FIRM—A Graph-Based Intermediate Representation

Matthias Braun Sebastian Buchwald Andreas Zwinkau
Karlsruhe Institute of Technology (KIT)
{matthias.braun,buchwald,zwinkau}@kit.edu

Abstract

We present our compiler intermediate representation FIRM. Programs are always in SSA-form enabling a concise graph-based representation. We argue that this naturally encodes context information simplifying many analyses and optimizations. Instructions are connected by dependency edges relaxing the total to a partial order inside a basic block. For example alias analysis results can be directly encoded in the graph structure.

The paper gives an overview of the representation and focuses on its construction. We present a simple construction algorithm which does not depend on dominance frontiers or a dominance tree. We prove that for reducible programs it produces a program in pruned and minimal SSA-form. The algorithm works incrementally so optimizations like copy propagation and constant folding can be performed on-the-fly during the construction.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Retargetable Compilers

General Terms Intermediate Representation, SSA

1. Introduction

A compiler is usually separated into front end, for reading the source code (like a programming language or bytecode), and back end, for emitting target code (like machine code or bytecode). In between, the program is stored in an *intermediate representation* (IR), which allows optimizations to be independent of source and target language. Hence, multiple front and back ends all profit from the same optimizations. Thus, the main goal for the design of an intermediate representation is to support the optimizations using it.

The heart of an intermediate representation are *operations*, e.g. subtractions, comparisons, or conditional jumps. Operations consume and produce *values*: raw data elements. A *variable* is a named storage place for values. Storing a value into a variable is also called *assignment*. State of the art IRs are in *Static Single Assignment* (SSA) form, where each variable is assigned exactly once. This makes analysis more efficient, because every value (statically) comes from exactly one single definition. In contrast, an optimization on a non-SSA representation needs to process a vector of definitions.

Compiler optimizations can be classified into three kinds. The common *explicit optimization* approach is to analyse and transform the program iteratively, e.g. according to equalities like `x*2 = x+x = x<<1`. However, this results in the well-known phase-ordering problem [20], which stems from the fact that transformations enable or inhibit each other, such that there is no globally optimal order of optimizations. Additionally, an implementation may perform *implicit optimizations*, which are not part of a certain compiler phase. For example, our LIBFIRM implementation performs constant folding, e.g. `2*3` → `6`, during the construction of an arithmetic operation, without any effort from the frontend, which uses the construction API. The downside of implicit optimization is that they are local by nature, so they can be performed in constant time. Therefore, a more powerful optimization is usually done explicitly anyways. A third kind are *inherent optimizations*, which "abstract away" inessential aspects of the program during the IR construction, because these aspects cannot be represented in the IR. For example, in FIRM dead code elimination, which removes unused operations, is inherent, because unused operations are just not reachable by a walk over the program graph. In contrast, an optimization pass working on an instruction list representation still encounters unused operations and requires a special optimization pass for dead code elimination. Inherent optimizations are the most desirable kind, because they neither succumb to the phase-ordering problem, nor are they limited to local optimizations.

The IR's purpose (supporting optimization) is realized by three properties, by which the quality of an IR should be measured:

1. Optimizations are inherent.
2. Analysis is efficient.
3. Transformation is simple.

While analysis and transformation are the basic parts of any optimization pass, the first property aims to perform as many optimizations as possible during the construction and render explicit optimization unnecessary. However, we cannot imagine an IR, which performs all known optimizations either implicitly or inherently, so all three properties are useful to judge IRs.

In this document, we present our IR called FIRM, which represents program graphs as explicit dependency graphs in SSA-form. A quality of FIRM is that even dependencies due to memory accesses are explicitly modelled. In contrast, most IRs encode these dependencies through an implicit operation schedule, but this approach introduces pseudo-dependencies and requires additional analysis overhead for optimizations that want to change the order of memory accesses. Our contributions are

- a thorough description of FIRM and its design decisions,
- a detailed description of an incremental SSA construction algorithm, which does not rely on dominance information, and
- a proof that the SSA construction algorithm constructs pruned SSA-form for all programs and minimal SSA-form for reducible programs.

Section 2 presents the structure and semantics of FIRM. Then, Section 3 considers SSA-form and its construction in detail. Finally, we present related work, future work and our conclusions.

2. FIRM

FIRM was initially built in 1996 for the Sather-K compiler Fiasco which manifests in its name: Fiasco's Intermediate Representation Mesh. Later, FIRM was extracted to the separate open-source library LIBFIRM [18]. The library can be used with the Java- and C-frontends by the Edison Design Group. Additionally, there are open-source frontends for C [9] and Java bytecode [4].

A FIRM program consists of a set of *entities* which are objects that will occur in the output/binary. This includes functions, global variables, string and number literals, struct initializers, classes, and vtables. To support this, firm contains a type system modeling nested structs and class hierarchies and function types. Code inside functions is modeled in graphs as described below.

2.1 SSA-Graphs

FIRM is based on static single assignment (SSA) form which is defined as follows:

Definition 1. *A program is in SSA-form iff each variable has exactly one definition.*

Translating a program into SSA-form involves the renaming of different assignments to the same variable and the insertion of special ϕ-functions at places where different assignments meet.

Since each variable has only one well-known definition, SSA-form allows a new style of intermediate representations: Instead of reading an operand from a variable we can directly link to the operation producing this value. Thus, the concept of variables becomes unnecessary. FIRM follows the *sea of nodes* idea by Click [7, 8] that makes dependencies between values explicit. In contrast to an instruction list representation, this implies that there is no implicit schedule and no total order of operations, which leads to a graph-based representation. A total ordering is not calculated until the instruction scheduling phase in the backend.

2.2 Explicit Dependency Graphs

FIRM-graphs were first described by Trapp [19]. While the details have changed in the last decade, his definition still fits.

Definition 2 (explicit dependency graph)**.** *An* explicit dependency graph *(EDG) is a directed, marked graph. The nodes are marked with a signature of Σ_{EDG}, which is their operation. The nodes have ordered in- and outputs. Their number and order correspond to the parameters and results of the respective term in Σ_{EDG}. The in- and outputs are marked with types of T_{EDG}. Edges connect outs with ins of the same type. The signatures Σ_{EDG} and types T_{EDG} are shown in Table 1.*

A function in FIRM is represented by a program graph. Every operation is a node. The edges denote common dependencies, e.g. data or control dependencies. Since FIRM models dependency instead of flow edges, the program graph is reversed compared to the common data flow representation. Nodes may produce multiple results combined in a tuple value. There is a special instruction called Proj which extracts values from tuples.

If there's a dependency between two nodes then there is always a path between them. Hence, the node semantics are *referentially transparent* for simple operations like Addition or Subtraction. However, Phi-nodes for example are an exception, because the behavior that all Phi-nodes must be evaluated simultaneously at beginning of a basic block is not represented with edges.

Most scalar variables can easily be brought into SSA-form. However, there may be additional data structures on the heap and some variables might be affected by aliasing effects. If the aliasing relations of a variable are unknown then the variable is simply not represented in SSA-form. Instead every read-/write-access to the variable is represented with an explicit Load-/Store-node. The order of these operations is important. To model this we introduce a new SSA-value called memory which represents aliased variables and heap data. Load- and Store-nodes consume a memory value and produce a new one.

This chaining through the memory value can be stricter than necessary. However, if we know that some operations are not dependent on each other because of aliasing information or because they only read memory, then we can represent this in the graph: we let all such operations use the same memory value as input, their outputs are combine in a Sync-node, which forms a barrier for subsequent potentially aliasing operations.

There have been proposals to directly model aliased variables in SSA-form [6] by virtual instructions updating a value at places with potential aliasing effects. We do not use this approach, because in program-parts with aliasing effects we expect optimisation opportunities to be very limit or non-existent, while Load- and Store-operations to the same address without aliasing will get promoted to SSA-values by scalar replacement.

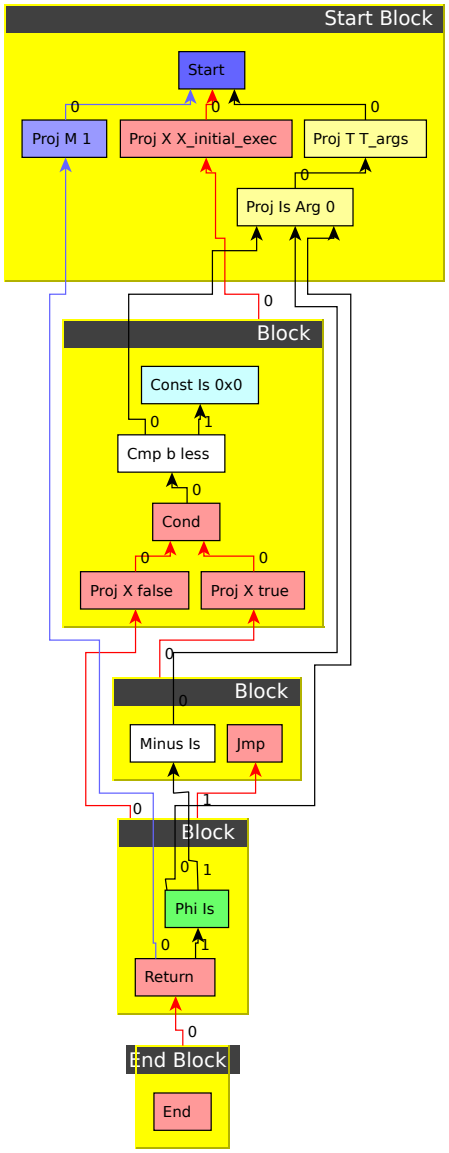


Figure 1. Example FIRM graph with control flow

An example FIRM graph is shown in Figure 1. It shows the FIRM representation of a simple function:

```
int abs(int x) {
  if (x < 0) {
    x = -x;
  }
  return x;
}
```

There are four basic blocks. The Start Block contains Start, where all values including function parameters originate. The second block contains the comparison with the zero Constant. The conditional jump is modeled with a Cond node which produces a tuple containing controlflow depending on the condition. The third block represents the body of the if-statement, where the value is negated. The last block contains the end of the function, where the value of variable x is returned. The Return operation references a Phi node, because the operand depends on the control flow. Every block (except the start block) references the operation, where control flow comes from (red). For example, the block with the Return operation points to the Proj of the Cond and to the Jmp. These control dependency edges form a "reversed CFG". The Return node

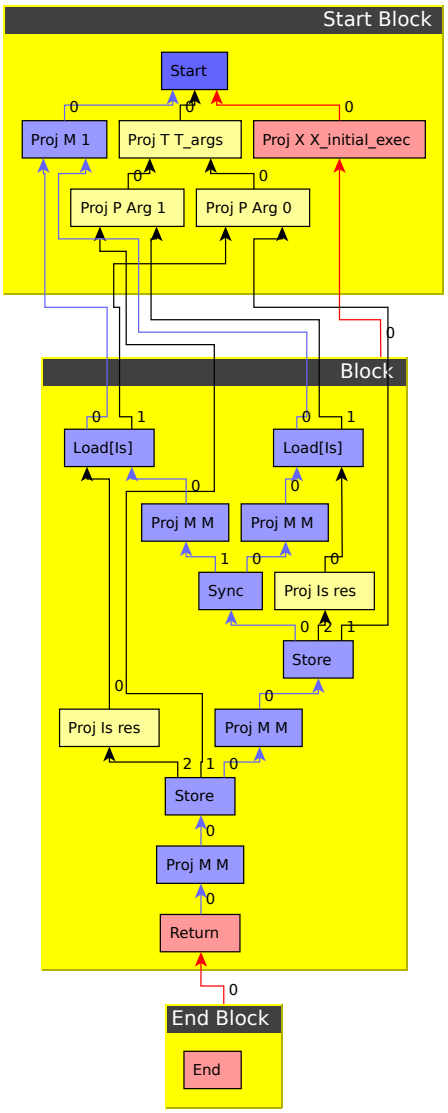


Figure 2. Example FIRM graph with memory operations

additionally references (blue) the initial memory state produced by Start.

A second example dominstrating memory-values is shown in Figure 2:

```
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = y;
}
```

The initial memory values is produced by the Start-node. Since the two Load- operations for the $*x$ and $*y$ expression can safely happen in parallel they both use the same memory value. They both produced the loaded value and a new memory value. The memory values are combined with a Sync node which which is then used by the Store nodes from the $*x \leftarrow$ and $*y \leftarrow$ expressions. The variable t was eliminated by copy-propagation. In fact a node for copying values doesn't even exist.

2.3 On-the-fly Optimizations

As already mentioned, the construction of an IR may include implicit and inherent optimizations. FIRM construction optimizes as follows:

Arithmetic Simplification (implicit) A node constructor is able to detect local simplifications, like `x-x = 0`, and returns a node, which represents the simplified expression.

Common Subexpression Elimination (implicit) This optimization merges duplicate expressions. In FIRM this is also done implicitly during node creation. For example, before a new Const 1 node is created, the constructor checks if there is already a Const 1 in the graph and then returns this instead.

Constant Folding (implicit) This optimization exchanges constant expressions for their resulting value, e.g. `2*3` is optimized to `6`. Due to the mostly referentially transparent nature of operations, the `2*3` expression is easily detected in FIRM. FIRM performs this optimization during every node creation, so when a frontend calls the constructor for the Add node with the two Const operands a Const 6 is constructed internally and returned instead of an Add.

Copy Propagation (inherent) This optimization removes unnecessary assignments, e.g. `x = y`. As FIRM has no local variables, there are no assignments between them either. A copy operation doesn't even exist in the representation. If `x` is a global variable, then the FIRM graph represents access with a Store/Load operation. However, if `x` is local, then its users will just reference the defining of `y` directly.

Dead Code Elimination (inherent) Since a FIRM node can only be accesses through its users an unused node is not seen, when walking the program graph. A garbage collection mechanism frees the memory occupied by dead code.

All these optimizations are only performed with local analysis to keep the graph construction efficient. This forces them to be conservative. There may be additional optimization possibilities detectable by optimistic dataflow analysis. Imagine for example, unreachable code elimination which is not reached because some conditional branch will never be taken. The local analysis may fail to detect this when the unreachable code contains further definitions influencing the condition of the branch. FIRM contains such optimistic optimizations as a separate pass.

3. Incremental SSA-Construction

SSA-form was invented by Rosen, Wegman, and Zadeck [16] and became popular after Cytron et al. [10] presented an efficient algorithm for constructing it. This algorithm can be found in any textbook presenting SSA and is used by the majority of compilers.

Cytrons algorithm is based on dominance frontiers which require a dominance tree and therefore a complete control flow graph to be built [17]. This in turn requires a traditional non-SSA-form representation of the complete program to be available. This is fine for compilers using SSA-form only for some of their optimization passes. However, in our setting it would be desirable to directly build the SSA-form while parsing the program.

Fortunately, there is a little-known construction algorithm by Click [8] which is efficient and does not depend on the dominance tree, so it can be used while the control flow graph is under construction. In the following, we give a detailed description of the algorithm and prove that it constructs pruned SSA-form for all programs and minimal SSA-form for reducible programs.

3.1 The Construction Algorithm

We use the usual definitions of the program being represented in a control flow graph (CFG) built from basic blocks connected by edges representing possible jumps between basic blocks. There are distinct entry and exit blocks through which all control flows into and out of the graph.

In the following we assume that all variables are enumerated. Each basic block has an array *variables* recording for each variable-number the operation with the latest assignment to that variable. The array is initialized with *null* entries for variables where the latest assignment is unknown yet. We process the operations in a

basic block in the order provided by the source program. When an assignment is encountered the corresponding operation is recorded in the variables array. When a variable v is read, then the last defining operation is looked up in the array. If the last assignment is not yet known, then it must be in another block. If we know that there will only be one predecessor block we look the definition up at this block. Otherwise, we pessimistically create a ϕ-function for v since we might not have added all predecessors to the block yet and therefore we might not be able to determine the latest assignment to v. For the same reasons we do not attempt to determine the arguments of the ϕ-function yet. This ϕ-function is recorded as the last definition for v in the variables array.

There's a special case for the start block: If we did not find a definition there, then the variable must have been used uninitialized and we create and record an unknown value.

Once we finished constructing all basic blocks, the arguments for the ϕ-functions are determined by looking into the variables arrays of the corresponding predecessor blocks. This can lead to the creation of further ϕ-functions whose arguments can be determined immediately. To avoid running in endless cycles, we record the ϕ-function in the variables array immediately after its creation before we determine the arguments.

For a construction example consider the pseudocodes Algorithm 1 and Algorithm 2 and the following C program fragment.

```
int foo(int x) {
  do {
    if (...) {
      ...
    } else {
      ...
    }
  } while (...);
  return x;
}
```

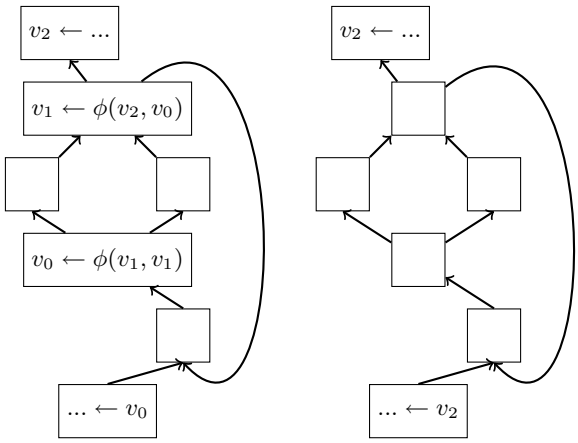
Let the varnum of x be v_x. The construction of the do-while loop is: First, a finish-block, where the program will proceed after the loop, and a condition-block, where the loop condition will be put, are built. Then a body-block is constructed and a recursive processStatement handles the loop body, where **continue** will jump to the condition-block and **break** to the finish-block. Afterwards, the condition is processed in the condition-block and makes a conditional jump to the finish- and the body-blocks.

Similarly, the if statement constructs a then-, an else-, and a finish-block. The condition is processed (using short-circuit evaluation).

The **return** statement itself is easy to process, but consider the readVariable(currentBlock, v_x) call for the expression x. As there is no definition of x in return-block, there are a recursive calls through the condition-block and the if-finish-block, which has two predecessors. Therefore, a $v_0 = \phi_0$ is inserted and the predecessors are inspected. Another $v_1 = \phi_0$ is inserted into the if-condition block (the loop-body-block). One argument is the initial v_2 from the start block's definition of x, the other one is v_0 from a previous loop iteration, so $v_1 = \phi(v_2, v_0)$. Now, the recursion returns to the v_0 block and finishes it, so $v_0 = \phi(v_1, v_1)$, which results in the graph shown in Figure 3(a).

3.2 Reducing the Number of ϕ-Functions

The algorithm described so far places a ϕ-function at all join-points in the control flow graph. This is correct but more conservative than necessary. The reason for the conservative ϕ-placement is the uncertainty of a block's predecessors during CFG construction. However during construction we know the predecessors for many blocks in advance. Typical examples are if-statements and structured loops where we know the exact predecessor for the then- and else-part or the loop-body. We use this knowledge to improve our construction algorithm: Blocks with known predecessors are called *mature* other blocks *immature*.



(a) Unoptimized. (b) Minimal SSA-form.

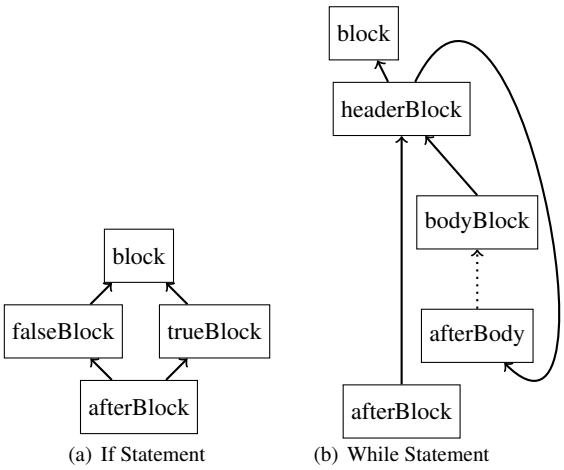
Figure 3. Control flow dependency graphs for program example.

```
proc writeVariable(block, varnum, node):
    block.variables[varnum] ← node

internal proc setPhiArguments(phi):
    phiArgs ← []
    for pred in phi.block.preds:
        arg ← readVariable(pred, phi.varnum)
        phiArgs.append(arg)
    phi.setArguments(phiArgs)
    RemoveUnnecessaryPhi(phi)

proc readVariable(block, varnum):
    if block.variables contains varnum:
        return block.variables[varnum]
    if block.matured:
        if |block.preds| = 0: # startblock
            return new Unknown(block)
        else if |block.preds| = 1:
            return readVariable(block.preds[0], varnum)
    phi ← new Phi(block)
    phi.varnum ← varnum
    writeVariable(block, varnum, phi);
    if block.matured:
        setPhiArguments(phi)
    return phi

proc matureBlock(block):
    for phi in block.phinodes:
        setPhiArguments(phi)
    block.matured ← true
```

Algorithm 1: Value identification and ϕ-creation

A newly created basic-block is *immature*. In an immature block we always place temporary ϕs like already described. As soon as all predecessor blocks have been added and processed we mature the block. This involves determining the arguments for all ϕ-functions in the block. If we have to create additional ϕ-functions in a mature block, we will immediately calculate their arguments. Obviously the ϕ can be omitted if a mature block has exactly 1 predecessor.

A second simplification called RemoveUnnecessaryPhi is applied when all arguments of a ϕ-function have been determined:



(a) If Statement (b) While Statement

Figure 4. Illustration of construction procedures.

1. If all arguments of a ϕ-function are the same value s or the ϕ-function itself, then we remove the ϕ-function and let all users directly use s. We call such a ϕ-function *obviously unnecessary*.
2. When we removed a ϕ-function p, then we recursively try to apply this simplification rule with all (former) users of p, because they may have become obviously unnecessary due to the removal of p.

Algorithm 1 shows pseudocode for the described algorithm. Due to the RemoveUnnecessaryPhi rule, the program of Figure 3(a) can be further optimized: v_0 is immediately replaced by v_1. The ϕ-function defining v_1, which now reads $\phi(v_2, v_1)$, is optimized recursively. Figure 3(b) shows the resulting program which is in minimal SSA-form.

Some typical scenarios when constructing an SSA-form program from an AST are shown in Algorithm 2. The resulting control dependency graphs are illustrated in Figure 4.

3.3 Quality of the Algorithm

Having too many ϕ-functions in an SSA-program results in an bigger number of operations overall but more importantly it can obscure information available to optimizations. This happens when seemingly different values always have the same value in reality. So a good SSA-construction algorithm should create as few ϕ-functions as possible. There are two sources of extraneous ϕ-functions:

1. There might be ϕ-functions that no one references.
2. There might be ϕ-functions where all arguments effectively have the same value at runtime.

While there are situations in which our algorithm produces extraneous ϕ-functions, in the majority of the cases it does not. As the next section proves this depends on the reducibility of the programs control flow graph. Intuitively: For goto-free programs we do not produce extraneous ϕ-functions. Programs in languages like Java, Javascript or Python always fall into this class.

3.3.1 Pruned SSA-form

A program is said to be in pruned SSA-form [5] if each ϕ-function has at least one user. We only create ϕ-functions as a result of a situation where someone will use it: Either a variable being read or another ϕ-function needing an argument. So our construction naturally produces a program in pruned SSA-form.

3.3.2 Minimal SSA-form

Minimal SSA-form requires that ϕ-functions for a variable v only occur at basic blocks where different definitions of v meet for the first time. Cytron's formal definition is based on the following two terms:

```
proc processExpression(block, ⟦x ← E⟧):
    node ← processExpression(E);
    writeVariable(block, vnum(x), node);
    return node

proc processExpression(block, ⟦x⟧):
    return readVariable(block, vnum(x));

proc processStatement(block, ⟦S1; S2⟧ ):
    block1 ← processStatement(block, S1)
    block2 ← processStatement(block1, S2)
    return block2

proc processStatement(block, ⟦while(C) S⟧ ):
    jump ← new Jump(block);
    headerBlock ← new Block();
    headerBlock.addPredecessor(jump);

    bodyBlock ← new Block();
    afterBlock ← new Block();
    processCondition(headerBlock, C, bodyBlock, afterBlock);
    matureBlock(bodyBlock);

    afterBody ← processStatement(bodyBlock, S);

    jump ← new Jump(afterBody);
    headerBlock.addPredecessor(jump);
    matureBlock(headerBlock);

    matureBlock(afterBlock);
    return afterBlock;

proc processStatement(block, ⟦if(C) S1 else S2⟧ ):
    trueBlock ← new Block();
    falseBlock ← new Block();
    afterBlock ← new Block();
    processCondition(block, C, trueBlock, falseBlock);
    matureBlock(trueBlock);
    matureBlock(falseBlock);

    trueEnd ← processStatement(trueBlock, S1);
    jump ← new Jump(trueEnd);
    afterBlock.addPredecessor(jump);

    falseEnd ← processStatement(falseBlock, S2);
    jump ← new Jump(falseEnd);
    afterBlock.addPredecessor(jump);

    matureBlock(afterBlock)
    return afterBlock
```

Algorithm 2: Typical scenarios when constructing from an AST

Definition 3 (path convergence)**.** *Two non-null paths $X_0 \to^+ X_J$ and $Y_0 \to^+ Y_K$ are said to* converge *at a block Z iff the following conditions hold:*

$$X_0 \neq Y_0; \tag{1}$$
$$X_J = Z = Y_K; \tag{2}$$
$$(X_j = Y_k) \Rightarrow (j = J \vee k = K). \tag{3}$$

Definition 4 (necessary ϕ-function)**.** *A ϕ-function for variable v is necessary in block Z iff two non-null paths $X \to^+ Z$ and $Y \to^+ Z$ converge at a block Z, such that the blocks X and Y contain assignments to v.*

A program with only necessary ϕ-functions is in *minimal SSA-form*. The following is a proof that our algorithm with the sim-

plification rule from Section 3.2 produces minimal SSA-form for reducible programs.

We say a block A *dominates* a block B if every path from the entry block to B passes through A. We say A *strictly dominates* B if A dominates B and $A \neq B$. Each block C except entry has a unique immediate dominator $idom(C)$, i.e. a strict dominator of C which does not dominate any other strict dominator of C. The dominance relation can be represented as a tree whose nodes are the basic blocks with a connection between immediately dominating blocks.

Definition 5 (reducible flow graph)**.** *A (control) flow graph G is reducible iff for each cycle C of G there is a node of C which dominates all other nodes in C.*

We now assume that our construction algorithm finished and produced a program with a reducible CFG [13]. We observe that the simplification rule RemoveUnnecessaryPhi was applied at least once to each ϕ-function with its current arguments. This is because we apply the rule each time a ϕ-function's parameters are set for the first time. In the case that a simplification on another operation leads to a change of parameters the rule is applied again. Furthermore, our construction algorithm fulfills the following property:

Definition 6 (SSA-property)**.** *In an SSA-form program a path from a definition of an SSA-value for variable v to its use cannot contain another definition or ϕ-function for v. The use of the operands of ϕ-function happens in the respective predecessor blocks not in the ϕ's block itself.*

The SSA-property ensures that only the "most recent" SSA-value of a variable v is used. Furthermore, it forbids multiple ϕ-functions for one variable in the same basic block.

Lemma 1. *Let p be a ϕ-function in a block P. Furthermore, let q in a block Q and r in a block R be two operands of p, such that p, q and r are pairwise distinct. Then at least one of Q and R does not dominate P.*

Proof. Assume that Q and R dominate P, i.e. every path from the start block to P contains Q and R. Since immediate dominance forms a tree, Q dominates R or R dominates Q. Without loss of generality, let Q dominate R. Furthermore, let S be the corresponding predecessor block of P where p is using q. Then, there is a path from the start block crossing Q then R and S. This violates the SSA-property. □

Lemma 2. *If a ϕ-function p in a block P for a variable v is unnecessary, but not obviously unnecessary, then it has an operand q in a block Q, such that q is an unnecessary ϕ-function and Q does not dominate P.*

Proof. The node p must have at least two different operands r and s which are not p itself, otherwise p is obviously unnecessary. They can either be:

- The result of a direct assignment to v.
- The result of a necessary ϕ-function r'. This however means that r' was reachable by at least two different direct assignments to v. So there is a path from a direct assignment of v to p.
- Another unnecessary ϕ-function.

Assume neither r in a block R nor s in a block S is an unnecessary ϕ-function. Then a path from an assignment to v in a block V_r crosses R and a path from an assignment to v in a block V_s crosses S. They converge at P or earlier. Convergence at P is not possible because p is unnecessary. An earlier convergence would imply a necessary ϕ-function at this point which violates the SSA-property.

So r or s must be an unnecessary ϕ-function. Without loss of generality, let this be r.

If R does not dominate P then r is the sought-after q. So let R dominate P. Due to Lemma 1 S does not dominate P. Employing the SSA-property, $r \neq p$ yields $R \neq P$. Thus, R strictly dominates P. This implies that R dominates all predecessors of P which contain the uses of p, especially the predecessor S' that contains the use of s. Due to the SSA-property, there is a path from S to S' that does not contain R. Employing R dominates S' this yields R dominates S.

Now assume that s is necessary. Let X contain the most recent definition of v on a path from the start block to R. By Definition 4 there are two definitions of v which render s necessary. Since R dominates S, the SSA-property yields that one of these definitions is contained in a block Y on a path $R \rightarrow^+ S$. Thus, there are paths $X \rightarrow^+ P$ and $Y \rightarrow^+ P$ rendering p necessary. Since this is a contradiction, s is unnecessary and the sought-after q. □

Theorem 1. *A program in SSA-form with a reducible CFG G without any obviously unnecessary ϕ-functions is in minimal SSA-form.*

Proof. Assume G is not in minimal SSA-form and contains no obviously unnecessary ϕ-functions. We choose an unnecessary ϕ-function p. Due to Lemma 2, p has an operand q, which is unnecessary and does not dominate p. By induction q has an unnecessary ϕ-function as operand as well.

This would lead to an endless number of ϕ-functions. Since the program only has a finite number of operations this is only possible if there is a cycle when following the q chain.

A cycle in the ϕ-functions is only possible with a cycle in the CFG. As the CFG is reducible this cycle contains one entry block which dominates all other blocks in the cycle. Since one of the ϕ-functions must be in the entry block, we get a contradiction to our choice of q. So our assumption must be wrong and G is either in minimal SSA-form or there exist obviously unnecessary ϕ-functions. □

Since we know, that our construction algorithm will have optimized all obviously unnecessary nodes, a Firm graph must be in minimal SSA-form for reducible CFGs.

3.3.3 Less Than Minimal

As the Firm construction algorithm implicitly performs some optimizations, the program graph may contain even fewer ϕ-functions than the minimal and pruned SSA definitions require. Consider the following code example:

```
x ← y;
if (...) { y ← x; }
... ← y;
```

While the Cytron algorithm will place a ϕ-function for y within the block after the if-statement, our algorithm performs copy propagation on-the-fly and will remove the ϕ.

3.3.4 Time Complexity

Let B be the number of basic blocks, E the number of CFG edges, and V the number of variables in the input program. Transforming the program into SSA-form inserts at most $\mathcal{O}(BV)$ ϕ-functions. The number of ϕ-operands can be limited by $\mathcal{O}(EV)$. Since our algorithm caches the `varnum` of each variable at each block, we obtain a time complexity of $\mathcal{O}(EV)$ without the optimization described in Section 3.2. For the optimization of a obviously unnecessary ϕ-function p we need to check at most V ϕ-functions which uses p. Checking these ϕ-functions requires at most $\mathcal{O}(E)$ time. Since there are at most $\mathcal{O}(BV)$ ϕ-functions, this leads to an overall time complexity of $\mathcal{O}(BVE)$.

3.3.5 Evaluation

As expected the construction algorithm is fast in practice. On a Core2-Duo with 2.6 GHz we spend 1308 ms for SSA-construction of all C-Programs of SPEC CINT 2000. This corresponds to more than 1200 graph-nodes per millisecond.

4. Related Work

4.1 Static Single Assignment

The commonly used algorithm for constructing SSA-form has been described by Cytron et al. [10]. It is based on the observation that ϕ-functions for a variable v are only necessary at the iterated dominance frontiers of blocks containing definitions of v. He presents an efficient algorithm for computing the iterated dominance frontiers. After marking blocks in the iterated dominance frontiers, a rename phase produces unique variable names. Compared to the algorithm presented here minimal SSA-form is produced for irregular control-flow too. This however comes at a higher complexity: A complete CFG and dominance tree is a requirement. The transformation is performed in two phases without any further implicit optimizations.

4.2 Usage of FIRM

While FIRM is an intermediate representation, its implementation also provides a backend, which does not destruct the IR and even SSA-form is retained in all steps. All work is performed on the IR by transforming the graph. So the usual work can be described like this:

instruction selection Match certain patterns within the program graph and replace these parts, such that the nodes represent instructions of the target machine. As every essential property is modelled by the graph structure in FIRM, instruction selection can be performed with graph transformation techniques [3].

scheduling Insert additional dependency edges, such that there is a total order for the instructions within each basic block.

register allocation Annotate each instruction with the register for its result. FIRM has been used by Hack [12] in his work on SSA-based register allocation, where he showed that SSA-form implicitly provides points in the program graph to insert a live-range split. This allows to guarantee a register allocation, because register pressure lowering can be separated in a phase before the allocation and does not need to be repeated, as it is necessary when using common graph coloring algorithms. Exploiting this property, there has been additional work on register allocation by Braun et al. [1, 2], which provides competitive results with significantly reduced compile times by avoiding the construction of an interference graph.

4.3 Other Intermediate Representations

We already explained why SSA is a useful abstraction and that it enables FIRM to make certain optimizations inherent. However, there is still room for additional abstraction.

There has been work on using the program dependence graph for optimizations [11] which manages to abstract away from a control flow graph in favor of control dependencies. This was further extended to the gated SSA-form [15]. More modern variants like the VSDG graphs model the program state more explicitly [14]. These representations lead to more powerful optimizations and simpler optimizations but generating good code from these appears to be very challenging. We have not seen any implementations handling bigger programs, e.g. the SPEC benchmarks.

5. Conclusion

We presented the intermediate representation FIRM, which provides a lot of desirable features, like being in SSA-form and being "mostly referentially transparent". This e.g. allows to shun the concept of variable names, such that optimizations like copy propagation are inherent. Additionally, SSA makes analyses efficient and referential transparency (in many cases) makes transformation simple, so FIRM is a high-quality IR according to our measures.

We also showed how to incrementally construct FIRM graphs, such that they are in pruned and minimal SSA-form. Transforming an AST or bytecode representation to FIRM is efficient and optimizations like copy propagation and constant folding can be performed along the way.

References

[1] M. Braun and S. Hack. Register spilling and live-range splitting for SSA-form programs. In *Proceedings of the International Conference on Compiler Construction*, pages 174–189. Springer, March 2009. doi: 10.1007/978-3-642-00722-4_13.

[2] M. Braun, C. Mallon, and S. Hack. Preference-guided register assignment. In *International Conference on Compiler Construction*. Springer, March 2010. doi: 10.1007/978-3-642-11970-5_12.

[3] S. Buchwald and A. Zwinkau. Instruction selection by graph transformation. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, pages 31–40, New York, NY, USA, October 2010. ACM. doi: 10.1145/1878921.1878926.

[4] Bytecode2Firm. `https://github.com/MatzeB/bytecode2firm`.

[5] J. D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 55–66, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99594.

[6] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In T. Gyimóthy, editor, *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267. Springer Berlin / Heidelberg, 1996. doi: 10.1007/3-540-61053-7_66. URL `http://dx.doi.org/10.1007/3-540-61053-7_66`.

[7] C. Click and M. Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30:35–49, March 1995. ISSN 0362-1340. doi: 10.1145/202530.202534.

[8] C. N. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, February 1995.

[9] cparser C-Frontend. `https://github.com/MatzeB/cparser`.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041.

[12] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In A. Zeller and A. Mycroft, editors, *Compiler Construction 2006*, volume 3923 of *Lecture Notes In Computer Science*, pages 247–262. Springer, March 2006. doi: 10.1007/11688839_20.

[13] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974. ISSN 0004-5411. doi: 10.1145/321832.321835.

[14] N. Johnson and A. Mycroft. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In G. Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, chapter 1, pages 1–16. Springer Berlin Heidelberg, February 2003. ISBN 978-3-540-00904-7. doi: 10.1007/3-540-36579-6_1.

[15] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7. doi: 10.1145/93542.93578.

[16] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-252-7. doi: 10.1145/73560.73562.

[17] R. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974. doi: 10.1137/0203006.

[18] The libFirm Compiler. `http://www.libfirm.org`.

[19] M. Trapp. *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, Oct 2001.

Name	Type	Description
Add	$B \times Num \times Num \to Num$	Addition
Alloc	$B \times M \times Num \to M \times P$	Allocate memory on the stack
And	$B \times Num \times Num \to Num$	Bitwise AND
ASM	$B \times variable \to variable$	Inline assembler
Bad	$\to Any$	Value of unreachable calculation
Block	$X_0 \times \cdots \times X_n \to\!\!\to B$	Basic block
Call	$B \times Num_0 \times \cdots \times Num_k \times M \to Num_0 \times \cdots \times Num_k \times M$	function call
Cmp	$B \times Num \times Num \to b_0 \times \cdots \times b_{15}$	Binary compare
Cond	$B \times b \to X \times X$	Conditional branch
Const	$\to Num$	Constant value
Conv	$B \times Num \to Num$	Type conversion
Div	$B \times Num \times Num \to Num$	Division
End	$B \times X \to$	End of function
Eor	$B \times Num \times Num \to Num$	Bitwise XOR
Free	$B \to M \times P \times Num$	Release memory on the stack
Jmp	$B \to X$	Unconditional jump
Load	$B \times P \times M \to Num \times M$	Load from memory
Minus	$B \times Num \to Num$	Negate number
Mod	$B \times Num \times Num \to Num$	Modulo
Mul	$B \times Num \times Num \to Num$	Multiplication
Mux	$B \times b \times Num \times Num \to Num$	Select value depending on boolean
NoMem	$\to M$	Empty subset of memory
Not	$B \times Num \to Num$	Bitwise NOT
Or	$B \times Num \times Num \to Num$	Bitwise OR
Phi	$B \times Num_0 \times \cdots \times Num_n \to Num$	ϕ-function
Proj	$B \times T \to Num$	Projection node
Return	$B \times M \times Num_0 \times \cdots \times Num_n \to X$	Return
Rotl	$B \times Num \times Num \to Num$	Rotate left
Sel	$B \times M \times P \to M \times P$	Select field from structure
Shl	$B \times Num \times Num \to Num$	Shift left
Shr	$B \times Num \times Num \to Num$	Shift right zero extended
Shrs	$B \times Num \times Num \to Num$	Shift right sign extended
Start	$B \to X \times M \times P \times P \times T$	Start of function
Store	$B \times M \times P \times Num \to M$	Write to memory
Sub	$B \times Num \times Num \to Num$	Subtraction
SymConst	$B \times P$	Symbolical constant
Sync	$B \times M_0 \times \cdots \times M_n \to M$	Memory barrier
Unknown	$\to Any$	Undefined value

The types T_{EDG} are interpreted as follows:

B basic block
X control flow
M memory
P pointer
I integer (32-bit)
S integer (16-bit)
B integer (8-bit)
F floating point (32-bit)
D floating point (64-bit)
E floating point (80-bit)
b boolean
T tuple
Num $\{P, I, S, B, F, D, E\}$
Any T_{EAG}

Table 1. FIRM Node Types and Signatures

[20] S. R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. *SIGMICRO Newsl.*, 13:125–133, October 1982. ISSN 1050-916X.