Praktikum Compilerbau
Sitzung 6 – libFirm Teil 2

Lehrstuhl für Programmierparadigmen
Universität Karlsruhe (TH)

27. Mai 2009

Letzte Woche

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

Probleme beim erzeugen von Firmgraphen 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("type_int", Mode.getIs());
PrimitiveType floatType = new PrimitiveType("type_float", Mode.getF());
MethodType methodType = new MethodType("float(int,int)", 2, 1);
methodType.setResType(0, floatType);
methodType.setParamType(0, intType);
methodType.setParamType(1, intType);
```

Methoden Entity: Methode foo mit obigem Typ.

```
Type globalType = Program.getGlobalType();
Entity methodEnt = new Entity(globalType, "foo", methodType);
methodEnt.setLdIdent("foo");
methodEnt.setVisibility(ir_visibility.visibility_external_visible);
```

Begin/Ende des Firmaufbaus

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);
```

Tupel Knoten, Projektionen

- Bei `DivMod`, `Load` gibt es ein zusätzliches Attribut, dass 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);
```

Speicher, Synchronisation

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);
```

Variablen

Analog wird mit Variablen verfahren. Jeder Variable wird einer 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);
```

Grundblöcke

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.newJmp();

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

Pin-States

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 einzelnen Knoten [1] nicht direkt entscheiden kann ob der Block wichtig ist, gibt es in firm das sogenannte „pinned"-flag:

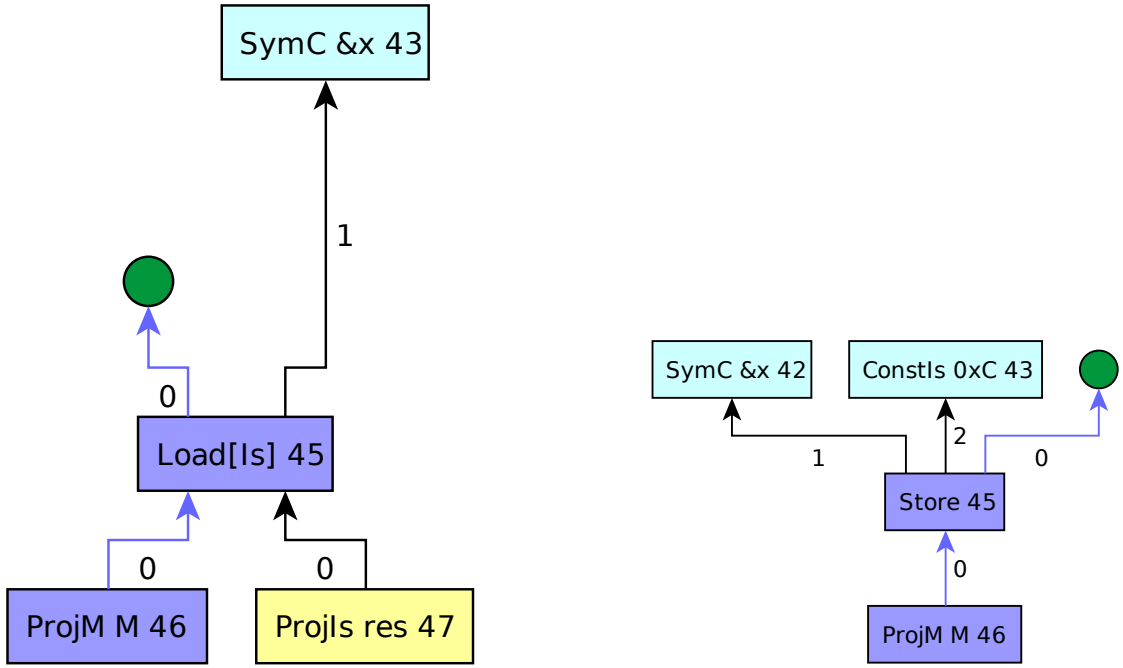
- `op_pin_state_floats` Block ist unwichtig, Knoten kann zwischen Blöcken verschoben werden.
- `op_pin_state_pinned` Knoten darf nicht zwischen Blöcken verschoben werden.

[1] Beispiel: `Div`, `Load`, `Store`

Laden/Speichern

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

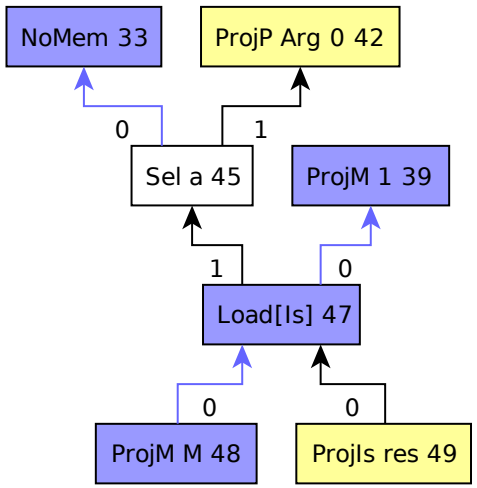
Beispiele: von Adresse der globalen Variable `x` laden; Den Wert 12 an diese Adresse schreiben.



Adresse von Feldern

- Adressen sind relativ zum `this`-Zeiger der Funktion. Adressberechnung kann mit `Sel`-Knoten erzeugt werden.
- Nimm `this`-Zeiger als Sel Vorgänger; Memory-Eingang für uns uninteressant: Dummy Knoten `NoMem` benutzen!
- Entity des Feldes ist Attribut des `Sel`-Knotens
- Eine spätere Phase in Firm ersetzt `Sels` durch echte Rechenoperationen.

Beispiel: Laden von Feld `a`:



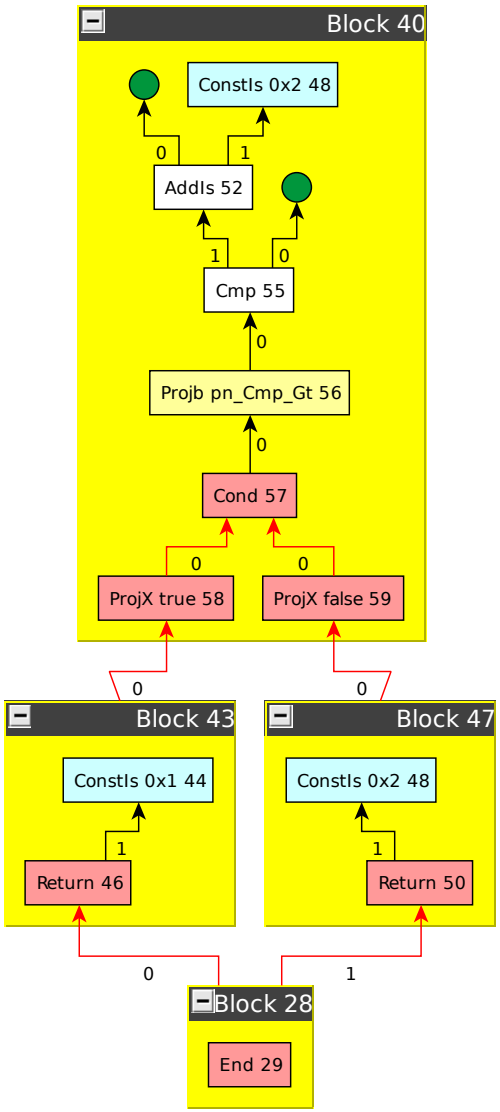
Vergleiche

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

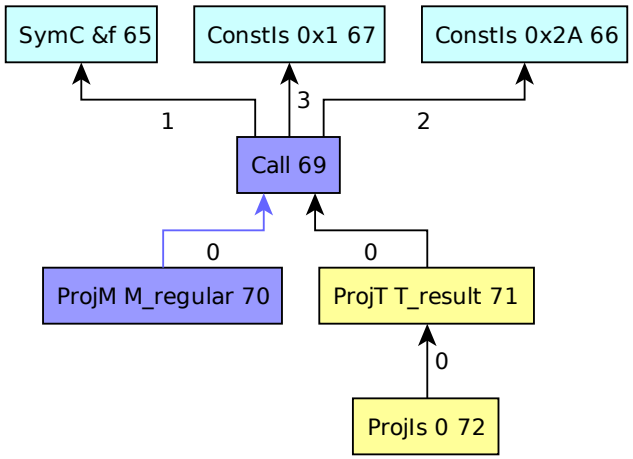
Beispiel: **if**(x>y+2) { **return** 1; } **else** { **return** 2; }



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 (kann sich aber bei variadischen Funktionen unterscheiden).
- Um Funktionsergebnisse abzufragen doppeltes `Proj` nötig!

Beispiel: f(42,1)



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.

Objekt-Orientierung / Klassen

- Methoden besitzen einen impliziten `this` Parameter: Dieser wird muss in der Firm-Darstellung explizit vorhanden sein.
- Die Main-Methode ist statisch und besitzt keinen `this` Parameter.

Eingebaute Checker (irverify)

Der eingebaute Checker prüft grundlegende Korrektheitsbedingungen eines Firmgraphs. Typische Beispiele sind:

- Vorgänger einer arithmetischen Operation haben alle den selben 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).

Debugging

Live-Demo

Benutzen des Firm x86 Backends

```
/* Architekturspezifische Optionen */
if (Platform.isMac()) {
  Backend.option("ia32-gasmode=macho");
  Backend.option("ia32-stackalign=4");
  Backend.option("pic");
} else if (Platform.isWindows()) {
  Backend.option("ia32-gasmode=mingw");
} else {
  Backend.option("ia32-gasmode=elf");
}
/* 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");
```

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.

Feedback! Fragen? Probleme?

- Anmerkungen?
- Probleme?
- Fragen?