Firm, a fully SSA-based IR

Michael Beck
beck@ipd.info.uni-karlsruhe.de

Institute for Innovative Computing and Program Structures (IPD)
University of Karlsruhe

30. April 2009

Chapter 1
Introduction

Introduction

FIRM – the Firm Immediate Representation Mesh – features:

- Graph-based
- SSA-based
- Dependency-based
- Easy construction directly from AST
- Contains powerful scalar optimisation
- SSA-Property is preserved even in backend
- Contains SSA-aware register allocator

libFIRM is our implementation of FIRM.

Some words on history . . .

- Was developed at the IPD Goos at the University of Karlsruhe
- Initially created 1996 as part of the Sather-K compiler FIASCO by Armbruster and von Roques as a clone of Click's Sea-of-Nodes IR
- Extended to Explicit Dependency Graphs by Trapp 2001
- Backend support added by Beck and Hack 2005
- SSA-based register allocator added by Hack 2005
- Lindenmaier adds a abstract type representation 2005
- Released to the public 2007
- New register-pressure-aware Scheduler added by Mallon 2008
- New Belady spiller added by Braun 2009
- Many others contribute . . .

Chapter 2
Construction

Firm-Construction

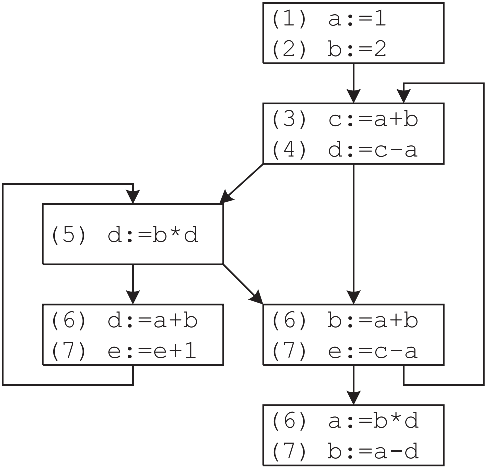
- Firm offers a simple interface to allow direct SSA construction from the AST.
- No CFG is needed in advance. In fact we can build the CFG in parallel.
- Only one precondition: We must know the number of alias-free local variables. This is simple if we use an under estimation (no address taken).

FIRM-Construction

1. Assign a number $[0 \dots n-1]$ to every alias-free local variable.
2. Start transformation with the start block.
3. Whenever all predecessor blocks of a basic block are processed, set them to mature, else to immature.
4. The construction is intertwined with (local) value numbering. To retrieve a value of a variable, use get_value (var).
5. For every assignment $a := e$, get the value of e and assign it to a using set_value (a).
6. For every expression *a op b* call the constructor new_Op(a, b) returning the value of this expression.

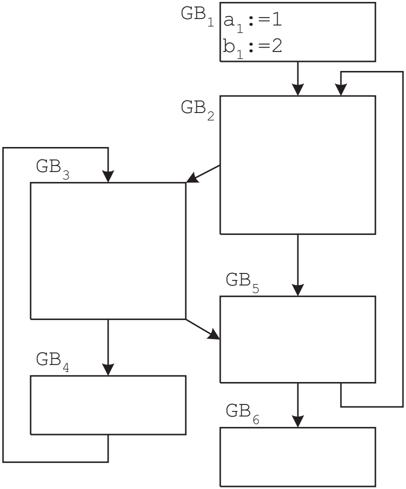
Construction Example

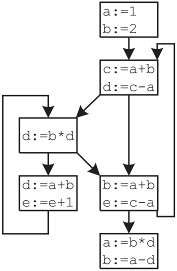
```
 (1) a:=1;
 (2) b:=2;
     while true {
 (3)   c:=a+b;
 (4)   if (d:=c-a)
 (5)     while (d:=b*d) {
 (6)       d:=a+b;
 (7)       e:=e+1;
          }
 (8)   b:=a+b;
 (9)   if (e:=c-a) break;
     }
(10) a:=b*d;
(11) b:=a-d;
```



SSA Construction Block 1

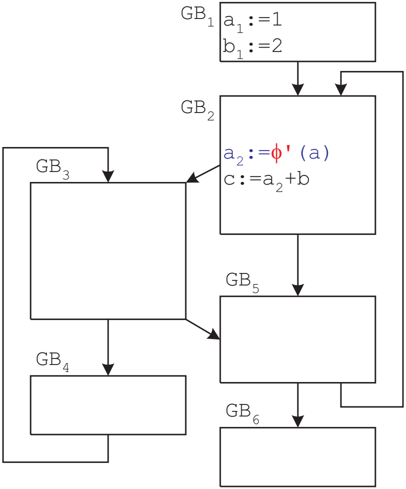
This block is mature. We assign the constants 1 and 2 to *a* and *b*.

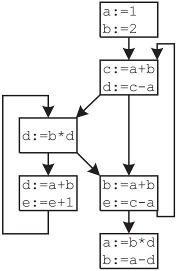




SSA Construction Block 2

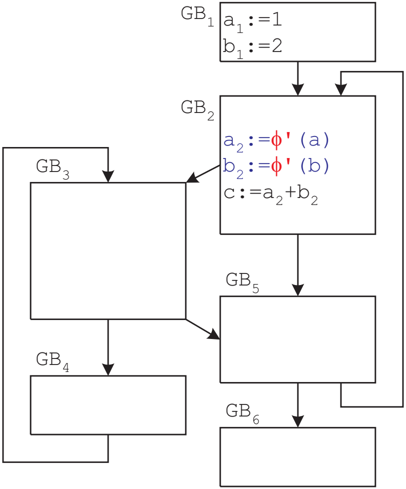
Getting the value for *a* creates a ϕ' for *a* because block 2 is not mature yet . . .

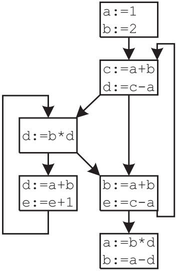




SSA Construction Block 2

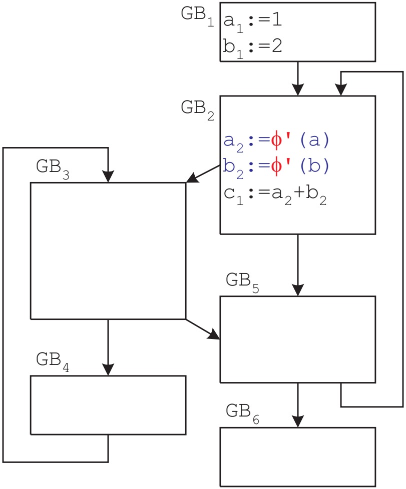
. . . same for *b* . . .

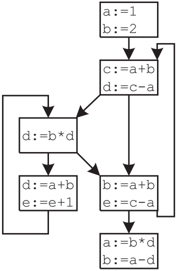




SSA Construction Block 2

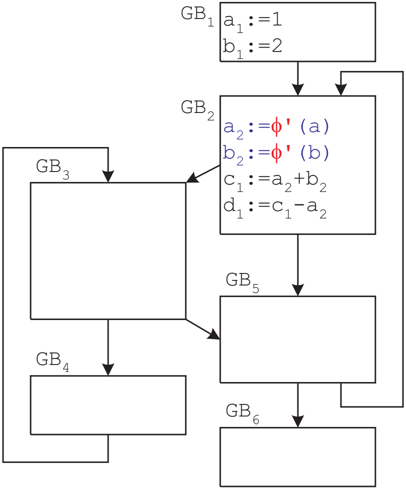
The construction of $c := a + b$ creates an *Add*-node and stores its value as the value for c.

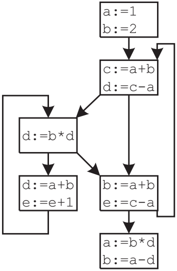




SSA Construction Block 2

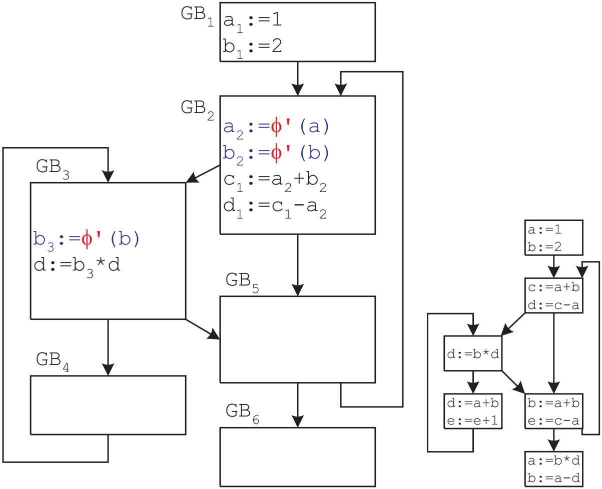
The construction of $d := c - a$ creates a *Sub*-node and stores its value as the value for d.



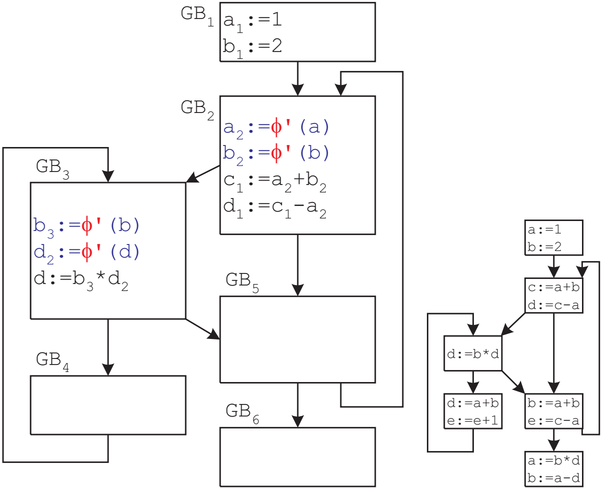


SSA Construction Block 3

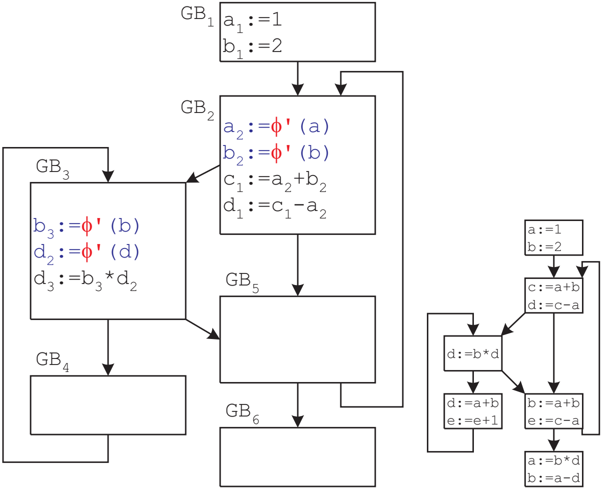
Same for block 3. It is not mature yet.



SSA Construction Block 3



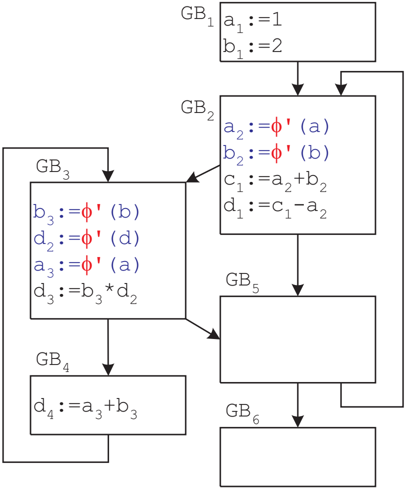
SSA Construction Block 3

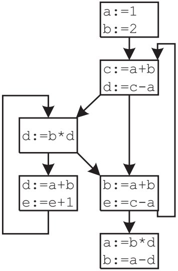


SSA Construction Block 4

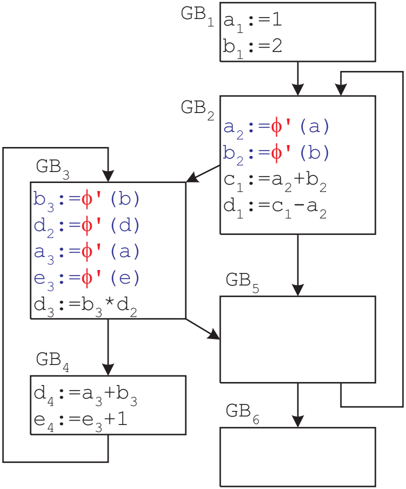
The get_value(a) call for block 4 leads to a recursive call of get_value(a) for block 3.

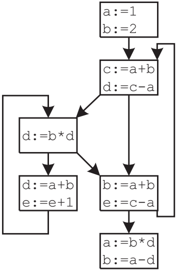
This creates a new ϕ' for a in block 3.





SSA Construction Block 4

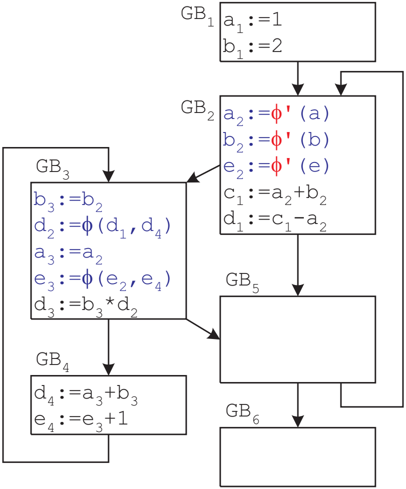


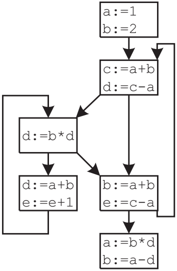


SSA Construction Block 4

Now all predecessors of block 3 are finish, we can mature it and calculate ϕ-nodes. (Later we put them all at the block start.)

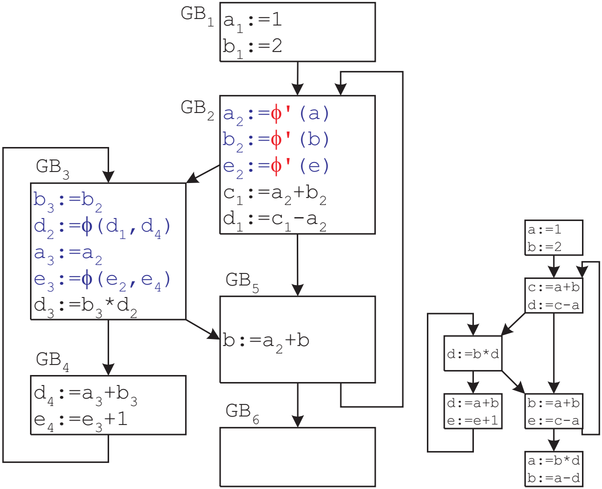
For e we recursively create a ϕ' in Block 2.



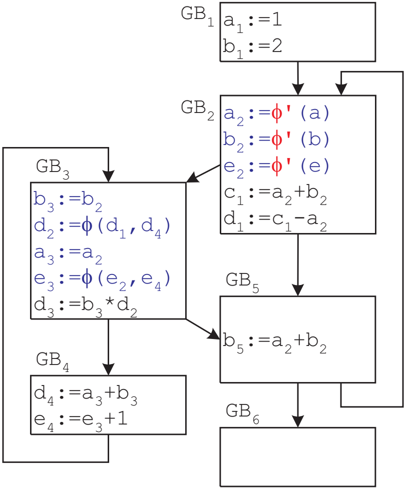


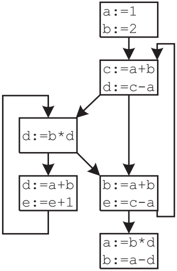
SSA Construction Block 5

get_value (a) in mature block 5 finds a definition in block 4, so no ϕ needed.

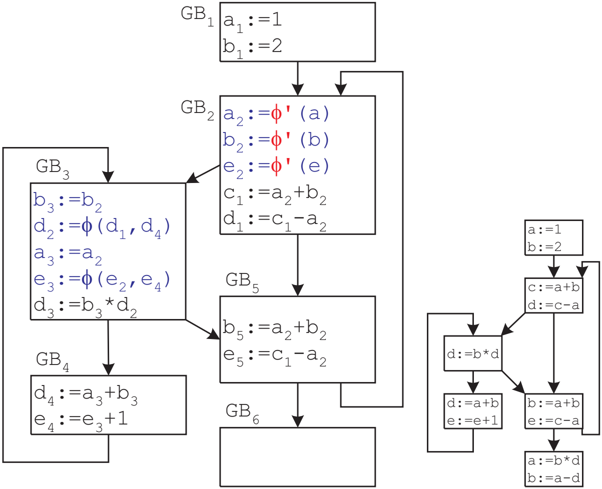


SSA Construction Block 5





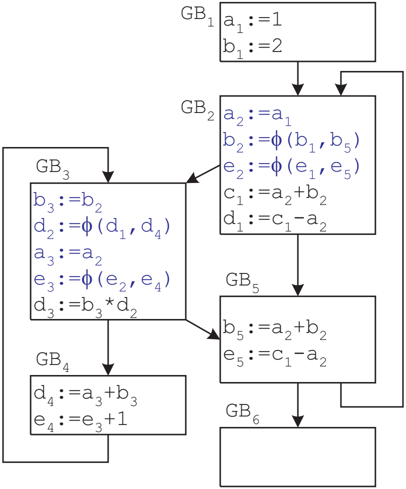
SSA Construction Block 5

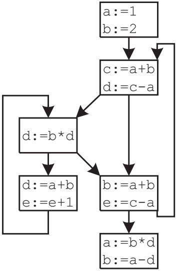


SSA Construction Block 5

Now all predecessors of block 2 are processed, we can mature it and calculate ϕ-functions.

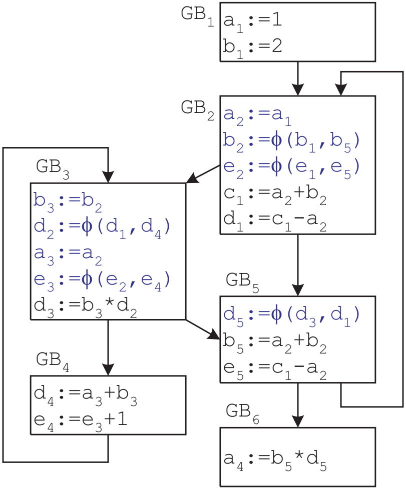
By doing this we detect:
e is uninitialised!
Assign some unknown value e_1.

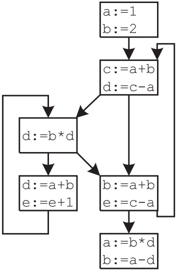




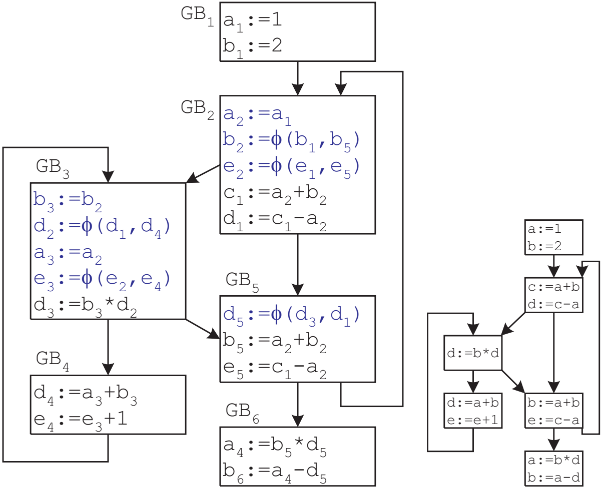
SSA Construction Block 6

Recursive call of get_value (a) in block 5 creates ϕ-Function for d_5.

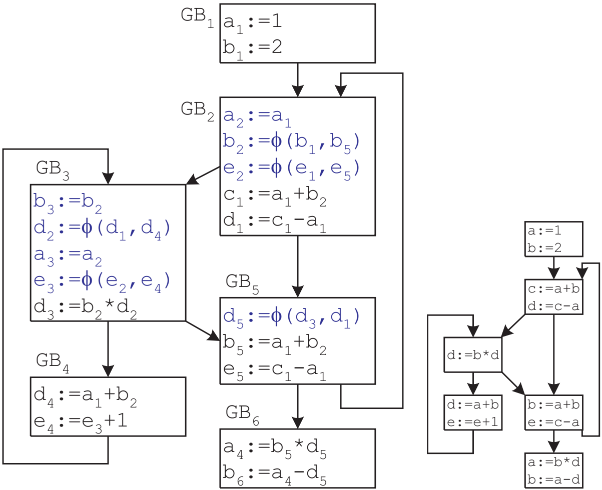




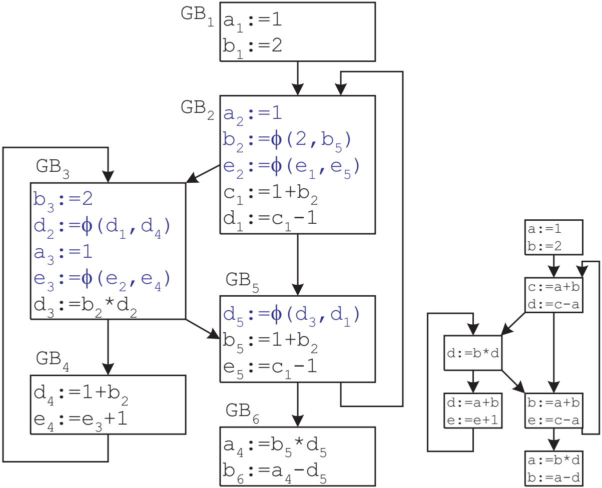
SSA Construction Block 6



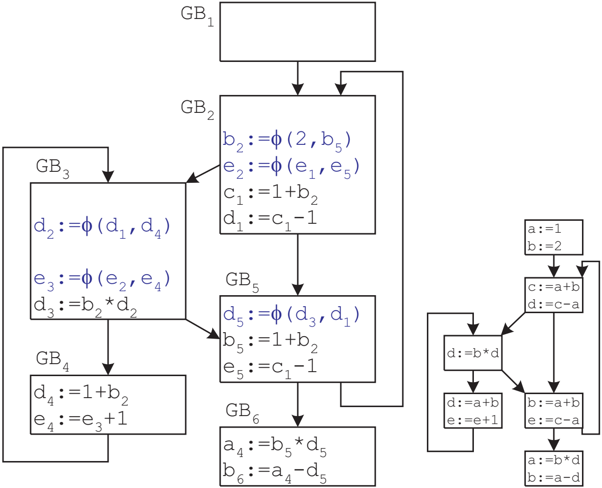
Simple Constant Propagation



Simple Constant Propagation



Dead Code Elimination



More Properties of the FIRM-Construction

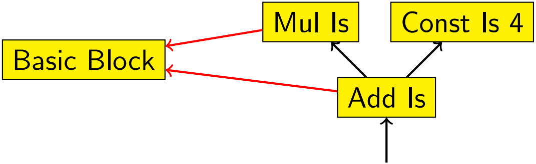
Construction of new nodes is always combined with:

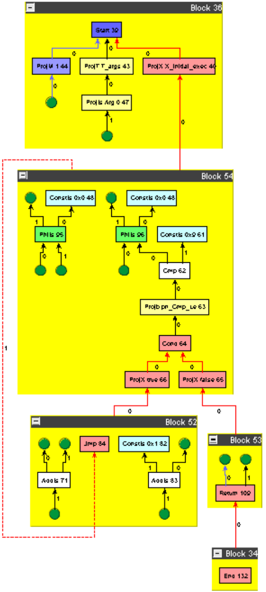
- Simple constant propagation
- Local Value Numbering
- Normalisation and Simplification by using algebraic identities and more complex local transformations

Chapter 3
The IR – Details

FIRM-Graphs

FIRM-Graphs are an union of a CFG and a DFG. The DFG is represented as a *Value Graph*. Every data node of this Value Graph has a control flow input. This input points to the basic block the node belongs to.



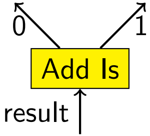


We typically render FIRM-Graphs in the classical way: Data nodes are included in basic blocks.

FIRM-Nodes

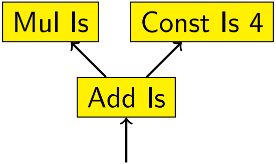
FIRM-Node features:

- Every node contains an operation code and a machine mode
- Every node produces only one result representing the computed value
- The result might be a tuple
- Projection nodes extract scalars from a tuple
- Most nodes have a fixed number of inputs
- Nodes points directly to its operands, so no names



Use of Dependency

FIRM-Edges are Dependency-Edges (Use-Def), not Dataflow Edges!



Here the *Add* node summarises the result of the *Mul* node and the constant 4.

There are

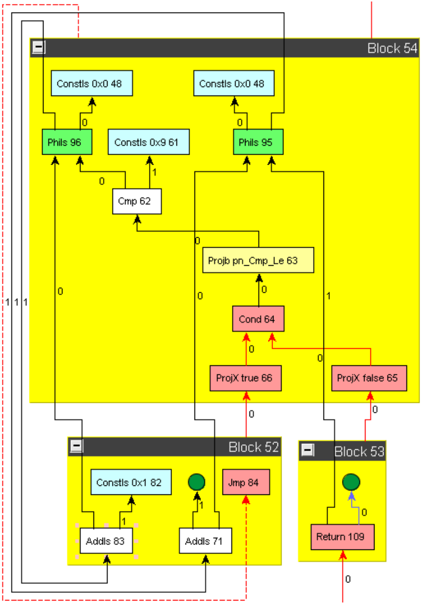
- no explicit assignments
- no distinction between expressions and statements
- no variables/temporaries

Use of Dependency

1. The (data) nodes in a FIRM-Graph have the SSA-Property.
2. Every node must be reachable by a dependency from the *End*-node.
3. Nodes that are not reachable from *End* do not add something to the computations of a graph and are dead. They automatically vanish from the graph.
4. There is no schedule inside a basic block. The dependency edges induce a partial order of a block nodes.
5. Every topological sort of the block nodes result in a valid schedule.

For a formal definition of Explicit Dependency Graphs see Trapp:99.

Loop-Example Reexamined



The Memory State

We handle the Memory State like every other SSA-variable:

- Every *Store*-node uses and defines a new memory state
- Every *Load*-node uses and defines a new memory state
- Every *Call*-node uses and defines a new memory state except when the optimiser can prove there is no memory access at all (pure function)
- Every fragile node (possibly throwing an exception) uses and defines a new memory state
- The are ϕ-nodes merging the memory in the graph

Why *Load*-nodes define Memory states

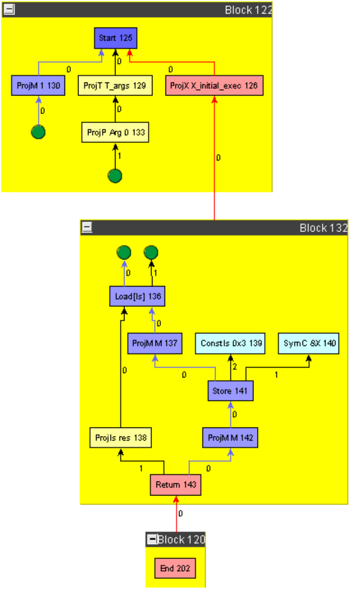
Answer: We use dependency between nodes. And between a *Load* and a following *Store* is an Anti-Dependency.

- So, the memory edge serialise all operations that could read or write the memory state.
- Additionally, the memory edge serialise all fragile instructions. We need this, because we do not have a schedule in the block.

Memory example

```
int X;

int func(int *a) {
  int t = *a;
  X = 3;
  return t;
}
```



Load After Load

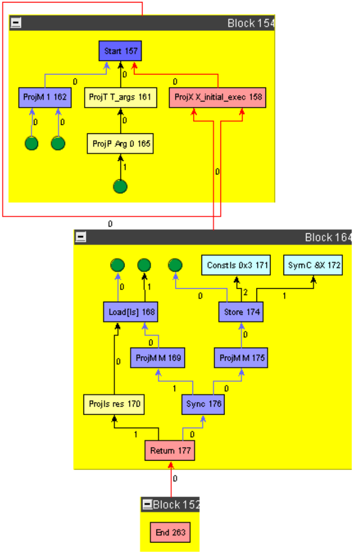
Q: There is no dependency between two *Load*-nodes, isn't it?
A: If we ignore exceptions no.

If we can prove that no dependency between two memory nodes exists, we can split the memory state and combine it later using the *Sync*-node.

Memory example with *Sync*

```
int X;

int func(int *restrict a) {
  int t = *a;
  X = 3;
  return t;
}
```



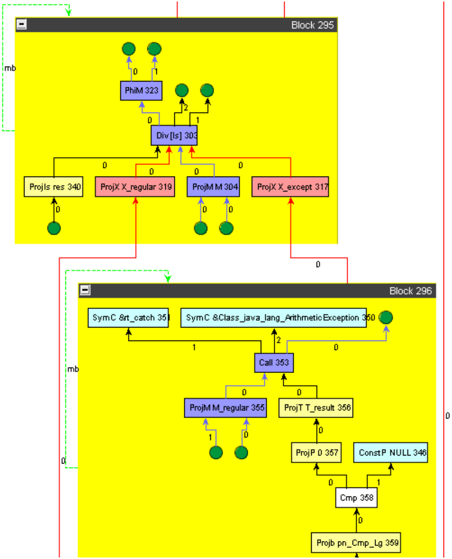
Handling of Exceptions

Every fragile node defines two additional control outputs:

- The regular edge represents the control flow if no exception is thrown.
- The exception edge represents the branching that occurs when an exception is thrown.
- So, fragile nodes behaves like control flow branches and ends the current basic block.

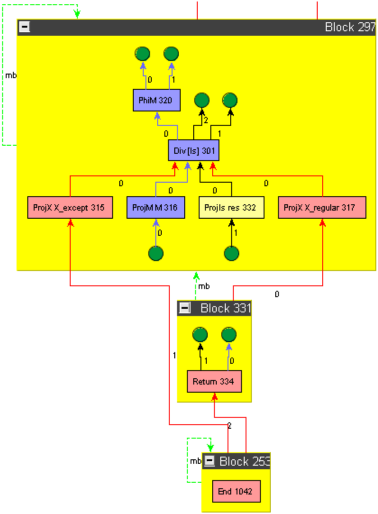
Exception Example

```
public static
int test(int a, int b) {
  try {
    return a/b;
  }
  catch (ArithmeticException
    return 0;
  }
}
```



Exception Example w/o catch

```
public static
int test(int a, int b) {
  return a/b;
}
```



Available Analyses

- Def-Use Edges (non-persistent or persistent)
- Dominance/Postdominance Calculation
- Loop-Tree
- Call Graph
- Interprocedural View (Interprocedural Graph)
- Execution frequency estimation
- Liveness check (Boissinot, Rastello, Hack)
- Interval/Structure Analysis
- Memory Disambiguation (Alias Analysis)

Available Transformations

- Extended version of Clicks Combined Analysis and Transformation (combo)
- GVN
- GVNPRE
- Inlining
- Procedure Cloning
- Load/Store Elimination
- If-Conversion (using Select Instructions)
- Conditional Evaluation
- Reassociation
- Scalar replacement for arrays and structures
- Tail recursion elimination
- Congruent block merging
- Rapid Type Analysis
- Class Hierarchy Analysis
- Control-flow Optimisations
- ...

Chapter 4
The Backend

SSA in the Backend

We want to preserve the SSA-Property (and the Value Graphs) even in the Backend. Then:

- We could use SSA-based Register Allocation
- Use our Transformations/Analyses on Target code (code placement for instance)
- We must not invent a new representation
- Could use our debugging tools

SSA in the Backend

What must been done here:

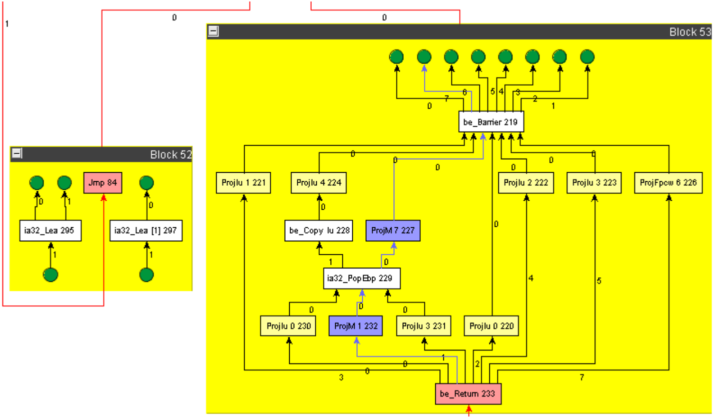
- Ensure the ABI: Lower procedure calls, create prolog/epilog
- Instruction selection preserving SSA-Property
- Instruction scheduling
- Register allocation
- Maybe Rescheduling
- Peephole Optimisation
- Code emitting

Not all these phases profit from SSA, but they are not more complicated!

Instruction Selection

- Instruction Selection is done by a hand-written code generator.
- For x86, a PBQP-based code-generator is available (slow, because based on an external tool yet), with equal code quality to the hand written approach.
- Other code generators for ARM, MIPS are available, but these are not mature yet.
- STA backend was implemented for the TU Dresden and is maintained there.
- Use-Def Edges are handy here for doing all the matching.

After Instruction Selection



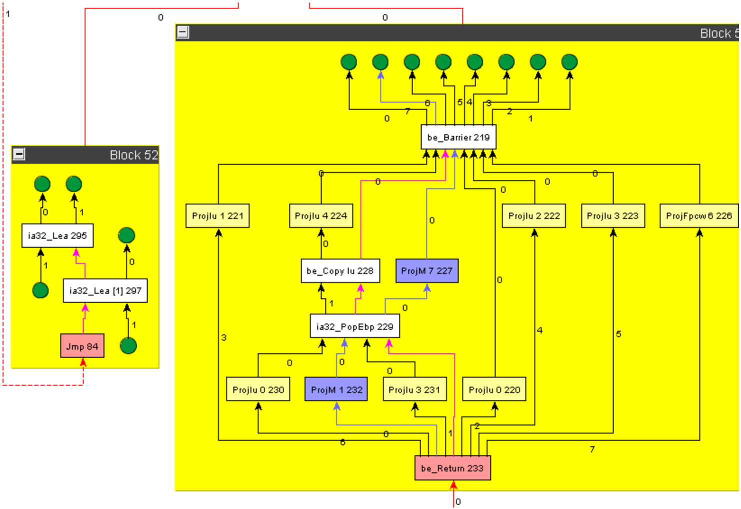
Instruction Scheduling

Several schedulers are available in libFIRM:

- Block Scheduler place blocks so that often executed edges are fall-troughs
- Trace Scheduler minimises the cycle-count on the critical path
- Register pressure aware scheduler tries to minimise the number of used registers

These schedulers simply add a new edge to nodes in every basic block that creates a total order of instructions or instruction bundles (VLIW).

After Instruction Scheduling



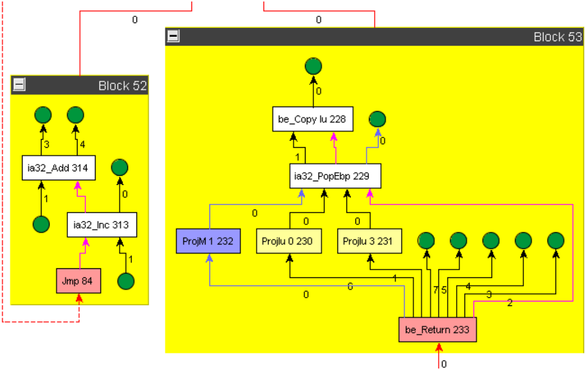
The purple edge represents the schedule.

Register Allocation

The SSA Register Allocator developed by Hack together with the enhanced Belady spiller by Braun is used.

This adds possibly spill code and finally annotates every data edge by a register name.

After RA and Peephole Optimisation



To emit the code, simply iterate over the block schedule and for every block over the schedule edges.

Chapter 5
Tools and Debugging Aid

Which Tools we need?

Users of libFIRM typically wants

- View the FIRM-Graph
- Inspect all FIRM-Nodes (these are complex data structures)
- Break on several FIRM-Events
- Selectively see log output
- Get warned as early as possible when doing a mistake and understand the error message

The yComp-Viewer

yComp

yComp is our viewer for all kind of Firm-Graphs.

It features:

- Layout optimised for "compiler graphs"
- Implemented in Java, so runs everywhere ... mostly
- Exports libFirm-VCG graphs into other formats (SVG and Tikz among them)
- Rendered most graphics in this talk
- Free of charge for academic use (uses the yFiles library)

Node Inspection

Contains a plugin for VisualStudio 6/7/8 that allows node inspection:

Locals

Name	Value	Type
irg	0x014c4888 {IRG: test3 [0, 30 nodes]}	ir_graph
start	0x014ce0f8 {node_list={...} node=0x014ce0c4 depth_nr=	node_t *
len	-858993460	int
initial_bl	0x014c5da4 {BlockBB [154:3]}	ir_node *
rem	0x014af568 {IRG: NULL [0, 16 nodes]}	ir_graph
env	{obst={...} worklist=0x014cdf90 cprop=0x00000000 ...}	environm

Same support is available for gdb through the use of gdb-macros.

Logging and FIRM-Debugger

Contains a built in logging facility allowing:

- to define new log classes
- emit log messages at different log levels
- redirect log messages

The FIRM-debugger is a built in command shell allowing:

- to change the log levels for log classes
- setting breakpoints on FIRM-events like node creation, node transformation, etc.
- can read command strings from environment variable or through a call of the libFIRM firm_debug() function

Chapter 6
Conclusion

Conclusion

FIRMis a IR that

- is fully based on SSA and Value Graphs
- includes SSA based backend using an SSA-aware Register allocator
- supports OO and imperative languages
- Frontsends available: cparser (C99, GPL), EDG C/C++ (only C yet), EDG Java (only 1.4 yet, old GNU classpath runtime)

http://www.libfirm.org