LLVM

Ein Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

Compiler II: Fortgeschrittene Themen
4. Juni 2013

Julian Oppermann

Eingebettete Systeme und Anwendungen, Technische Universität Darmstadt

Embedded Systems & Applications

Was ist LLVM (nicht)?

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM ist **kein** Compiler!
- Die LLVM-IR ist eine Zwischendarstellung, auf die man eine Vielzahl von Quellsprachen abbilden kann.
- “The LLVM Compiler Infrastructure Project” koordiniert die Entwicklung der IR und vieler weiterer Unterprojekte.
- Typischer Einsatz als Compiler:
  `clang` liest Quelltext und *benutzt* die LLVM-Bibliotheken zum Aufbau der IR, Optimierung und Codeerzeugung.

LLVM-Projekt
Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

LLVM Core	LLVM-IR, Analysen, Optimierungen, Codeerzeugung
clang	C/C++/Objective-C Frontend
dragonegg	Schnittstelle zu den Frontends der GCC
LLDB	Debugger
libc++	C++-Standardbibliothek
compiler-rt	Laufzeitumgebung für Architekturen, denen bestimmte Instruktionen fehlen
vmkit	LLVM-basierte virtuelle Maschinen für Java und .NET
polly	Schleifentransformationen zur Verbesserung der Cachelokalität und zur automatischen Parallelisierung
	⋮

LLVM-Projekt
Geschichte

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM stand ursprünglich für **L**ow **L**evel **V**irtual **M**achine.
  - Aktuell wird nur noch das Akronym verwendet, da das Projekt mittlerweile viel umfassender geworden ist.
- Begonnen Dezember 2000 von Chris Lattner und Vikram Adve an der University of Illinois.
- Heute ein erfolgreiches Open Source-Projekt unter BSD-kompatibler Lizenz.
- Gewinner des ACM Software System Award 2012.
- Bekannte Unterstützer aus der Industrie: Apple (LLVM ist Grundlage von Apples Entwicklungswerkzeugen), Google.
- Morgen (5.6.): Release von LLVM 3.3!

LLVM-Projekt

Was macht LLVM besonders?

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM-IR ist eigenständige Sprache, enthält alle Programminformationen.
- Ermöglicht echte Entkopplung von Frontend, Optimierungen und Codeerzeugung.
  - Im Gegensatz zu allen anderen Sprachimplementierungen zu Beginn der Entwicklung!

Embedded Systems & Applications

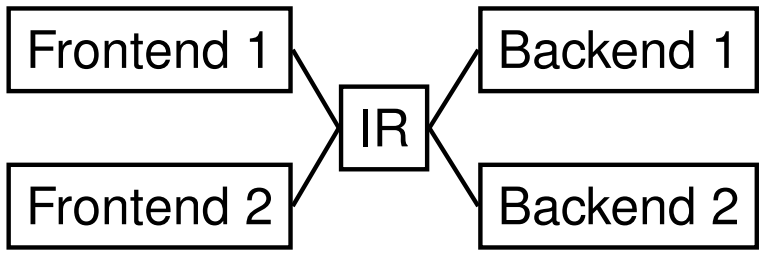
LLVM-Projekt

Was macht LLVM besonders?

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM-IR ist eigenständige Sprache, enthält alle Programminformationen.
- Ermöglicht echte Entkopplung von Frontend, Optimierungen und Codeerzeugung.
  - Im Gegensatz zu allen anderen Sprachimplementierungen zu Beginn der Entwicklung!

ideal:



praktisch:

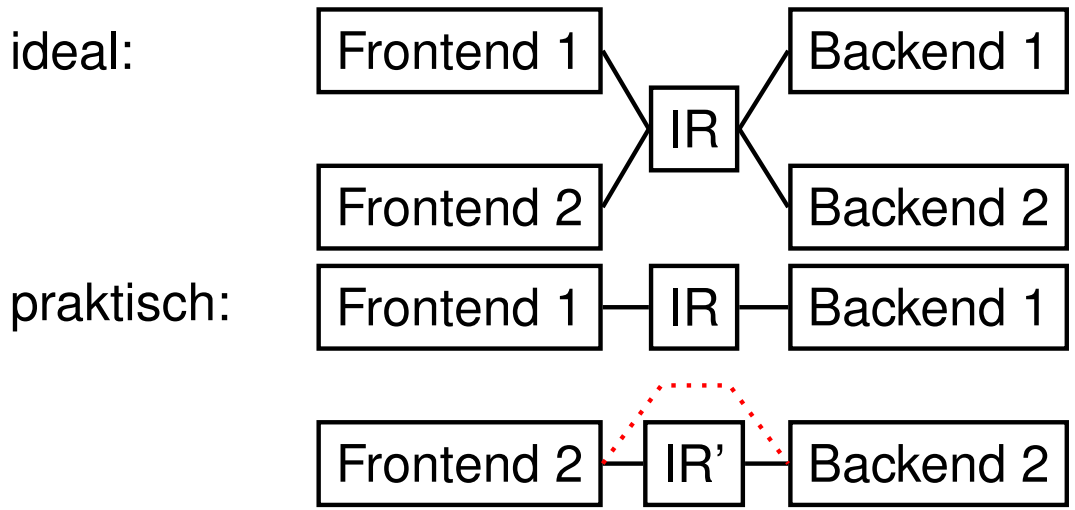
Embedded Systems & Applications

LLVM-Projekt

Was macht LLVM besonders?

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM-IR ist eigenständige Sprache, enthält alle Programminformationen.
- Ermöglicht echte Entkopplung von Frontend, Optimierungen und Codeerzeugung.
  - Im Gegensatz zu allen anderen Sprachimplementierungen zu Beginn der Entwicklung!



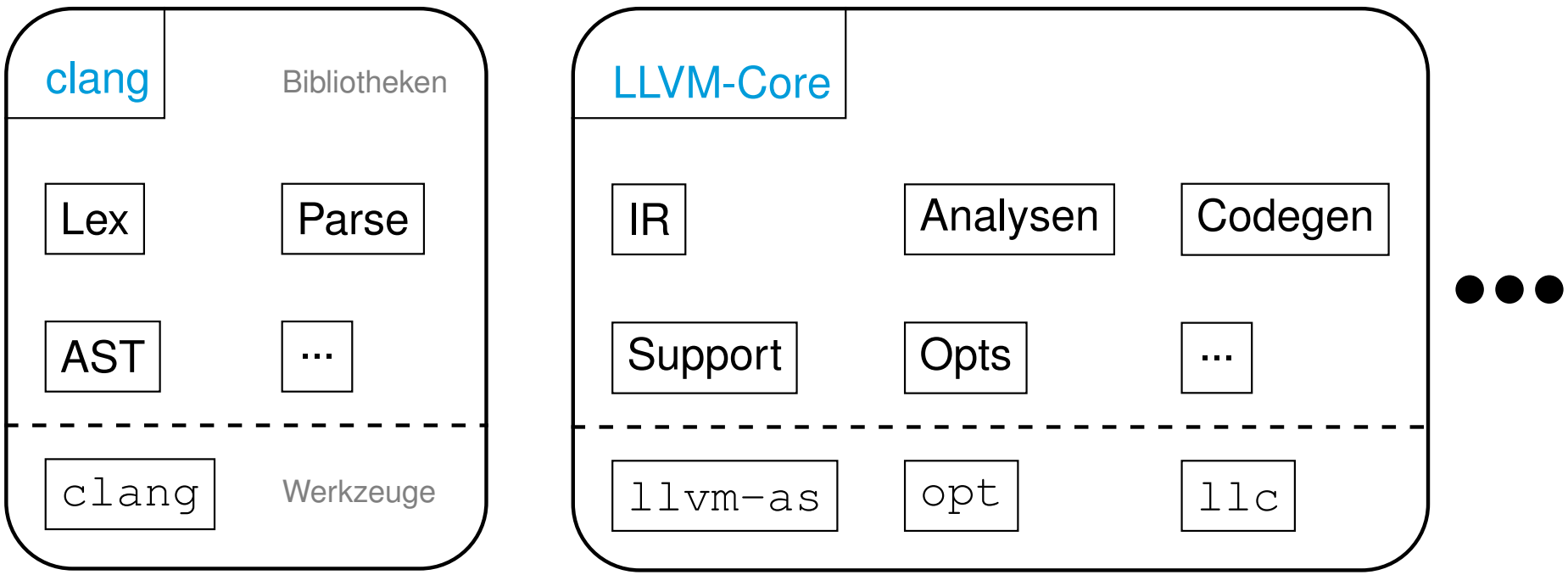
Embedded Systems & Applications

LLVM-Projekt

Was macht LLVM besonders?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Modulare Architektur, bestehend aus wiederverwendbaren Bibliotheken



Embedded Systems & Applications

LLVM-Projekt

Was macht LLVM besonders?

TECHNISCHE UNIVERSITÄT DARMSTADT

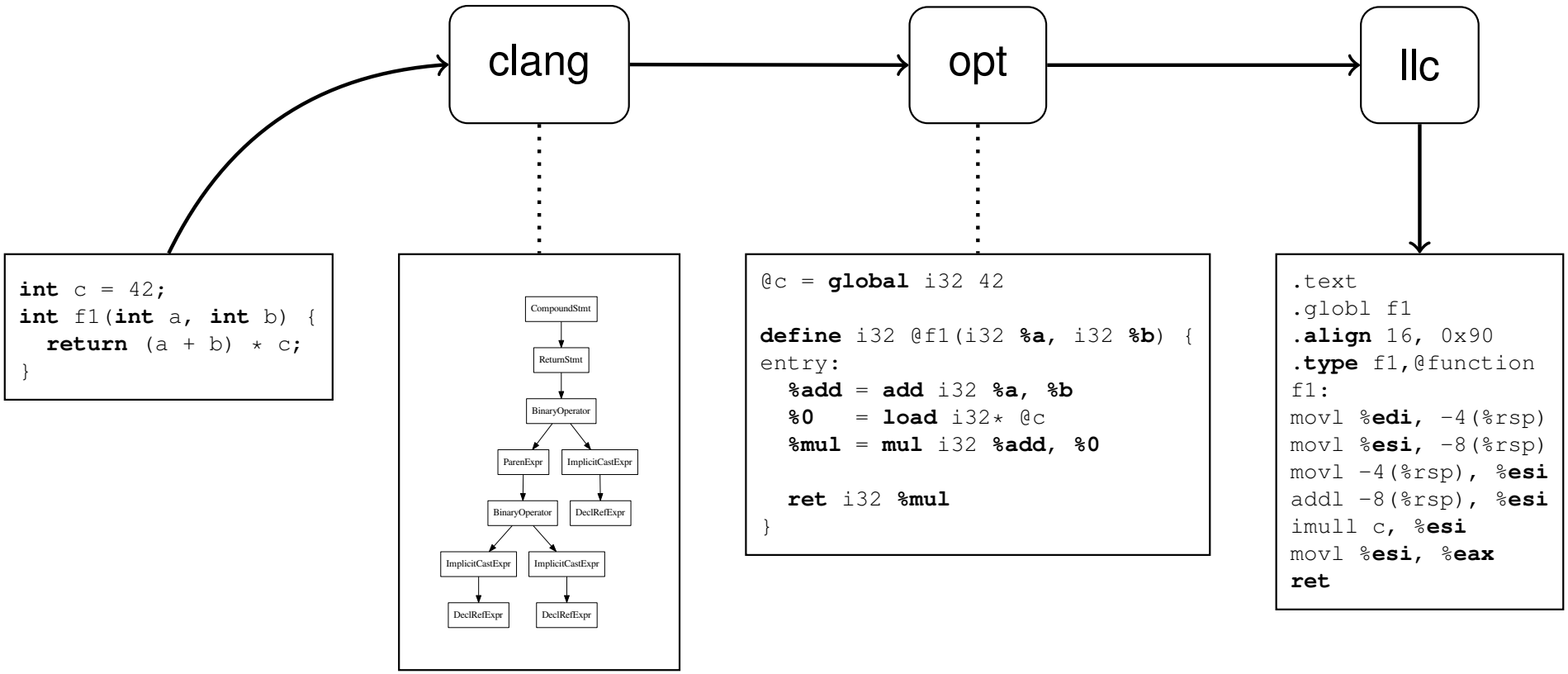
- clang
  - Schneller als gcc (compile time)
  - Bessere Fehlermeldungen

```
$ gcc-4.2 -fsyntax-only t.c
t.c:7: error: invalid operands to binary + (have 'int' and 'struct A')
$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
  return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                      ~~~~~~~~~~~~~~~ ^ ~~~~~
```

- LLVM Core
  - Aggressive skalare Optimierungen
  - Link-time optimization
  - “easily hackable”

Übersicht

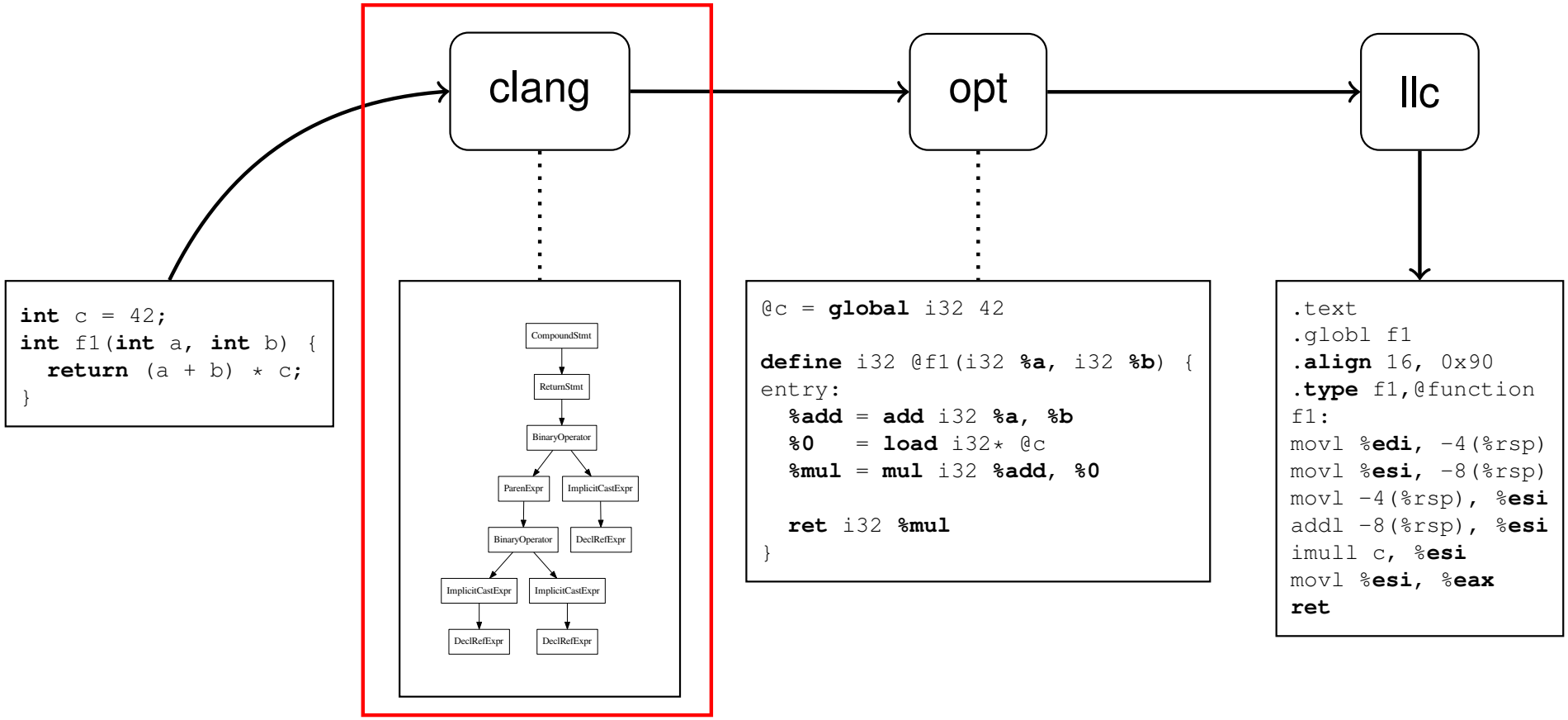
TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

clang
Parser

TECHNISCHE UNIVERSITÄT DARMSTADT

- Benutzt einen handgeschriebenen Parser nach dem Prinzip des rekursiven Abstiegs.
- Beispiel: if-Statement (stark vereinfacht)

```
StmtResult Parser::ParseIfStatement(SourceLocation *TrailingElseLoc) {
  SourceLocation IfLoc = ConsumeToken();  // eat the 'if'.

  if (Tok.isNot(tok::l_paren)) return StmtError();

  if (ParseParenExprOrCondition(CondExp, CondVar, IfLoc, true)) return StmtError();

  StmtResult ThenStmt(ParseStatement(&InnerStatementTrailingElseLoc));

  StmtResult ElseStmt;
  if (Tok.is(tok::kw_else))
    ElseStmt = ParseStatement();

  return Actions.ActOnIfStmt(...);
}
```

clang
AST

TECHNISCHE UNIVERSITÄT DARMSTADT

- Klassenhierarchien für Deklarationen (`Decl`), Anweisungen (`Stmt`) und Typen (`Type`).
  - Ausdrücke (`Expr`) sind Unterklassen von `Stmt`.
- Wurzelknoten ist `TranslationUnitDecl`.
- Keine gemeinsame Oberklasse, jeder Knotentyp spezifiziert seine eigenen Zugriffsmethoden:

```
class IfStmt : public Stmt {
  ...
  Expr *getCond() { return reinterpret_cast<Expr*>(SubExprs[CO
  Stmt *getThen() { return SubExprs[THEN]; }
  Stmt *getElse() { return SubExprs[ELSE]; }
  ...
}
```

- Traversierung mittels RecursiveASTVisitor (“Makromonster”).

clang
AST (Beispiel)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int
addabs(int a,
    int b)
{
  int x;
  if (a*b >= 0)
    x = a+b;
  else
    x = a-b;
  return x;
}
```

```
TranslationUnitDecl 0x5ff6ba0 <<invalid sloc>>
`-FunctionDecl 0x5ff75d0 <../llvm-vortrag/cfg.c:1:1, line:8:1> addabs 'int (int, int)'
  |-ParmVarDecl 0x5ff7490 <line:1:12, col:16> a 'int'
  |-ParmVarDecl 0x5ff7500 <col:19, col:23> b 'int'
  `-CompoundStmt 0x6023f10 <col:26, line:8:1>
    |-DeclStmt 0x5ff76e8 <line:2:2, col:7>
    | `-VarDecl 0x5ff7690 <col:2, col:6> x 'int'
    |-IfStmt 0x6023e80 <line:3:2, line:6:9>
    | |-<<<NULL>>>
    | |-BinaryOperator 0x5ff77c8 <line:3:6, col:13> 'int' '>='
    | | |-BinaryOperator 0x5ff7780 <col:6, col:8> 'int' '*'
    | | | |-DeclRefExpr 0x5ff7700 <col:6> 'int' lvalue ParmVar 0x5ff7490 'a' 'int'
    | | | `-DeclRefExpr 0x5ff7728 <col:8> 'int' lvalue ParmVar 0x5ff7500 'b' 'int'
    | | `-IntegerLiteral 0x5ff77a8 <col:13> 'int' 0
    | |-BinaryOperator 0x6023d60 <line:4:3, col:9> 'int' '='
    | | |-DeclRefExpr 0x5ff77f0 <col:3> 'int' lvalue Var 0x5ff7690 'x' 'int'
    | | `-BinaryOperator 0x5ff7898 <col:7, col:9> 'int' '+'
    | |   |-DeclRefExpr 0x5ff7818 <col:7> 'int' lvalue ParmVar 0x5ff7490 'a' 'int'
    | |   `-DeclRefExpr 0x5ff7840 <col:9> 'int' lvalue ParmVar 0x5ff7500 'b' 'int'
    | `-BinaryOperator 0x6023e58 <line:6:3, col:9> 'int' '='
    |   |-DeclRefExpr 0x6023d88 <col:3> 'int' lvalue Var 0x5ff7690 'x' 'int'
    |   `-BinaryOperator 0x6023e30 <col:7, col:9> 'int' '-'
    |     |-DeclRefExpr 0x6023db0 <col:7> 'int' lvalue ParmVar 0x5ff7490 'a' 'int'
    |     `-DeclRefExpr 0x6023dd8 <col:9> 'int' lvalue ParmVar 0x5ff7500 'b' 'int'
    `-ReturnStmt 0x6023ef0 <line:7:2, col:9>
      `-DeclRefExpr 0x6023eb0 <col:9> 'int' lvalue Var 0x5ff7690 'x' 'int'
```

clang
Codeerzeugung (LLVM-IR)

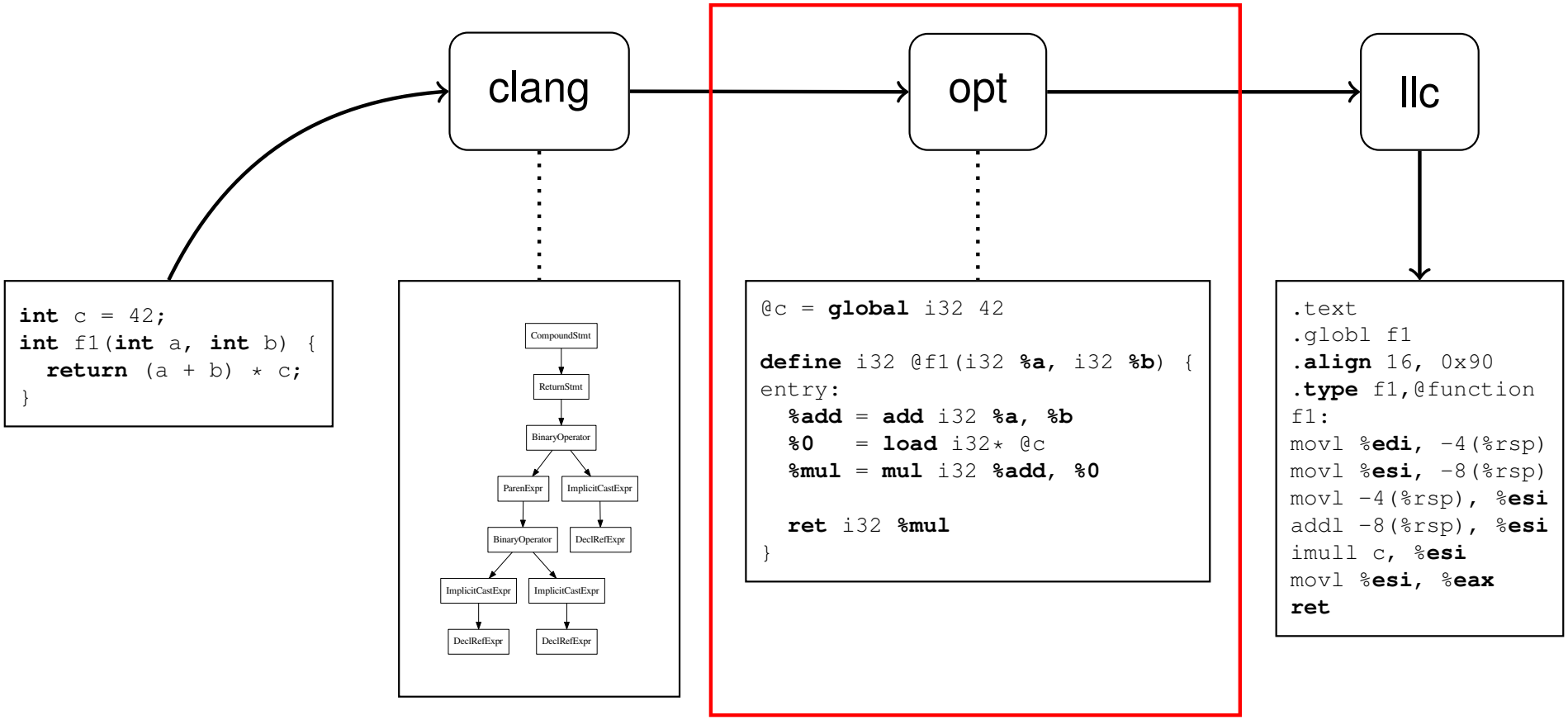
TECHNISCHE UNIVERSITÄT DARMSTADT

- (Gekürzte) LLVM-IR-Generierung für ein If-Statement

```
void CodeGenFunction::EmitIfStmt(const IfStmt &S) {
  llvm::BasicBlock *ThenBlock = createBasicBlock("if.then");
  llvm::BasicBlock *ContBlock = createBasicBlock("if.end");
  llvm::BasicBlock *ElseBlock = ContBlock;
  if (S.getElse())
    ElseBlock = createBasicBlock("if.else");
  EmitBranchOnBoolExpr(S.getCond(), ThenBlock, ElseBlock);

  EmitBlock(ThenBlock);
  EmitStmt(S.getThen());
  EmitBranch(ContBlock);

  if (const Stmt *Else = S.getElse()) {
    EmitBlock(ElseBlock);
    EmitStmt(Else);
    EmitBranch(ContBlock);
  }

  EmitBlock(ContBlock, true);
}
```

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```


Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42

define i32 @f1(i32 %a, i32 %b) {
entry:
  %add = add i32 %a, %b
  %0   = load i32* @c
  %mul = mul i32 %add, %0

  ret i32 %mul
}
```


Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42
```

Globale Variable

```
define i32 @f1(i32 %a, i32 %b) {
entry:
  %add = add i32 %a, %b
  %0   = load i32* @c
  %mul = mul i32 %add, %0

  ret i32 %mul
}
```

Funktionsdefinition

Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42
```

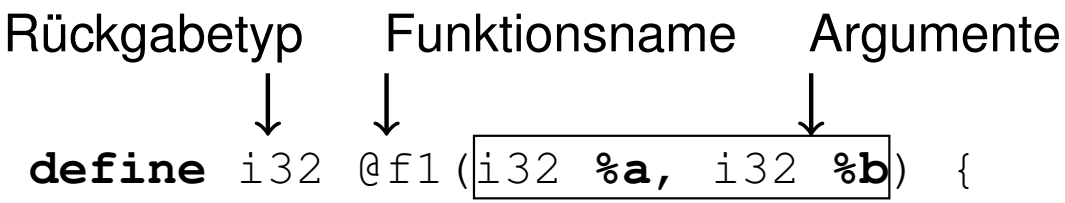
Name — Typ — Initialer Wert

Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```



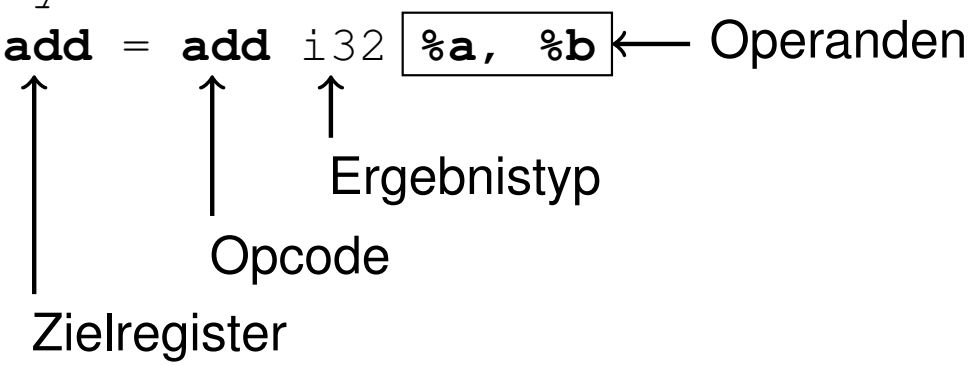
Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
define i32 @f1(i32 %a, i32 %b) {
entry:
  %add = add i32 %a, %b
```



Embedded Systems & Applications

LLVM-IR
Ein Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42

define i32 @f1(i32 %a, i32 %b) {
entry:
  %add = add i32 %a, %b
  %0   = load i32* @c
  %mul = mul i32 %add, %0

  ret i32 %mul
}
```

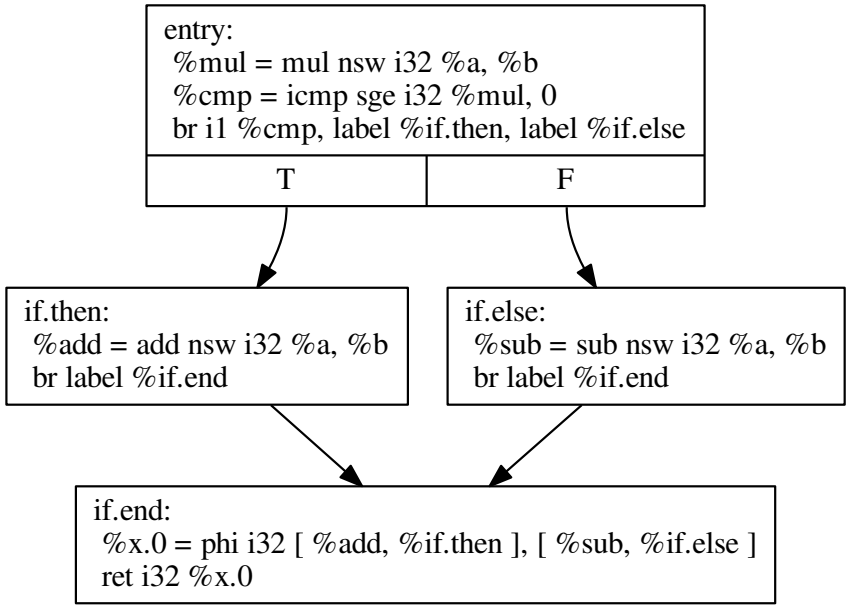

Embedded Systems & Applications

LLVM-IR

Beispiel mit Steuerfluss

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int addabs(int a, int b) {
  int x;
  if (a*b >= 0)
    x = a+b;
  else
    x = a-b;
  return x;
}
```



CFG for 'addabs' function

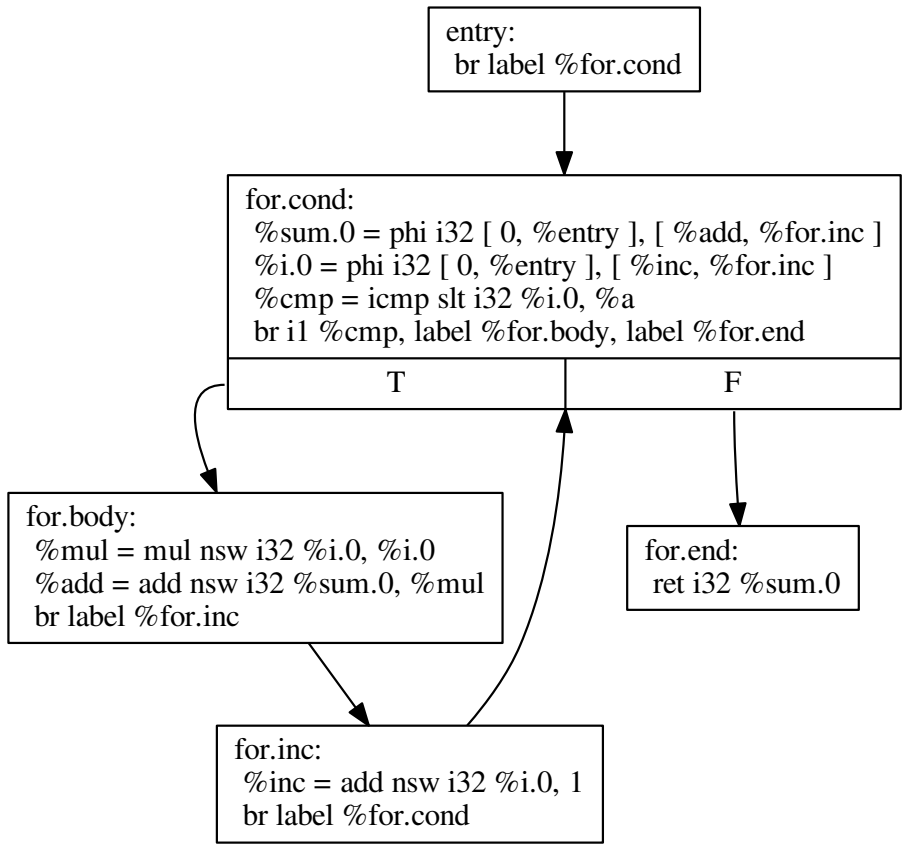
Embedded Systems & Applications

LLVM-IR

Beispiel mit Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sumsqares(int a) {
  int sum = 0, i;
  for (i=0; i<a; i++) {
    sum += i*i;
  }
  return sum;
}
```



CFG for 'sumsqares' function

Embedded Systems & Applications

LLVM-IR

“Virtuelle Maschine”

TECHNISCHE UNIVERSITÄT DARMSTADT

- Erkenntnis: Sieht aus wie eine Assembler-Darstellung für einen RISC-Prozessor.
  - unendlich viele Register
  - Jedes Register kann nur einmal von einer eindeutig bestimmten Instruktion beschrieben werden ($\rightarrow$ **SSA-Form**).
  - typisiert
- “low level”: im Kontrast zu Java / .NET VMs
  - keine Klassen/Objekte
  - keine Vererbung
  - keine Polymorphie
  - kein Exception Handling
  - ...
  - **Aber:** alle diese Konstrukte lassen sich auf LLVM-IR abbilden!

LLVM-IR

Befehlssatz

TECHNISCHE UNIVERSITÄT DARMSTADT

- Steuerfluss: ret | br | switch | indirectbr | invoke | resume | unreachable
- Arithmetisch: add | fadd | sub | fsub | mul | fmul | udiv | sdiv | fdiv | urem | srem | frem | shl | lshr | ashr | and | or | xor
- Elementzugriff: extractelement | insertelement | shufflevector | extractvalue | insertvalue
- Speicher und Adressierung: alloca | load | store | fence | cmpxchg | atomicrmw | getelementptr
- Konversionen: trunc | zext | sext | fptrunc | fpext | fptoui | fptosi | uitofp | sitofp | ptrtoint | inttoptr | bitcast
- Andere: icmp | fcmp | phi | select | call | va_arg | landingpad

LLVM-IR

Befehlssatz (Besonderheiten)

TECHNISCHE UNIVERSITÄT DARMSTADT

phi Die Φ-Funktion der SSA-Form

- explizite Instruktion, Tupel von Wert und Label als Argumente
- Beispiel:

```
%x = phi i32 [ %add, %then ], [ %sub, %else ]
```

call Funktionsaufruf

- abstrahiert Aufrufkonventionen, erhält Funktionsname und -argumente als Parameter
- Beispiel: `%y = call i32 @Get_Bits(i32 1)`

getelementptr Typsichere Adressrechnung

- Berechnet die Adresse von Array- oder Struktur-Elementen
- Beispiel: `int arr[2]; arr[1] = ...` →

```
%z = getelementptr [2 x i32]* %arr, i32 0, i32 1
```


Embedded Systems & Applications

LLVM-IR
Typsystem

TECHNISCHE UNIVERSITÄT DARMSTADT

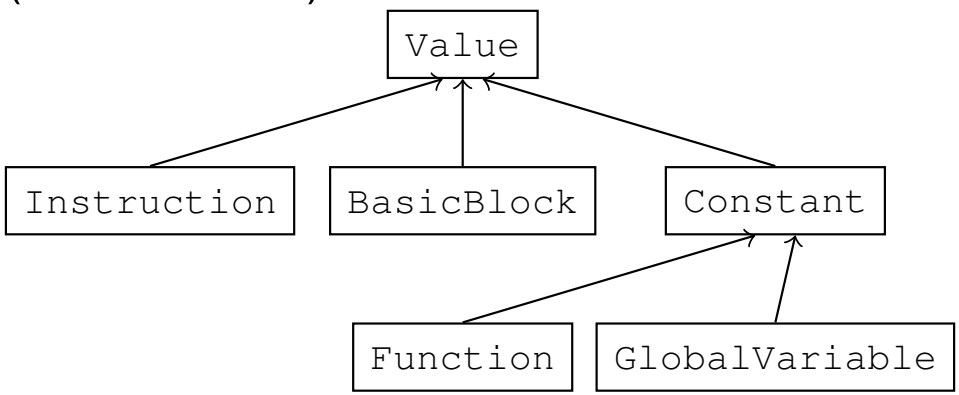
Jeder Wert hat einen Typ (unabhängig von der Quellsprache)!

- Integerwerte: `i1, i8, i16, i32, ...`
  alle Bitbreiten möglich, keine signed/unsigned-Unterscheidung
- Fließkommazahlen: `half, float, double, ...`
- Zeiger: `i64*`
- Arrays: `[10 x i32], [2 x [2 x float]]`
- Strukturen: `{i32, float, i32}`
- Vektoren (SIMD): `<i8, i8, i8, i8>`
- ...

LLVM-IR
API

TECHNISCHE UNIVERSITÄT DARMSTADT

- Oberklasse für fast alle IR-Elemente: `Value` modelliert (SSA-)Werte.
  - Jeder Wert hat einen `Type`.
- (Vereinfachte) Klassenhierarchie:



- `Instruction`s speichern ihre Operanden als Zeiger zu anderen `Value`-Objekten.
- Beispiel: Konstruktor von `BranchInst`:

```
BranchInst(BasicBlock *IfTrue, BasicBlock *IfFalse,
Value *Cond)
```

LLVM-IR
API

TECHNISCHE UNIVERSITÄT DARMSTADT

- Toplevel-Konstrukt ist das `Module`: enthält Liste von `Function`s, globalen Variablen, ...
  - `getFunctionList(), getGlobalList(), ...`
- `Function` enthält Liste von `BasicBlock`s, organisiert als Steuerflussgraph.
  - `getEntryBlock(), getBasicBlockList(), ...`
  - Vorgänger-/Nachfolgerblöcke über spezielle Iteratoren `pred_iterator`, `succ_iterator`
- `BasicBlock` enthält Liste von `Instruction`s.
  - `getTerminator(), getInstList(), ...`

API-Dokumentation: “llvm::<Klassenname>” googlen.

LLVM-IR
Darstellungsformen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt drei **äquivalente** Darstellungsformen:
  - textuell (Assembler-Format, siehe Beispiel) `prog.ll`
  - binär (Bitcode-Format) `prog.bc`
  - im Speicher (C++-Objekte)
- Jede Darstellungsform enthält stets alle Details des Programms
  $\rightarrow$ klare Schnittstelle für Analysen und Transformationen.
- Für alle Analysen und Transformation wird ausschließlich diese IR verwendet.

Optimierungen
Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM-IR enthält genug Informationen, um auch “high-level” Analysen und Transformationen durchzuführen.
- Gekapselt als Pässe.
- Auszug aus der Liste der mitgelieferten Pässe:
  - Analysen: (Post-)Dominatorbaum, natürliche Schleifen, Aliasanalyse(n), . . .
  - Transformationen: Dead Code Elimination, Reassociation, Loop Invariant Code Motion, Global Value Numbering, . . .
- Abhängigkeiten zwischen Pässen werden automatisch aufgelöst.
- Man kann sogar Transformationen einzeln auf ein Programm anwenden:

```
$ opt -S -reassociate -o prog_opt.ll prog.ll
$ opt -S -licm -o prog_opt2.ll prog_opt.ll
```

Optimierungen
Konstantenpropagation

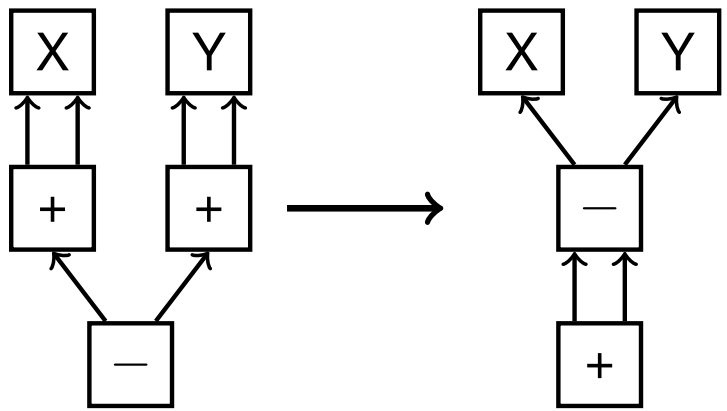
TECHNISCHE UNIVERSITÄT DARMSTADT

```
while (!WorkList.empty()) {
  Instruction *I = *WorkList.begin(); WorkList.erase(WorkList.begin());

  if (Constant *C = ConstantFoldInstruction(I, TD, TLI)) {
    // Add all of the users of this instruction to the worklist, they might
    // be constant propagatable now...
    for (Value::use_iterator UI = I->use_begin(), UE = I->use_end(); UI != UE; ++UI)
      WorkList.insert(cast<Instruction>(*UI));

    // Replace all of the uses of a variable with uses of the constant.
    I->replaceAllUsesWith(C);

    // Remove the dead instruction.
    WorkList.erase(I);
    I->eraseFromParent();
  }
}
```

- Inhalt von `ConstantFoldInstruction`: “Erzeuge neue Konstante, wenn alle Operanden von `I` konstant sind”

TECHNISCHE UNIVERSITÄT DARMSTADT

Implementierung einer Peephole-Optimierung



$(X + X) - (Y + Y)$ $\qquad$ $(X - Y) + (X - Y)$

(aus: Bersch, Thomas: Generierung lokaler Optimierungen. Diplomarbeit, 2012)

Optimierungen
Pass-Implementierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class SpecialSub : public FunctionPass {
  static char ID;
  SpecialSub() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) {
    for (inst_iterator II = inst_begin(F), IE = inst_end(F);
         II != IE; ++II)
      performSpecialSubOptimization(&*II);
    return true;
  }
};

static RegisterPass<SpecialSub> X("specialsub",
  "Special␣subtraction␣transformation␣pattern");
```


Embedded Systems & Applications

Optimierungen
Pass-Implementierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class SpecialSub : public FunctionPass {
  static char ID;
  SpecialSub() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) {
    for (inst_iterator II = inst_begin(F), IE = inst_end(F);
         II != IE; ++II)
      performSpecialSubOptimization(&*II);
    return true;
  }
};

static RegisterPass<SpecialSub> X("specialsub",
  "Special␣subtraction␣transformation␣pattern");
```

erben

Embedded Systems & Applications

Optimierungen
Pass-Implementierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class SpecialSub : public FunctionPass {
  static char ID;
  SpecialSub() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) {
    for (inst_iterator II = inst_begin(F), IE = inst_end(F);
         II != IE; ++II)
      performSpecialSubOptimization(&*II);
    return true;
  }
};

static RegisterPass<SpecialSub> X("specialsub",
  "Special␣subtraction␣transformation␣pattern");
```

erben

implementieren

Embedded Systems & Applications

Optimierungen
Pass-Implementierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class SpecialSub : public FunctionPass {
  static char ID;
  SpecialSub() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) {
    for (inst_iterator II = inst_begin(F), IE = inst_end(F);
         II != IE; ++II)
      performSpecialSubOptimization(&*II);
    return true;
  }
};
```

← erben

← implementieren

```
static RegisterPass<SpecialSub> X("specialsub",
  "Special␣subtraction␣transformation␣pattern");
```

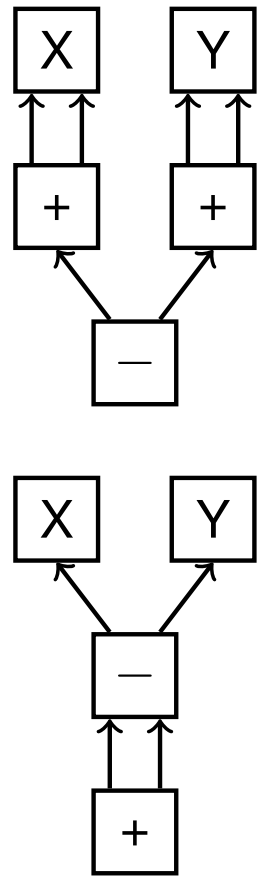
← registrieren

Embedded Systems & Applications

Optimierungen
Transformation

TECHNISCHE UNIVERSITÄT DARMSTADT

```
void performSpecialSubOptimization(Instruction *I) {
  if (match(I, m_Sub(
    m_Add(m_Value(X1), m_Value(X2)),
    m_Add(m_Value(Y1), m_Value(Y2))
    )) && X1 == X2 && Y1 == Y2) {

      newSub = CreateSub(X1, Y1, "newsub");
      newAdd = CreateAdd(newSub, newSub, "newadd");

      newSub->insertBefore(I);
      newAdd->insertBefore(I);

      I->replaceAllUsesWith(newAdd);
  }
}
```



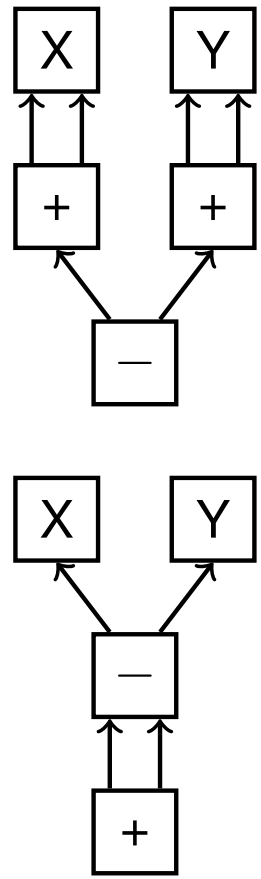
Embedded Systems & Applications

Optimierungen
Transformation

TECHNISCHE UNIVERSITÄT DARMSTADT

```
void performSpecialSubOptimization(Instruction *I) {
  if (match(I, m_Sub(
    m_Add(m_Value(X1), m_Value(X2)),
    m_Add(m_Value(Y1), m_Value(Y2))
    )) && X1 == X2 && Y1 == Y2) {

      newSub = CreateSub(X1, Y1, "newsub");
      newAdd = CreateAdd(newSub, newSub, "newadd");

      newSub->insertBefore(I);
      newAdd->insertBefore(I);

      I->replaceAllUsesWith(newAdd);
  }
}
```

⟵ Muster finden



Embedded Systems & Applications

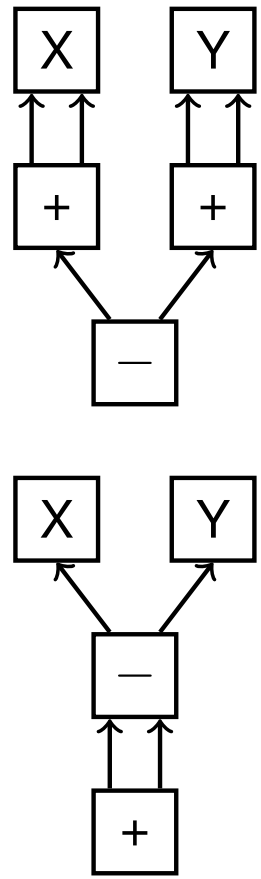
Optimierungen
Transformation

TECHNISCHE UNIVERSITÄT DARMSTADT

```
void performSpecialSubOptimization(Instruction *I) {
  if (match(I, m_Sub(
    m_Add(m_Value(X1), m_Value(X2)),
    m_Add(m_Value(Y1), m_Value(Y2))
    )) && X1 == X2 && Y1 == Y2) {

      newSub = CreateSub(X1, Y1, "newsub");
      newAdd = CreateAdd(newSub, newSub, "newadd");

      newSub->insertBefore(I);
      newAdd->insertBefore(I);

      I->replaceAllUsesWith(newAdd);
  }
}
```

⟵ Muster finden

↑erzeugen



Embedded Systems & Applications

Optimierungen
Transformation

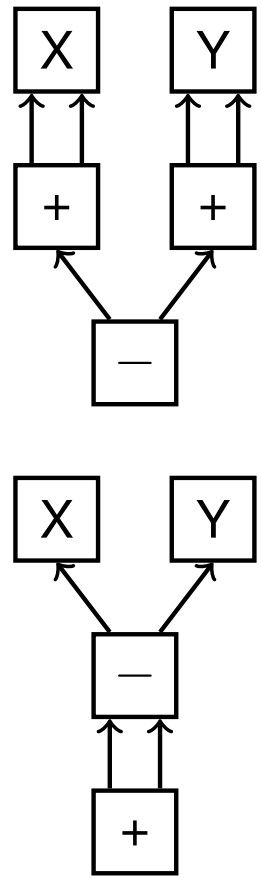
TECHNISCHE UNIVERSITÄT DARMSTADT

```
void performSpecialSubOptimization(Instruction *I) {
  if (match(I, m_Sub(
    m_Add(m_Value(X1), m_Value(X2)),
    m_Add(m_Value(Y1), m_Value(Y2))
    )) && X1 == X2 && Y1 == Y2) {

      newSub = CreateSub(X1, Y1, "newsub");
      newAdd = CreateAdd(newSub, newSub, "newadd");

      newSub->insertBefore(I);
      newAdd->insertBefore(I);

       I->replaceAllUsesWith(newAdd);
  }
}
```

← Muster finden

↑erzeugen

← einfügen



Embedded Systems & Applications

Optimierungen
Transformation

TECHNISCHE UNIVERSITÄT DARMSTADT

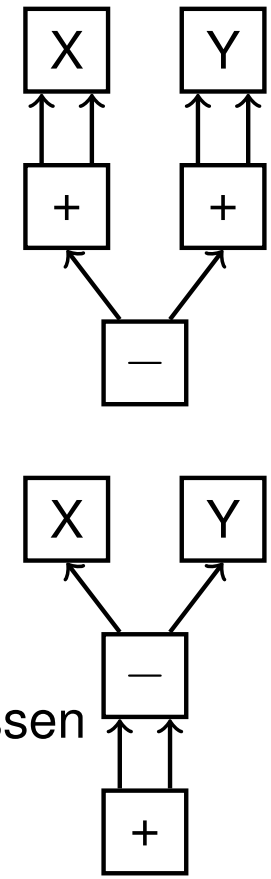
```
void performSpecialSubOptimization(Instruction *I) {
  if (match(I, m_Sub(
    m_Add(m_Value(X1), m_Value(X2)),
    m_Add(m_Value(Y1), m_Value(Y2))
    )) && X1 == X2 && Y1 == Y2) {

      newSub = CreateSub(X1, Y1, "newsub");
      newAdd = CreateAdd(newSub, newSub, "newadd");

      newSub->insertBefore(I);
      newAdd->insertBefore(I);

      I->replaceAllUsesWith(newAdd);
  }
}
```

← Muster finden

↑erzeugen

← einfügen

← Verwender anpassen



Embedded Systems & Applications

Optimierungen
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int func(int a, int b) { return (a+a)-(b+b); }
```


Embedded Systems & Applications

Optimierungen
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int func(int a, int b) { return (a+a)-(b+b); }
```

```
%add  = add i32 %a, %a
%add1 = add i32 %b, %b
%sub  = sub i32 %add, %add1
ret i32 %sub
```


Embedded Systems & Applications

Optimierungen
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int func(int a, int b) { return (a+a)-(b+b); }
```

```
%add  = add i32 %a, %a
%add1 = add i32 %b, %b
%sub  = sub i32 %add, %add1
ret i32 %sub
```

⇓ `opt -load SpecialSub.so -specialsub -dce prog.ll` ⇓

Embedded Systems & Applications

Optimierungen
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int func(int a, int b) { return (a+a)-(b+b); }
```

```
%add  = add i32 %a, %a
%add1 = add i32 %b, %b
%sub  = sub i32 %add, %add1
ret i32 %sub
```

⇓ `opt -load SpecialSub.so -specialsub -dce prog.ll` ⇓

```
%newsub = sub i32 %a, %b
%newadd = add i32 %newsub, %newsub
ret i32 %newadd
```


Embedded Systems & Applications

Optimierungen
Analysen nutzen

TECHNISCHE UNIVERSITÄT DARMSTADT

Optimierungen können Analyseinformationen anfordern:

```
virtual void getAnalysisUsage(AnalysisUsage &AU) const {
  AU.addRequired<AliasAnalysis>();
}
```

Verwendung:

```
...
AliasAnalysis &AA = getAnalysis<AliasAnalysis>();
if (AA.alias(V1, V2)) {
...
```

Aliasanalyse
Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

- Einheitliche Schnittstelle für ...
  - Alias-Anfragen: $(\text{Addr}, \text{Size}) \times (\text{Addr}, \text{Size}) \rightarrow \{\text{No}, \text{May}, \text{Must}, \text{Partial}\}\text{Alias}$
  - ModRef-Anfragen: $(\text{Instr}, \text{Addr}, \text{Size}) \rightarrow \{\text{No}, \text{Mod}, \text{Ref}, \text{ModRef}\}$
  - `doesNotAccessMemory, onlyReadsMemory` für Funktionen
- Konvention: Analysen werden verkettet, deswegen auch sehr spezielle Verfahren möglich.
- Bequemere Verwendung der Information:
  - Memory Dependence-Analyse: findet Abhängigkeiten zwischen zustandsverändernden Instruktionen.
  - AliasSets: Einteilung von Zeigern in disjunkte Mengen.

Aliasanalyse

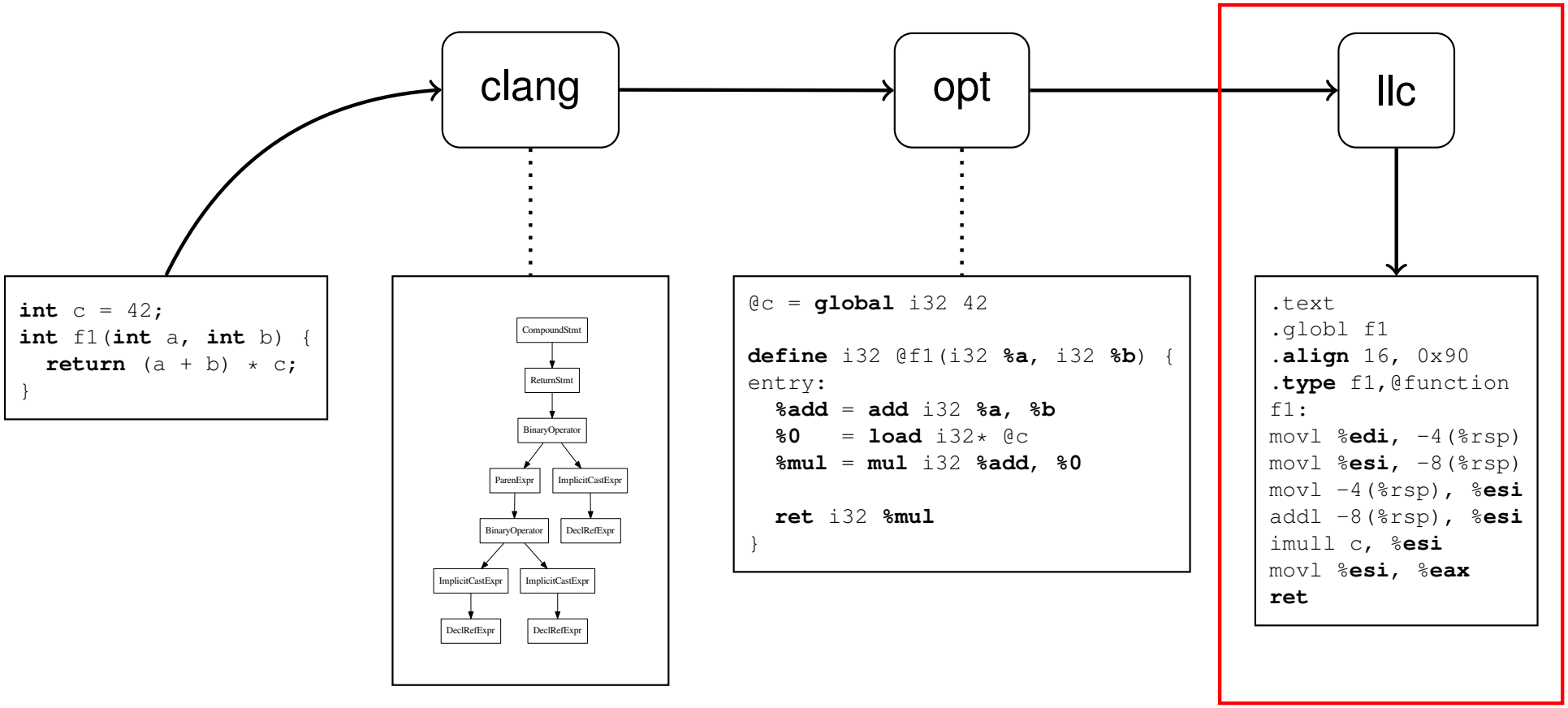
Verfügbare Analysen

TECHNISCHE UNIVERSITÄT DARMSTADT

basic	Analyse basierend auf Kenntnis von Fakten, z.B. *“globale Variablen und allozierter Speicher auf Stack und Heap aliasen nicht und sind nie* `NULL`*”*.
steens	Steensgard Points-to-Analyse.
ds	Data Structure Analysis (von den LLVM-Autoren).
scev	Einbeziehung der Scalar-Evolution-Information, d.h. Entwicklung von Werten in Schleifen.
globalsmodref	Spezielle Analyse für globale Variablen, deren Adressen nicht ausgewertet werden.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

Codeerzeugung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Verfügbare Targets im Backend: x86, ARM, PowerPC, SPARC, MIPS, ...
- Abstrakte Beschreibung der Zielarchitektur: Befehlsformat, Registersatz, ...
  - Standardaufgaben der Codeerzeugung als architekturunabhängige Pässe verfügbar.
  - Instruktionsauswahl lässt sich größtenteils generieren.
  - Target-unabhängige JIT-Codeerzeugung

Mehr dazu beim nächsten Mal

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- LLVM ist eine Sammlung von Bibliotheken und Werkzeugen zur Compilerentwicklung.
- LLVM-IR $\approx$ Assembler für virtuelle Maschine mit typischen RISC-Befehlssatz plus ein paar Extras.
- Beim Design der Infrastruktur wurde größter Wert auf Modularität und Wiederverwendbarkeit der Komponenten gelegt.

Quellen / weiterführende Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

- http://llvm.org
- http://llvm.org/docs/LangRef.html
- http://llvm.org/docs/ProgrammersManual.html
- http://llvm.org/docs/WritingAnLLVMPass.html
- http://llvm.org/docs/WritingAnLLVMBackend.html
- http://llvm.org/docs/AliasAnalysis.html
- http://www.aosabook.org/en/llvm.html
- Chris Lattner und Vikram Adve: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation".

Backup-Folien

Cheatsheet

TECHNISCHE UNIVERSITÄT DARMSTADT

- C-Programm nach LLVM übersetzen:
  ```
  $ clang -S -emit-llvm -o prog.ll prog.c
  ```
- IR in “richtige” SSA-Form bringen:
  ```
  $ opt -S -mem2reg -o prog-ssa.ll prog.ll
  ```
- Zwischen Assembler- und Bitcode-Format konvertieren:
  ```
  $ llvm-as prog.ll # erzeugt prog.bc
  $ llvm-dis prog.bc # erzeugt prog.ll
  ```
- Codeerzeugung (= System-Assemblercode generieren)
  ```
  $ llc prog.ll # erzeugt prog.s
  ```

clang vs. gcc

TECHNISCHE UNIVERSITÄT DARMSTADT

- Codequalität: kein klarer Sieger, leichter Vorteil für gcc 4.8 ggü. clang 3.2
- gcc unterstützt mehr Sprachen und Architekturen
- clang ist modularer, schneller und braucht weniger Speicher
- Linux Kernel: muss noch gepatcht werden, ARM besser unterstützt als x86.

http://www.phoronix.com/scan.php?page=article&item=llvm_clang32_final
http://clang.llvm.org/comparison.html

http://llvm.linuxfoundation.org/index.php/Main_Page