Documentation of the Intermediate Representation Firm

Tech Report Nr. 1999-44

Martin Trapp, Götz Lindenmaier and Boris Boesler

Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe
ISSN 1432-7864
{trapp|goetz|boesler}@ipd.info.uni-karlsruhe.de

December 1999

Contents

Chapter 1

Introduction

This documentation presents Firm, an intermediate representation (IR) presented first by Armbruster and von Roques in [AvR96], subsequently refined and extended by Trapp ([Tra99]) in his Phd Thesis at the Institute for Program Structures and Data Organization at University of Karlsruhe. Armbruster and von Roques implemented Firm in an experimental compiler, Fiasco, for the language Sather-K ([GS96]).

Compilers translating object oriented programs do not produce code of comparable quality to compilers translating common imperative languages. Partially this is due to the functionality of such languages that imposes additional runtime costs such as resolving polymorphy dynamically. But a lot of program runtime is wasted in unnecessary program and data structures that arise from straight forward translation of object oriented programs. For example, these have more and smaller procedures and allocate more variables dynamically. Optimizations developed for traditional imperative languages do not deal with these specific issues. Further, IRs originally developed for imperative languages are not tuned to expose the additional problems of translating object oriented programs.

The design goal for a new intermediate representation was to develop an IR that supports fast and powerful optimization of object oriented programs. Therefore Firm differs in several aspects from traditional IRs.

Firm is based on static single assignment (SSA) form. Variables represented in SSA are resolved to data flow edges so that the IR contains no objects to hold local variables. The dataflow representation further allows to include value numbering in the representation. In addition to the dependencies represented in SSA by traditional IRs, Firm represents dependencies between dynamic allocated objects explicitly. Such program objects can not be represented in SSA, so that anti- and output dependencies between these objects are modeled. Further Firm implements a new concept to model exceptions reducing the loss of preciseness of dataflow analysis if exceptions are modeled as control flow changes. [AvR96] provide an efficient implementation of standard dataflow analysis on their initial version of Firm. [Tra99] developed a heap analysis and optimizations of object oriented constructs on Firm improving program performance of object oriented programs to a level of clever implemented imperative programs.

Firm is a low IR, it's operations are similar to operations on target machines. This supports to perform scheduling directly in Firm, so that the scheduler can use all information gathered by any optimization to achieve maximal parallelism. Firm's dataflow representation of variables that can be allocated to registers represents the information needed for scheduling syntactically.

The next Chapter defines the structure of Firm. Section 2.1 lists the syntax of Firm. Section 2.2 explains in detail the semantics of Firm modes and operations. Section 2.3 explains the concept of representating exceptions. The last section gives some examples of Firm graphs.

Chapter 2

Syntax and Semantics

This chapter gives the specification of the syntax and semantics of the intermediate representation Firm. The section about syntax gives a number of types for values called modes known in Firm. Then it introduces a list of operations with operands and results that are typed with these modes. It explains how these operations can be assembled to form a graph and formulates a set of restrictions to such graphs.

The section about the semantics of Firm pragmatically introduces semantic meaning of the operations and modes. From this further restrictions for Firm graphs can be derived. The third Section explains the representation of exceptions, and the last gives some examples of Firm graphs.

2.1 Syntax of Firm

This section defines the syntax of Firm. Firm knows 36 different operations that operate on values of 17 different modes, i.e., it is very lean. Each operation uses a certain number of operands to produce several results. Operands and results are typed with modes.

Firm can easily be extended by additional operations and modes. The intention of Firm is though, to represent a program in a firm manner, using as few different operations and modes as possible. The present specification of Firm allows to represent programs written in any of the major programming languages.

2.1.1 Modes in Firm

The modes in Firm are BB, X, F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b, M.

2.1.2 Operations of Firm

A Firm operation is an operation on a set of operands producing a set of results. The size of these sets is not fixed by the operation. The operation can limit the possible modes of operands and results. The syntax of a firm operation is specified by a name of the operation, the number of operands and results and the modes possible for them.

Table 2.1 lists the syntax of all Firm nodes. The first column mentions the name of the operation. The second column specifies the number and possible modes of operands, the third does so for results. If several modes are possible for a single operand or result the table uses a generic mode. Generic modes are resolved by Table 2.2.

2.1.3 Firm as a graph

Firm represents a program as a directed graph in contradiction to traditional IRs. These view a program as a list of basic blocks where each basic block is an ordered list of instructions or expression trees. Firm knows no such hierarchical decomposition.

A Firm operation is associated with each node. We call a node associated with operation x an x node. Each node has sockets for incoming and outgoing edges which correspond to the operands and results of the operation, so that the operation of a node specifies the number of these sockets and the mode of the edges beginning or ending at these sockets. Each edge is labeled with a mode. There can be only one edge coming into a socket for incoming edges, as the operands of an operation have to be unambiguous. Several edges can start at a socket for outgoing edges. Sockets for incoming edges are also called *inputs* to the node, such for outgoing edges *outputs* of the node.

Figure 2.1 shows an example of a node. Sockets for incoming edges are at the top of the node, such for outgoing edges at the bottom. With this convention we do not need to draw the directed edges as arrows.

Operation	: Modes of Operands	→ Modes of Results
$Block$	: X^n	$\rightarrow BB$
$Start$	: BB	$\rightarrow X \times M \times P \times P \times data_1 \times \ldots \times data_n$
End	: BB	$\rightarrow$
Jmp	: BB	$\rightarrow X$
$Cond$	: $BB \times b$	$\rightarrow X \times X$
$Cond$	: $BB \times I_u$	$\rightarrow X^n$
$Return$	: $BB \times M \times data_1 \times \ldots \times data_n$	$\rightarrow X$
$Raise$	: $BB \times M \times P$	$\rightarrow X \times M$
$Const$	: BB	$\rightarrow data$
$SymConst$	: BB	$\rightarrow I_u$
$SymConst$	: BB	$\rightarrow P$
Sel	: $BB \times M \times P \times I_u^n$	$\rightarrow P$
$Call$	: $BB \times M \times P \times data_1 \times \ldots \times data_n$	$\rightarrow M \times X \times data_{n+1} \times \ldots \times data_{n+m}$
Add	: $BB \times num \times num$	$\rightarrow num$
Add	: $BB \times P \times I_s$	$\rightarrow P$
Add	: $BB \times I_s \times P$	$\rightarrow P$
Sub	: $BB \times num \times num$	$\rightarrow num$
Sub	: $BB \times P \times I_s$	$\rightarrow P$
Sub	: $BB \times I_s \times P$	$\rightarrow P$
Sub	: $BB \times P \times P$	$\rightarrow I_s$
$Minus$	: $BB \times float$	$\rightarrow float$
Mul	: $BB \times num \times num$	$\rightarrow num$
$Quot$	: $BB \times M \times float \times float$	$\rightarrow M \times X \times float$
$DivMod$	: $BB \times M \times num \times num$	$\rightarrow M \times X \times I_s \times I_s$
Div	: $BB \times M \times num \times num$	$\rightarrow M \times X \times I_s$
Mod	: $BB \times M \times num \times num$	$\rightarrow M \times X \times I_s$
Abs	: $BB \times num$	$\rightarrow num$
And	: $BB \times int \times int$	$\rightarrow int$
Or	: $BB \times int \times int$	$\rightarrow int$
Eor	: $BB \times int \times int$	$\rightarrow int$
Not	: $BB \times int$	$\rightarrow int$
Shl	: $BB \times int \times I_u$	$\rightarrow int$
Shr	: $BB \times int \times I_u$	$\rightarrow int$
$Shrs$	: $BB \times int \times I_u$	$\rightarrow int$
Rot	: $BB \times int \times I_u$	$\rightarrow int$
Cmp	: $BB \times datab \times datab$	$\rightarrow b^{16}$
$Conv$	: $BB \times datab_1$	$\rightarrow datab_2$
Phi	: $BB \times dataM^n$	$\rightarrow dataM$
$Load$	: $BB \times M \times P$	$\rightarrow M \times X \times data$
$Store$	: $BB \times M \times P \times data$	$\rightarrow M \times X$
$Alloc$	: $BB \times M \times I_u$	$\rightarrow M \times X \times P$
$Free$	: $BB \times M \times P$	$\rightarrow M$
$Sync$	: $BB \times M^n$	$\rightarrow M$

Table 2.1: Syntax of Firm operations. For resolution of generic modes *data* etc. see Table 2.2.

$data, data_i$	$\in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$
$datab, datab_i$	$\in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b\}$
$dataM$	$\in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, M\}$
num, num_i	$\in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$
int	$\in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$
$float$	$\in \{F, D, E\}$

Table 2.2: Generic modes of operations in Table 2.1

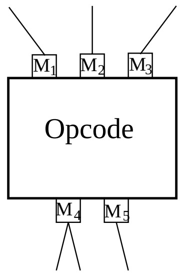


Figure 2.1: Graphic representation of a Firm node *Opcode* : $M_1 \times M_2 \times M_3 \to M_4 \times M_5$

2.1.4 Further restrictions to Firm Graphs

We say, a node is in a *Block*, or, the *Block* contains the node if the *Block* is an input of the node. We define two classes of subgraphs of Firm Graphs.

Definition 2.1.1 (control flow graph) *The control flow graph,* CFG, *consists only of* Block *nodes. Control flow edges are edges between two* Block *nodes A and B, if there is an execution edge between the* Jmp/Cond *or other control flow node in* Block *A and* Block *B.*

Definition 2.1.2 (intra-block graph) *The intra-block graph,* IBG, *of a* Block *B contains B and all its direct successors. Further it contains all Firm edges between its nodes, except inputs to* Phi *nodes and the* Block *node, even if their origin is within the* IBG.

Further we can define a subclass of all edges.

Definition 2.1.3 (inter-block edge) *An inter-block edge is an edge not contained in any* IBG.

Based on these definitions, we can give the following restrictions to legal Firm graphs.

- There exists a *Block* that contains a single *Start* node. This is the only *Start* node in the Firm graph. This *Block* has no predecessors. We call it the start block.
- There exists a *Block* that contains a single *End* node. This is the only *End* node in the Firm graph. This *Block* has no successors. We call it the end block.
- All *Block* nodes in a Firm graph have to be on a path in the CFG from the start block to the end block.
- IBGs in Firm graphs are acyclic.
- *Phi* nodes have as many inputs as the *Block* they belong to.
- An inter-block edge from node a in *Block* A to node b in *Block* B, b not a *Phi* node, is legal if there exists a path in the CFG from A to B.
- An inter-block edge from node a in *Block* A to the i-th input of a *Phi* node b in *Block* B is legal if there exists a path in the CFG from A to the i-th predecessor of B.

Further restrictions apply to edges of mode memory. These can not be expressed as pure syntactic concepts, as they refer to the semantics of the Firm graph. Therefore their discussion is delayed to Section 2.2.

2.2 Semantics of Firm modes and operations

A Firm graph represents a program. Its operations are operations of the program, the operands and results represent data or control flow dependencies between these operations. The modes give the kind of dependency. Firm operations are strict, i.e., they can only execute if all their operands are available. Exceptions are the *Block* and *Phi* operations which execute if only one of their operands is available.

2.2.1 Modes

Several different kinds of Firm modes can be distinguished. Section 2.2.1.1 lists modes used to specify the control flow in the program represented in

Firm. Further there are modes labeling all data objects known by the program represented by a Firm graph. These are separated into primitive value modes for operands represented by Firm in SSA-Form (Section 2.2.1.2) and data objects not representable by SSA (Section 2.2.1.3). SSA assumes infinite resources, i.e., value numbers, so that false and anti dependencies are removed from the representation. A later transformation, register allocation, remaps the infinite representation to the finite register set. Data objects stored in the heap can not be represented in SSA-form because some language semantics require that a variable is always mapped to the same memory location. In general dynamically created objects can not be value numbered. Therefore Firm introduces a special mode *Memory*. Edges of mode *Memory* represent all dependencies, i.e., true, false and anti, introduced by data transfers not expressible in SSA.

2.2.1.1 Control flow modes

There are two different control flow modes. The first, blockmode (BB), specifies the block structure as implicitly given by the source program[1]. In traditional IRs, where operations are specified in triple or quadruple form, the order of the operations specifies the separation into basic blocks. In a graph based IR such as Firm it is not necessary to order instructions sequentially. Firm nodes can be executed as soon as their inputs are available. Therefore the traditional definition of basic blocks is not applicable to Firm. Still it must be guaranteed for some operations that they are executed before a distinguished control flow altering instruction. This is expressed by BB edges.

To represent the semantics of a program correctly it is not necessary to link every operation to a basic block node. The data dependencies often suffice to fix the operation to several basic blocks where they can be executed legally. A steady state of Firm though requires that all nodes (except *Block* nodes) are attached to a *Block* node.

BB	blockmode
X	execution

Table 2.3: Control flow modes

Control flow can be altered explicitly by the program. An operation

[1]There is a mode 'Block', also called blockmode and a node '*Block*'.

mode		signedness	size	alignment
F	float	–	32 bit	4
D	double	–	64 bit	4
E	extended	–	80 bit	4
B	byte	signed, unsigned	8 bit	1
H	short integer	signed, unsigned	16 bit	2
I	integer	signed, unsigned	32 bit	4
L	long integer	signed, unsigned	64 bit	4
C	char	–	8 bit	1
P	pointer	–	32 bit	4
b	boolean	–	–	–

Table 2.4: Primitive value modes

altering the control flow produces a result of mode execution, X, which gives the next basic block to execute. I.e., execution edges point form control flow operations to *Block* nodes. Conditional jumps produce two or more values of mode X, one of which is 'execute', the others are 'do not execute'. Only one control flow operation can be attached to each *Block*.

The concept of exceptions in Firm involves that operations that can raise exceptions have an output of mode X. These operations are not considered as control flow operations, several of them can be attached to a block. Firm does not express the ordering constraints imposed by the exception semantics as control flow, see Section 2.3.

2.2.1.2 Primitive value modes

Table 2.4 lists the modes for primitive values. These are the modes corresponding to data types as requested by the target of the compilation. A value that is the operand or result of an operation has to be of the mode specified by the operation for that operand / result.

The mode boolean takes an extra role. It stands for the truth value produced by compare operations. This is not a boolean variable as it might be defined by the source language, and therefore can not be written to memory. To save the outcome of the *Cmp* operation in a variable, it needs to be converted to a data mode, e.g., byte. A Compare operation allows to convert an integer mode to boolean.

2.2.1.3 The Memory mode

M	Memory

Table 2.5: Memory mode

Memory expresses dependencies through memory. It is defined as a function from pointers P to primitive values V, where V is the union of all primitive value modes, except boolean:

$$M : P \to V$$

$$V = F \cup D \cup E \cup B_s \cup B_u \cup H_s \cup H_u \cup I_s \cup I_u \cup L_s \cup L_u \cup C \cup P$$

As most operations only depend on a part of the memory, we define

$$M_i : P_i \to V$$

on a subset P_i of P. Any operation with a memory operand or result needs only a part M_i of the full memory M as operand.

2.2.2 Operations

Each Operation is given as a Function from zero or more operands to zero or more results. As all operations except *Block* have an operand of mode BB, an the meaning of this operand is always the same, we leave out this mode. This section describes which results an operation derives from its operands, and how to use the operation.

2.2.2.1 Basic Blocks

Block : $X^n \to BB$

A *Block* operation groups operations to a basic block. All operations consuming the result of the *Block* belong to this basic block. Operands represent control flow from other basic blocks to this one. The *Block* operation is not strict, i.e., it is executable if one of the operands is available.

A basic block in Firm has a completely different concept than the traditional notion of a basic block, as there exists no total order on the operantions of a Firm program. For a detailed discussion see section 2.2.1.1.

Block operations are no real operations, as they are an auxiliary construct to specify the control flow. They are not performing an explicit operation on any inputs. *Block* operations just transport the control flow from the predecessor operation of the basic block to the operations in the basic block.

As all operations except the *Block* operation have an input of mode BB, we implicitly assume it as zeroth input in the following paragraphs.

2.2.2.2 Control flow operations

A control flow operation has to be executed as the last operation in a basic block. This is not expressed explicitly by dependencies between all other operations in the basic block and the control flow operation. This reduces the number of operands in the graph considerably and simplifies many optimizations. Only the code generation might need these dependencies, but the code generator can compute them by finding the control flow operation attached to a basic block.

Firm does not consider all operations producing a result of mode execution to be control flow operations. Operations that can cause exceptions have a result of mode execution consumed by the basic block with the exception handler. The result passed to the handler is usually 'do not execute', only if an exception is raised it is 'execute'. These operations are not considered control flow operations so that they do not end a basic block. This would reduce the size of basic blocks and increase the number of basic blocks which complicates dataflow analysis and reduces the effect of optimizations. In general, the code generation will not generate branch instruction for this control flow as implicit exceptions are handled by hardware and the exception handler.

Start : $\rightarrow X \times M \times P \times P \times T_1 \times \ldots \times T_n$
where $T_1, \ldots, T_n \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$

The *Start* operation belongs to a basic block that solely contains this node. It marks the procedure entry. Its execution result indicates the first basic block to be executed, i.e, is an operand of that block. The memory result describes the state of the memory when the procedure is called. The first pointer result gives the procedure's stack frame, i.e., it models the stack pointer. Firm also could model the stack frame as part of the memory by allocating it explicitly to the initial memory with an Alloc node. The second result of mode pointer represents a pointer to the global memory of the program containing global objects as well as procedures, i.e., the heap pointer. $T_1, \ldots, T_n$ are the parameters of the procedure. For details of the implementation of the *Start* operation see section 2.2.2.8.

Firm represents all alias free variables and values as edges in a dataflow graph. Therefore no representation of the stack frame location of atomic local variables is needed. Unfortunately the stack frame still needs to be modeled to contain statically allocated arrays and other compound data types which can not be represented as a single operand.

End $: \rightarrow$

The *End* operation models the end of the control flow in this procedure. It belongs to a basic block that solely contains this node. All results of *Return* operations must be operands of the basic block containing the *End* operation.

Jmp $: \rightarrow X$

Jmp is an unconditional branch to the basic block that has its result as operand.

Cond $: b \rightarrow X \times X$
Cond $: I_u \rightarrow X^n$

Cond is the conditional branch. There are two versions of the *Cond* operation. The result of the first are two control flows, the first is taken if the boolean operand is true, else the second is taken. Its input is the result of a Comp operation, i.e., the truth value computed by a Comp, not a boolean variable as it might be defined by the source language. The other *Cond* operation models switch commands. If its operand is i, then control flow will proceed along result x_i. If its operandd is $\geq n$, control flow will proceed along result x_n. A *Cond* has to be executed as the last operation in a basic block. This is not expressed explicitly by dependencies between all other operations in the basic block and the *Cond* operation, see section 2.2.2.2.

Return $: M \times T_1 \times \ldots \times T_n \rightarrow X$
where $T_1, \ldots, T_n \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$

The operands of the *Return* operation are the results of the procedure and the state of memory after execution of the procedure. The result of the *Return* operation passes execution to the basic block of the procedure that contains the *End* operation.

Raise : $M \times P \rightarrow X \times M$

Raise raises an explicit exception. The operands to the operation are a pointer to an *Except* variable, as described in section 2.3 and the part of the memory that contains this variable. The execution result gives the exception handler handling the appropriate exception if it is defined within this procedure. Else it points to the basic block of the procedure that contains the *End* operation.

2.2.2.3 Constants

$\boldsymbol{Const^{tv}}$: $\rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$

The *Const* operation returns a constant value of mode T. This value is stored as a target value in the constant table, *tv* is an attribute that points to the proper entry in a constant table.

$\boldsymbol{SymConst^{type}}$: $\rightarrow I_u$
$\boldsymbol{SymConst^{name}}$: $\rightarrow P$

SymConst (symbolic constant) allows to delay decisions about the data layout. It is used for values depending on the memory layout of data objects which can be changed by optimizations. It also simplifies finding a clean representation of type tags if several modules are linked together, as breaking tags down to integers can be delayed until all tags are known. The attribute *type* gives the type the symbolic information refers to. If *SymConst* stands for a type tag, the attribute *type* gives the type the tag has to represent. If it is a size it is the size of the type given in the attribute.

Further *SymConst*s can be used to represent symbolic information needed for the linker. E.g., addresses of global variables are not known at compile time. Information about these variables is communicated to the linker by introducing explicit names. *SymConst*s representing such addresses have a result of mode pointer and an attribute containing the explicit name. These *SymConst*s should not be introduced by a frontend or before the major optimizations. They are introduced by lowering *Sel* nodes.

2.2.2.4 The Select node

$\boldsymbol{Sel^{ent}} \quad : \quad M \times P \times I_u^n \rightarrow P$

The *Sel* operation selects a single attribute out of an object. This object can be the stack frame, a compound object or an array. Further *Sel* can select a global variable from a virtual global frame that will be further specified by the linker. Its operands are the state of the memory, a pointer to the object and eventually several indexes. The attribute to select is given as an annotation. *Sel* returns a pointer to the attribute specified by *ent* in the given object. In case the owner of the *ent* is an array, it returns a pointer to the array element given by the indexes.

Sel allows to hide the addressing mechanism within the object. This restricts the parts of the program represented in Firm to the operations intended by the user, enabling a compact representation and giving room for optimizations of, e.g, the memory layout. The *Sel* operation for example implements polymorphy, i.e., if *ent* labels a polymorphic method, a pointer to the proper method is returned.

A lowering phase exposes this implicit functionality by generating the addressing code, which now should be subject to further optimization. Lowering *Sel* nodes that select global variables generates *SymConst* nodes.

2.2.2.5 Arithmetic operations

The semantic of the arithmetic operations is the obvious one for a given mode.

$\boldsymbol{Call^{type}} \quad : \quad M \times P \times T_1 \times \ldots \times T_n \rightarrow M \times T'_1 \times \ldots \times T'_m$
where $T_1, \ldots, T_n, T'_1, \ldots, T'_m \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$

The memory state, a pointer to the procedure to be called and the procedure parameters of modes $T_1, \ldots, T_n$ are operands to the *Call* operation. It returns the results with the modes $T'_1, \ldots, T'_m$ as computed by the procedure called, and a changed memory state. The pointer to the procedure can be the result of a *Sel* operation. The *type* attribute points to the entry for the called procedure in the type table. For details about the implementation of the *Call* operations in the Firm graph see 2.2.2.8.

$\boldsymbol{Add} \quad : \quad T \times T \rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$,

Add : $P \times I_s \rightarrow P$
Add : $I_s \times P \rightarrow P$

The *Add* operation has two operands for two data items, and a single result for the result of the addition. In general the modes of the operands and the result have to be identical, an exception is made for pointer arithmetic.

Sub : $T \times T \rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$,
Sub : $P \times I_s \rightarrow P$
Sub : $I_s \times P \rightarrow P$
Sub : $P \times P \rightarrow I_s$

The *Sub* operation takes two operands for two data items, and yields the result of the subtraction. The second operand is subtracted from the first one. In general the modes of the operands and the result have to be identical, an exception is made for pointer arithmetic. The last typing allows to compute sizes of the memory region between the two pointers.

Minus : $T \rightarrow T$
where $T \in \{F, D, E\}$,

The *Minus* operation additive inverts its operand. It is necessary as an additive inversion of a floating point value can not be modeled as a subtraction from zero. This can cause rounding errors.

Mul : $T \times T \rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The result of the *Mul* operation is the multiplication of its two operands.

Quot : $M \times T \times T \rightarrow M \times X \times T$
where $T \in \{F, D, E\}$

The *Quot* operation performs exact division. It has two operands for two data items, and one result for the result of the division. It divides the first operand by the second one. The memory operand and result are used to model exceptions, see section 2.3.

DivMod : $M \times T \times T \rightarrow M \times X \times I_s \times I_s$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *DivMod* operation has two operands for two data items, and two results for the results of the operations integral division and integral remainder. The first operand is divided by the second operand. The memory operand and result are used to model exceptions, see section 2.3.

Div : $M \times T \times T \rightarrow M \times X \times I_s$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Div* operation has two operands for two data items, and returns the result of the integral division. The first operand is divided by the second operand. The memory operand and result are used to model exceptions, see section 2.3.

Mod : $M \times T \times T \rightarrow M \times X \times I_s$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Mod* operation has two operands for two data items, and returns the result of the integral remainder operation. The first operand is divided by the second operand. The memory operand and result are used to model exceptions, see section 2.3.

Abs : $T \rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Abs* operation returns the absolute value of its single operand.

And : $T \times T \rightarrow T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *And* operation performs bitwise and.

Or : $T \times T \rightarrow T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Or* operation performs bitwise or.

Eor : $T \times T \to T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Eor* operation performs bitwise exclusive or.

Not : $T \to T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Not* operation performs bitwise negation.

Shl : $T \times I_u \to T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Shl* operation shifts the first operand by as many bits as given by the second operand to the left.

Shr : $T \times I_u \to T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Shr* operation shifts the first operand by as many bits as given by the second operand to the right. It performs logical shifts, i.e. the result is zero extended.

Shrs : $T \times I_u \to T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Shrs* operation shifts the first operand by as many bits as given by the second operand to the right. It performs arithmetic shifts, i.e. the result is sign extended.

Rotate : $T \times I_u \to T$
where $T \in \{B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u\}$

The *Rotate* operation rotates its first operand by as many bits to the left as the second operand specifies.

Cmp : $T \times T \to b^{16}$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b\}$

Compares are treated differently in various processor architectures. Firm separates compares and branches by breaking up conditional branches to

notation	semantics				
	?	$>$	$<$	$=$	
False	f	f	f	f	(always false)
Eq	f	f	f	t	$=$
Lt	f	f	t	f	$<$
Le	f	f	t	t	$\leq$
Gt	f	t	f	f	$>$
Ge	f	t	f	t	$\geq$
Lg	f	t	t	f	$<$ or $>$
Leg	f	t	t	t	ordered
Uo	t	f	f	f	unordered
Ue	t	f	f	t	unordered or $=$
Ul	t	f	t	f	unordered or $<$
Ule	t	f	t	t	unordered or $\leq$
Ug	t	t	f	f	unordered or $>$
Uge	t	t	f	t	unordered or $\geq$
Ne	t	t	t	f	$\neq$
True	t	t	t	t	(always true)

Table 2.6: Compare operations

achieve a straightforward representation that can be mapped to any implementation of compares in a processor. To avoid that optimizations separate compares and branches, a special mode boolean is introduced whose values can not be saved to memory. This simplifies code generation.

The *Cmp* operation implements all compares simultaneously. It compares its two operand values and produces sixteen boolean values as result, one for each compare operation. Firm uses four orthogonal compare predicates, equal, less, greater and unordered. The sixteen results are the possible combinations of these predicates. They are illustrated in table 2.6. For an example see section 2.4.

Conv : $T_1 \rightarrow T_2$
where $T_1, T_2 \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b\}$

The *Conv* operation converts a value of mode T_1 to a value of mode T_2. The conversion in Figure 2.2 and all transitive conversions are supported by the implementation.

$$B_s \rightarrow H_s \rightarrow I_s \rightarrow L_s$$

$$B_u \rightarrow H_u \rightarrow I_u \rightarrow L_u$$

$$B \rightarrow H, \quad H \rightarrow I, \quad I \rightarrow L$$

$$F \rightarrow D \rightarrow E$$

$$C \rightarrow H_u$$

$$b \rightarrow B$$

$$P \rightarrow I_u$$

Figure 2.2: Conversion between Primitive Value Modes. (If signedness is omitted both are possible.)

2.2.2.6 The Phi operation

Phi : $T^n \rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, M\}$

A *Phi* operation is always attached to a basic block. It has one operand for each control flow predecessor of the basic block. The i-th operand is a definition valid at the end of the i-th predecessor of the basic block. During execution of the compiled program the basic block will have an unambiguous predecessor. The operand of the *Phi* operation corresponding to this basic block will be available when the operation is executed, and the *Phi* operation returns this operand, i.e., if the *Phi* operation is reached through the i-th predecessor block, it returns its i-th operand. The *Phi* operation is not strict, it can execute if only one of its operands is available.

2.2.2.7 Operations to manage memory explicitly

The following operations model dependencies through memory explicitly by specifying the memory touched by the operation. They all need an operator of mode memory as operand and produce an eventually modified memory as result in addition to the standard operands and results.

To allow as precise information about the dependencies as possible partial memory functions can be specified. (See also Section 2.2.1.3.) These include only those parts of the memory that actually might be touched by the operation. A heap or alias analysis can sharpen this information.

Load : $M \times P \rightarrow M \times X \times T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$

This operation reads a data item of mode T from the memory location given as second operand. The result of mode memory is necessary to model anti dependencies through memory. For the treatment of exceptions see Section 2.3.

Store : $M \times P \times T \rightarrow M \times X$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$

This operation writes a data item of mode T to the memory location given as second parameter. The memory operand and result are used to model memory dependencies. For the treatment of exceptions see Section 2.3.

Alloc$^{\boldsymbol{type}}$: $M \times I_u \rightarrow M \times X \times P$

The *Alloc* operation allocates memory for a variable. The size of the memory needed is given by the second operand. The operation is annotated with the type *type* for which the memory is allocated. This is a pointer to the type information of the program representation. *Alloc* returns a new memory expanded by the new location, and a pointer to this location. The new memory location is not initialized. For the treatment of exceptions see Section 2.3.

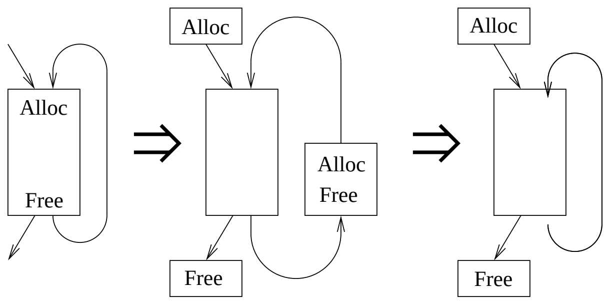
For many optimizations on Firm it is essential to know about memory allocation. A frontend compiling languages as C should transform calls as to malloc to an *Alloc* operation. Later the backend will lower these to a funcion call again.

Free$^{\boldsymbol{type}}$: $M \times P \rightarrow M \times X$

The *Free* operation frees the memory of a variable. It is annotated with the type *type* for which memory is freed. The memory returned no longer contains space for this variable.

If Firm is used to translate a language with explicit memory deallocation, *Free* operations can be generated by the frontend. For garbage collected languages a static garbage collection can introduce free operations.

Representing allocation and deallocation as explicit operations allows interesting optimizations, as, e.g., shown in Figure 2.3.



1. Move *Alloc* against control flow out of loop.
2. Move *Free* with control flow out of loop.
3. Merge adjacent *Alloc Free* pairs.

Figure 2.3: Moving allocation out of a loop.

Sync : $M^n \to M$

The *Sync* operation unifies several partial memory regions. These regions have to be pairwise disjunct, or the values in common locations have to be identical. This operation allows to specify all operations that eventually need several partial memory regions as operand with a single entrance by unifying these memories with a preceding *Sync* operation.

2.2.2.8 Special operations simplifying the representation

We want to implement edges in a Firm graph as pointers stored in the nodes themselves to allow fast navigation. This means that an edge can not point to a particular result of a predecessor in the graph, only to the whole node. If all operations have only one result this implementation is possible. Therefore we view nodes returning several values as nodes that return a single value that is a tuple consisting of the individual values. We introduce a special operation extracting particular values from these tuples and a special mode, *Tuple*, that is used for operations that produce a tuple as result. Figure 2.4 illustrates this concept.

T	Tuple

Table 2.7: Auxiliary mode Tuple

$Proj^i$: $T \to U$

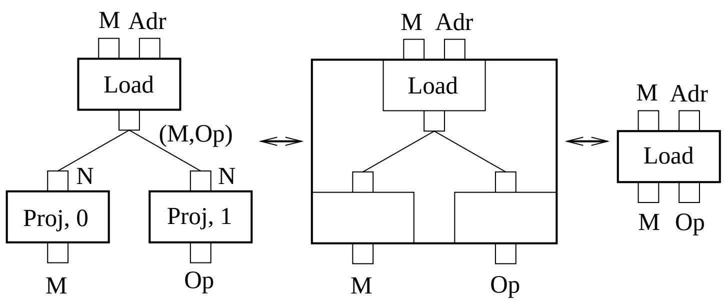


Figure 2.4: Conversion between one-exit and multi-exit representations

where $U \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b, M, X, T\}$

This operator is used to extract a single value from a tuple. Its operand is a tuple. The operation is annotated with the position i of the value to extract from the operand tuple. It returns the extracted value. Figure 2.4 shows how a graphic representation with operations that have one exit can be projected to a representation with several exits using the *Proj* operations.

Start **and** ***Call*** **revisited**

The operations *Start* and *Call* are implemented with nested tuples. Two *Proj* operations are needed to extract a procedure parameter. An example can be seen in Section 2.4, Figure 2.17.

Start : $\rightarrow T_{start}$
where $T_{start} = X \times M \times P \times P \times T_{args}, T_{args} = data_1 \times \ldots \times data_n$

Call^{type} : $M \times P \times T_1 \times \ldots \times T_n \rightarrow T_{call}$
where $T_1, \ldots, T_n \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P\}$,
$T_{call} = M \times X \times T_{results}$,
$T_{results} = data_1 \times \ldots \times data_n$.

2.2.2.9 Operations used to hold intermediate information during optimization

Id : $T \rightarrow T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b, X, M\}$

The *Id* operation is used to simplify the implementation of optimizations.

It has no functionality, i.e., its results are the same as its operands. If an optimization replaces an operation by other ones or removes it, an *Id* is inserted taking the place of the original operation. (No new operation is allocated, only the label of the operation is changed to *Id*.) This allows that the references to the original operation need not be changed. Dead operation elimination will later remove the *Id* operations. Figure 2.5 shows the use of *Id* operations: The optimization iterated over the Firm graph looking for opportunities for strength reduction. Arriving at the *Mult* node it decided to replace the multiplication by two by an addition. The optimization has no access to the users of the value produced by the *Mult* as the edges are directed backwards. Therefore it turns the *Mult* into an *Id* node and inserts the new *Add* before.

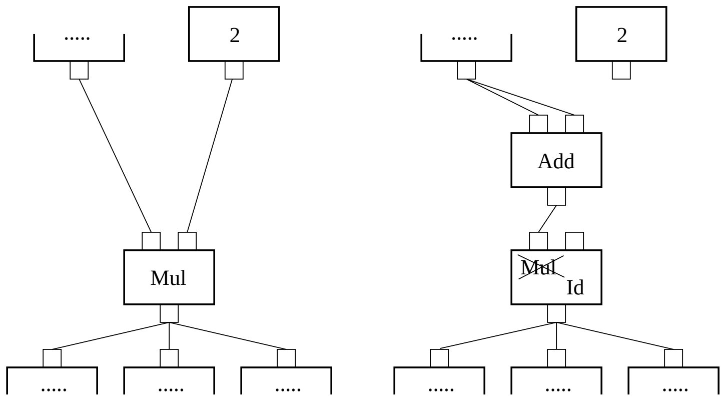


Figure 2.5: Transformation of Firm graph using Id.

Tuple : $T_1 \times \ldots \times T_n \to N$
where $T_1, \ldots, T_n \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b, X, M\}$

The *Tuple* operation combines single values into tuples. It is used to transform Firm graphs. Optimizations that replace operations that produce tuples as result by several operations use this operation to form a tuple so that the proceeding *Proj* operations need not to be touched. (No new operation is allocated, only the label of the operation is changed to *Tuple*.) A later pass can fold the *tuple* and *Proj* operations.

Bad : $\to T$
where $T \in \{BB, X, F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b, M, T\}$

The *Bad* node is used to represent dead values. An operand of a node that is a *Bad* node indicates that this value will never be computed during runtime of the program. If this is a strict operation this means that also this operation will never be executed. If it is a *Phi* or *Block* node this means that control will never reach this node by this edge.

Figure 2.6 shows how to use the *Tuple* and *Bad* operation. The optmization decided that it can evaluate the *Cond* operation statically. Therefore it replaces the *Cond* by a *Tuple* of two static control flow operations: *Bad* for the branch that is never taken, and *Jmp* for the branch that is always taken. To replace the operation an *Id* is used. Further optimization steps will remove the *Tuple*, *Id* and *Proj* nodes.

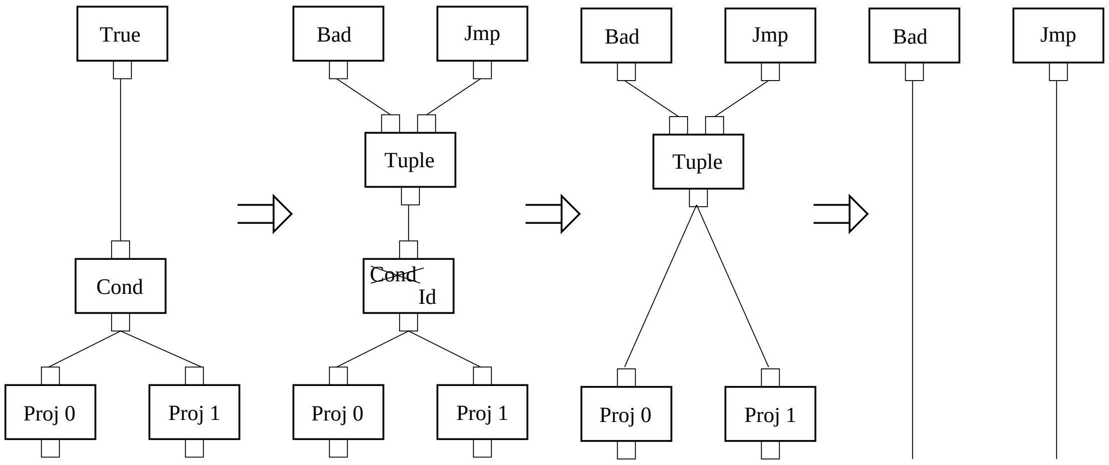


Figure 2.6:

Confirm : $T \to T$
where $T \in \{F, D, E, B_s, B_u, H_s, H_u, I_s, I_u, L_s, L_u, C, P, b, M\}$

Used to represent abstract knowledge about a value that is higher than no information, i.e., the value can be anything allowed by the mode, and lower than a constant value. This information can be derived from Cmp/Cond operation combinations.

2.3 Exceptions

Exceptions pose two problems for optimizations. They complicate the control flow, reducing the effect of optimizations that depend on large basic blocks. Further they slow down optimizations whose runtime depends on the number of nodes in the control flow graph. Exceptions also decrease the preciseness of optimization algorithms that make conservative assumptions if information is incomplete. Therefore modeling exceptions efficiently is crucial for an intermediate representation as Firm, which is designed to support aggressive optimization.

In traditional intermediate representations exceptions are modeled as conditional branches, i.e., as control dependencies. An operation that can raise an exception (a fragile operation) ends the basic block. The control flow branches to the exception code if this operation fails . If an optimization shows that the fragile operation can not cause an exception, the control flow edge to the exception code can be removed. This model sequentializes all fragile operations, even if this is not enforced by the source language, restricting optimizations severely.

Therefore Firm represents exceptions by data dependencies according to the new approach developed by [Tra99]. It introduces an abstract variable *Except*, which models dependencies between fragile operations through memory.

At the beginning and end of a protected region the variable *Except* is defined by two auxiliary operations. The fragile operations in the region use and define this variable — they all have an operand and result of mode M. E.g., they might write information about the position of the instruction in the source program to this variable. This guarantees that these operations can not be moved out of the protected region, the definitions mark the beginning and end of the protected region, but the operations within the region are not ordered. Further restrictions by the source language on the order of exceptions can be modeled by adding uses and definitions of *Except* or by introducing additional abstract variables.

At the beginning of the exception code Firm introduces a *Phi* operation, that merges the memory results of all fragile operations in the protected region. The control flow predecessors of the basic block with the exception code are the fragile operations in the protected region. If there are several fragile operations in a single region there are as many control flow edges to the block containing the exception code. With this concept the precise memory state at the point where the exception was raised is known, so that a program analysis has all information about the environment if it analyzes the exception code. This reduces the loss of preciseness when the analysis merges the information after the exception code with that after the basic block containing the protected region.

2.4 Examples

This section gives some examples of interesting Firm graph sections.

The example in figure 2.7 to 2.9 illustrate how code for a basic blocks is represented in Firm. Figure 2.7 gives a small basic block ended by a Jmp operation. Figure 2.8 shows how the basic block is represented in Firm. One can see that all arithmetic nodes are attached to the Block node. The order of the arithmetic operations is relaxed. Figure 2.9 shows an alternative drawing of the *BB* edges: affiliation to a block is represented by grouping the operations into a sqare box. This reduces the number of edges making the graph better readable.

```
a = 2 + 1
b = a + 1
c = a - b
Jmp
```

Figure 2.7: Code of a basic block.

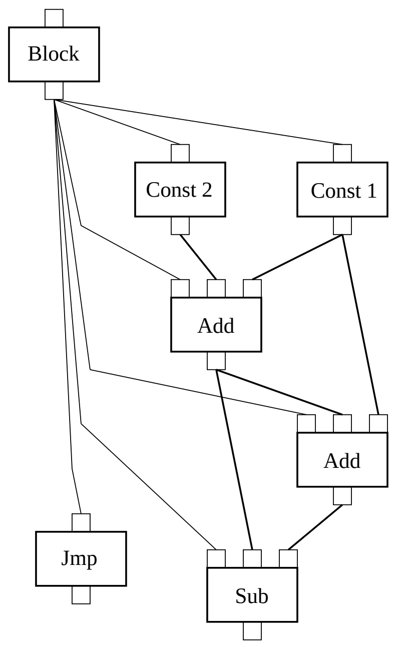


Figure 2.8: A Firm graph of a basic block according to Figure 2.7 illustrating the use of BB edges.

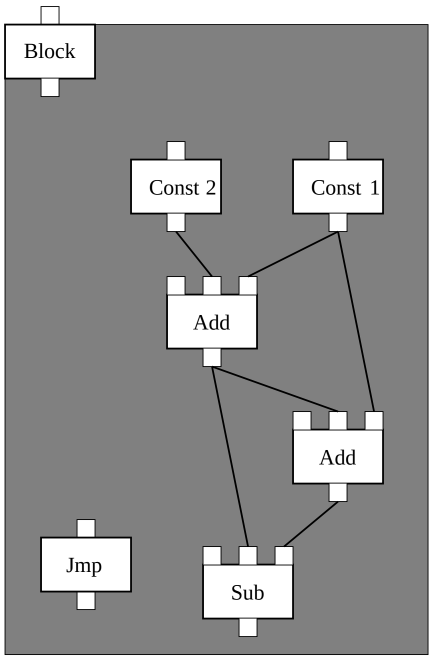


Figure 2.9: A different representation of Block operations illustrated on the basic block in figure 2.7.

The example in Figures 2.10 and 2.11 illustrates the representation of control flow and the use of *Phi* nodes.

```
a = a + 2
if (...) { a = a + 2 }
b = a + 2
```

Figure 2.10: An if statement with two reaching definitions.

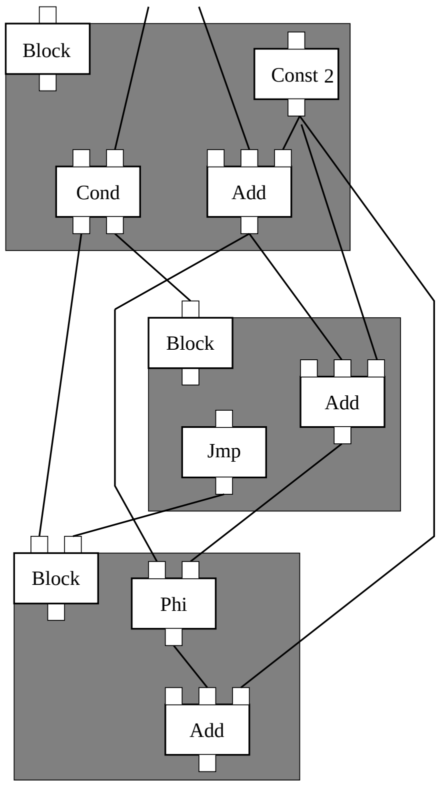


Figure 2.11: SSA representation of several reaching definitions according to example 2.10.

Figures 2.12 and 2.13 demonstrate the use of memory edges to sequentialize operations. The *Alloc* node allocates a piece of memory of the size given by operator *size*. It adds the new piece of memory to the memory passed as operand and returns the extended memory as well as a pointer to the new location. The *Load* node now reads a value from this location. The pointer passed as operand must point to a location in the memory passed. The *Load* returns the unchanged memory and the loaded value. Arithmetic operations now can change the value. Later the *Store* writes the new value to the same location. The *Store* could as well receive the memory output by the *Alloc* as this is the exact same one as returned by the *Load*. But by consuming the *Loads* memory the sequentialization of the *Store* after the *Load* is guaranteed. The *Store* returns a memory that contains the new value at the location.

```
a = malloc(1)
*a = *a + 1
b = *a
```

Figure 2.12: An example with dynamic allocated variables. For sake of brevity we ignore that the value `*a` is undefined.

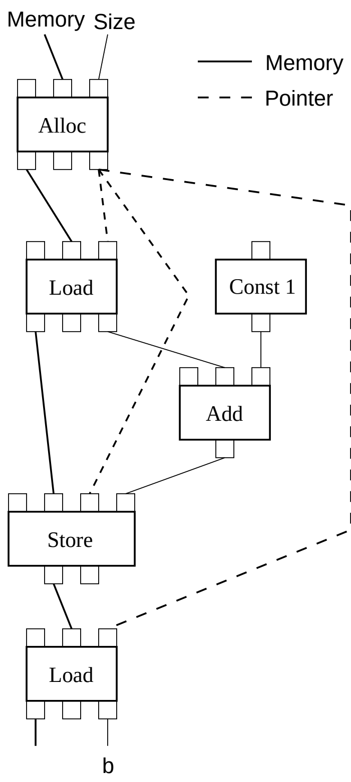


Figure 2.13: *Load* and *Store* operations sequentialized by memory edges. This graph implements the code in Figure 2.12.

Figures 2.14 to 2.16 illustrate the use of the *Sel* and *Call* nodes. Figure 2.14 shows a code fragment that allocates an object, writes a field of the object and calls a method of the object. Figure 2.15 illustrates how a *Sel* node is used to generate the address of the field: The pointer to the object, in this case a result of the *Alloc* node, is passed to the *Sel* node. The node knows about the type X of the object as well as the entity a to select. It returns the pointer to a. Figure 2.16 implements the same for the call.

```
class X {
  int foo(int);
  int a;
}

X x = new X();

x.a = 17;
x.foo(17);
```

Figure 2.14: Use of an object field `a` and a method `foo()`.

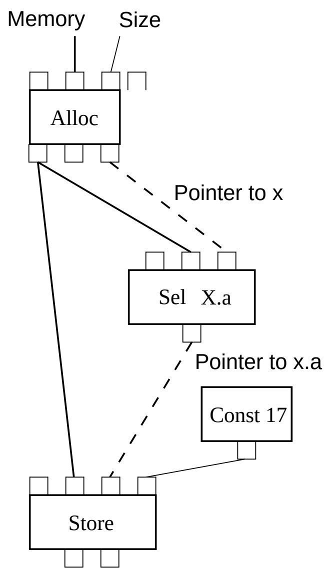


Figure 2.15: Assignment to a field of an object. This graph implements the assignment in the code in Figure 2.14.

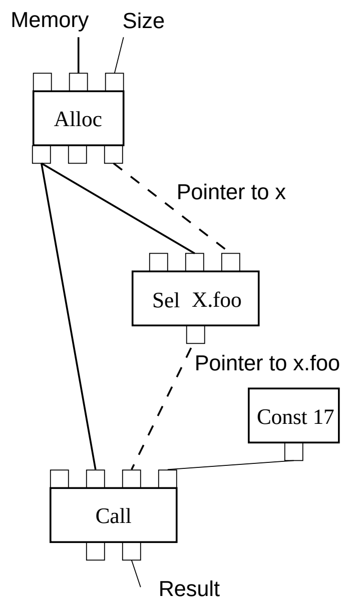


Figure 2.16: Call to a method. An example for resolving polymorphy. This graph implements the call in the code in Figure 2.14.

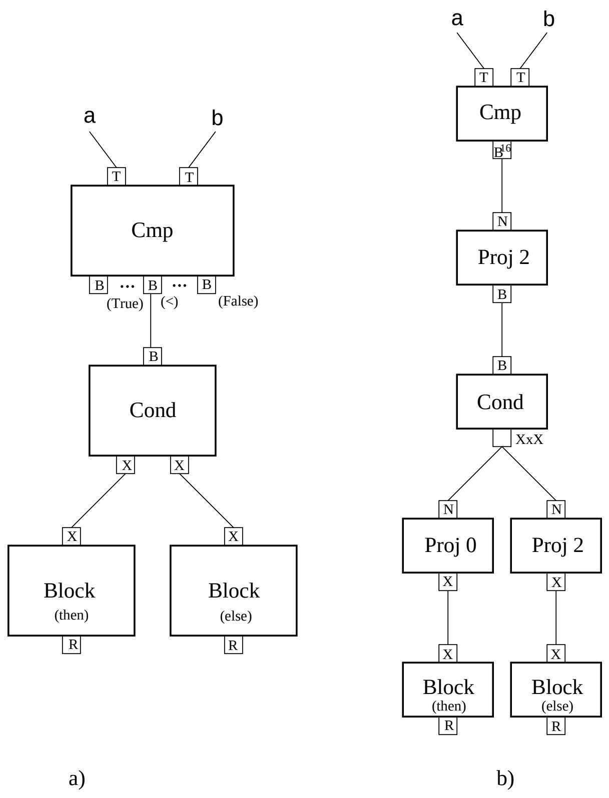


Figure 2.17: A Firm graph for the statement `if (a<b) then ... else ....` This illustrates the representation of conditional branches with the *Cmp* operation. a) shows the representation with multiple exits, b) with single exits and *Proj* nodes.

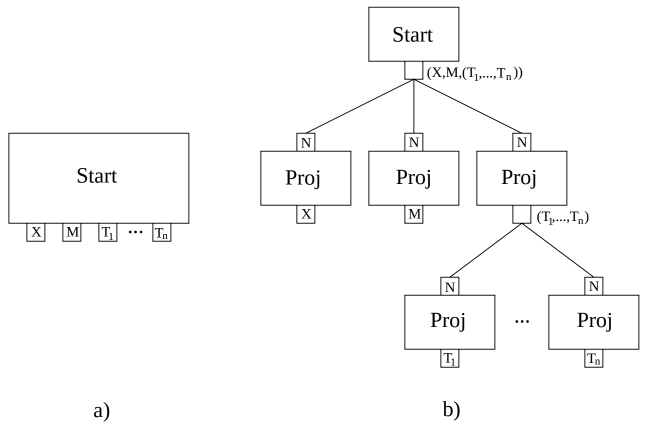


Figure 2.18: A *Start* node. a) shows the representation with multiple exits, b) with single exits and *Proj* nodes.

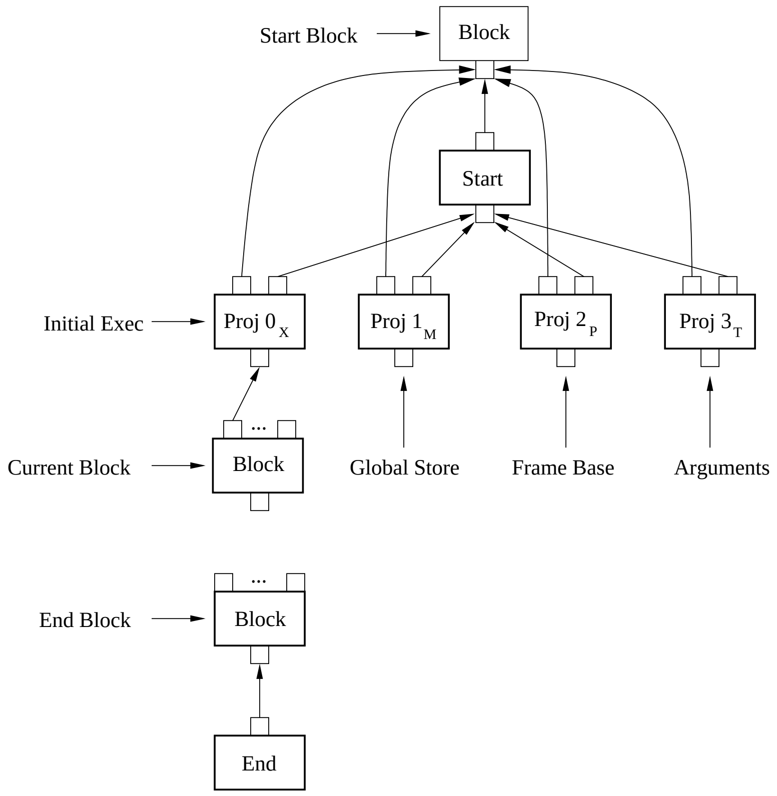


Figure 2.19: Initial graph built by new_ir_graph()

Bibliography

[AvR96] Markus Armbruster and Christian von Roques. Entwurf und Realisierung eines Sather-K Übersetzers. Master's thesis, Dept. of Computer Science, University of Karlsruhe (TH), December 1996. In German.

[GS96] Gerhard Goos and Heinz Schmidt. Sather-K the language. Technical report, Dept. of Computer Science, University of Karlsruhe (TH), October 1996.

[Tra99] Martin Trapp. *Optimierung Objektorientierter Programme.* PhD thesis, Dept. of Computer Science, University of Karlsruhe (TH), December 1999. In German.