Karlsruher Institut für Technologie

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

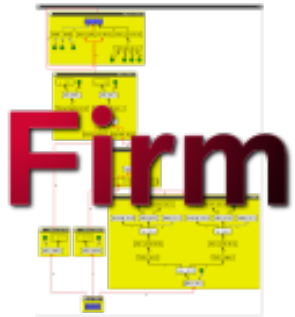
- Was waren die Probleme?
- Hat soweit alles geklappt?

Wette

Probleme:

- Keine Team-Abgabe
- {} statt { }
- **null** == **null** is valide
- **int** x; **int** x; nicht erlaubt ala Java
- **return** foo() nicht erlaubt bei **void** foo()
- Kein Fehler bei leerer Klasse ohne main

Übersicht

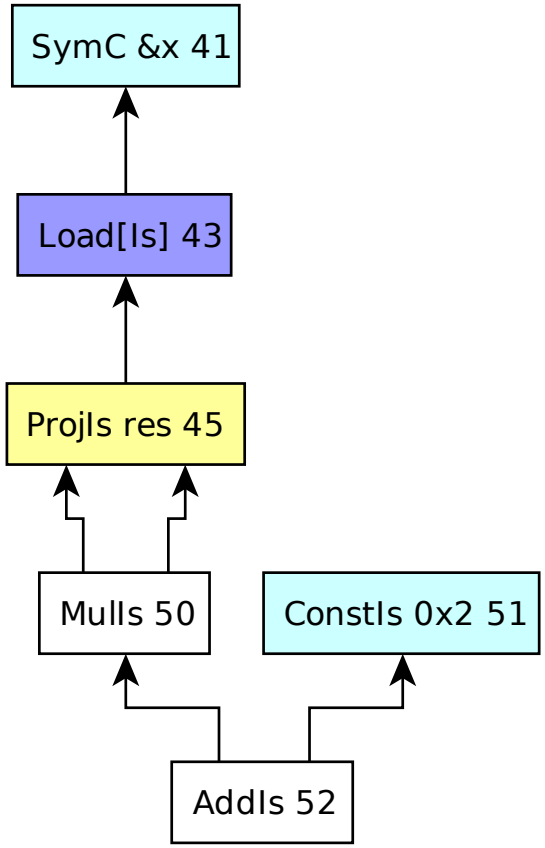


- libFirm ist die Implementierung einer low-level Programmrepräsentation.
- Low-level: Näher an der Maschine als an der Quellsprache.
- Komplett Graph-basiert; keine Instruktionslisten oder Tripelcode, stattdessen Datenabhängigkeiten und Steuerflussgraphen.
- Komplett SSA-basiert.
- Enthält zahlreiche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

Besonderheiten

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

Datenabhängigkeiten



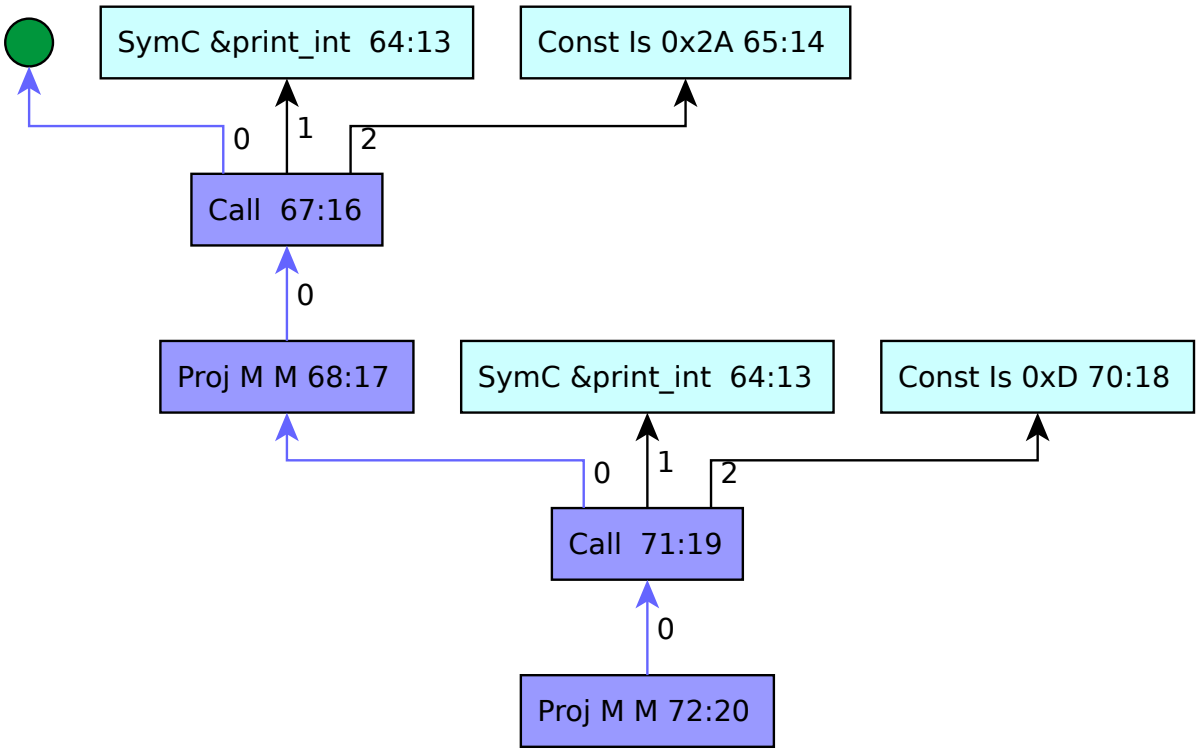
Beispiel: `x*x + 2`

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

„Speicher“-abhängigkeiten

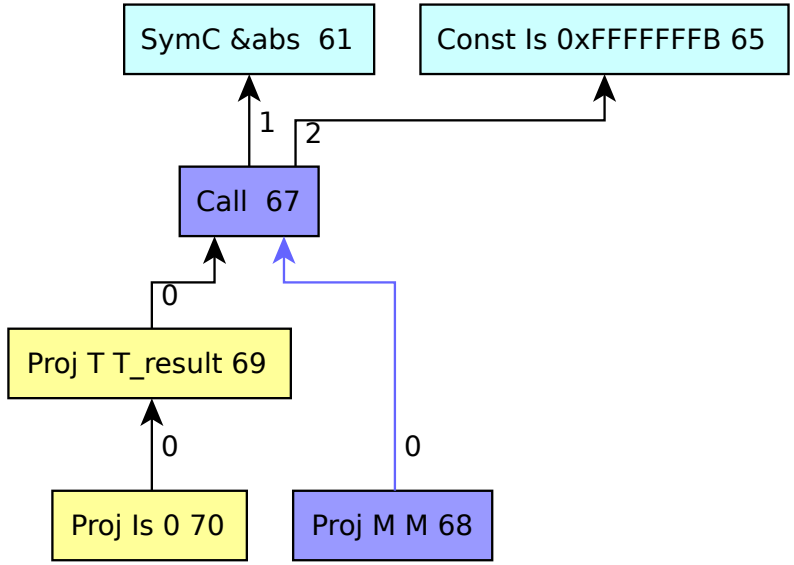
Beispiel: print_int(42); print_int(13);

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



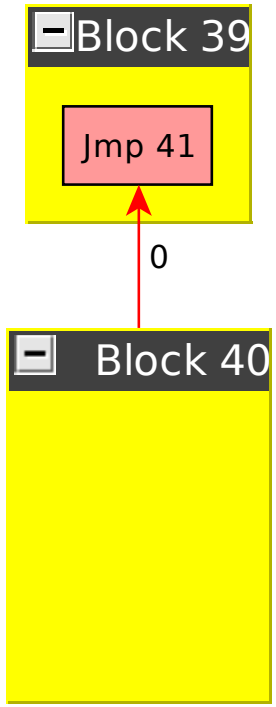
Tupelwerte und Projektionsknoten

Beispiel: `abs(-5)`



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

Grundblöcke und Steuerfluss



- Grundblöcke sind normale Knoten, die Sprungbefehle als Vorgänger besitzen.
- Jeder Knoten ist einem Grundblock zugeordnet (Vorgänger Nummer -1)

Modes

- Knoten im Graphen sind typisiert: Sie besitzen einen `Mode`.
- Es existieren verschiedene Klassen von Modi:
  - Datenwerte (schwarz)
  - Speicher/Synchronisation (blau)
  - Steuerfluss (rot)
- Modi werden im Namen des Knotens mit angegeben: `Add Is` ist ein „Add“-Knoten mit Modus „Is“ (Integer Signed).

Typische Modi

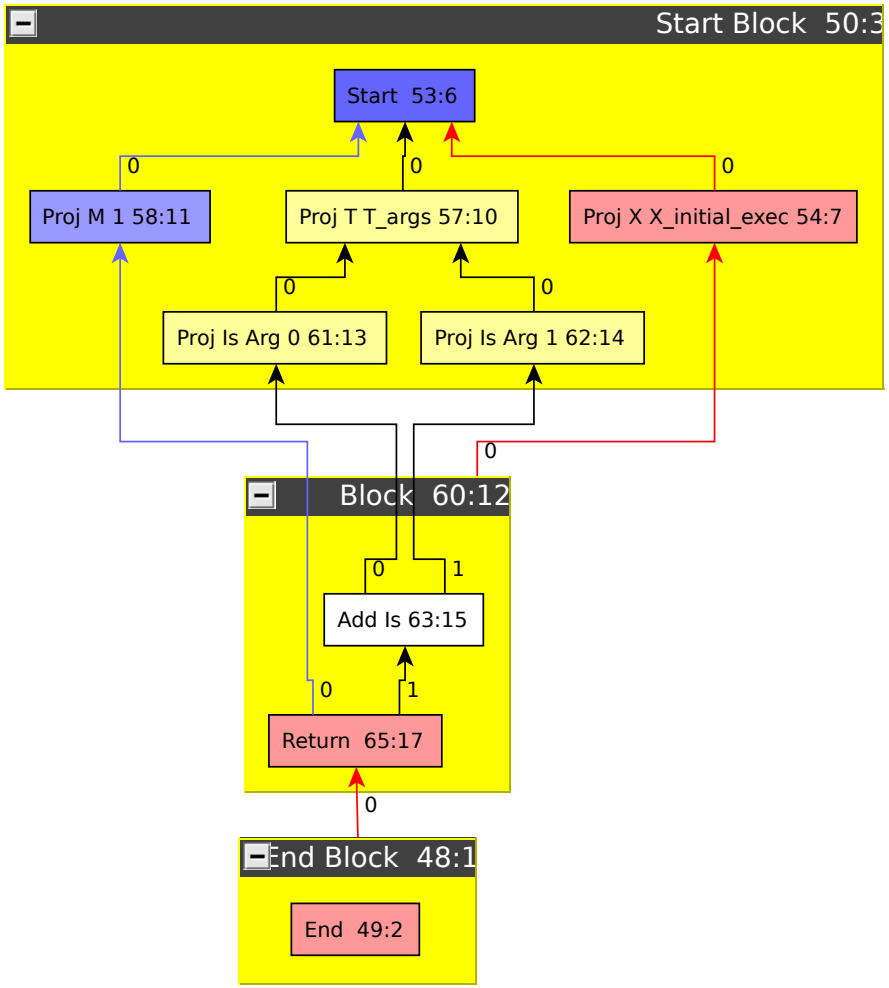
Bezeichnung	Bitbreite	Vorzeichen	Art
`Bs`	8	Ja	Ganzzahl
`Bu`	8	Nein	Ganzzahl
`Hs`	16	Ja	Ganzzahl
`Hu`	16	Nein	Ganzzahl
`Is`	32	Ja	Ganzzahl
`Iu`	32	Nein	Ganzzahl
`P`	32	Nein	Zeiger auf Daten/Code
`F`	32	Ja	Gleitkomma
`D`	64	Ja	Gleitkomma
`b`			(interne) Wahrheitswerte
`X`			Steuerfluss
`M`			Speicher/Synchronisation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

Methoden

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

Beispiel: Komplette Methode

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



Typen

- Zu jedem Programm existiert ein (minimales) Typsystem um Methoden und Datenstrukturen zu typisieren.
- Typen:
  - `Primitive` „Atomare“ Datentypen, Werte haben genau einen FirmMode.
  - `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)

Karlsruher Institut für Technologie

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, Sichtbarkeit

Hierarchie

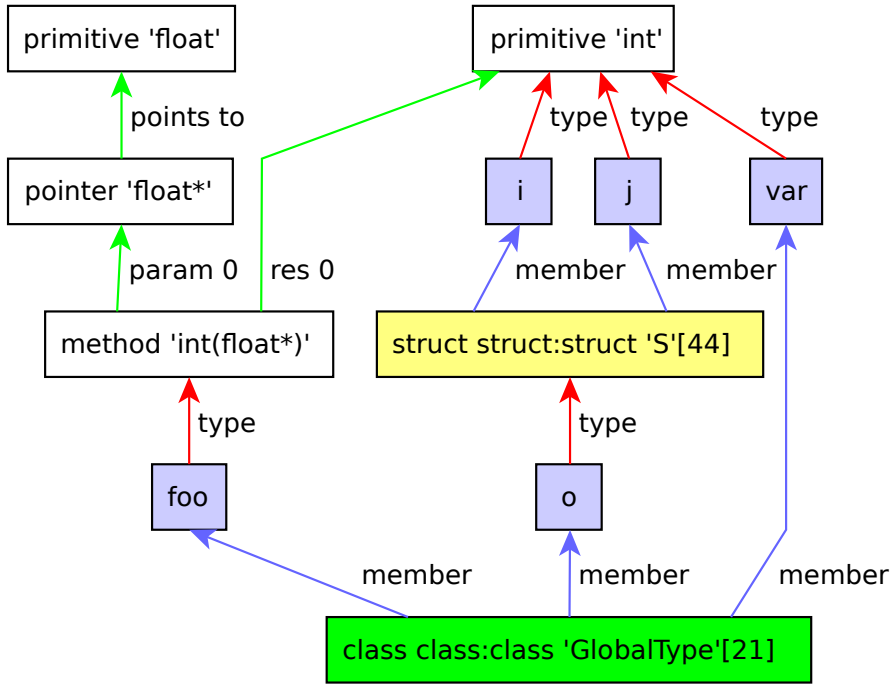
- Entitäten sind stets Kinder eines Compound-Typs (Klassentyp, Structtyp, . . .)
- Für „globale“ Entitäten existiert ein spezieller Typ Namens „global“.

Für globale Entitäten lässt sich die Sichtbarkeit festlegen:

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

Beispiel Entitäten/Typen

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



Probleme beim Erzeugen von FIRM-Graphen aus einem AST

Karlsruher Institut für Technologie

- 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/Entitäten Erzeugen

MethodType: Erzeuge Methodentyp mit 2 Integer Parametern und einem Gleitkomma 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-Entität: Methode foo mit obigem Typ.

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

Beginn/Ende der Graphkonstruktion

Karlsruher Institut für Technologie

Beginn

```
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 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);
```

Grundblöcke

Knoten werden im `CurrentBlock` erzeugt. Nach dem 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 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.

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

Vergleiche

Der `Cmp`-Knoten vergleicht 2 Werte. Mögliche Relationen (in der `Relation` Klasse):

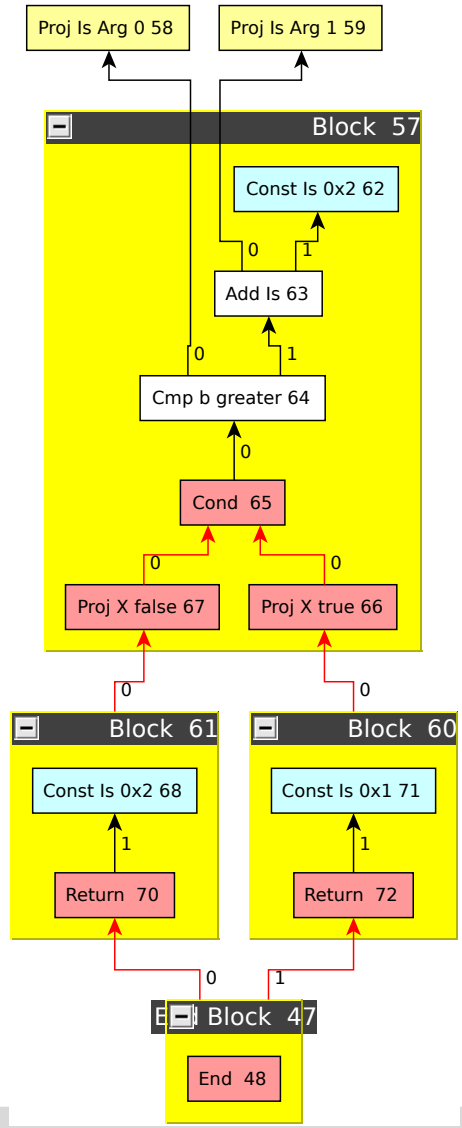
Name	Vergleich
`False`	immer falsch
`Equal`	$x = y$
`Less`	$x < y$
`Greater`	$x > y$
`Unordered`	unordered

Relationen lassen sich kombinieren (Beispiel (`UnorderedLessEqual`).
Operationen auf Relationen:

- Inverse Relation bilden: $a \leq b \Rightarrow a \geq b = b \leq a$
- Negieren: $a \leq b \Rightarrow a >_u b = \neg(a \leq b)$

If-Konstruktion

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



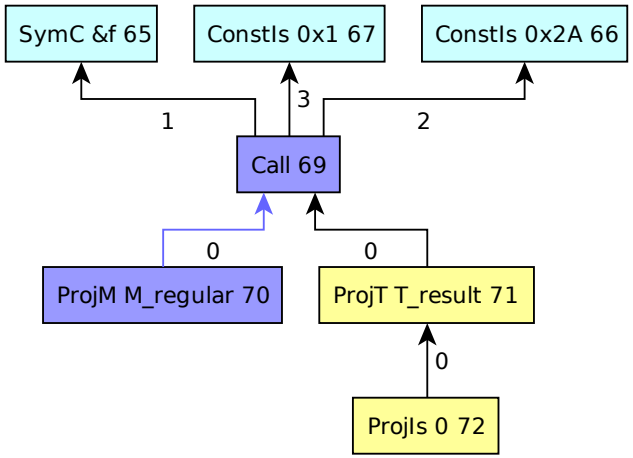
Objektorientierung / Methoden

- 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 Methodentyp angegeben werden. Im Allgemeinen der Typ der Funktion.
- Um Funktionsergebnisse abzufragen doppeltes `Proj` nötig!
- `this`-Zeiger nicht vergessen!

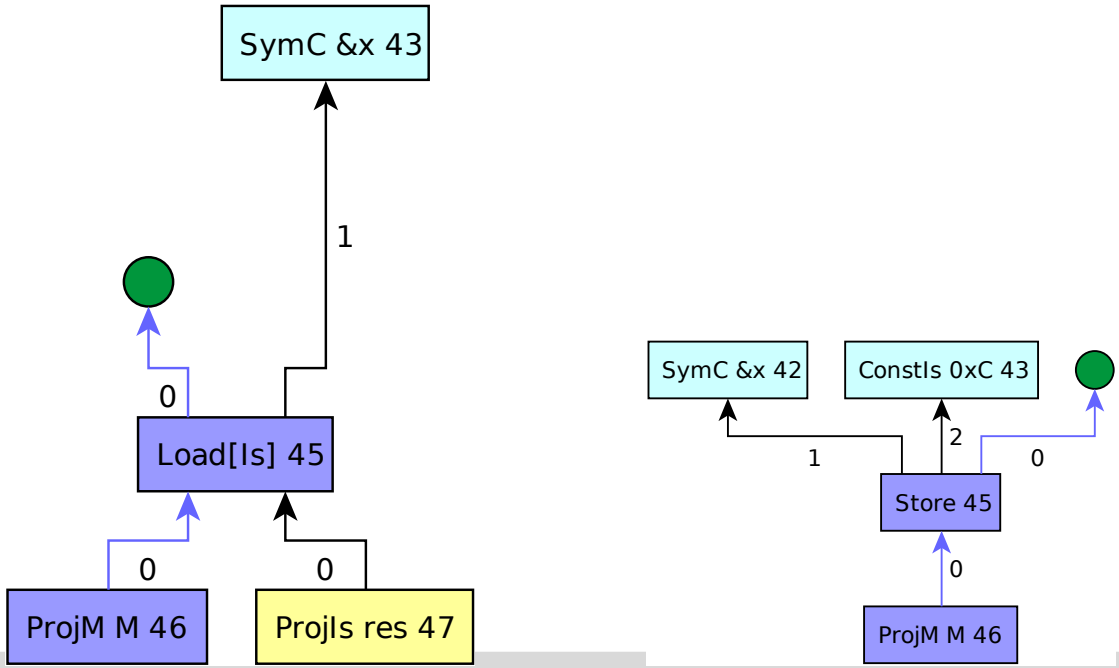
Beispiel: f(42,1)



Laden/Speichern

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

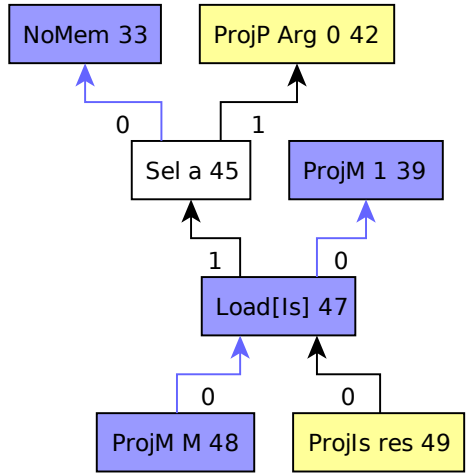
Beispiele: Von Adresse der globalen Variable `x` laden; Den Wert 12 an diese Adresse schreiben.



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!
- Entität des Feldes als Attribut setzen.

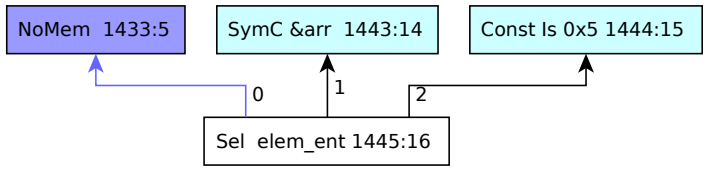
Beispiel: Laden von Feld `a`:



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.
- Entität 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.)

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

Karlsruher Institut für Technologie

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 auf der Praktikums-Webseite).

Debugging

Karlsruher Institut für Technologie

Live-Demo

High-level -> Low-level

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 Low-level 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");
```

Feedback! Fragen? Probleme?

Karlsruher Institut für Technologie

- Anmerkungen?
- Probleme?
- Fragen?