

Praktikum Compilerbau

Sitzung 6 – libFIRM

Prof. Dr.-Ing. Gregor Snelting
Matthias Braun und Sebastian Buchwald

IPD Snelting, Lehrstuhl für Programmierparadigmen



1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
6. Typische Konstrukte
7. Hilfsmittel
8. x86-Backend
9. Sonstiges

Letzte Woche

- Was waren die Probleme?
- Hat soweit alles geklappt?

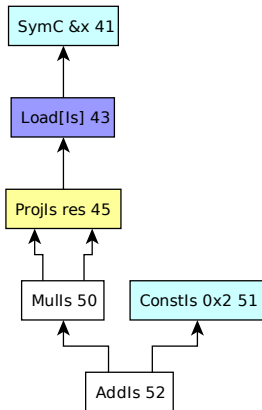
1. Letzte Woche
2. **libFirm**
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
6. Typische Konstrukte
7. Hilfsmittel
8. x86-Backend
9. Sonstiges



- libFIRM ist die Implementierung einer low-level Programmrepräsentation.
- low-level: Näher an der Maschine als an der Quellsprache.
- Komplette Graph-basiert; keine Instruktionslisten oder Tripelcode, stattdessen Datenabhängigkeiten und Steuerflussgraphen.
- Komplette SSA-basiert.
- Enthält zahlreiche Optimierungen.
- Sehr ausgereift (für ein Forschungsprojekt).

- Keine Befehlslisten – Abhängigkeitsgraphen genügen um Reihenfolge vorzugeben.
- Keine Variablen – Wir betrachten berechnete Werte; „*Namen sind Schall und Rauch*“.
- Konsistente Benutzung der SSA-Form (erzwingen durch Programmrepräsentation).
- Konstantenfaltung, CSE, DCE, algebraische Identitäten werden On-The-Fly optimiert (keine separate Phase notwendig).

Datenabhängigkeiten



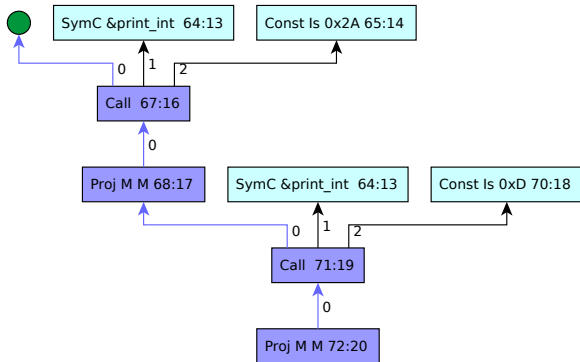
Beispiel: $x*x + 2$

- Operationen sind Knoten in einem Graph.
- Kanten geben Datenabhängigkeiten an.

„Speicher“-abhängigkeiten

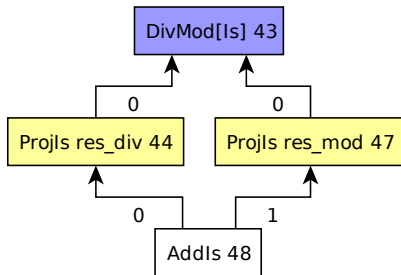
Beispiel: `print_int(42); print_int(13);`

- Operationen können Nebeneffekte haben (Speicher verändern, Bildschirmausgaben).
- Ordnung muss durch weitere Abhängigkeiten erzwungen werden.

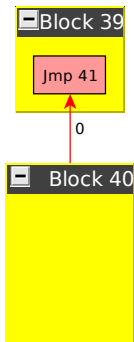


Tupelwerte und Projektionsknoten

Beispiel: $x/y + x\%y$



- Manche Operationen liefern mehrere Werte zurück. Dieser werden in einem Tupelwert zusammengefasst.
- Mit Hilfe der (virtuellen) Proj-Operation kann man einzelne Werte aus einem Tupel extrahieren.



- Grundblöcke sind normale Knoten, die Sprungbefehle als Vorgänger besitzen.
- Jeder Knoten ist einem Grundblock zugeordnet (Ausnahme Grundblöcke)

- Jeder Knoten hat einen festgelegten Mode der angibt was von einer Operation produziert wird.
- Moditypen werden im Graphen als unterschiedliche Farben dargestellt:
 - schwarz – Datenwerte
 - blau – Speicher/Synchronisation
 - rot – Steuerfluss
- Modi werden im Namen des Knotens mit angegeben: AddIs ist ein „Add“-Knoten mit Modus „Is“ (Integer Signed).

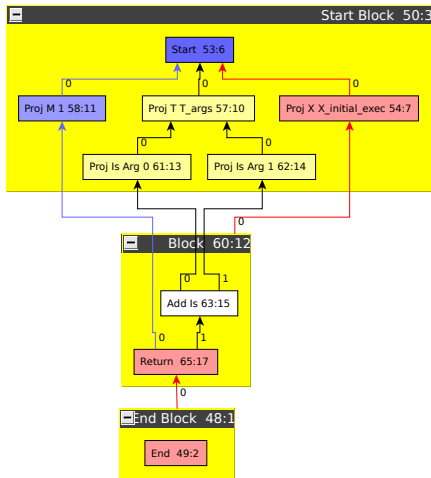
Typische Modi

Bezeichnung	Bitbreite	Vorzeichen	Art
Bs	8	Ja	Ganzzahl
Bu	8	Nein	Ganzzahl
Hs	16	Ja	Ganzzahl
Hu	16	Nein	Ganzzahl
Is	32	Ja	Ganzzahl
Iu	32	Nein	Ganzzahl
P	32	Nein	Zeiger auf Daten
F	32	Ja	Fließkomma
D	64	Ja	Fließkomma
b			(interne) Wahrheitswerte
X			Steuerfluss
M			Speicher/Synchronisation
T			Tupelwerte

- Eine Funktion beginnt am Start-Knoten im Startblock.
- Der Startknoten erzeugt einen initialen Speicherwert und die Funktionsargumente.
- Sie endet am End-Knoten im Endblock.
- Der Endknoten hat Return-Operationen als Vorgänger.

Beispiel: Komplette Methode

Beispiel: `int f(int a, int b) { return a + b; }`



1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
6. Typische Konstrukte
7. Hilfsmittel
8. x86-Backend
9. Sonstiges

- Zu jedem Programm existiert ein (minimales) Typsystem um Methoden und Datenstrukturen zu typisieren.
- Typen:
 - Primitive „Atomare“ Datentypen, Werte haben genau einen Firm Mode.
 - Method Beschreibt Methoden: Gibt Anzahl der Parameter und Rückgabewerte, sowie deren Typen an.
 - Pointer Zeiger/Referenz auf einen anderen Typ.
 - Struct Zusammengesetzter Datentyp. Enthält eine Liste von Entitäten. Adressen der Entitäten dürfen nicht überlappen.
 - Union Zusammengesetzter Datentyp. Enthält eine Liste von Entitäten. Adressen der Entitäten dürfen überlappen.
 - Class Zusammengesetzter Datentyp. Enthält eine Liste von Entitäten darf im Gegensatz zu Struct und Union auch Methoden enthalten.

Entitäten (Entities)

Eine Entität (Entity) beschreibt ein Objekt im Arbeitsspeicher:

- Typ des Objekts
- (relative) Adresse im Arbeitsspeicher
- Elterntyp (Entitäten sind stets einem Typ zugeordnet)
- (optional) Länge
- (optional) zugehöriger Firm-Graph
- (optional) initiale Wertebelegung

Typische Entitäten:

- Methoden
- globale Variablen
- Felder in einem Struct-, Union- oder Class-Type.

Hierarchie

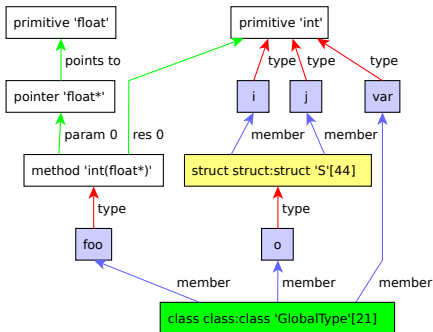
- Entitäten sind stets Kinder eines Compound-Typs (Klassentyp, Structtyp, ...)
- Für „globale“ Entitäten existiert ein spezieller Typ Namens „global“.

Sichtbarkeit Für globale Entitäten lässt sich die Sichtbarkeit für den Linker festlegen. Es lassen sich außerdem virtuelle Entitäten für externe Objekte in anderen Übersetzungseinheiten anlegen.

- **Visibility:**
 - **Local:** Definition und Sichtbarkeit auf Objektdatei beschränkt.
 - **Default:** Definiert; für andere Objektdateien sichtbar.
 - **External:** In fremder Objektdatei definiert (und sichtbar gemacht).
- **LdName (mangled) Linker Name.** Wird für Linker benutzt. Laufzeitumgebungen haben hier unterschiedliche Konventionen. (main unter Linux, _main unter Windows, Mac)

Beispiel Entities / Typen

```
int var;  
struct S { int i, j; };  
struct S o;  
extern int foo(float* x);
```



1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
- 5. Firm-Graph Aufbau**
6. Typische Konstrukte
7. Hilfsmittel
8. x86-Backend
9. Sonstiges

Probleme beim Erzeugen von Firm-Graphen aus einem AST

- Transformation der expliziten Ausführungsreihenfolge in Abhängigkeitsgraphen.
- SSA-Aufbau – platzieren der ϕ -Funktionen.
- Ersetzen von Variablen durch Use-Def-Beziehungen.

Firm kommt mit einigen Hilfsmitteln um diesen Aufbau zu vereinfachen.

Firm initialisieren

Initialisieren

```
Firm.init();  
System.out.println("Initialized libFirm Version: %1s.%2s\n",  
    Firm.getMinorVersion(),  
    Firm.getMajorVersion());
```

Typen/Entities Erzeugen

MethodType: Erzeuge Methodentyp mit 2 Integer Parametern und einem Fließkomma Rückgabewert.

```
PrimitiveType intType = new PrimitiveType(Mode.getIs());  
PrimitiveType floatType = new PrimitiveType(Mode.getF());  
MethodType methodType = new MethodType(new Type[] {intType, intType},  
                                         new Type[] {floatType});
```

Methoden Entity: Methode foo mit obigem Typ.

```
Type globalType = Program.getGlobalType();  
Entity methodEntity = new Entity(globalType, "foo", methodType);
```

Begin/Ende der Graphkonstruktion

Begin

```
int n_vars = 23; /* lokale Variablen zaehlen */  
Graph graph = new Graph(methodEnt, n_vars);  
Construction construction = new Construction(graph);
```

- Entität für Methode erzeugen, Graph erzeugen.
- Lokale Variablen zählen und Instanz von Construction anlegen.

Ende

```
construction.finish();  
/* dump graph (optional) */  
Dump.dumpBlockGraph(graph, "-after-construction");
```

- Aufruf von `finish` erzeugt fehlende ϕ -Operationen.
- Guter Zeitpunkt um Graph auszugeben.

Erzeugen von Knoten

Konstanten 2 und 5 addieren:

```
Mode mode = Mode.getIs();  
Node c5 = construction.newConst(5, mode);  
Node c2 = construction.newConst(2, mode);  
Node add = construction.newAdd(c5, c2, mode);
```

- Bei DivMod, Load gibt es ein zusätzliches Attribut, das den Typ der berechneten/des geladenen Wertes angibt.
- Die entsprechenden Knotenklassen besitzen vordefinierte Konstanten die man als Projektionsnummern benutzen sollte (DivMod.pnResMod).

```
Node mem = construction.getCurrentMemory();
Node divmod = construction.newDivMod(memory, left, right, mode,
    op_pin_state.op_pin_state_floats);
Node projResDiv = construction.newProj(divmod, mode, DivMod.pnResDiv);
Node projResMod = construction.newProj(divmod, mode, DivMod.pnResMod);
Node projMem = construction.newProj(divmod, mode, DivMod.pnM);
construction.setCurrentMemory(projMem);
```

Befehle bei denen die Ausführungsreihenfolge wichtig ist, besitzen in Firm Speicherkanten. Während des Aufbaus zeigt deshalb `CurrentMem` auf den letzten erzeugten Speicherwert. Beispiel:

```
Node mem = construction.getCurrentMem();
Node load = construction.newLoad(mem, pointer, mode);
Node loadResult = construction.newProj(load, mode, Load.pnRes);
Node loadMem = construction.newProj(load, Mode.getM(), Load.pnM);
construction.setCurrentMem(loadMem);
```

Analog wird mit Variablen verfahren. Jeder Variable wird eine Nummer zugeordnet. Jede Nummer hat eine aktuelle Definition:

```
/* Abfrage der Variable */
```

```
int var_num = ... ;
```

```
Mode mode = ... ;
```

```
Node currentVal = construction.getVariable(var_num, mode);
```

```
/* Setzen der Variable */
```

```
int var_num = ... ;
```

```
Node value = ... ;
```

```
construction.setVariable(var_num, value);
```

Knoten werden im `CurrentBlock` erzeugt. (Nachdem erzeugen der `Construction` Klasse ist bereits der „initiale“ Block erzeugt und als `CurrentBlock` gesetzt. Beispiel:

```
/* Sprung erzeugen */  
Node jump = construction.newJump();  
  
/* Neuen Block erzeugen */  
Block newBlock = construction.newBlock();  
newBlock.addPred(jump);  
construction.setCurrentBlock(newBlock);
```

Bei den meisten Knoten ist es nicht wichtig in welchem Block sie sich befinden, so lange ihre Datenabhängigkeiten erfüllt sind. Ausnahmen sind Knoten wie Sprungbefehle, oder ϕ -Knoten. Da man bei einigen Knoten ¹ nicht direkt entscheiden kann ob der Block wichtig ist, gibt es in Firm das sogenannte „pinned“-flag:

- `op_pin_state_floats` Block ist unwichtig; Knoten kann zwischen Blöcken verschoben werden.
- `op_pin_state_pinned` Knoten darf nicht zwischen Blöcken verschoben werden.

¹Beispiel: `Div`, `Load`, `Store`

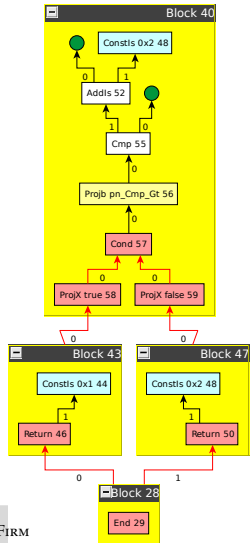
1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
- 6. Typische Konstrukte**
7. Hilfsmittel
8. x86-Backend
9. Sonstiges

Der Cmp-Knoten vergleicht 2 Werte. Alle möglichen Vergleiche werden durchgeführt und als Tupel zurückgeliefert. Proj-Nummern:

Ganzzahlig / Ordered		Unordered	
Name	Vergleich	Name	Vergleich
False	immer falsch	Uo	unordered
Eq	$x = y$	Ue	$x = y \vee$ unordered
Lt	$x < y$	Ul	$x < y \vee$ unordered
Le	$x \leq y$	Ule	$x \leq y \vee$ unordered
Gt	$x > y$	Ug	$x > y \vee$ unordered
Ge	$x \geq y$	Uge	$x \geq y \vee$ unordered
Lg	$x < y \vee x > y$	Ne	$x \neq y$
Leg	$x < y \vee x > y \vee x = y$	True	immer wahr

If-Konstruktion

Beispiel: `if(x>y+2) { return 1; } else { return 2; }`

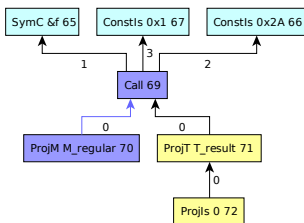


- Methoden besitzen einen impliziten `this` Parameter: Dieser muss in der Firm-Darstellung explizit vorhanden sein.
- statische Methoden (in MiniJava also genau die `main`-Methode) besitzen keinen `this` Parameter.

Funktionsaufrufe

- Adresse der aufzurufenden Funktion berechnen.
- Adresse, CurrentMem und Argumente sind Eingänge des Call-Knotens.
- Auch für den Call muss ein Methodtyp angegeben werden. Im allgemeinen der Typ der Funktion.
- Um Funktionsergebnisse abzufragen doppeltes Proj nötig!
- this-Zeiger nicht vergessen!

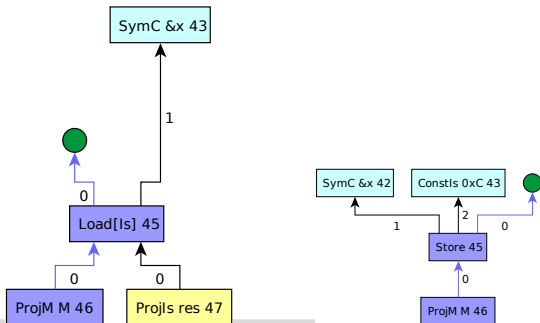
Beispiel: $f(42,1)$



Laden/Speichern

- Berechne Speicheradresse von der geladen wird / auf die geschrieben wird.
- Benutze `CurrentMem` als Speichervorgänger, nach der Operation `CurrentMem` auf Speicher-Proj setzen.

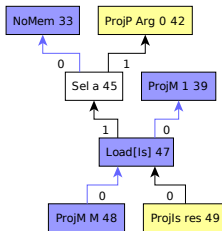
Beispiele: von Adresse der globalen Variable `x` laden; Den Wert 12 an diese Adresse schreiben.



Adresse von Feldern

- Adressberechnung kann mit Sel-Knoten erzeugt werden.
- Adressen sind relativ zu Referenz (oder this-Zeiger).
- Speicher-Eingang für uns uninteressant: Dummy-Knoten NoMem benutzen!
- Entity des Feldes als Attribut setzen.

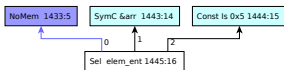
Beispiel: Laden von Feld a:



Adresse von Array-Elementen

- Adressberechnung mit Sel-Knoten.
- Adressen sind relativ zu Referenz auf das Array.
- Speicher-Eingang für uns uninteressant: Dummy-Knoten NoMem benutzen!
- weiterer Eingang für Array-Index.
- Entity aus `ArrayType.getElementEntity()`

Beispiel: Adresse von `array[5]`:



Speicher reservieren („new“)

- Speicher kann auf dem Heap oder dem Aufrufkeller mit Hilfe eines `Alloc`-Knotens erzeugt werden.
- (Freigabe mit `Free` möglich, aber in MiniJava nicht sinnvoll.)
- (Klassenkonstruktoren müssen mit separatem `Call`-Knoten aufgerufen werden.)

1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
6. Typische Konstrukte
- 7. Hilfsmittel**
8. x86-Backend
9. Sonstiges

Eingebaute Checker (irverify)

Der eingebaute Checker prüft grundlegende Korrektheitsbedingungen eines Firm-Graphen. Typische Beispiele sind:

- Vorgänger einer arithmetischen Operation haben alle denselben Mode.
- Nur Proj-Knoten als Nachfolger eines Knotens mit `mode_T`
- Proj-Nummern im erlaubten Bereich
- Modi und Anzahl von Parametern und Rückgabewerte stimmen mit den Methodentypen überein.
- ...

Der Checker läuft immer nach dem anlegen neuer Knoten und beim Ausgeben der Graphen als `.vcg-Datei`.

Graphen ausgeben, betrachten

Ausgeben

```
for(Graph g : Program.getGraphs()) {  
    /* vcg graph in Datei "GRAPHNAME-finished.vcg" ausgeben */  
    Dump.dumpBlockGraph(g, "-finished");  
}
```

Betrachten

Benutze das yComp-Tool (Link steht im Wiki).

Live-Demo

1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
6. Typische Konstrukte
7. Hilfsmittel
- 8. x86-Backend**
9. Sonstiges

Highlevel -> Lowlevel

Einige Konstruktionen können nach unserem Aufbau nicht direkt in Maschinencode abgebildet werden. Deshalb ist eine zusätzliche Lowering Phase nötig, falls das Firm x86-Backend benutzt werden soll:

- Sel-Knoten durch Adressrechnung ersetzen. Geschieht durch Aufruf von Util.lowerSels()
- Alloc-Knoten durch Aufrufe von malloc ersetzen (oder echten Garbage-Collector benutzen).
- Methoden vom ClassType in den GlobalType verschieben
- LdNames erzeugen, die nur die Zeichen [a-zA-Z0-9_] enthalten.

Achtung: Die High- nach Lowlevel Transformation darf nicht durchgeführt werden wenn Java Bytecode erzeugt wird

Benutzen des Firm x86-Backends

```
/* Erzeuge Assembler Datei foo.s (input Datei war "bla.java") */  
Backend.createAssembler("foo.s", "bla.java");  
/* Externen assembler aufrufen um Programm "foo" zu erzeugen */  
Runtime.getRuntime().exec("gcc foo.s -o foo");
```

1. Letzte Woche
2. libFirm
3. Programmdarstellung
4. Typen und Entitäten
5. Firm-Graph Aufbau
6. Typische Konstrukte
7. Hilfsmittel
8. x86-Backend
- 9. Sonstiges**

Feedback! Fragen? Probleme?

- Anmerkungen?
- Probleme?
- Fragen?