Termination Analysis of Imperative Programs Using Bitvector Arithmetic

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

[1]Institute for Theoretical Computer Science, KIT, Germany

[2]Dept. of Computer Science, University of New Mexico, USA

Verified Software: Theories, Tools, Experiments
VSTTE 2012

Summary

This paper proposes a novel method for encoding the wrap-around behavior of bitvector arithmetic within integer arithmetic, such that existing methods for reasoning about the termination of integer arithmetic programs can be employed for reasoning about the termination of bitvector arithmetic programs.

Outline

Outline

1. Introduction
2. Termination Analysis in KITTeL
3. Encoding Bitvector Arithmetic
4. Conclusion

Existing Methods

Bitvectors are treated as (unbounded) integers (or as real numbers).

This approximation can cause errors in both directions:

- Terminating wrt bitvector arithmetic, but non-terminating wrt integer arithmetic.

Example

```
void f(int i) {
  while (i > 0)
    i++;
}
```

- Terminating wrt integer arithmetic, but non-terminating wrt bitvector arithmetic.

Example

```
void g(int i, int j) {
  while (i <= j)
    i++;
}
```

Related Work [CKR+10]

Main differences:

- The use of quantifiers. Quantifier elimination before ranking function synthesis is expensive.
- Template matching for linear ranking functions allows to use SAT, QBF and SMT solver.

[CKR+10] Byron Cook, Daniel Kroening, Philipp Rümmer, Christoph M. Wintersteiger: Ranking Function Synthesis for Bit-Vector Relations. TACAS 2010: 236-250

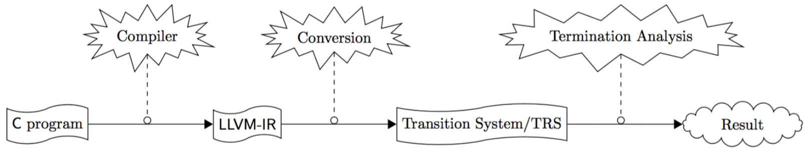
Outline

1. Introduction
2. Termination Analysis in KITTeL
3. Encoding Bitvector Arithmetic
4. Conclusion

The Approach of KITTeL

KITTeL is a termination analysis tool on LLVM-IR.

Bird's-eye view of KITTeL's approach

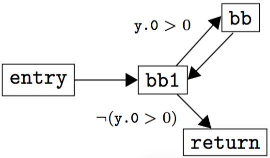


From C to LLVM-IR

From C code to LLVM-IR code, and then *basic block graph*:

Example 1

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

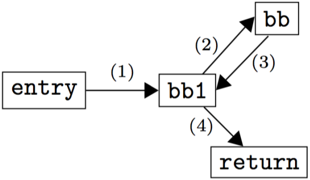



```
define i32 @power(i32 %x, i32 %y) {
entry:
  br label %bb1

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
}
```

From LLVM-IR to Transition System

The corresponding transition system of Example 1:

Example 1



(1) $\mathtt{y.0}' \simeq \mathtt{y} \wedge \mathtt{r.0}' \simeq 1$
(2) $\mathtt{y.0} > 0$
(3) $\mathtt{y.0}' \simeq \mathtt{y.0} - 1 \wedge \mathtt{r.0}' \simeq \mathtt{r.0} * \mathtt{x} \wedge$
$\%1' \simeq \mathtt{r.0} * \mathtt{x} \wedge \%2' \simeq \mathtt{y.0} - 1$
(4) $\neg(\mathtt{y.0} > 0)$

From Transition Systems to TRSs

- A transition system can be represented in the form of an `int`-based TRS.
- Each transition $s_1 \to^{\lambda} s_2$ gives rise to a rewrite rule

$$s_1(x_1, \ldots, x_n) \to s_2(e_1, \ldots, e_n) \; [\![\varphi]\!]$$

TRS of Example 1

$$\begin{array}{rcl}
\mathtt{entry}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \%1, \%2) & \to & \mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y}, 1, \%1, \%2) \\
\mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \%1, \%2) & \to & \mathtt{bb}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \%1, \%2) \; [\![\mathtt{y.0} > 0]\!] \\
\mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \%1, \%2) & \to & \mathtt{return}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \%1, \%2) \; [\![\neg(\mathtt{y.0} > 0)]\!] \\
\mathtt{bb}(\mathtt{x}, \mathtt{y}, \mathtt{y.0}, \mathtt{r.0}, \%1, \%2) & \to & \mathtt{bb1}(\mathtt{x}, \mathtt{y}, \mathtt{y.0} - 1, \mathtt{r.0}, \mathtt{r.0} * \mathtt{x}, \mathtt{y.0} - 1)
\end{array}$$

Reducing TRSs

The number of arguments, hence the TRS, can be further reduced by standard compiler techniques:

Reduced TRS of Example 1

$$
\begin{array}{rcl}
\mathtt{entry}(\mathtt{y}) & \to & \mathtt{bb1}(\mathtt{y}) \\
\mathtt{bb1}(\mathtt{y.0}) & \to & \mathtt{bb}(\mathtt{y.0})\ [\![\mathtt{y.0} > 0]\!] \\
\mathtt{bb1}(\mathtt{y.0}) & \to & \mathtt{return}()\ [\![\neg(\mathtt{y.0} > 0)]\!] \\
\mathtt{bb}(\mathtt{y.0}) & \to & \mathtt{bb1}(\mathtt{y.0} - 1)
\end{array}
$$

Reducing TRSs

The number of arguments, hence the TRS, can be further reduced by standard compiler techniques:

- Backward slicing removes all variables that are not relevant for the control flow of the program. (`x`, `r.0`, %1, and %2 in Example 1)

Reduced TRS of Example 1

$$
\begin{array}{rcl}
\mathtt{entry}(\mathtt{y}) & \to & \mathtt{bb1}(\mathtt{y}) \\
\mathtt{bb1}(\mathtt{y.0}) & \to & \mathtt{bb}(\mathtt{y.0})\ [\![\mathtt{y.0} > 0]\!] \\
\mathtt{bb1}(\mathtt{y.0}) & \to & \mathtt{return}()\ [\![\neg(\mathtt{y.0} > 0)]\!] \\
\mathtt{bb}(\mathtt{y.0}) & \to & \mathtt{bb1}(\mathtt{y.0} - 1)
\end{array}
$$

Reducing TRSs

The number of arguments, hence the TRS, can be further reduced by standard compiler techniques:

- Backward slicing removes all variables that are not relevant for the control flow of the program. (`x`, `r.0`, %1, and %2 in Example 1)
- Liveness analysis removes variables before they are defined or after they are no longer needed. (`y.0` in `entry` and `return`; `y` in all function symbols but `entry`)

Reduced TRS of Example 1

$$\begin{array}{rcl} \mathtt{entry}(\mathtt{y}) & \to & \mathtt{bb1}(\mathtt{y}) \\ \mathtt{bb1}(\mathtt{y.0}) & \to & \mathtt{bb}(\mathtt{y.0}) \ [\![\mathtt{y.0} > 0]\!] \\ \mathtt{bb1}(\mathtt{y.0}) & \to & \mathtt{return}() \ [\![\neg(\mathtt{y.0} > 0)]\!] \\ \mathtt{bb}(\mathtt{y.0}) & \to & \mathtt{bb1}(\mathtt{y.0} - 1) \end{array}$$

Outline

Overview of the Encoding

Overview of the Encoding

The bitvector $b_{n-1} \cdots b_0$ is represented by its *signed* (two's complement) integer

$$-b_{n-1} \cdot 2^{n-1} + \Sigma_{i=0}^{n-2} b_i \cdot 2^i \quad \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}.$$

Overview of the Encoding

The bitvector $b_{n-1} \cdots b_0$ is represented by its *signed* (two's complement) integer

$$-b_{n-1} \cdot 2^{n-1} + \Sigma_{i=0}^{n-2} b_i \cdot 2^i \quad \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}.$$

The semantics of the bitvector operations and the wrap-around behavior of bitvector arithmetic is handled by two phases:

Overview of the Encoding

The bitvector $b_{n-1} \cdots b_0$ is represented by its *signed* (two's complement) integer

$$-b_{n-1} \cdot 2^{n-1} + \Sigma_{i=0}^{n-2} b_i \cdot 2^i \quad \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}.$$

The semantics of the bitvector operations and the wrap-around behavior of bitvector arithmetic is handled by two phases:

1. The conversion of comparison, arithmetical, bitwise and extension/truncation instructions is adapted. In particular, signed and unsigned operations are distinguished.

Overview of the Encoding

The bitvector $b_{n-1} \cdots b_0$ is represented by its *signed* (two's complement) integer

$$-b_{n-1} \cdot 2^{n-1} + \Sigma_{i=0}^{n-2} b_i \cdot 2^i \quad \in \{-2^{n-1}, \ldots, 2^{n-1} - 1\}.$$

The semantics of the bitvector operations and the wrap-around behavior of bitvector arithmetic is handled by two phases:

1. The conversion of comparison, arithmetical, bitwise and extension/truncation instructions is adapted. In particular, signed and unsigned operations are distinguished.
2. The wrap-around behavior of the arithmetical instructions is modeled in order to ensure that they are within the appropriate ranges.

Phase 1

Phase 1 is performed during the generation of the transition system/`int`-based TRS. The conversion of instructions is adapted in order to correspond to their semantics on bitvectors.

Phase 1

Phase 1 is performed during the generation of the transition system/`int`-based TRS. The conversion of instructions is adapted in order to correspond to their semantics on bitvectors.

The comparison instructions:

`icmp eq i`k x`,` y	$x \simeq y$	`icmp ne i`k x`,` y	$x \not\simeq y$
`icmp ugt i`k x`,` y	$\mathsf{ugt}(x, y)$	`icmp sgt i`k x`,` y	$x > y$
`icmp uge i`k x`,` y	$\mathsf{ugt}(x, y) \vee x \simeq y$	`icmp sge i`k x`,` y	$x \geq y$
`icmp ult i`k x`,` y	$\mathsf{ult}(x, y)$	`icmp slt i`k x`,` y	$x < y$
`icmp ule i`k x`,` y	$\mathsf{ult}(x, y) \vee x \simeq y$	`icmp sle i`k x`,` y	$x < y$

Phase 1

Phase 1 is performed during the generation of the transition system/`int`-based TRS. The conversion of instructions is adapted in order to correspond to their semantics on bitvectors.

The comparison instructions:

`icmp eq i`k x, y	$x \simeq y$	`icmp ne i`k x, y	$x \not\simeq y$
`icmp ugt i`k x, y	$\mathsf{ugt}(x, y)$	`icmp sgt i`k x, y	$x > y$
`icmp uge i`k x, y	$\mathsf{ugt}(x, y) \vee x \simeq y$	`icmp sge i`k x, y	$x \geq y$
`icmp ult i`k x, y	$\mathsf{ult}(x, y)$	`icmp slt i`k x, y	$x < y$
`icmp ule i`k x, y	$\mathsf{ult}(x, y) \vee x \simeq y$	`icmp sle i`k x, y	$x < y$

For example,

$$\begin{aligned} ugt(x, y) \quad &= \quad (x \geq 0 \wedge y \geq 0 \wedge x > y) \\ &\quad\; \vee (x \geq 0 \wedge y < 0) \\ &\quad\; \vee (x < 0 \wedge y < 0 \wedge x > y) \end{aligned}$$

Phase 1 (cont'd)

arithmetic	$z = \texttt{add}\ ik\ x,\ y$	$z' \simeq x + y$
	$z = \texttt{sub}\ ik\ x,\ y$	$z' \simeq x - y$
	$z = \texttt{mul}\ ik\ x,\ y$	$z' \simeq x * y$
	$z = \texttt{sdiv}\ ik\ x,\ y$	$z' \simeq n_i \wedge \mathsf{sdiv}(x, y, n_i)$
	$z = \texttt{udiv}\ ik\ x,\ y$	$z' \simeq n_i \wedge \mathsf{udiv}(x, y, n_i)$
	$z = \texttt{srem}\ ik\ x,\ y$	$z' \simeq n_i \wedge \mathsf{srem}(x, y, n_i)$
	$z = \texttt{urem}\ ik\ x,\ y$	$z' \simeq n_i \wedge \mathsf{urem}(x, y, n_i)$
bitwise operations	$z = \texttt{and}\ ik\ x,\ y$	$z' \simeq n_i \wedge \mathsf{and}(x, y, n_i)$
	$z = \texttt{or}\ ik\ x,\ y$	$z' \simeq n_i \wedge \mathsf{or}(x, y, n_i)$
	$z = \texttt{xor}\ ik\ x,\ y$	$z' \simeq n_i$
extension/truncation	$z = \texttt{sext}\ ik\ x\ \texttt{to}\ il$	$z' \simeq n_i$
	$z = \texttt{uext}\ ik\ x\ \texttt{to}\ il$	$z' \simeq x \wedge \mathsf{uext}(z, n_i)$
	$z = \texttt{trunc}\ ik\ x\ \texttt{to}\ il$	$z' \simeq x$

Phase 1 (cont'd)

Some instructions are approximated by introducing new variables and constraints.

For example, the `and` instruction, its constraint:

$$\begin{aligned} and(x, y, n_i) \quad = \quad & (x \geq 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \leq x \wedge n_i \leq y) \\ & \vee (x \geq 0 \wedge y < 0 \wedge n_i \geq 0 \wedge n_i \leq x) \\ & \vee (x < 0 \wedge y \geq 0 \wedge n_i \geq 0 \wedge n_i \leq y) \\ & \vee (x < 0 \wedge y < 0 \wedge n_i < 0 \wedge n_i \leq x \wedge n_i \leq y) \end{aligned}$$

Phase 2

Phase 2 is performed after generating the `int`-based TRS. The wrap-around behavior of the arithmetical instructions is modeled by modifying the generated `int`-based TRS.

Phase 2

Phase 2 is performed after generating the `int`-based TRS. The wrap-around behavior of the arithmetical instructions is modeled by modifying the generated `int`-based TRS.

For this, the wrap-around behavior is simulated by explicitly normalizing the resulting integers to be in the appropriate ranges.

Phase 2

Phase 2 is performed after generating the `int`-based TRS. The wrap-around behavior of the arithmetical instructions is modeled by modifying the generated `int`-based TRS.

For this, the wrap-around behavior is simulated by explicitly normalizing the resulting integers to be in the appropriate ranges.

Each rewrite rule

$$\rho : f(x_1, \ldots, x_n) \to g(p_1, \ldots, p_m) \; [\![\varphi]\!]$$

Phase 2

Phase 2 is performed after generating the `int`-based TRS. The wrap-around behavior of the arithmetical instructions is modeled by modifying the generated `int`-based TRS.
For this, the wrap-around behavior is simulated by explicitly normalizing the resulting integers to be in the appropriate ranges.
Each rewrite rule

$$\rho : f(x_1, \ldots, x_n) \to g(p_1, \ldots, p_m) \; [\![\varphi]\!]$$

should be replaced by:

$$\begin{aligned}
f(x_1, \ldots, x_n) &\to g^\sharp(p_1, \ldots, p_m) \; [\![\varphi \wedge \mathsf{inrange}(\mathcal{V}(\rho))]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1 + 2^{\|x_1\|}, \ldots, x_m) \; [\![x_1 < \mathsf{intmin}(\|x_1\|)]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1 - 2^{\|x_1\|}, \ldots, x_m) \; [\![x_1 > \mathsf{intmax}(\|x_1\|)]\!] \\
&\;\;\vdots \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1, \ldots, x_m + 2^{\|x_m\|}) \; [\![x_m < \mathsf{intmin}(\|x_m\|)]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g^\sharp(x_1, \ldots, x_m - 2^{\|x_m\|}) \; [\![x_m > \mathsf{intmax}(\|x_m\|)]\!] \\
g^\sharp(x_1, \ldots, x_m) &\to g(x_1, \ldots, x_m) \; [\![\mathsf{inrange}(\{x_1, \ldots, x_m\})]\!]
\end{aligned}$$

Phase 2 (cont'd)

For example, the rule

$$\mathtt{bb}(\mathtt{i.0}) \to \mathtt{bb1}(\mathtt{i.0}) + 1)$$

is replaced by

$$\begin{array}{rcl}
\mathtt{bb}(\mathtt{i.0}) & \to & \mathtt{bb1}^{\#}(\mathtt{i.0} + 1) \; [\![\mathit{inrange}(\mathtt{i.0})]\!] \\
\mathtt{bb1}^{\#}(\mathtt{i.0}) & \to & \mathtt{bb1}^{\#}(\mathtt{i.0} + 2^{32}) \; [\![\mathtt{i.0} < \mathit{intmin}(32)]\!] \\
\mathtt{bb1}^{\#}(\mathtt{i.0}) & \to & \mathtt{bb1}^{\#}(\mathtt{i.0} - 2^{32}) \; [\![\mathtt{i.0} > \mathit{intmax}(32)]\!] \\
\mathtt{bb1}^{\#}(\mathtt{i.0}) & \to & \mathtt{bb1}(\mathtt{i.0}) \; [\![\mathit{inrange}(\mathtt{i.0})]\!]
\end{array}$$

Outline

Conclusion

- The approach is sound.

Conclusion

- The approach is sound.
  - The instructions are *soundly* approximated.

Conclusion

- The approach is sound.
  - The instructions are *soundly* approximated.
  - The wrap-around behavior is correctly modeled.

Conclusion

- The approach is sound.
  - The instructions are *soundly* approximated.
  - The wrap-around behavior is correctly modeled.
- The implementation is powerful than the related approach in [CKR+10], with a comparable average time.

Conclusion

- The approach is sound.
  - The instructions are *soundly* approximated.
  - The wrap-around behavior is correctly modeled.
- The implementation is powerful than the related approach in [CKR+10], with a comparable average time.
- The approach can be combined with other existing techniques for analyzing termination of transition systems/`int`-based TRSs.

Thank you!