 ]
  %0 = icmp ne i32 %x.0, 0
  br i1 %0, label %bb, label %return

bb:
  %1 = icmp sgt i32 %x.0, 0
  br i1 %1, label %bb1, label %bb2
```

[10] Dually, it is possible to consider the `int`-based rewrite rules that may *precede* it.

```
bb1:
  %2 = sub i32 1, %x.0
  br label %bb3

bb2:
  %3 = sub i32 -1, %x.0
  br label %bb3

return:
  ret void
}
```

Then, the following `int`-based TRS is generated from it:

$$\mathsf{state}_{\mathsf{start}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_{x.0}) \tag{7}$$
$$\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_x) \tag{8}$$
$$\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_{x.0}) \quad [\![v_{x.0} < \mathsf{0}]\!] \tag{9}$$
$$\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_{x.0}) \quad [\![v_{x.0} > \mathsf{0}]\!] \tag{10}$$
$$\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_{x.0}) \quad [\![v_{x.0} \simeq \mathsf{0}]\!] \tag{11}$$
$$\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_{x.0}) \quad [\![v_{x.0} > \mathsf{0}]\!] \tag{12}$$
$$\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, v_{x.0}) \quad [\![v_{x.0} \le \mathsf{0}]\!] \tag{13}$$
$$\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, -v_{x.0} + 1) \tag{14}$$
$$\mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, -v_{x.0} - 1) \tag{15}$$
$$\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{stop}}(v_x, v_{x.0}) \tag{16}$$

Using the termination graph, the `int`-based TRS {(7)–(16)} is transformed into the `int`-based TRS {(9), (10), (12), (13), (14), (15)}. This `int`-based TRS cannot be handled by the techniques presented so far. Notice that in any chain, each occurrence of the `int`-based rewrite rule (9) is followed by an occurrence of the `int`-based rewrite rule (12) or an occurrence of the `int`-based rewrite rule (13). Thus, (9) may be replaced by new `int`-based rewrite rules that simulate an application of (9) followed by an application of (12) or (13), respectively. These new `int`-based rewrite rules are

$$\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_{x.0})[\![v_{x.0} < \mathsf{0} \wedge v_{x.0} > \mathsf{0}]\!] \tag{9.12}$$
$$\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, v_{x.0})[\![v_{x.0} < \mathsf{0} \wedge v_{x.0} \le \mathsf{0}]\!] \tag{9.13}$$

The `int`-based TRS {(9.12), (9.13), (10), (12), (13), (14), 15)} is transformed into the `int`-based TRS {(9.13), (10), (12), (14), (15)} using the termination graph. Then, the `int`-based rewrite rule (10) can be combined with all `int`-based rewrite rules that may follow it in chains. This produces

$$\mathsf{state}_{\mathsf{bb3}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_{x.0})[\![v_{x.0} > \mathsf{0}]\!] \tag{10.12}$$

and the `int`-based TRS {(9.13), (10.12), (12), (14), (15)} which is transformed into the `int`-based TRS {(9.13), (10.12), (14), (15)} using the termination graph. Combining (14) with all rules that may follow it yields

$$\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, -v_{x.0} + 1)[\![-v_{x.0} + 1 < \mathsf{0} \wedge -v_{x.0} + 1 \le \mathsf{0}]\!] \tag{14.9.13}$$
$$\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, -v_{x.0} + 1)[\![-v_{x.0} + 1 > \mathsf{0}]\!] \tag{14.10.12}$$

and the int-based TRS {(9.13), (10.12), (14.9.13), (14.10.12), (15)} which contains the non-trivial SCC {(9.13), (10.12), (14.9.13), (15)}. Next, considering the rules that may follow (15), the new rules

$$\mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, -v_{x.0} - 1) \quad [\![-v_{x.0} - 1 < \mathsf{0} \land -v_{x.0} - 1 \leq \mathsf{0}]\!] \quad (15.9.13)$$

$$\mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}(v_x, v_{x.0}) \to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, -v_{x.0} - 1) \quad [\![-v_{x.0} - 1 > \mathsf{0}]\!] \quad (15.10.12)$$

give rise to the int-based TRS {(9.13), (10.12), (14.9.13), (15.9.13), (15.10.12)}. There is one non-trivial SCC {(14.9.13), (15.10.12)}. This final int-based TRS can now easily be handled using a polynomial interpretation with $\mathcal{P}ol(\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}) = x_2$ and $\mathcal{P}ol(\mathsf{state}_{\mathsf{bb2}_{\mathsf{in}}}) = -x_2$. ◀

Formally, this idea can be stated as the following termination processor. Notice that chaining of int-based rewrite rules is easily possible since the left-hand sides have the form $f(x_1, \ldots, x_n)$. Also, notice that the rule $l \to f(s_1, \ldots, s_n)[\![\varphi]\!]$ is *replaced* by the rules that are obtained by chaining.

▶ **Theorem 37** (Processor Based on Chaining). *The termination processor with* $\mathsf{Proc}(\mathcal{R} \uplus \{l \to f(s_1, \ldots, s_n)[\![\varphi]\!]\}) = \{\mathcal{R} \cup \mathcal{R}'\}$ *where* $\mathcal{R}' = \{l \to r'\mu[\![\varphi \land \varphi'\mu]\!] \mid f(x_1, \ldots, x_n) \to r'[\![\varphi']\!] \in \mathcal{R} \cup \{l \to f(s_1, \ldots, s_n)[\![\varphi]\!]\}, \mu = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}\}$ *is sound.*

Proof. It needs to be shown that every occurrence of (a variable-renamed version of) $l \to r[\![\varphi]\!]$ and the int-based rewrite rule following it in an infinite chain can be replaced by some int-based rewrite rule from $\mathcal{R}'$. Thus, assume some infinite chain contains $\ldots, l \to r[\![\varphi]\!], l' \to r'[\![\varphi']\!], v \to w[\![\psi]\!], \ldots$. Let the infinite chain be based on the substitution σ, i.e., $\mathsf{norm}(r\sigma) = l'\sigma$, $\mathsf{norm}(r'\sigma) = v\sigma$, and $\varphi\sigma$ and $\varphi'\sigma$ are int-valid. Now $\mathsf{norm}(r\sigma) = \mathsf{norm}(l'\mu\sigma) = l'\mathsf{norm}(\mu\sigma)$ since $r = l'\mu$ for the substitution μ used in the definition of $\mathcal{R}'$. Therefore $l'\mathsf{norm}(\mu\sigma) = l'\sigma$ and thus $x_i\mathsf{norm}(\mu\sigma) = x_i\sigma$ for all variables x occurring in l'. This implies that $\varphi'\mu\sigma$ is int-valid since $\varphi'\sigma$ is int-valid. Thus, $l \to r[\![\varphi]\!], l' \to r'[\![\varphi']\!]$ can be replaced by $l \to r'\mu[\![\varphi \land \varphi'\mu]\!]$ since $\varphi\sigma \land \varphi'\mu\sigma$ is int-valid and $\mathsf{norm}(r'\mu\sigma) = \mathsf{norm}(r'\mathsf{norm}(\mu\sigma)) = \mathsf{norm}(r'\sigma) = v\sigma$. ◀

11 Implementation

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 [18], KITTeL has been written in OCaml and consists of about 2400 lines of code. The input to KITTeL is a Simple program or an int-based TRS. The translation from LLVM-IR programs into int-based TRSs has been implemented in the separate tool llvm2kittel using about 3800 lines of C++ code.

The first decision that has to be made for the implementation of KITTeL is the order in which the termination processors from Sections 7–10 are applied. The order employed by KITTeL is given in Figure 3.

Here, DNF is the termination processor of Theorem 23 that splits an int-based rewrite rule into several int-based rewrite rules according to the DNF of the rule's int-constraint. SCC is the termination processor of Theorem 25 that returns the non-trivial SCCs of the termination graph, polo is the termination processor of Theorem 28 using int-polynomial interpretations that removes int-based rewrite rules which are decreasing w.r.t. $\succ_{\mathcal{P}ol}$, and chain is the termination processor of Theorem 37 that combines int-based rewrite rules.

SCC approximates the termination graph using a decision procedure for int-satisfiability. More precisely, KITTeL uses the SMT-solver Yices [16] for this. Then, the standard graph

```
R := DNF(R)
todo := SCC(R)
while todo ≠ ∅ do
  P := pick-and-remove(todo)
  P′ := polo(P)
  if P = P′ then
    P′ := chain(P)
    if P = P′ then
      return "Failure"
    end if
  end if
  todo := todo ∪ SCC(P′)
end while
return "Termination shown"
```

Figure 3 Main loop of `KITTeL`.

algorithm as implemented in the library ocamlgraph [9] is used to compute the non-trivial SCCs. The most complex part of the implementation is the function polo for the automatic generation of `int`-polynomial interpretations.

11.1 Automatic Generation of `int`-Polynomial Interpretations

For the automatic generation, a linear[11] *parametric* `int`-polynomial interpretation is used, i.e., an interpretation where the coefficients of the polynomials are not integers but parameters that have to be determined. Thus, $\mathcal{P}ol(\mathsf{state}_i) = a_{i,1}x_1 + \ldots + a_{i,n}x_n + c_i$ for each function symbol state_i, where the $a_{i,j}$ and c_i are parameters.

Recall that the termination processor of Theorem 28 operating on an `int`-based TRS $\mathcal{R}$ aims at generating an `int`-polynomial interpretation $\mathcal{P}ol$ with

- $l \succ_{\mathcal{P}ol}^{\varphi} r$ for all $l \to r⟦\varphi⟧ \in \mathcal{R}'$ for some non-empty $\mathcal{R}' \subseteq \mathcal{R}$ and
- $l \gtrsim_{\mathcal{P}ol}^{\varphi} r$ for all $l \to r⟦\varphi⟧ \in \mathcal{R} - \mathcal{R}'$.

As shown in Section 9, it suffices to show that

- $\varphi \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} > 0$ and $\varphi \Rightarrow [l]_{\mathcal{P}ol} \geq 0$ are `int`-valid for all $l \to r⟦\varphi⟧ \in \mathcal{R}'$ for some non-empty $\mathcal{R}' \subseteq \mathcal{R}$ and
- $\varphi \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} \geq 0$ is `int`-valid for all $l \to r⟦\varphi⟧ \in \mathcal{R} - \mathcal{R}'$.

Notice that $[l]_{\mathcal{P}ol}$ and $[l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol}$ are (possibly non-linear) polynomials whose coefficients are linear polynomials over the parameters (so-called *polynomials with linear coefficients*). For instance, if $[l] = \mathsf{state}(x, x+x*y)$ and $\mathcal{P}ol(\mathsf{state}) = ax_1+bx_2+c$, then $[l]_{\mathcal{P}ol} = (a+b)x+bxy+c$.

In order to determine the parameters such that $\varphi \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} \geq 0$ is `int`-valid for all $l \to r⟦\varphi⟧ \in \mathcal{R}$, sufficient conditions on the parameters are derived and it is checked whether these conditions are satisfiable. Since the conditions on the parameters will be linear, it is decidable whether they are satisfiable. Furthermore, SMT-solvers such as Yices can compute a satisfying assignment which immediately gives rise to a polynomial interpretation. The derivation of the conditions is done independently for the `int`-based rewrite rules, but the

[11] The method presented in this section can also be applied in order to generate non-linear `int`-polynomial interpretations.

check for satisfiability of the conditions considers all `int`-based rewrite rules since they need to be oriented using the same `int`-polynomial interpretation.

For a single `int`-based rewrite rule $l \to r[\![\varphi]\!]$, the conditions on the parameters are obtained as follows, where $p = [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol}$:

1. φ is transformed into a conjunction of atomic `int`-constraints of the form $\sum_{i=1}^{n} a_i x_i + c \geq 0$ where $a_1, \ldots, a_n, c \in \mathbb{Z}$.
2. The `int`-constraints from step 1 are used to derive upper and/or lower bounds on the variables in p.
3. The bounds from step 2 are used to derive conditions on the parameters.

Here, the first two steps are identical to [18], but the third step is more complex than in [18].

Step 1: Transformation of φ. For linear `int`-terms s and t, $s \simeq t$ is transformed into $s - t \geq 0 \wedge t - s \geq 0$, $s \geq t$ is transformed into $s - t \geq 0$, and $s > t$ is transformed into $s - t - 1 \geq 0$. Non-linear atoms are discarded.

Step 2: Deriving upper and/or lower bounds. The `int`-constraints obtained after step 1 might already contain upper and/or lower bounds on the variables, where a lower bound has the form $x + c \geq 0$ and an upper bound has the from $-x + c \geq 0$ for some $c \in \mathbb{Z}$. Otherwise, it might be possible to obtain such bounds as follows.

An atomic constraint of the form $ax + c \geq 0$ with $a \neq 0, 1, -1$ that contains only one variable gives a bound on that variable that can be obtained by dividing by $|a|$ and rounding. For example, the `int`-constraint $2x + 3 \geq 0$ is transformed into $x + 1 \geq 0$, and $-3x - 2 \geq 0$ is transformed into $-x - 1 \geq 0$.

An atomic `int`-constraint with more than one variable can be used to express a variable x occurring with coefficient 1 in terms of the other variables and a fresh slack variable w with $w \geq 0$. This allows to eliminate x from the polynomial p and at the same time gives the lower bound 0 on the slack variable w. For example, $x - 2y \geq 0$ can be used to eliminate the variable x by replacing it with $2y + w$. Similar reasoning applies if the variable x occurs with coefficient -1.

These ideas are formalized in the transformation rules from Figure 4 that operate on triples $\langle C_1, C_2, q \rangle$ where C_1 and C_2 are sets of atomic `int`-constraints and q is a polynomial with linear coefficients. Here, C_1 only contains `int`-constraints of the form $\pm x_i + c \geq 0$ giving upper and/or lower bounds on the variable x_i and C_2 contains arbitrary atomic `int`-constraints. The initial triple is $\langle \emptyset, C, p \rangle$.

$$\textsf{Strengthen}\quad \frac{C_1,\ C_2 \uplus \{a_i x_i + c \geq 0\},\ q}{C_1 \cup \left\{ \frac{a_i}{|a_i|} x_i + \lfloor \frac{c}{|a_i|} \rfloor \geq 0 \right\},\ C_2,\ q} \quad \text{if } a_i \neq 0$$

$$\textsf{Express}^{+}\quad \frac{C_1,\ C_2 \uplus \{\sum_{i=1}^{n} a_i x_i + c \geq 0\},\ q}{C_1 \cup \{w \geq 0\},\ C_2\sigma,\ q\sigma} \quad \begin{array}{l}\text{if } a_j = 1 \text{ and } \sigma \text{ is the substitution} \\ \{x_j \mapsto -\sum_{i \neq j} a_i x_i - c + w\} \\ \text{for a fresh slack variable } w\end{array}$$

$$\textsf{Express}^{-}\quad \frac{C_1,\ C_2 \uplus \{\sum_{i=1}^{n} a_i x_i + c \geq 0\},\ q}{C_1 \cup \{w \geq 0\},\ C_2\sigma,\ q\sigma} \quad \begin{array}{l}\text{if } a_j = -1 \text{ and } \sigma \text{ is the substitution} \\ \{x_j \mapsto \sum_{i \neq j} a_i x_i + c - w\} \\ \text{for a fresh slack variable } w\end{array}$$

Figure 4 Transformation rules to derive upper and/or lower bounds.

Step 3: Deriving conditions on the parameters. After finishing step 2, a final triple $\langle C_1, C_2, q\rangle$ is obtained. If C_1 contains more than one bound on a variable x_i, then it suffices to consider the maximal lower bound and the minimal upper bound. The bounds in C_1 are used in combination with absolute positiveness[12] [24] in order to obtain conditions on the parameters that make $q = \sum_{i=1}^{k} p_i x_1^{i_1} \cdots x_n^{i_n} + p_0$ non-negative for all instantiations of the variables that satisfy $C_1 \cup C_2$.

This is done similarly to [18] but the method is more complex since q may be non-linear. If q contains a monomial $p_l x_1^{l_1} \cdots x_n^{l_n}$ such that at least one of the x_i occurring with positive odd degree[13] does not have an upper or lower bound, then the absolute positiveness test requires $p_l \simeq 0$ as a condition on the parameters.

Otherwise, for simplicity of presentation, assume that all variables occur with positive degree, that $x_1, \ldots, x_o$ occur with odd degree, that $x_1, \ldots, x_{o'}$ have upper bounds $-x_i + c_i \geq 0$, that $x_{o'+1}, \ldots, x_o$ have lower bounds $x_j + c_j \geq 0$, and that $x_{o+1}, \ldots, x_n$ have even degree. Then, notice that $m := p_l x_1^{l_1} \cdots x_n^{l_n}$ can also be written as $r + (m - r)$ where $r := (-1)^{o'} p_l (-x_1 + c_1)^{l_1} \cdots (-x_{o'} + c_{o'})^{l_{o'}} (x_{o'+1} + c_{o'+1})^{l_{o'+1}} \cdots (x_o + c_o)^{l_o} x_{o+1}^{l_{o+1}} \cdots x_n^{l_n}$. The absolute positiveness test then requires $(-1)^{o'} p_l \geq 0$ as a condition on the parameters.[14]

Summarizing this method, the algorithm from Figure 5 is used in order to obtain conditions D on the parameters. Here, "$\oplus$" means that the monomials of $m - r$ (if non-zero) are added to monomials with the same variable degrees or inserted into `todo`, respectively. Furthermore, $\mathsf{sign}(C)$ is 1 if C is of the form $x_i + c \geq 0$ and -1 if C is of the form $-x_i + c \geq 0$.

Automatically finding strictly decreasing rules. For the termination processor of Theorem 28, it also has to be ensured that $\mathcal{R}'$ is non-empty, i.e., that at least one `int`-based rewrite rule is decreasing w.r.t. $\succ_{\mathcal{P}ol}$. Let $l \to r[\![\varphi]\!]$ be an `int`-based rewrite rule that should satisfy $l \succ_{\mathcal{P}ol}^{\varphi} r$. Then, $\varphi \Rightarrow [l]_{\mathcal{P}ol} \geq 0$ gives rise to conditions D_1 on the parameters as above. The second condition, i.e., $\varphi \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} > 0$, gives rise to conditions D_2 just as above, with the only difference that the constant monomial (i.e., the monomial where all variables have degree 0) now needs to be strictly bigger than 0.

Given a set of rules $\{l_1 \to r_1[\![\varphi_1]\!], \ldots, l_n \to r_n[\![\varphi_n]\!]\}$, the final constraint on the parameters is then $\bigwedge_{i=1}^{n} D^i \wedge \bigvee_{i=1}^{n} (D_1^i \wedge D_2^i)$ where the D^i are obtained from $\varphi_i \Rightarrow [l_i]_{\mathcal{P}ol} - [r_i]_{\mathcal{P}ol} \geq 0$, the D_1^i are obtained from $\varphi_i \Rightarrow [l_i]_{\mathcal{P}ol} \geq 0$, and the D_2^i are obtained from $\varphi C_i \Rightarrow [l_i]_{\mathcal{P}ol} - [r_i]_{\mathcal{P}ol} > 0$. Notice that this constraint is linear and can thus be given to an SMT solver for linear integer arithmetic which, in case the constraint is satisfiable, can also produce a satisfying assignment. This satisfying assignment then gives rise to an `int`-polynomial interpretation.

▶ **Example 38.** The method is illustrated on the `int`-based TRS consisting of the `int`-based rewrite rule $\mathsf{state}(x, y, z) \to \mathsf{state}(x, y * x, z)[\![y < z \wedge y > \mathsf{0} \wedge x > \mathsf{1}]\!]$.

For this, a parametric `int`-polynomial interpretation with $\mathcal{P}ol(\mathsf{state}) = ax_1 + bx_2 + cx_3 + d$ is used, where a, b, c, d are parameters that need to be determined. Thus, the goal is to instantiate the parameters in such a way that $\mathsf{state}(x, y, z) \succ_{\mathcal{P}ol}^{y<z\wedge y>\mathsf{0}\wedge x>\mathsf{1}} \mathsf{state}(x, y * x, z)$, i.e., such that

$$y < z \wedge y > \mathsf{0} \wedge x > \mathsf{1} \quad \Rightarrow \quad [\mathsf{state}(x, y, z)]_{\mathcal{P}ol} \geq 0$$

[12] For a polynomial p in the indeterminates $x_1, \ldots, x_n$ which may only be instantiated by natural numbers, the absolute positiveness test concludes that p is non-negative for all instantiations of the indeterminates if all coefficients of p are non-negative.

[13] Notice that even powers of variables are always non-negative.

[14] This is only one possibility. It is of course also possible to consider the bounds for variables occurring with even degree, and the implementation in `KITTeL` actually supports both possibilities.

$D := \mathsf{true}$
$\texttt{todo} := \mathsf{monomials}(q)$
for $p_l x_1^{l_1} \cdots x_n^{l_n} \in \texttt{todo}$ **do**
  if one of the x_i occurring with odd degree does not have a bound in C_1 **then**
    $D := D \wedge p_l \simeq 0$
    $\texttt{todo} := \texttt{todo} - \{p_l x_1^{l_1} \cdots x_n^{l_n}\}$
  end if
end for
while $\texttt{todo} := \emptyset$ **do**
  pick monomial $m := p_l x_1^{i_1} \cdots x_n^{i_n} \in \texttt{todo}$ with maximal degree
  $\texttt{todo} := \texttt{todo} - \{m\}$
  $r := p_l$
  $o' := 0$
  for $1 \leq k \leq n$ **do**
    if i_k is odd **then**
      pick bound C of the form $\pm x_k + c \geq 0$ from C_1
      $r := r * \mathsf{sign}(C) * (\pm x_k + c)^{l_k}$
      if $\mathsf{sign}(C) = -1$ **then**
        $o' := o' + 1$
      end if
    else
      $r := r * x_k^{l_k}$
    end if
  end for
  $\texttt{todo} := \texttt{todo} \oplus \{m - r\}$
  $D := D \wedge (-1)^{o'} p_l \geq 0$
end while

Figure 5 Deriving conditions on the parameters

and

$$y < z \wedge y > 0 \wedge x > 1 \quad \Rightarrow \quad [\mathsf{state}(x, y, z)]_{\mathcal{P}ol} - [\mathsf{state}(x, y * x, z)]_{\mathcal{P}ol} > 0$$

are `int`-valid. Notice that $[\mathsf{state}(x, y, z)]_{\mathcal{P}ol} = ax + by + cz + d$ and $[\mathsf{state}(x, y * x, z)]_{\mathcal{P}ol} = ax + byx + cz + d$. Therefore, $[\mathsf{state}(x, y, z)]_{\mathcal{P}ol} - [\mathsf{state}(x, y * x, z)]_{\mathcal{P}ol} = by - byx$.

For the first formula, the constraint $y < z$ is transformed into $z - y - 1 \geq 0$ in step 1 while the other two constraints are transformed into $y - 1 \geq 0$ and $x - 2 \geq 0$, respectively. In step 2, the transformation rules from Figure 4 are applied to the triple $\langle \emptyset, \{z - y - 1 \geq 0, y - 1 \geq 0, x - 2 \geq 0\}, ax + by + cz + d \rangle$. A possible transformation sequence is as follows, where the $\mathsf{Express}^{+}$-step uses $\sigma = \{z \mapsto y + w + 1\}$.

$$\begin{array}{l} \mathsf{Strengthen}^2 \\ \mathsf{Express}^{+} \end{array} \begin{array}{c} \emptyset,\ \{z - y - 1 \geq 0, y - 1 \geq 0, x - 2 \geq 0\},\ ax + by + cz + d \\ \hline \{y - 1 \geq 0, x - 2 \geq 0\},\ \{z - y - 1 \geq 0\},\ ax + by + cz + d \\ \hline \{y - 1 \geq 0, x - 2 \geq 0, w \geq 0\},\ \emptyset,\ ax + (b + c)y + cw + c + d \end{array}$$

Step 3 gives $a \geq 0 \wedge b + c \geq 0 \wedge c \geq 0 \wedge 2a + b + 2c + d \geq 0$ as conditions on the parameters.

For the second formula from above, step 1 is as above while step 2 does not perform any "interesting" transformation, i.e., the (relevant parts of the) result of the transformation is

$C_1 = \{y-1 \geq 0, x-2 \geq 0\}$, $q = -bxy+by$. Step 3 first removes the monomial $-bxy$ and adds the monomial $r = -bxy-(-b(x-2)(y-1)) = -bxy-(-bxy+bx+2by-2b) = -bx-2by+2b$ to the set `todo`, thus obtaining `todo` $= \{-bx, -by, 2b\}$. Furthermore, $-b \geq 0$ is obtained as a condition on the parameters. The set `todo` is subsequently transformed into $\{-by\}$ and then $\{-b\}$, giving rise to the conditions $-b \geq 0$ and $-b \geq 0$. Finally, the set $\{-b\}$ requires $-b > 0$ since the constant monomial needs to be strictly bigger than 0.

The final constraint on the parameters is thus $a \geq 0 \wedge b+c \geq 0 \wedge c \geq 0 \wedge 2a+b+2c+d \geq 0 \wedge -b > 0$. This constraint is satisfiable and the satisfying assignment $a = 0, b = -1, c = 1, d = 0$ gives rise to the `int`-polynomial interpretation $\mathcal{P}ol(\mathsf{state}) = x_3 - x_2$. ◀

12 Evaluation

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 [4, 5, 6, 7, 8, 11, 12, 31, 32], the textbook [34], from the Java category of TPDB [36] and converted to C, and the `zlib` compression library. The collection of examples includes "classical" algorithms such as binary search and sorting algorithms, cyclic redundancy check and hash code algorithms, encryption algorithms, image processing algorithms, and numerical algorithms. 14 out of these 174 examples (e.g., the heapsort example from [15]) 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` [19].

The implementation has been able to show termination of all[15] examples fully automatically, on average taking less than 0.3 seconds[16] 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 LLVM-IR into a TRS, and the termination analysis of the obtained TRS. The following table contains details for some of the examples. Here, the "LOC" column gives the number of code lines in the C program, and the "RR" column gives the number of rewrite rules that are generated. The full results for all examples are provided in Appendix A.

C program	LOC	RR	Time / s	C program	LOC	RR	Time / s
`allroots`	200	77	0.861	`dijkstra`	78	58	0.693
`almabench`	390	42	0.370	`fft`	99	30	0.342
`barr-crc16`	265	45	0.398	`hash`	241	80	0.566
`barr-crc32`	265	45	0.402	`jfdctint`	366	15	0.374
`barr-crc-ccitt`	265	35	0.318	`lpbench`	419	134	1.155
`bellman-ford`	75	39	0.369	`mergesort-recursive`	42	50	0.634
`blowfish`	476	43	0.389	`sort`	138	90	0.757
`bmpfile`	749	254	3.050	`zlib-adler32`	124	34	0.891
`c-aes`	236	64	0.385	`zlib-crc32-BYFOUR`	335	41	1.182
`c-des`	399	64	0.477	`zlib-crc32`	333	13	0.170

Thus, `KITTeL` clearly shows the practicality and effectiveness of the proposed approach on a collection of "typical" examples. Notice that an empirical comparison with the methods from [4, 5, 6, 7, 8, 11, 12, 31, 32] is not possible since implementations of those methods are not publicly available. The examples, detailed results, the termination proofs generated by

[15] If the invariants are omitted from the aforementioned 14 examples, then termination cannot be shown.
[16] On a 2.4 GHz Intel® Core™2 Duo processor with 4 GB main memory.

KITTeL, an a link to a web interface for KITTeL are available at http://baldur.iti.kit.edu/~falke/kittel/.

13 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 rewriting system. In this paper, we have concentrated on LLVM and its intermediate language LLVM-IR [28]. Finally, termination of the obtained TRS is shown using term rewriting techniques.

Recall that it is assumed in this paper that 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 whose termination is to be investigated. The methods from [4, 5, 6, 7, 8, 11, 12, 31, 32] also exhibit this problem, only [10] 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 Gilles Barthe and Frank S. de Boer, editors, *Proceedings of the 10th Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2008.

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

3 Thomas Ball and Robert Jones, editors. *Proceedings of the 18th Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

4 Aaron Bradley, Zohar Manna, and Henny Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram Rajamani, editors, *Proceedings of the 17th Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer-Verlag, 2005.

5 Aaron Bradley, Zohar Manna, and Henny Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *Proceedings of the 6th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 113–129. Springer-Verlag, 2005.

6 Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In Sophia Drossopoulou, editor, *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, volume 4960 of *Lecture Notes in Computer Science*, pages 148–162. Springer-Verlag, 2008.

7 Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2001.

8 Michael Colón and Henny Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer-Verlag, 2002.

9 Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. In Marco T. Morazán, editor, *Proceedings of the 8th Symposium on Trends in Functional Programming (TFP '07)*, volume 8 of *Trends in Functional Programming*, pages 124–140. Intellect Books, 2008.

10 Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In Javier Esparza and Rupak Majumdar, editors, *Proceedings of the 16th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer-Verlag, 2010.

11 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *Proceedings of the 12th Symposium on Static Analysis (SAS '05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer-Verlag, 2005.

12 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, pages 415–426, 2006.

13 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Ball and Jones [3], pages 415–418.

14 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.

15 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Symposium on Principles of Programming Languages (POPL '78)*, pages 84–96, 1978.

16 Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Ball and Jones [3], pages 81–94.

17 Stephan Falke and Deepak Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In Andrei Voronkov, editor, *Proceedings of the 19th Conference on Rewriting Techniques and Applications (RTA '08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 94–109. Springer-Verlag, 2008. An expanded version is Technical Report TR-CS-2007-21, available from `http://www.cs.unm.edu/research/tech-reports/`.

18 Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In Renate A. Schmidt, editor, *Proceedings of the 22nd Conference on Automated Deduction (CADE '09)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 277–293. Springer-Verlag, 2009. An expanded version is Technical Report TR-CS-2009-02, available from `http://www.cs.unm.edu/research/tech-reports/`.

19 Paul Feautrier and Laure Gonnord. Accelerated invariant generation for C programs with Aspic and C2fsm. *Electronic Notes in Theoretical Computer Science*, 267(2):3–13, 2010.

20 Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In Ralf Treinen, editor, *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA '09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 32–47. Springer-Verlag, 2009.

21 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei

Voronkov, editors, *Proceedings of the 11th Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '04)*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 301–331. Springer-Verlag, 2005.

22 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp. Proving termination by bounded increase. In Frank Pfenning, editor, *Proceedings of the 21st Conference on Automated Deduction (CADE '07)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 443–459. Springer-Verlag, 2007. An expanded version is Technical Report AIB-2007-03, available from `http://aib.informatik.rwth-aachen.de/`.

23 Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

24 Hoon Hong and Dalibor Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.

25 Keiichirou Kusakari, Masaki Nakamura, and Yoshihito Toyama. Argument filtering transformation. In Gopalan Nadathur, editor, *Proceedings of the 1st Conference on Principles and Practice of Declarative Programming (PPDP '99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 1999.

26 Gaël Lalire, Mathias Argoud, and Bertrand Jeannet. Interproc analyzer for recursive programs with numerical variables, 2010. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.

27 Dallas Lankford. On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech University, Ruston, 1979.

28 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd Symposium on Code Generation and Optimization (CGO '04)*, pages 75–88. IEEE Computer Society, 2004.

29 Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

30 Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java bytecode by term rewriting. In Christopher Lynch, editor, *Proceedings of the 21st Conference on Rewriting Techniques and Applications (RTA '10)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 259–276. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

31 Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer-Verlag, 2004.

32 Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proceedings of the 19th Symposium on Logic in Computer Science (LICS '04)*, pages 32–41. IEEE Computer Society, 2004.

33 Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

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

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

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

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

A Empirical Results

We have evaluated `KITTeL`/`llvm2kittel` on a collection of 174 examples. 14 out of these 174 examples require simple invariants on the program variables for a successful termination proof. In the following table, C programs with a name ending in `no-inv` have these invariants omitted and are otherwise identical to the corresponding C programs containing the invariants.

The following table contains the details of our empirical evaluation. All times where obtained on a 2.4 GHz Intel® Core™2 Duo processor with 4 GB main memory. The "LOC" column gives the number of code lines in the C program, and the "RR" column gives the number of rewrite rules that are generated. We have recorded the times taken by the compiler frontend (`llvm-gcc`), the translation into a TRS (`llvm2kittel`), and the termination proof (`KITTeL`). A (total) timeout of 5 seconds was used. The examples and the termination proofs generated by `KITTeL` are available at `http://baldur.iti.kit.edu/~falke/kittel/`.

C program	**LOC**	**RR**	`llvm-gcc`	`llvm2kittel`	`KITTeL`	**Total**	**Result**
`a.01.c`	10	9	0.014	0.007	0.071	0.092	YES
`a.02.c`	19	17	0.013	0.014	0.107	0.134	YES
`a.02.real.c`	12	14	0.013	0.010	0.100	0.124	YES
`a.03.c`	44	29	0.014	0.036	0.495	0.545	YES
`a.03-no-inv.c`	43	28	0.013	0.035	5.007	5.055	∞
`a.03.real.c`	38	28	0.014	0.027	0.369	0.409	YES
`a.03.real-no-inv.c`	37	27	0.013	0.026	5.006	5.046	∞
`a.04.c`	6	6	0.014	0.006	0.037	0.057	YES
`a.05.c`	6	6	0.013	0.007	0.037	0.057	YES
`a.06.c`	7	6	0.013	0.007	0.039	0.059	YES
`a.07.c`	7	7	0.013	0.007	0.064	0.084	YES
`a.08.c`	7	6	0.013	0.006	0.037	0.056	YES
`a.09.c`	10	8	0.013	0.007	0.046	0.066	YES
`a.10.c`	10	10	0.013	0.007	0.159	0.178	YES
`a.11.c`	16	17	0.013	0.012	0.193	0.217	YES
`ack.c`	9	11	0.013	0.008	0.174	0.195	YES
`Ack.c`	27	16	0.023	0.009	0.198	0.230	YES
`allroots.c`	200	77	0.034	0.050	0.763	0.847	YES
`almabench.c`	390	42	0.039	0.027	0.300	0.366	YES
`b.01.c`	6	6	0.013	0.006	0.037	0.055	YES
`b.02.c`	7	6	0.013	0.007	0.037	0.057	YES
`b.03.c`	10	8	0.013	0.006	0.047	0.066	YES
`b.03-no-inv.c`	9	7	0.013	0.006	5.010	5.029	∞
`b.04.c`	8	5	0.014	0.006	0.029	0.048	YES
`b.05.c`	10	9	0.013	0.008	0.058	0.079	YES
`b.06.c`	7	7	0.013	0.007	0.044	0.064	YES
`b.07.c`	7	7	0.013	0.007	0.057	0.076	YES
`b.08.c`	18	16	0.013	0.010	0.113	0.136	YES
`b.09.c`	15	12	0.013	0.007	0.090	0.110	YES
`b.09-no-inv.c`	14	10	0.013	0.007	5.006	5.026	∞
`b.10.c`	14	10	0.013	0.009	0.100	0.122	YES

C program	LOC	RR	llvm-gcc	llvm2kittel	KITTeL	Total	Result
b.11.c	14	13	0.013	0.009	0.103	0.124	YES
b.12.c	14	11	0.013	0.008	0.124	0.146	YES
b.13.c	14	11	0.013	0.008	0.124	0.145	YES
b.14.c	9	10	0.013	0.008	0.067	0.087	YES
b.15.c	9	10	0.013	0.008	0.072	0.093	YES
b.16.c	9	9	0.013	0.007	0.067	0.087	YES
b.17.c	9	9	0.013	0.008	0.080	0.100	YES
b.18.c	14	14	0.013	0.009	0.097	0.119	YES
barr-crc16.c	265	45	0.021	0.037	0.337	0.395	YES
barr-crc32.c	265	45	0.021	0.038	0.336	0.395	YES
barr-crc-ccitt.c	265	35	0.021	0.030	0.269	0.319	YES
bellman-ford.c	75	39	0.019	0.026	0.318	0.363	YES
binary_search.c	13	13	0.013	0.011	0.131	0.155	YES
binsearch.c	30	20	0.013	0.014	0.448	0.476	YES
binsearch-recursive.c	17	15	0.013	0.010	0.910	0.934	YES
blit.c	98	28	0.019	0.126	0.166	0.311	YES
blowfish.c	476	43	0.026	0.045	0.324	0.395	YES
bmpfile.c	749	254	0.046	0.424	2.536	3.006	YES
break.c	9	6	0.013	0.006	0.034	0.053	YES
bresenham.c	36	9	0.014	0.026	0.067	0.106	YES
brutesearch.c	21	15	0.016	0.012	0.150	0.178	YES
brutesearch-no-inv.c	17	13	0.016	0.011	5.007	5.034	∞
bubble_nice.c	50	25	0.024	0.014	0.170	0.208	YES
bubble_sort.c	13	12	0.013	0.010	0.102	0.125	YES
bubblesort.c	14	12	0.013	0.011	0.104	0.128	YES
Bubblesort.c	172	41	0.024	0.024	0.270	0.317	YES
c.01.c	13	10	0.013	0.007	0.079	0.100	YES
c.01-no-inv.c	10	9	0.013	0.007	5.009	5.029	∞
c.02.c	11	10	0.014	0.008	0.081	0.103	YES
c.03.c	10	9	0.018	0.007	0.126	0.151	YES
c.04.c	19	13	0.013	0.009	0.150	0.172	YES
c.04-no-inv.c	16	12	0.013	0.008	5.005	5.026	∞
c.05.c	19	14	0.013	0.009	0.124	0.146	YES
c.06.c	24	17	0.014	0.014	0.590	0.618	YES
c.07.c	9	7	0.013	0.007	0.048	0.068	YES
c.08.c	10	9	0.013	0.008	0.071	0.092	YES
c.09.c	13	11	0.013	0.008	0.115	0.136	YES
c.10.c	14	8	0.013	0.008	0.048	0.069	YES
c.11.c	18	13	0.013	0.009	0.096	0.119	YES
cacm.c	15	11	0.013	0.009	0.079	0.100	YES
c_aes.c	236	64	0.028	0.022	0.331	0.381	YES
cav1.c	12	10	0.013	0.007	0.065	0.085	YES
cav2.c	23	15	0.013	0.011	0.335	0.359	YES

C program	LOC	RR	llvm-gcc	llvm2kittel	KITTeL	Total	Result
c_des.c	399	64	0.037	0.066	0.370	0.472	YES
chaining1.c	16	14	0.013	0.009	0.442	0.464	YES
chaining2.c	12	9	0.013	0.007	0.129	0.148	YES
chaining3.c	10	10	0.013	0.060	0.456	0.529	YES
crc.c	98	33	0.031	0.118	0.211	0.360	YES
cube.c	146	68	0.035	0.050	0.532	0.616	YES
diff.c	25	25	0.013	0.015	0.249	0.277	YES
dijkstra.c	78	58	0.020	0.038	0.622	0.681	YES
dt4.c	49	35	0.019	0.184	1.940	2.143	YES
eratosthenes.c	21	22	0.020	0.014	0.159	0.194	YES
euclid.c	30	13	0.018	0.009	0.078	0.105	YES
euclid-no-inv.c	29	12	0.019	0.009	5.010	5.038	∞
ex1.c	8	6	0.014	0.007	0.035	0.056	YES
ex2.c	15	14	0.013	0.009	0.178	0.200	YES
ex3a.c	6	7	0.013	0.006	0.050	0.069	YES
ex3b.c	6	7	0.013	0.007	0.052	0.072	YES
factorial.c	8	6	0.013	0.006	0.034	0.053	YES
fermat.c	22	38	0.013	0.028	0.206	0.248	YES
fft16.c	188	2	0.020	0.007	0.010	0.038	YES
fft.c	99	30	0.028	0.025	0.285	0.338	YES
fft-no-inv.c	97	29	0.023	0.025	5.006	5.054	∞
fibcall.c	79	7	0.014	0.007	0.051	0.072	YES
fibo.c	75	11	0.027	0.010	0.070	0.108	YES
fibonacci.c	8	7	0.013	0.006	0.043	0.063	YES
flag.c	7	8	0.017	0.006	0.058	0.082	YES
floyd-warshall.c	23	15	0.014	0.011	0.144	0.169	YES
hanoi.c	45	8	0.023	0.007	0.047	0.077	YES
hash.c	241	80	0.025	0.045	0.478	0.548	YES
hoist_call.c	14	7	0.013	0.007	0.038	0.058	YES
hoist_load.c	14	6	0.014	0.006	0.035	0.056	YES
inline.c	15	7	0.013	0.006	0.041	0.060	YES
insertion_sort.c	13	12	0.014	0.009	0.100	0.123	YES
insertionsort.c	12	12	0.013	0.009	0.100	0.123	YES
insertsort.c	85	11	0.014	0.009	0.085	0.107	YES
inside.c	44	14	0.015	0.023	0.092	0.131	YES
intersect.c	24	2	0.014	0.005	0.008	0.027	YES
IntMM.c	161	36	0.023	0.016	0.220	0.259	YES
java_Ackermann.c	5	11	0.013	0.008	0.172	0.193	YES
java_AG313.c	9	9	0.013	0.008	0.061	0.082	YES
java_AProVEMath.c	22	26	0.013	0.013	0.162	0.189	YES
java_AProVEMathRecursive.c	14	20	0.019	0.015	0.435	0.470	YES
java_Avg.c	9	10	0.013	0.007	0.327	0.347	YES
java_Break.c	8	6	0.013	0.005	0.034	0.052	YES

C program	LOC	RR	llvm-gcc	llvm2kittel	KITTeL	Total	Result
java_BubbleSort.c	11	12	0.013	0.010	0.107	0.130	YES
java_Continue1.c	8	6	0.013	0.006	0.034	0.053	YES
java_Diff.c	19	25	0.016	0.015	0.253	0.285	YES
java_DivMinus1.c	8	7	0.014	0.006	0.046	0.066	YES
java_DivMinus2.c	21	14	0.013	0.010	0.211	0.235	YES
java_DivWithoutMinus.c	19	16	0.013	0.009	0.132	0.154	YES
java_Double1.c	10	8	0.013	0.006	0.114	0.133	YES
java_Double2.c	10	8	0.019	0.009	0.052	0.080	YES
java_Double3.c	8	8	0.013	0.006	0.048	0.067	YES
java_Duplicate.c	11	7	0.013	0.007	0.046	0.066	YES
java_EqUserDefRec.c	7	11	0.013	0.007	0.067	0.087	YES
java_Factorial.c	4	6	0.013	0.006	0.034	0.053	YES
java_FactSum.c	17	13	0.013	0.007	0.068	0.089	YES
java_FibRecursive.c	9	10	0.013	0.007	0.063	0.083	YES
java_Hanoi.c	11	9	0.013	0.007	0.055	0.074	YES
java_LeUserDefRec.c	7	9	0.014	0.007	0.056	0.076	YES
java_LogBuiltIn.c	14	8	0.014	0.006	0.046	0.066	YES
java_MinusBuiltIn.c	16	6	0.013	0.006	0.037	0.056	YES
java_MinusMin.c	24	10	0.013	0.008	0.054	0.076	YES
java_Nested.c	7	9	0.013	0.007	0.063	0.083	YES
java_NestedLoop.c	20	19	0.019	0.017	0.163	0.199	YES
java_PlusSwap.c	19	6	0.013	0.006	0.047	0.066	YES
java_Recursions.c	44	36	0.015	0.015	0.263	0.293	YES
java_Sequence.c	7	9	0.013	0.006	0.053	0.072	YES
java_TimesPlusUserDef.c	17	18	0.013	0.009	0.112	0.134	YES
jfdctint.c	366	15	0.015	0.269	0.083	0.367	YES
knapsack.c	14	15	0.013	0.014	0.147	0.174	YES
lis.c	28	24	0.019	0.023	0.242	0.284	YES
lpbench.c	419	134	0.036	0.065	1.051	1.151	YES
lpbench-no-inv.c	419	133	0.036	0.064	5.013	5.113	∞
matmul.c	78	22	0.014	0.015	0.191	0.220	YES
matrix.c	65	32	0.019	0.017	0.235	0.271	YES
Matrix.c	71	40	0.024	0.021	0.313	0.357	YES
matrix_chain.c	26	26	0.015	0.035	0.291	0.340	YES
max-array.c	15	10	0.013	0.008	0.068	0.089	YES
max_sum.c	69	35	0.020	0.020	0.356	0.396	YES
mergesort.c	55	42	0.023	0.026	0.398	0.447	YES
mergesort-no-inv.c	53	40	0.023	0.026	5.006	5.055	∞
mergesort-recursive.c	42	50	0.036	0.065	0.522	0.624	YES
mutual1.c	11	9	0.013	0.006	0.052	0.070	YES
mutual2.c	21	15	0.013	0.007	0.084	0.104	YES
n-body.c	141	35	0.034	0.016	0.237	0.287	YES
nsieve-bits.c	38	29	0.024	0.035	0.302	0.362	YES

C program	LOC	RR	llvm-gcc	llvm2kittel	KITTeL	Total	Result
nsieve-bits-no-inv.c	36	28	0.024	0.034	5.006	5.065	∞
opt-tree.c	30	33	0.016	0.055	0.425	0.496	YES
Oscar.c	323	77	0.027	0.053	0.568	0.647	YES
Oscar-no-inv.c	320	74	0.027	0.051	5.006	5.084	∞
perfect.c	57	31	0.024	0.024	0.328	0.376	YES
perfect-no-inv.c	54	30	0.023	0.023	5.006	5.052	∞
Perm.c	172	35	0.025	0.013	0.195	0.233	YES
pi.c	31	17	0.023	0.019	0.112	0.154	YES
power.c	23	26	0.013	0.014	0.162	0.188	YES
prim.c	83	44	0.019	0.036	0.432	0.487	YES
puzzle.c	67	31	0.023	0.016	0.194	0.233	YES
qsort.c	35	20	0.013	0.011	0.141	0.165	YES
RealMM.c	162	36	0.024	0.015	0.218	0.257	YES
selection_nice.c	59	24	0.023	0.013	0.159	0.195	YES
selection_sort.c	15	11	0.013	0.010	0.096	0.119	YES
selectionsort.c	18	11	0.016	0.010	0.101	0.127	YES
shell_sort.c	20	27	0.014	0.019	0.268	0.301	YES
snu-crc.c	124	33	0.015	0.118	0.211	0.343	YES
sort.c	138	90	0.024	0.038	0.682	0.745	YES
spectral-norm.c	52	39	0.032	0.018	0.263	0.312	YES
sphere.c	157	68	0.035	0.050	0.533	0.617	YES
spiral.c	176	80	0.034	0.068	0.619	0.722	YES
Towers.c	220	37	0.025	0.014	0.176	0.215	YES
twisted.c	18	15	0.013	0.011	0.139	0.164	YES
wrap.c	38	24	0.019	0.025	0.211	0.255	YES
zlib-adler32.c	124	34	0.017	0.642	0.223	0.883	YES
zlib-crc32-BYFOUR.c	335	41	0.027	0.851	0.276	1.154	YES
zlib-crc32.c	333	13	0.017	0.074	0.074	0.166	YES