Praktikum Compilerbau
Sitzung 5 – libFirm Teil 1

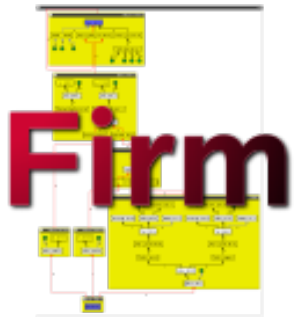
Lehrstuhl für Programmierparadigmen
Universität Karlsruhe (TH)

20. Mai 2009

Letzte Woche

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

Übersicht

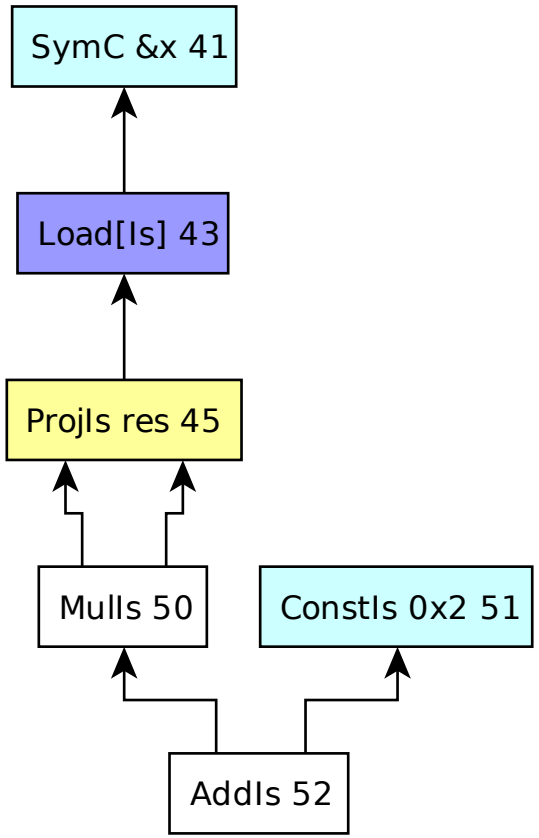


- libFirm ist die Implementierung einer low-level Programmrepräsentation.
- low-level: Näher an der Maschine als an der Quellsprache.
- Komplett Graphbasiert; keine Instruktionslisten oder Tripelcode, stattdessen Datenabhängigkeiten und Kontrollflußgraphen.
- Komplett SSA basiert.
- Enthält zahlreiche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

Unterschiede zu klassischen Compilern

- Keine Befehlslisten – Abhängigkeitsgraphen genügen um Reihenfolge vorzugeben.
- Keine Variablen – Wir betrachten berechnete Werte; *Variablennamen 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).

Programmdarstellung



Beispiel: `x*x + 2`

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

Programmdarstellung - Modes

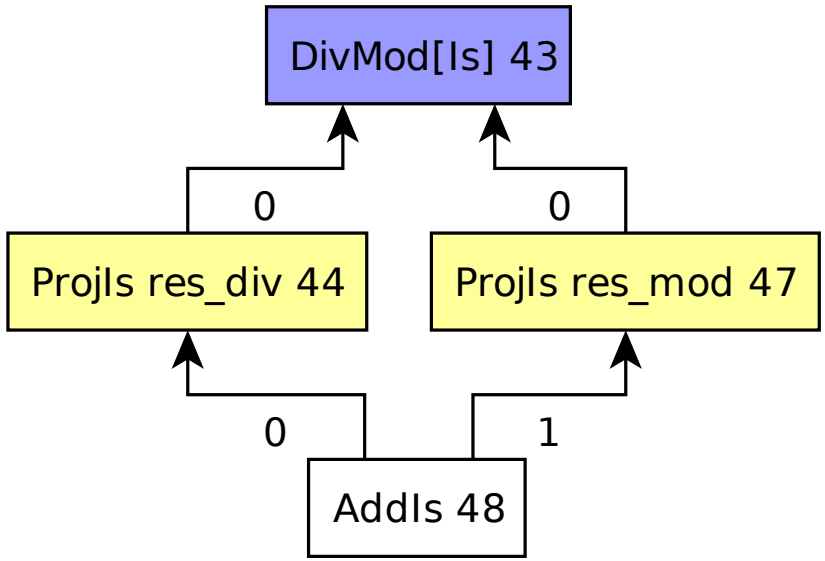
- Jeder Knoten hat einen festgelegten `Mode` der angibt was von einer Operation produziert wird.
- Moditypen werden im Graphen als unterschiedliche Farben dargestellt:
  - schwarz – Datenwerte
  - blau – Speicher/Synchronisation
  - rot – Kontrollfluß
- Modi werden im Namen des Knotens mit Angegeben: `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
`Ls`	64	Ja	Ganzzahl
`Lu`	64	Nein	Ganzzahl
`F`	32	Ja	Fließkomma
`D`	64	Ja	Fließkomma
`b`			(interne) Wahrheitswerte
`X`			Kontrollfluß
`M`			Speicher/Synchronisation
`T`			Tupelwerte

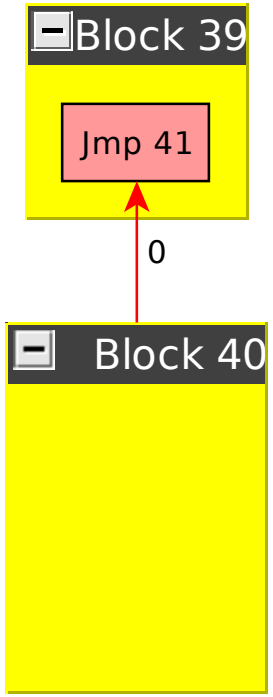
Tuple und Projektionsknoten

Beispiel: `x/y + x%y`



- Liefert eine Operationen mehrere Werte zurück, so wird ein spezieller Mode namens `T` (Tupel) benutzt.
- Mit Hilfe der (virtuellen) `Proj` Operation kann man einzelne Werte aus einem Tupel extrahieren.

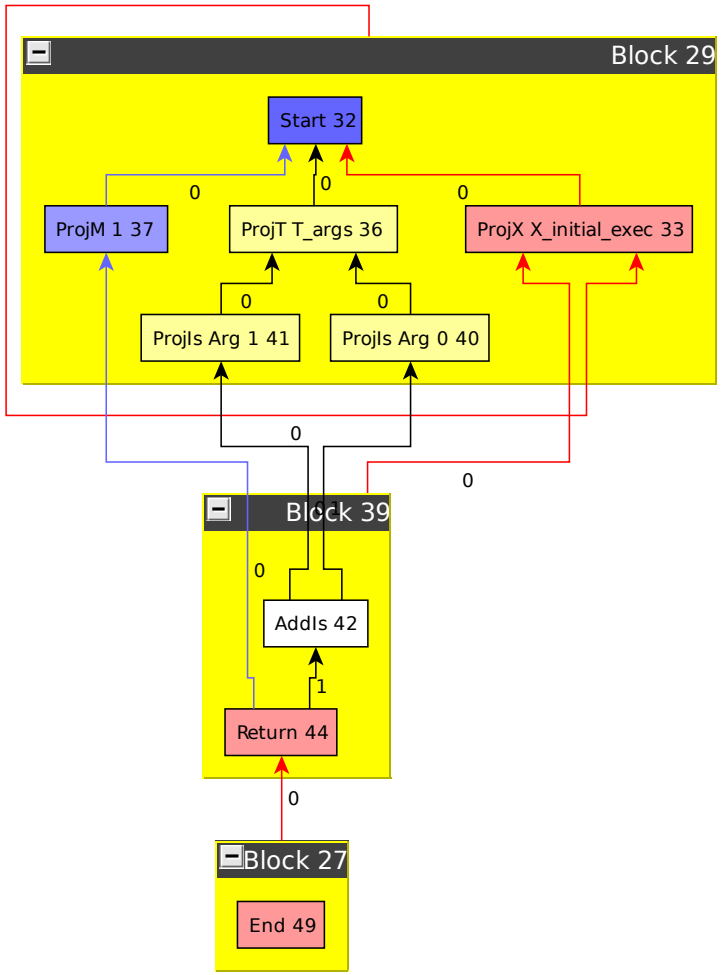
Grundblöcke und Kontrollfluß



- Jeder Knoten ist einem Grundblock zugeordnet.
- Grundblöcke sind selbst wieder Knoten, die Sprungbefehle als Vorgänger besitzen.

Komplette Methoden

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



Methoden

- 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.

Typen

- 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` Zusammengesetzer Datentyp. Enthält eine Liste von Entitäten. Adressen der Entitäten dürfen nicht überlappen.
  - `Union` Zusammengesetzer Datentyp. Enthält eine Liste von Entitäten. Adressen der Entitäten dürfen überlappen.
  - `Class` Zusammengesetzer 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.

Für globale Entitäten existiert ein vorgegebener `Class`-Type namens `GlobalType`.

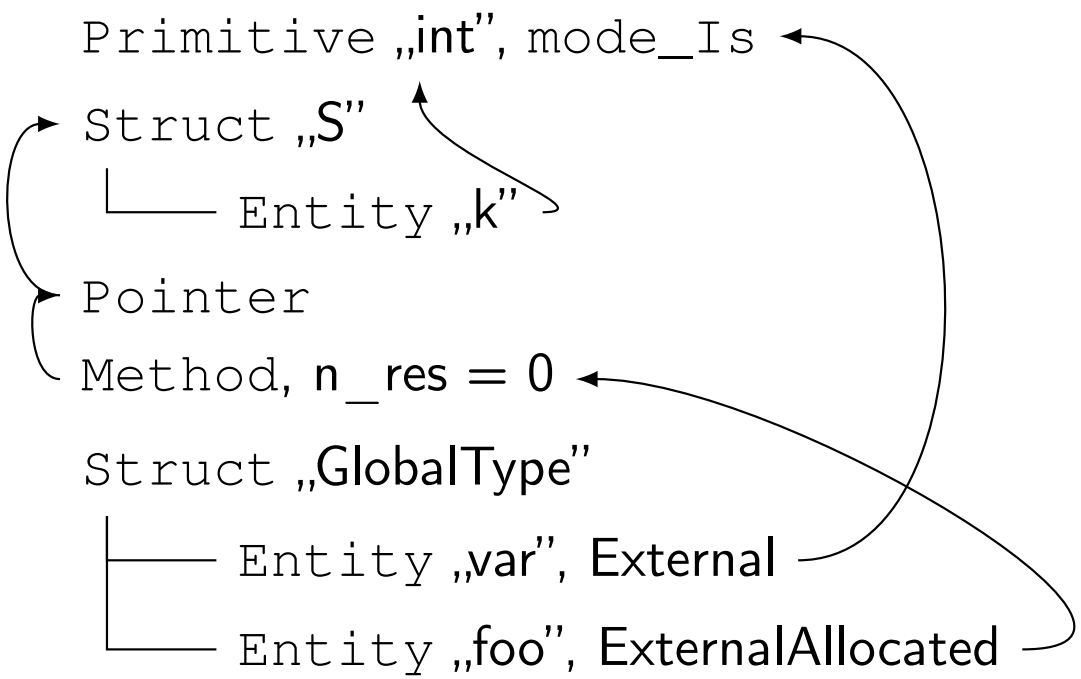
Sichtbarkeit und Linker Namen

Es werden typischerweise nur einzelne Objekte übersetzt die später von einem Linker bearbeitet werden. Entitäten müssen deshalb mit zusätlichen Informationen annotiert werden:

- `Visibility`:
  - `Local`: Definition und Sichtbarkeit auf Objektdatei beschränkt.
  - `ExternalVisible`: In Objektdatei definiert; für andere Objektdateien sichtbar.
  - `ExternalAllocated`: In fremder Objektdatei definiert (und sichtbar gemacht).
- `Variability`:
  - `Uninitialized` Inhalt nicht bekannt
  - `Initialized` Mit bekanntem Wert initialisiert
  - `Constant` Mit bekanntem Wert initialisiert, konstant
- `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;
typedef struct S { int k; };
extern void foo(S *x);
```



Probleme beim erzeugen von Firmgraphen aus einem AST

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

Firm kommt mit einigen Hilfsmitteln um diesen Aufbau zu vereinfachen.

Beginn/Ende des Firmaufbaus

Begin

```
int n_vars = 20;
Graph graph = new Graph(entity, 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);
```

Ausführungsreihenfolge

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, Loa
Node loadMem    = construction.newProj(load, Mode.getM
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 = ... ;
Node currentVal = construction.getVariable(0, mode);

/* setzen der Variable */
int var_num = ... ;
construction.setVariable(0, mode);
```

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);
```

Feedback! Fragen? Probleme?

- Anmerkungen?
- Probleme?
- Fragen?