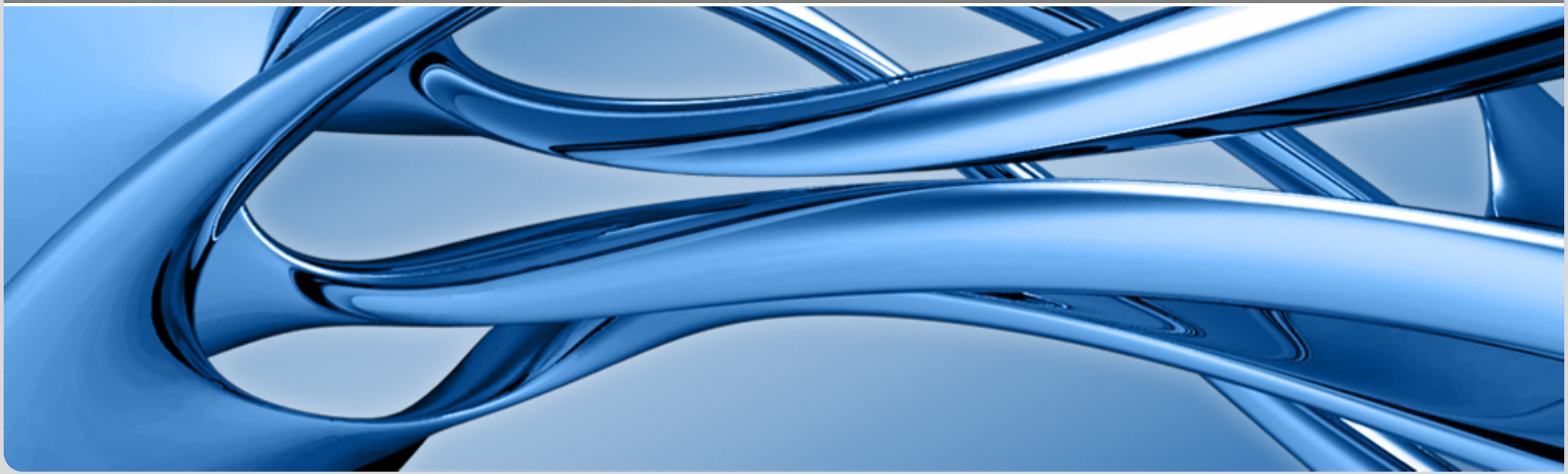
Karlsruher Institut für Technologie

LLVM

Entscheidungsverfahren mit Anwendungen in der Softwareverifikation

STEPHAN FALKE — INSTITUT FÜR THEORETISCHE INFORMATIK (ITI)

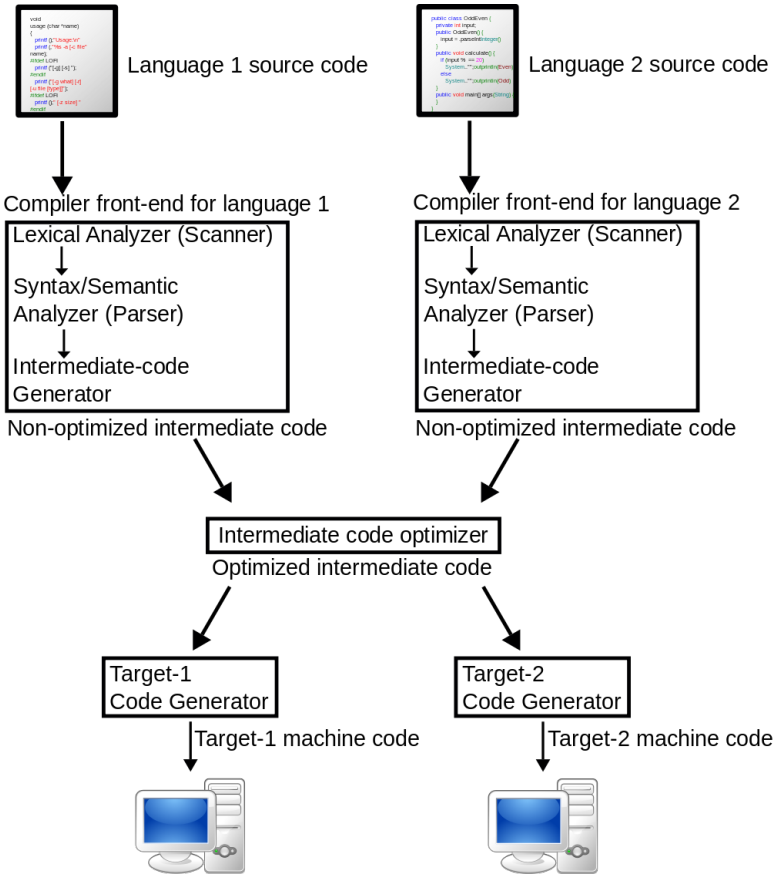


Inhalt

Motivation

- Ziel: Verifikationssystem das alle Features einer komplexen Programmiersprache (C, C++, Java, . . .) unterstützt
- Probleme:
  - Komplexer Syntax
  - Komplexe Semantik
  - Viele Grenzfälle
- Hohe Anfangshürde, insbesondere für akademische Forschungsprojekte
- Lösung: Benutze ein Werkzeug, welches diese Probleme ebenfalls lösen muss (z.B. einen Compiler)

Struktur Moderner Compiler



- Scanner und Parser behandeln den komplexen Syntax
- Bei der Erzeugung der Zwischensprache wird die komplexe Semantik inklusive der Grenzfälle umgesetzt
- Nach Anwendung von Optimierungen wird die Zwischensprache in Maschinencode übersetzt

Verifikation auf der Ebene der Zwischensprache

- Das Verifikationssystem auf der Ebene eine Zwischensprache operieren zu lassen hat viele Vorteile:
  - Die Zwischensprache hat in der Regel einen einfachen Syntax und eine weniger komplexe Semantik
  - Das Verifikationssystem kann auf Programmen eingesetzt werden die in unterschiedlichen Programmiersprachen geschrieben wurden
    - Voraussetzung: Es gibt ein Front-End das die Programmiersprache in die Zwischensprache übersetzt
  - Das untersuchte Programm ist “näher” am ausgeführten Programm
- Viele in der letzten Jahren entwickelte Verifikationssysteme verfolgen diesen Ansatz

LLVM

- LLVM ist ein beliebtes Compiler Framework
- Front-Ends für
  - C, C++, Objective-C, Objective-C++
  - Fortran
  - Ada
  - D, Lua, Hydra, ...
- Zwischensprache:
  - RISC-artige Assemblersprache für eine Registermachine mit einer unbeschränkten Anzahl von Registern
  - Programme sind in SSA-Form (static single assignment)
    - Jedes Register wird im Programmcode nur einmal geschrieben
- Codegeneratoren für
  - X86
  - PowerPC
  - ARM
  - Mips
  - Sparc
  - AArch64, Hexagon, MBlaze, MSP430, NVPTX, R600, SystemZ, XCore, ...

Beispiel

```
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
}
```

Von LLVM nach QF_ABV 1

- Im Moment: Funktionen mir nur einem Basic Block
- Funktionen ohne Speicherzugriffe (`load` und `store`) können praktisch direkt als Formeln in der Bitvektorlogik interpretiert werden

Beispiel

```
void f(int x, int y) {
    assert((x − y > 0) == (x > y));
}
```

```
define void @f(i32 %x, i32 %y) {
entry:
  %sub = sub nsw i32 %x, %y
  %cmp = icmp sgt i32 %sub, 0
  %conv = zext i1 %cmp to i32
  %cmp1 = icmp sgt i32 %x, %y
  %conv2 = zext i1 %cmp1 to i32
  %cmp3 = icmp eq i32 %conv, %conv2
  %conv4 = zext i1 %cmp3 to i32
  call void @assert(i32 %conv4)
  ret void
}
```

Assertion kann fehlschlagen
gdw.
Formel erfüllbar

$$
\begin{aligned}
& sub = \texttt{bvsub}\ x\ y \\
\wedge\ & cmp = (\texttt{bvsgt}\ sub\ \texttt{bv}_{32,0})\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv = \texttt{zero\_extend}_{31}\ cmp \\
\wedge\ & cmp1 = (\texttt{bvsgt}\ x\ y)\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv2 = \texttt{zero\_extend}_{31}\ cmp1 \\
\wedge\ & cmp3 = (conv = conv2)\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv4 = \texttt{zero\_extend}_{31}\ cmp3 \\
\wedge\ & conv4 = \texttt{bv}_{32,0}
\end{aligned}
$$

Von LLVM nach QF_ABV 2

- Speicher kann als großes Array von Bytes modelliert werden
- In LLVM ist der Speicherzustand implizit, in der Formel muss er explizit gemacht werden

Beispiel

```
void f(int *p, int *q) {
    *p = 10;
    *q = 100;
    assert(*p == 10);
}
```

```
define void @f(i32* %p, i32* %q) {
entry:
  store i32 10, i32* %p
  store i32 100, i32* %q
  %0 = load i32* %p
  %cmp = icmp eq i32 %0, 10
  %conv = zext i1 %cmp to i32
  call void @assert(i32 %conv)
  ret void
}
```

Assertion kann fehlschlagen
gdw.
Formel erfüllbar

$$
\begin{aligned}
& a_1 = \text{write}(a_0, p, \texttt{bv}_{32,10}) \\
\wedge\ & a_2 = \text{write}(a_1, q, \texttt{bv}_{32,100}) \\
\wedge\ & n_0 = \text{read}(a_2, p) \\
\wedge\ & cmp = (n_0 = \texttt{bv}_{32,10})\ ?\ \texttt{bv}_{1,1}\ :\ \texttt{bv}_{1,0} \\
\wedge\ & conv = \texttt{zero\_extend}_{31}\ cmp \\
\wedge\ & conv = \texttt{bv}_{32,0}
\end{aligned}
$$