Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

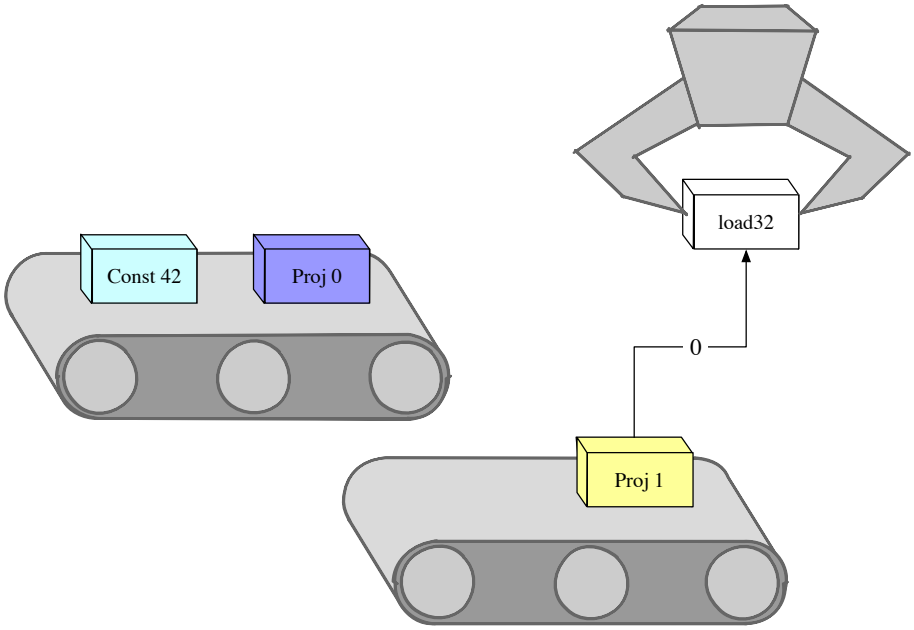
Lehrstuhl Prof. Dr.-Ing. Snelting

Generating Instruction Selectors For Large Pattern Sets

Masterarbeit von

Markus Schlegel

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald M.Sc. Andreas Fried

Bearbeitungszeit: 5. Oktober 2017 – 4. April 2018

Zusammenfassung

Es ist sehr aufwendig, effiziente Befehlsauswahlphasen für neue Instruktionssätze zu entwickeln, obwohl sich diese in ihrer Struktur oft ähnlich sind. In dieser Arbeit entwickeln wir einen Algorithmus, um eine effiziente Befehlsauswahlphase automatisch aus einer gegebenen Regelmenge zu generieren. Wir bauen auf der Arbeit von Buchwald et al. [1] auf, die beschreibt, wie man aus einer formalen Beschreibung eines Teils des x86-Befehlssatzes eine umfassende Regelmenge mit mehr als 60000 Regeln berechnen lassen kann. In dieser Arbeit generieren wir aus dieser großen Regelmenge wiederum eine effiziente Befehlsauswahlphase für das libFirm Compiler Framework. Wir führen zwei Algorithmen ein: Pattern Decomposition und Bottom-up Annotation. Wir vergleichen unseren Ansatz mit der handgeschriebenen Befehlsauswahl in libFirm.

Development of new instruction selectors is hard and time-consuming, even though most instruction sets share a lot of similarities. In this thesis we propose a method to generate efficient instruction selectors from a given rule set. We make use of foundational work by Buchwald et al. [1]. Their work describes a means to generate a rule set with more than 60000 rules from a formal description of x86 instructions. In turn, we use this large rule set to generate an efficient instruction selector for the libFirm compiler framework. We introduce two algorithms: pattern decomposition and bottom-up annotation. We compare the performance of our generated instruction selector with the performance of the handwritten instruction selector in libFirm.

Contents

1 Introduction

In the beginning programmers wore suits and dresses and they were coding in absolute binary but soon the first programmer ate from the forbidden fruit of assembly and humankind was sent down the spiral of abstraction. Programming used to be about manipulating 0s and 1s on a computer. Today, programming is mainly about communicating ideas with other humans. Today's high-level programming languages rely on a sophisticated tool called a *compiler* that translates these high-level programs into the language of the machine.

Compilers are complex. GCC [2] and LLVM [3], two major industrial compiler projects, each have millions of lines of source code. Decades of work has gone into these projects. This is not surprising because a compiler has to account for the complexity of many programming languages and many target machines. On the side of the programming languages, the frontend, a lot of work has gone into formalizing methods and thus automating the process. On the side of the target machine, the backend, such work is rarer. Compiler backends are usually written by hand even though a lot of this work is repetitive. In this thesis we investigate one way to automate a lot of this work by generating an instruction selector, part of the backend, from a formal specification of the target machine.

2 Basics

A compiler translates a program written in one language (*source*) into a semantically equivalent program in another language (*target*). In the compilers we concern ourselves with in this thesis, the source is a general-purpose machine-independent imperative language such as C [4] and the target is a machine language targeted at a specific architecture such as x86 assembly [5].

Compilers are structured as pipelines consisting of different passes. These passes are commonly divided into a *frontend*, an optimization phase[1] and a *backend* [6]. The frontend parses the source program, performs semantic analysis on it, and translates it into an *intermediate representation* (IR). Ideally, this IR is designed such that it is independent of any source and target language considerations. For that reason IR is abstract and simple and it is therefore well-suited as a basis for optimizations such as constant folding [7] or common subexpression elimination [8]. After these optimizations the backend translates the final IR program into a target language.

Such a design, which decouples frontend and backend, makes it easy to adapt the compiler to both new source languages and new target languages. If you want to add a new source language to the compiler you just have to write a new frontend. The rest of the pipeline doesn't have to be touched at all. Similarly, if a new architecture comes along, you just have to write a new backend.

2.1 Intermediate representation

An IR that allows proper decoupling of frontend and backend must be carefully designed not to contain any biases towards specific source or target languages. We also want the IR to be designed in a way that makes it easy to reason about and implement optimization passes, so ideally all primitives are pure functions (free of side-effects). We want to make all effects of a primitive explicit. An instruction that computes a value but also sets a flag should therefore be modelled to have two results.

An IR consists of a small but complete set of abstract primitives. Luckily, in the compilers we are dealing with, both source and target are imperative in nature. We can therefore use the bedrock of imperative programming as the foundation of a good IR design. The bedrock of imperative programming are data and control flow. Consequently the primitives of an IR are operations that modify data and change the control flow. These operations are connected by data and control flow dependencies.

[1] This phase is often called the middleend, which is a misnomer, because there's no end in the middle of a pipeline.

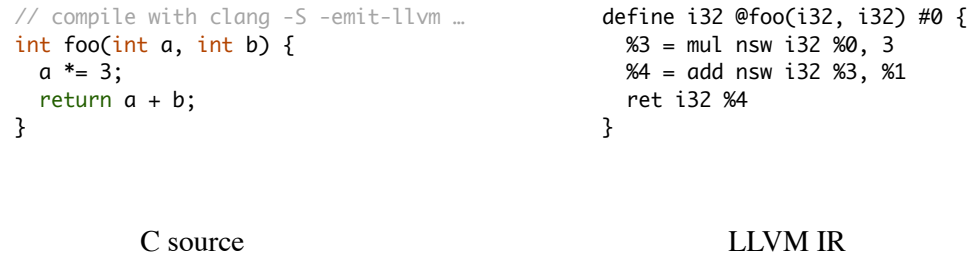


Figure 2.1: A C function (left) and its equivalent in LLVM IR (right).

Such IR designs are therefore inherently graph-based and we call the program in IR a *program graph*.

LLVM IR, the intermediate representation of the widely used LLVM compiler framework [3], is implicitely graph-based. Figure 2.1 shows an example of a simple C function and its intermediate representation in LLVM.

2.1.1 Static single assignment form

In the C source program in Figure 2.1 the variable a is reassigned the new value $a * 3$, yet this reassignment is missing in the resulting LLVM IR program. Indeed, in the IR program a new temporary variable is introduced for every operations' result. This is a deliberate choice of the designers of the IR, which is said to be in *static single assignment* (SSA) form. In SSA form, variables can statically be assigned only once. This design simplifies a lot of optimizations such as constant folding because the usage of a variable has a direct correspondence to the definition of that variable. In addition to that, SSA makes some further optimizations easier such as *global value numbering* [9] or *sparse conditional constant propagation* [10].

When control flow comes into play, a single variable might be assigned different values depending on the path taken through the control flow graph. In SSA we deal with this phenomenon by making it explicit through so called ϕ nodes. A ϕ node represents a variable that can take on different values depending on the control flow predecessor. A ϕ node has multiple predecessors and it can be regarded as representing all these predecessor values in unison.

Since ϕ nodes emerge whenever there are branches in the control flow, ϕ nodes are often to be found when there are control flow loops. In turn, ϕ nodes are a source of loops in the data dependency graph as well. In our work we ignore ϕ nodes (and blocks, discussed later), so we can view program graphs as directed acyclic graphs

(DAG), which means we can visit its nodes in a topological order. In a topological order whenever we visit a node we are guaranteed to have visited all of that node's operands before.

2.1.2 FIRM

In this thesis we work with the research compiler framework libFirm. The libFirm intermediate representation is simply called FIRM. In contrast to LLVM IR, where the program graph structure is encoded implicitely, FIRM uses explicit dependency graphs, which are by their very nature in SSA form. Figure 2.2 shows a comparison of a simple C function translated into FIRM and LLVM IR.

In FIRM each node consumes zero or more SSA values and produces zero or one SSA value as output. Each node in the dependency graph has a label or *opcode* in firm parlance, which determines the instruction kind. We may sometimes refer to these opcodes as *types*. Each opcode corresponds to a signature that determines the number of operands. Outgoing edges are indexed.

2.1.3 Data dependencies

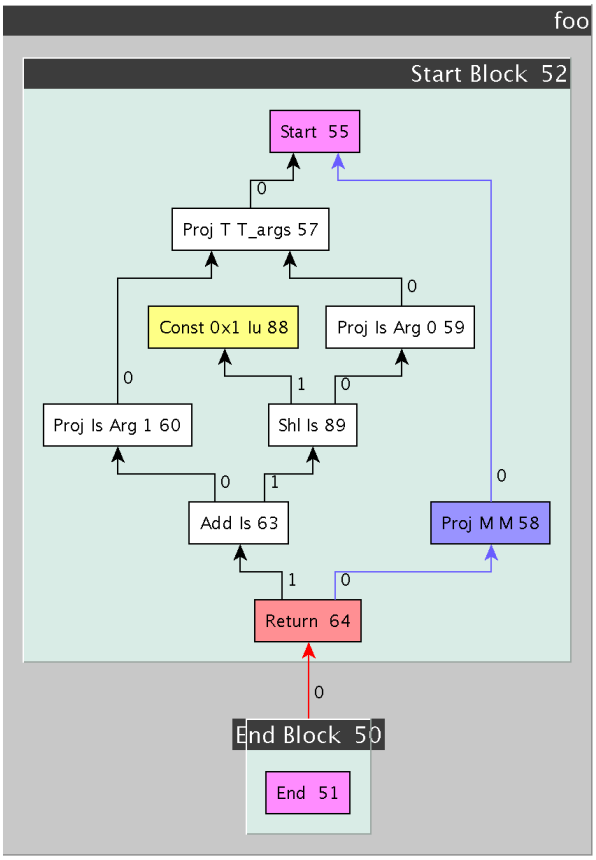
The most straightforward kind of data dependencies are to be found in arithmetic and logical expressions. These dependencies form *define-use-relationships*. One node defines a value that another node uses as an argument for a later computation. Side-effects produce dependencies as well. Reading from and writing to memory are considered such side-effects. In general these reads and writes cannot be arbitrarily reordered without breaking the original semantics. FIRM therefore introduces the notion of memory dependencies through so call M-values, which represent the state of the entire memory and therefore help to force an ordering on memory operations. A load operation, for example, consumes an M-value and produces another one as part of its output. Not all memory operations have to be totally ordered, though. Two reads can be reordered regardless of the addresses they access. Furthermore, often it can be shown that two memory operations do not interfere because they work with different locations. The ordering imposed by M-value dependencies is therefore only required to be partial.

We have already seen that LLVM IR uses named temporary variables to implement data flow. However, this data flow only witnesses define-use-relationships. The order of memory operations is fixed implicitely by the order of the LLVM IR instructions. By the virtue of LLVM IR being instruction based, this order is total. A scheduler has to analyze this total order for parallelizability in a separate step later on. The libFirm approach avoids this problem entirely.

2.1.4 Control flow

All nodes in FIRM are directly assigned to a basic block. Basic blocks can be considered special nodes that take care of control flow. Two nodes A and B belong

```
define i32 @foo(i32, i32) #0 {
  %3 = shl i32 %0, 1
  %4 = add nsw i32 %3, %1
  ret i32 %4
}
```



LLVM IR FIRM

Figure 2.2: A comparison of LLVM IR and FIRM for the same C source. On the left we see the LLVM IR code that clang produces. On the right we see the FIRM graph that the libFIRM C compiler cparser produces. LLVM IR uses temporary variables to form data dependency edges between operations. In contrast FIRM is explicitly graph-based, so the data dependencies are directly drawn as edges. Whenever we present drawings of FIRM graphs or patterns in this thesis we always lay the nodes out from bottom to top. Edges denote dependencies and go from the node that depends on a value at the bottom to the node that provides this value at the top. We may omit drawing arrowheads whenever nodes can be laid out in this fashion. In the few cases when we need to draw edges that go from top to bottom, we draw arrowheads to make the direction clear.

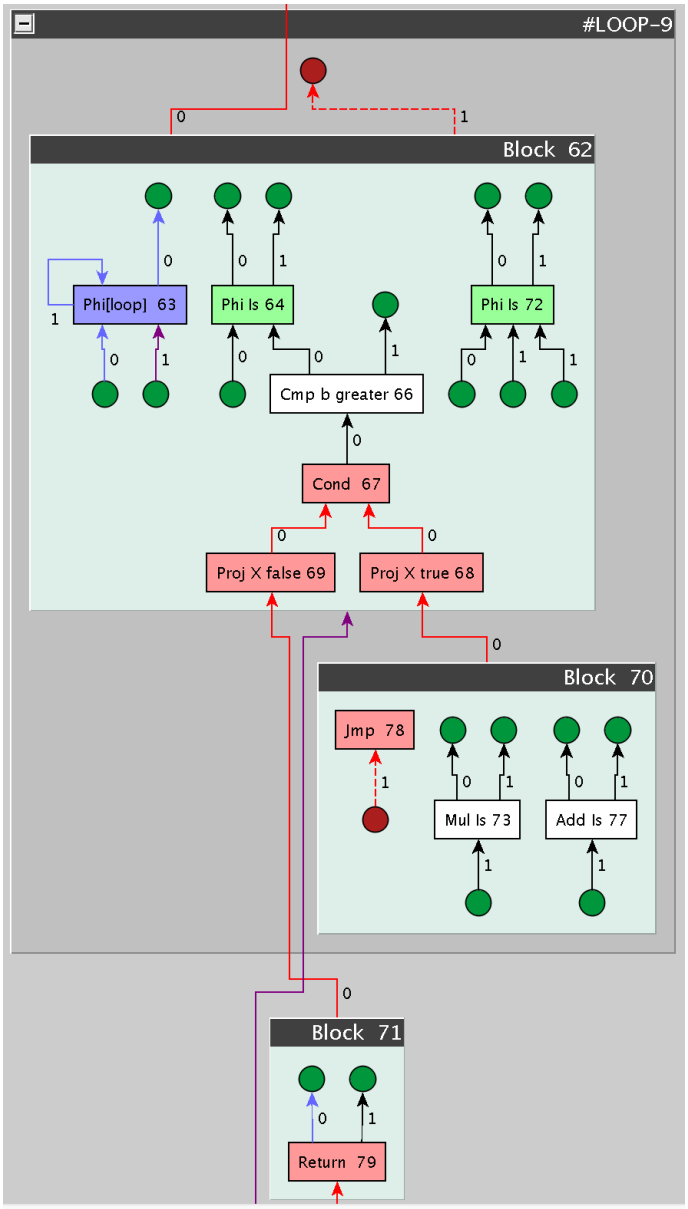


Figure 2.3: A FIRM graph containing a loop, multiple blocks, and ϕ nodes.

to the same basic block if in any conceivable situation where A is run, B is also run and vice versa. Figure 2.3 shows the FIRM graph of a simple loop. We can discern three basic blocks. Block 62 corresponds to the original loops head, block 70 corresponds to the loop body and block 71 is the part after the loop ends. We can also see that there's a control flow dependency between block 70 and the Proj X true node and between block 71 and the Proj X false node. In our work we do not visit basic block nodes during our topological walk through the program graph. We therefore simply leave all control flow dependencies intact and rely on the instruction scheduler to find a good total order.

2.1.5 Modes

Each FIRM node has a *mode*. A mode can be regarded as a type of a value flowing through the IR graph. For integers, for example, there are modes for signed and unsigned versions of 8, 16, 32, and 64 bit widths. Other modes serve as a means to categorize node operations. The most important categories are summarized in the following list. In graph drawings we will also color code the modes. The color is given in parentheses.

I Integer value (white)

B Basic block (gray)

M M-value (blue)

P Memory address (white)

X Control flow (red)

b Boolean (white)

P Pointer (white)

T Tuple (white)

F Floating point (white, not relevant to our work)

Tuple nodes are used to implement operations with multiple output values. A Load node takes an address and an M-value as operands and produces a single tuple as output, so the mode of the Load node is Tuple. The tuple actually holds two results: the retrieved value and another M-value. We can get to these individual results with the help of **Proj** (projection) nodes. Each of the individual results of a tuple operation has a distinct projection number that we can use to get to that value through a **Proj** node. Tuples and **Proj** nodes can greatly simplify analyses because we can always take on the view that each operation produces a single output value only.

2.1.6 Nodes

We will now give a brief overview of some of the node types (opcodes) that are to be found in a FIRM graph. The complete list of opcodes can be found in the official FIRM API documentation [11].

A FIRM graph represents a single procedure in the underlying source language. In our case the source language is C, so a FIRM graph represents a single C function.

Every program graph starts with a **Start** node as an entry point. As we said earlier, every operation produces exactly one output value. If we want to describe an operation with multiple outputs, we have to have the node be a **Tuple** and use **Proj** nodes to access the individual output values. For the sake of brevity we will give signatures that directly show the possible projections and omit the tuple indirection. The **Start** node is an example of such a multi-output operation.

$$\begin{aligned} Start : B &\rightarrow M \times P \times r^n \\ \text{with } r &\in \Big\{P, I, F\Big\} \end{aligned}$$

This **Start** node produces an initial control flow node, a node representing the state of memory when entering the function and a pointer to the frame base pointer. We can use the frame base pointer to access arguments on the stack. We also obtain a set of values representing the initial register values. Depending on the calling convention the function arguments are either to be found in these register values or on the stack.

We can see that **Start** is assigned a basic block by feeding the block to the node as an operand. Blocks can have zero or more control flow nodes as predecessors.

$$Block : X^n \rightarrow B$$

Among others control flow nodes can be **Jmp** for unconditional jumps, and **Cond** for branches.

$$\begin{array}{rll} \text{Jmp}: & B & \rightarrow X \\ \text{Cond}: & B \times b & \rightarrow X \times X \end{array}$$

In addition to the opcode and its arguments a node may specify *attributes*. A **Proj** node specifies the projection index as an attribute and a **Const** node specifies the contant value it represents as an attribute. We will write nodes of type T with an attribute value of w as **T w** such as **Const 1** for a constant one or **Proj 0** for a projection with index zero. When we compare labels for equality we will take the attribute part of the label into consideration as well. This means that **Const 1** is not equal to **Const 0** for example.

FIRM has all the usual arithmetical and logical operations such as **Add**, **Sub**, **And**, **Or** etc. Division fails if the divisor is zero and since failure is a control

flow side-effect, **Div** and **Rem** nodes additionally take an M-value as an operand and produce another M-value as part of their output. These nodes also produce a control flow node that represents normal execution and a control flow node that represents exceptional execution. The source language C doesn't feature exceptions on a language level so if the graph is a transformed C function, these outputs will usually not be used.

Another class of opcodes are memory-access related. The two most important memory-access opcodes are **Load** and **Store**.

$$\begin{array}{lll} \text{Load}: & B \times M \times P & \rightarrow M \times r \\ \text{Store}: & B \times M \times P \times r & \rightarrow M \end{array}$$

There are no surprises here. **Load** takes an M-value and an address and produces an M-value and a result value ($r \in \left\{P, I, F\right\}$). **Store** takes an M-value, an address, and a value to store and returns a new M-value.

We said earlier that each node is assigned to exactly one basic block. As we can see from the example nodes above, this assignment is realized by giving each node a block as an operand. For the following discussion this block operand is irrelevant. We therefore treat all nodes as having no block operand. When we translate the IR graph to a target graph, we simply leave all the block nodes and the block assignments as they appeared in the IR graph. FIRM provisions for special treatment of these block operands by giving them an index of -1.

2.2 IA-32

In this work we implement an instruction selection generator that works with arbitrary architectures as long as they follow certain reasonable prerequisites. We will, however, only test our implementation with one architecture: 32-bit x86, also referred to as IA-32. IA-32 is widely implemented in industrial and research compilers. A lot of work has gone into instruction selectors tailored to this architecture. It therefore provides a good benchmark to compare our efforts with.

IA-32 defines a 32-bit wide address bus and 32-bit general purpose registers. IA-32 processors are CISC machines and therefore feature a great number of instructions. In our work we are handling 535 different instructions. We disregard floating point arithmetic completely.

The IA-32 CISC design makes heavy use of addressing modes. Adressing modes let the programmer specify the source and targets of operands for instructions in great detail. As an example we look at the `MOV` instruction. In its simplest form, `MOV` can load a value from an adress in a register.

```
MOV (%ESI), %EAX
```

We can additionally specify a constant offset. The constant is part of the instruction. This form can be used to load values off the stack.

```
MOV (%EBP-8), %EAX
```

In its fullest extent the `MOV` operation can load a value from an address that is calculated from a base register, an index register, a scale constant, and a displacement constant.

```
MOV -8(%EBP, %EDX, 4) %EAX
```

As we can see, an instruction that is touted as having a single responsibility can actually perform a lot of different things dependening on the addressing modes we choose to use. On the other hand, addressing modes work the same across a wide number of different instructions. Adressing modes can therefore be regarded orthogonal to the instruction set. This view is beneficial for someone who implements a tailored instruction selector because they can use some good heuristics that assume this orthogonality. Unfortunately we want our instruction selection generator to work well independent of such assumptions. For us, adressing modes significantly increase the set of distinct instructions, yet we can't employ any tailored heuristics.

2.3 Instruction selection

Just as a compiler can be structured as a pipeline consisting of a frontend, an optimization phase, and a backend, the backend itself can be divided into different phases. These phases are commonly called instruction selection, instruction scheduling, and register allocation. Instruction selection is the phase in a compiler pipeline where we translate a program graph in IR into a semantically equivalent graph in machine representation. Instruction selectors are mostly written and fine-tuned by hand. This might be worthwhile for widely used architectures such as those of Intel and ARM, but handwritten instruction selection is very costly for architectures with less users such as in the embedded market. Our goal is to find a systematic aproach to instruction selection that lets us generate efficient instruction selectors solely based on an abstract description of an instruction set.

Instruction selection works on the basis of *rules*. A rule is a simple mapping of a *source pattern* in IR to a semantically equivalent *target pattern* in machine representation. In the simplest setup, target patterns correspond directly with instructions in the instruction set. An example rule can be seen in Figure 2.4. Note that in addition to IR and target nodes, these patterns consist of input nodes[2]. These

[2]We use the word 'input' to refer to inputs to patterns. We use 'operand' to refer to inputs to

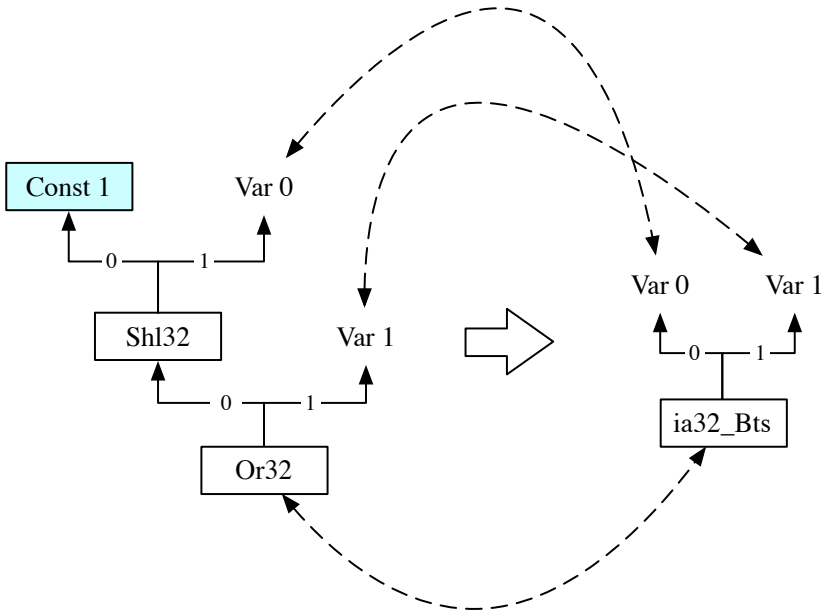


Figure 2.4: A simple single-rooted rule. On the left we see the IR pattern. On the right we see the equivalent target pattern. Roots in the IR pattern and roots in the target pattern are mapped 1 to 1.

nodes represent the input values to the patterns. There exists a mapping of input nodes in the IR pattern to input nodes in the target pattern.

IR languages are designed with semantic compositionality in mind. That means that if we plug IR nodes together to produce patterns, the resulting structure will have the semantics that we would expect. We can fit target languages into such a semantic compositionality framework as well by making all side-effects explicit. In IA-32 some instructions set flags implicitely and some take flags as input implicitely. By making these hidden inputs and outputs explicit, we essentially transform the target instructions into pure functions and therefore obtain this very useful semantic compositionality property. It is this compositionality that makes instruction selection a straightforward exercise on graphs.

2.4 Patterns

In order to be rigorous let us define what program graphs and patterns are precisely.

Definition (Program graph)**.** *Given a signature* Σ *and a function* $\pi : \Sigma \mapsto \mathbb{N} \cup \{\_\}$ *that determines the number of operands* $\pi(l)$ *a node with a label of* l *must have. Operations* l *where* $\pi(l) = \_$ *are variadic. A program graph* $G = (V, E, \gamma, \delta)$ *is a graph with nodes* V*, edges* E*, a node label function* $\gamma : V \mapsto \Sigma$*, and an edge index function* $\delta : E \mapsto \{0, 1, \ldots\}$*. The program graph is well-formed if every node* v *has*

single (FIRM) nodes.

exactly $n = \pi(\sigma(v))$ *operands (if* $n \neq \_$*) and the corresponding edges are labelled* 0 *through* $n - 1$*.*

Definition (Pattern)**.** *Given a signature* Σ*. A pattern* $P = (V_P, E_P, \sigma, e)$ *is a graph with nodes* V_P*, directed edges* E_P*, a node label function* $\sigma : V_P \mapsto \Sigma \cup \left\{ Var, Imm \right\}$*, and an edge index function* $e : E_P \mapsto \mathbb{N}$*. We call a node* v *with a label* $\sigma(v) = l \in \Sigma$ *an instruction of kind* l *or simply an* l *instruction. We call a node with a label of Var a variable input and we call a node with a label of Imm an immediate input. We call the edge labels indices. In a wellformed pattern, all outgoing edges of a single node have distinct edge indices.*

In our implementation we will describe patterns with indexed adjacency lists. The indexed adjacency list of a node v is a mapping $a_v : \mathbb{N} \mapsto V_P$. The edge labels are therefore integrated into these data structures.

Our definition of patterns allows for graphs that are disconnected or contain loops yet in this work we only handle connected DAG patterns. This restriction has no big impact on the amount of patterns that we can handle since most of the patterns that occur in our IA-32 rule set are loop-free anyway.

In DAG patterns we can identify root nodes. A root node is one that has no predecessors in the pattern.

Definition (Pattern root)**.** *Given a pattern* $P = (V_P, E_P, \sigma, e)$ *and a node* $v \in V_P$*. We call* v *a root node (*$v \in root(P)$*) iff there are no edges* $(v_i, v) \in E_P$*.*

There are patterns with a single root node. These are easy to handle because we can view them as trees with some minor caveats. The bulk of our test pattern set is multi-rooted patterns, however.

Similarly we can define leaf nodes.

Definition (Pattern leaf)**.** *Given a pattern* $P = (V_P, E_P, \sigma, e)$ *and a node* $v \in V_P$*. We call* v *a leaf node iff there are no edges* $(v, v_i) \in E_P$*.*

In a well-formed pattern all inputs are leaf nodes. Not all leaves are input nodes, however. Another kind of leaves are nodes that represent constants, for example.

All nodes that are neither leaves nor roots are called inner nodes. We call the set of all inner nodes the *trunk* of a pattern.

All IR patterns follow the signature of the IR language. All target patterns follow the signature of the target language. The input to our instruction selection generator is a set of rules. A *rule* is a tuple of an IR pattern and a corresponding semantically equivalent target pattern. Each root in the target pattern is mapped to exactly one root in the IR pattern. Each input node in the target pattern is mapped to exactly one input node in the IR pattern.

Figure 2.5 shows a multi-rooted rule and its node mappings. Roots and variable leaves are mapped as expected. These mappings help to preserve the semantics of the patterns.

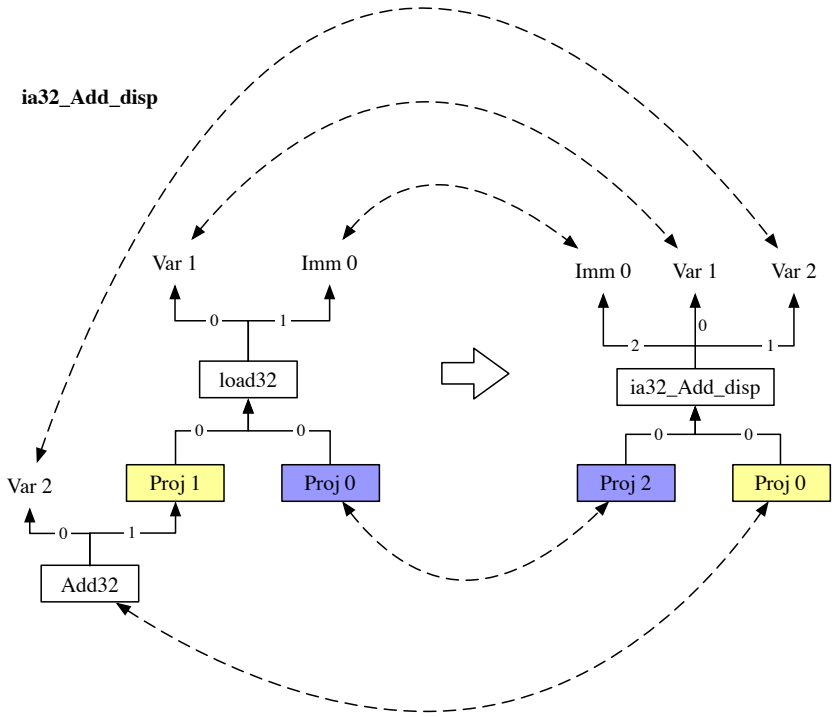


Figure 2.5: A rule with multiple roots. Roots in the IR pattern (left) and roots in the target pattern (right) are mapped 1 to 1. Inputs are mapped to inputs.

All rules in our rule set are of the form of either the rule in Figure 2.4 (single-rooted) or the rule in Figure 2.5 (multi-rooted).

Sometimes we need to talk about the height of a pattern. Since we are only dealing with DAG patterns, we define pattern height as DAG height.

Definition (Pattern height)**.** *Given a pattern $P = (V_P, E_P, \sigma, e)$. Let $L \subseteq V_P$ denote the set of leaves and let $R \subseteq V_P$ denote the set of roots. We define the height h of this pattern as the minimum over all the path lengths between any root and any leaf. (If such a path doesn't exist, we define the path length as ∞.)*

$$h = \min_{l \in L, r \in R} (\mathit{length}(\mathit{path}(P, r, l)))$$

The length of a path is the number of edges on that path.

We can identify special kinds of patterns: those that only consist of a single root node and a set of variable inputs. We call these patterns *atoms*.

Definition (Atom)**.** *A single-rooted pattern $P = (V_P, E_P, \sigma, e)$ over a signature Σ is called an atom iff for all nodes $v \in V_P$:*

$$v \notin \mathit{roots}(P) \implies \sigma(v) = \mathit{Var}$$

Each node has a *downwards view* of the pattern it is a part of[3].

Definition (Downwards view)**.** *Given a pattern $P = (V_P, E_P, \sigma, e)$ over a signature Σ and a node $v \in V_P$. Let V_v denote the set of nodes that are reachable from v in P. The downwards view D of v is the subgraph induced by V_v.*

2.5 Theoretical foundations

Instruction selection can be further divided into two distinct responsibilities: *pattern matching* and *pattern selection.* Again, we can think of these two responsibilities as two separate phases.

2.5.1 Pattern matching

In the first phase, pattern matching, we take a program graph and a set of IR patterns. For each node in the program graph we want to determine the set of rules that *fit the program graph structure* at that particular node (the set of *matches*, defined below). In the second phase, pattern selection, we use these sets to produce a *program graph tiling.* Matching and program graph tiling correspond to the theoretical graph problems of *subgraph isomorphism* and *graph tiling.*

[3]Note again that we draw all of our graphs upside down, so a downwards view is directed upwards in all our figures.

Definition (Graph isomorphism)**.** *Given two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$. We say that G and H are isomorphic ($G \cong H$) if we can find a bijection $f : V_G \mapsto V_H$ such that for any vertices $a, b \in V_G$ there is an edge $(a, b) \in E_G$ iff there is an edge $(f(a), f(b)) \in E_H$.*

Definition (Subgraph Isomorphism)**.** *Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be graphs. The subgraph isomorphism problem asks the question: Is there a graph $G_0 = (V_0, E_0)$, which is a subgraph of G, such that $G_0 \cong H$?*

Subgraph isomorphism is NP-complete for arbitrary graphs. It is easy to verify a given subgraph isomorphism in polynomial time. Cook [12] showed NP-hardness by a reduction from the 3-satisfiability problem, however, it is easier to show NP-hardness by a reduction from the graph clique problem which is known to be NP-hard. In the clique problem you are given a graph G and an integer k and you have to find out if there is a complete subgraph with k vertices in G. If we simply make H such a complete graph of k vertices, we could solve the clique problem with the help of subgraph isomorphism, so subgraph isomorphism must be NP-hard as well.

The definition of subgraph isomorphism loosely maps to our idea of a program graph match. However, we do not operate on arbitrary graphs but on program graphs. We have to take the node labels and edge indices into consideration.

Definition (Match)**.** *Given a program graph $G = (V_G, E_G, \gamma, \delta)$, a pattern $P = (V_P, E_P, \sigma, e)$ and a node $v \in V_G$. We say that P matches in G at v if there is a subgraph isomorphism $f : V_P \mapsto V_G$ that preserves edge indices:*

$$\forall u, w \in V_P : e(u, w) = \delta(f(u), f(w))$$

and node labels:

$$\forall u \in V_P : \gamma(f(u)) = \begin{cases} \_, & \sigma(u) = \mathit{Var} \\ \textbf{\textit{Const}}\ \_, & \sigma(u) = \mathit{Imm} \\ \sigma(u), & \mathit{otherwise} \end{cases}$$

and there is a root $r \in V_P$ such that $f(r) = v$.

We can see from the definition that immediate input nodes only match **Const** nodes, although the value of the constant doesn't matter. Variable input nodes can match any label, as described by the underscore.

If we find a match of P in G at v, we also say that the match is *anchored* at v. Figure 2.6 shows an example of a match.

If we are given a node in the program graph $v \in V$ and a single-rooted pattern $P = (V_P, E_P, \sigma, e)$, it is easy to verify if there is a match anchored at v. We just have to go through both the program graph starting at v and the pattern starting at the root step by step simultaneously in a depth-first order. We follow outgoing edges in the order of their edge indices. On each visited node we compare the labels of both graphs. If we encounter a label mismatch, the pattern doesn't match, else

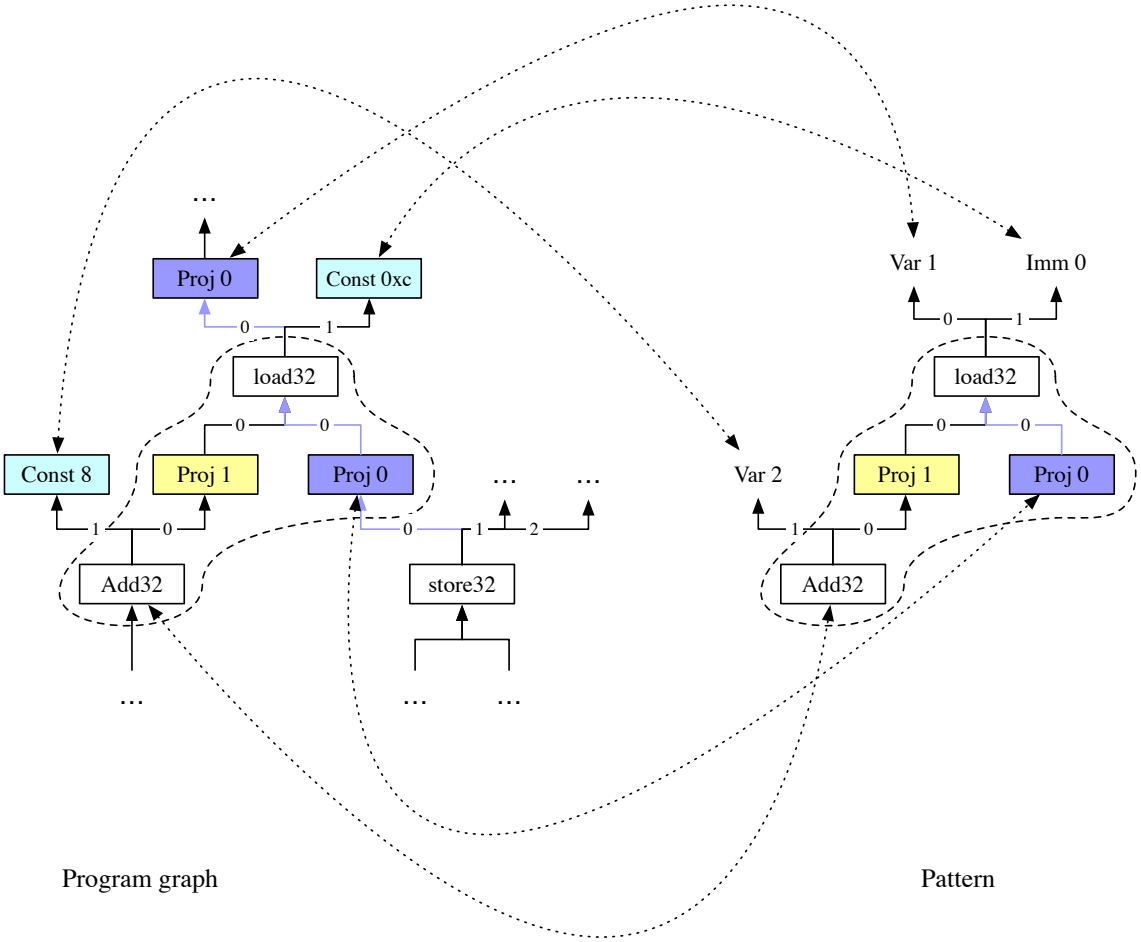


Figure 2.6: An example pattern match. The pattern on the right is matched in the program graph on the left. Pattern nodes are mapped onto a subgraph of the program graph such that (1) edges, (2) edge indices, and (3) node labels are preserved. Notice that an immediate input such as **Imm 0** can be mapped to any **Const _** node such as the **Const 1024** node in the example. Variable inputs such as **Var 1** and **Var 2** can be mapped to nodes of any label. The pattern has two roots, so we say the pattern is anchored both at the **Add32** and the **Proj 0** node.

it does. The special Var nodes are considered wildcards. A comparison of such a variable input always succeeds. The special Imm nodes are considered immediate wildcards. A comparison of such an input node always succeeds if the other node's label is **Const**.

If our rule set does not contain any patterns with variadic operations, pattern matching becomes very simple. Let o denote the maximum edge index as defined by the function $\pi : \Sigma \mapsto \mathbb{N} \cup \{\_\}$ limited on non-variadic operations only. This maxmimum edge index is small. In FIRM $o = 3$. The time complexity of a lookup of outgoing edges of a certain index i can therefore be regarded constant. We can retrieve the i-th outgoing edge of a node in constant time.

With these considerations the simple matching procedure above takes just $|V_P|$ steps. If we want v to enumerate all the nodes in the program graph, the subgraph isomorphism problem has a worst case time complexity of just $|V_P||V_G|$. We have *parameterized* the problem with parameter o. Program graph matching therefore becomes *fixed parameter tractable* for single-rooted patterns. It is easy to see that it is also fixed parameter tractable for arbitrary DAG patterns.

In the pattern matching phase of instruction selection, we want to annotate all matches for all patterns in our very large rule set R. We can perform the procedure above for every pattern $P \in R$. Pattern matching would then take $m|V_G||R|$ steps where m is the maximum pattern size. While this linear time complexity is nice in theory, in practice it is still way too slow, because $|R|$ is very large. We will use this linear approach in order to compare our later solution in terms of quality of the produced output (matches found) and running time.

2.5.2 Pattern selection

In the second phase of instruction selection, pattern selection, we take a program graph that has been annotated according to the pattern selection phase and produce a *program graph tiling*. The theoretical underpinning of program graph tiling is the *graph tiling* problem. In graph tiling, for a given graph G and a set of graphs H (the tiles) we have to find a set of graphs in H that are subgraphs of G such that these subgraphs cover all nodes in G but do not overlap. In our case, the set of tiles H is the set of patterns and G is the program graph. Often we do not just want to find any graph tiling but rather we want to find the best graph tiling according to a given metric. One approach is to attach costs to the patterns. In program graph tiling we want to minimize the accumulated costs of all patterns involved in the tiling.

Graph tiling has a more neat correspondence to program graph tiling than subgraph isomorphism and pattern matching, yet we still need to point out a differing detail. The tiles in program graph tiling are patterns. Patterns have immediate and variable inputs. In this discussion about graph tiling, we consider these inputs as not belonging to the pattern's nodes. This essentially allows some nodes to be matched by two separate patterns: once as a root node and once as an input node which is what we need at the edges of a match. Figure 2.7 shows an example of part of a program graph tiling.

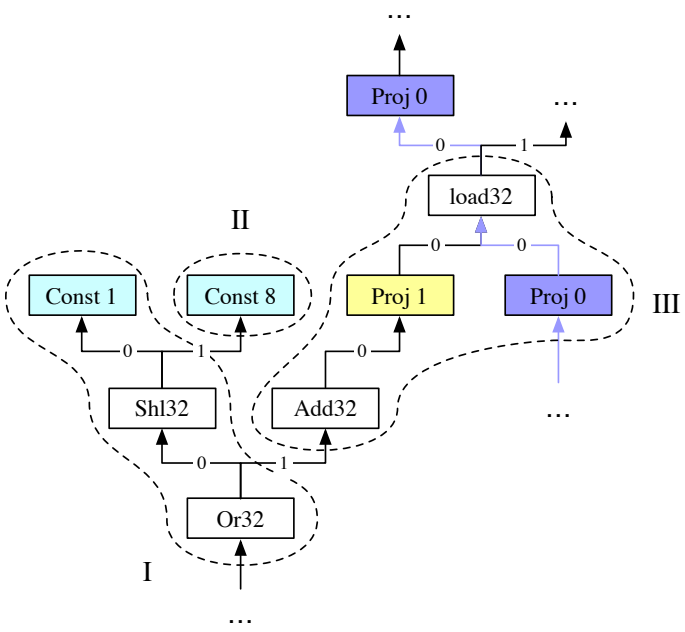


Figure 2.7: An excerpt of a graph tiling. Note how the inputs of pattern I are roots of patterns II and III. By disowning the input nodes from their patterns the tiling becomes non-overlapping.

Since we have already established which tiles fit at which nodes in the program graph due to the preceding pattern matching phase, the pattern selection phase becomes a lot easier. We use a simple greedy approach to solve the program graph tiling problem. Instead of attaching costs to patterns, we use prioritization. The set of all patterns is totally ordered according to their priorities. At each occasion where we have the choice to use one of multiple patterns we always chose the one with the highest priority.

The greedy matcher starts at a designated root node of the program graph (an **End** node) with a partial graph tiling which is empty in the beginning. It then looks up all the annotated pattern matches at that node and chooses the one with the highest priority that doesn't overlap with the current partial graph tiling. The chosen pattern is added to the partial graph tiling. The algorithm then recursively calls itself with the inputs of that pattern as the new root.

```
greedy(partial graph tiling T, node v, program graph G) {
  let ms = sort-by-priority(matches(G, v))
  let res = T
  for (m ∈ ms):
    if m overlaps with T:
      continue
    else:
      // nodes that correspond to the inputs of the pattern
```

```
      let in = inputs(m, G, v)
      for (i ∈ in:
        res = res ∪ greedy(T ∪ m, i, G)
  return res
}
```

Such a greedy approach may fail if our partial graph tiling produces gaps that we are unable to close because all of the patterns are too large. Buchwald and Zwinkau [13] have identified a constraint on the pattern set called *atomic completeness* that helps our case.

Definition (Atomic completeness)**.** *A pattern set H is called atomically complete over a signature Σ if for every distinct label $l \in \Sigma$ there exists an atomic pattern $P \in H$ such that $\sigma(root(P)) = l$.*

We can give a trivial program graph tiling of any program graph by assigning each node its respective atom pattern. It is therefore easy to see that any gap that might occur while we perform greedy matching can trivially be closed by a set of atoms.[4]

The problem we are dealing with in instruction selection is only partly comparable to pure graph tiling. Tiling the program graph is not an end in itself, we rather want to produce a target graph with the same semantics. In the pattern matching phase we do not just annotate nodes with patterns but with rules. We then translate the program graph tiling into a target graph by performing translations according to the rules corresponding to the selected tiles. As we have seen, in a rule each root and input node in the target pattern is assigned a root and input node in the IR pattern. This enables the pattern selector to stick the target patterns together in the correct fashion.

We can translate some graph tilings to a target graph but not all of them. As an example let us look at the rule in Figure 2.5 again. In Figure 2.8 we show a part of a program graph where the IR pattern of this rule matches. However there is no way in which we can translate this match reasonably. In the program graph the **Sub32** node uses one of the inner nodes of the pattern, the **Proj 1**, as an operand. This **Proj 1** node will no longer exist after the translation, however. It will be gobbled up by the ia32_Add_disp node together with the **load32** node. The ia32_Add_disp node condenses the semantics of the entire pattern trunk into a single node. There is no chance to get access to the individual parts after the translation.

We call matches such as the one described above *useless*. Fortunately we can identify useless matches very easily. Note how the match is rendered useless by an edge that goes from outside the match to its trunk. Any match where we notice such edges are useless. We will later see how we can make arrangements that help us to identify such edges early in the pattern matching phase.

[4] Buchwald and Zwinkau introduced another constraint called *composability*. In our solution, however, we only need our pattern sets to be atomically complete.

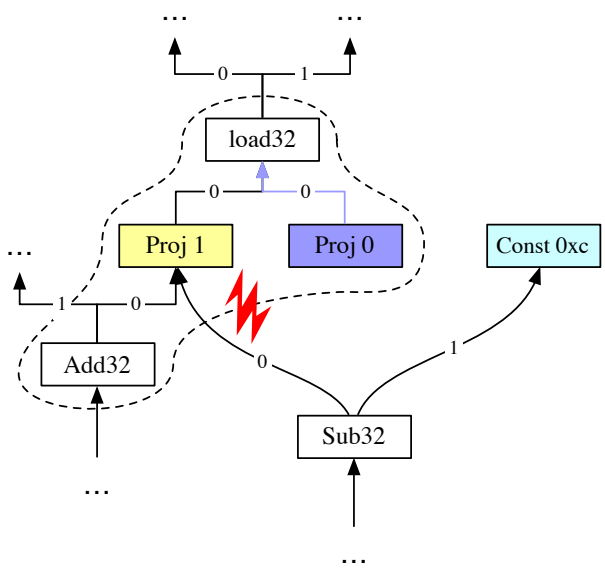


Figure 2.8: An excerpt of a program graph. We see that a pattern matches. However, there is an edge going from the outside of the pattern to the inside. In the resulting target graph such an edge cannot be satisfied. This match is therefore useless to our goal.

2.6 Formal architecture description

Even in RISC architectures, but certainly in CISC architectures such as IA-32, instruction sets can hold a big number of instructions. Since IR languages are usually a lot simpler than target languages, each target instruction will likely correspond to a big number of source patterns and will therefore lead to a big number of rules. For example, the IA-32 **andn** instruction can be respresented by either of these terms (which can be regarded as IR source patterns):

$$(\sim x)\&y \qquad y - (x\&y) \qquad x \oplus (x|y) \qquad y \oplus (x\&y)$$

It is a tedious task to enumerate all these rules by hand. We want to generate the set of rules exhaustively by using a formal description of the architecture. If we have a semantic description of each target instruction and we have a semantic description of the components of the IR language, we can use solvers to have all rules be generated for us by a computer. Up until recently, enumerating rules this way has taken weeks. Buchwald, Fried, and Hack [1] have found a new approach using a combination of enumerative techniques along counterexample-guided inductive synthesis (CEGIS), that brings this down to a few hours for integer instructions in IA-32. The resulting rule set holds more than 60.000 rules. We can use this rule set as input for an instruction selection generator. However, instruction selection algorithms used previously don't cope well with these excessive amounts of rules.

The hypothesis of our work is then to find a new approach to instruction selection that can incorporate huge rule sets. We want to generate an instruction selector which is comparable in both running time and quality of the produced output with industry compilers such as clang [14] or GCC [2] and research compilers such as FIRM [15].

2.7 Generation and compilation

The purpose of our work is to build what is often called a code-generator generator. Code-generator is another word for compiler backend. We build software that generates a compiler from an architecture specification. When we now explain the details of our proposed algorithm we have to be careful not to confuse which parts happen at *generation time* and which parts happen at compile time. We call generation time the run time of our generator. At generation time we are provided with a big set of rules and we produce an instruction selector written in C. This instruction selector is part of a libFirm based C compiler called cparser. When we run cparser with an input C program, we talk about *compile time.* cparser reads its input file, runs a parser, semantic analyzer, optimization passes and then calls our instruction selector. Afterwards, an instruction scheduler takes the final target graph from our instruction selector and schedules it. This step is followed by a register allocator and a peephole optimizer. The final output is IA-32 object code. After calling a linker, we can run the output. This is the start of what we call *run time.*

3 Related work

Compiler construction has a long history of research. Instruction selection, however, is given the least amount of attention out of all the parts of the compiler pipeline. In contrast to compiler frontends, where formal frameworks such as (E)BNF [16] are widely used, compiler backends are often written by hand in an ad-hoc fashion. Blindell provides the most comprehensive overview of formalized instruction selection techniques to date [17].

3.1 Macro expansion

The earliest instruction selectors used macro expansion, where text templates are matched over a representation of the source program. In the beginning macro expansion was used to match templates on the program source code directly instead of on an intermediate representation. Elson and Rake introduced the idea of performing macro expansion on nodes of the abstract syntax tree instead of the program source code [18]. This was the first step towards IR-based instruction selection. According to Blindell, Wilcox was one of the first to implement macro expansion on an intermediate program representation called *source language machine* (SLM) [19].

Writing macros by hand was still a difficult job and macro expansion generators were called for. Snyder was one of the first to come up with a fully operational macro expansion generator that produced a macro expander from a machine description [20]. In contrast to earlier attempts it could handle addressing modes, branching, and function calls.

A major drawback of macro expansion is that each macro can only substitute a single AST or IR node. The resulting code is therefore often of poor quality. Peephole optimization [21] can mitigate this problem. Notably, GCC uses a combination of macro expansion and peephole optimization for its instruction selector in what is called the Davidson-Fraser approach [22].

Cattell provides a survey of macro expanders [23]. Ganapathi et al. provide a survey of early code-generator generators [24].

3.2 Tree matching

In tree matching we are given a program tree and a set of tree patterns. The goal is to cover the tree with the patterns without gaps and overlaps. With tree matching it is possible to divide instruction selection into pattern matching and pattern selection.

Due to the tree structure of patterns and program trees, instructions can be modelled by rules of context-free grammars. Translating a program tree can be achieved by parsing the tree and invoking actions for each grammar rule. In each rule's action we emit assembly code for example. Glanville and Graham use LR parsing to translate program trees in this manner [25]. Their approach consists of translating the pattern set to a table which is then being used for the parsing of program trees.

Hoffmann and O'Donnel introduced Bottom-Up Pattern Matching (BUPM) in 1982 [26]. BUPM is also table-based. We determine matches and costs for all nodes going from the leaves towards the root. At each node we use the information computed earlier in order to find the match with the least cost. Only that match is annotated.

Pelegri-Llopart and Graham introduced the Bottom-Up Rewrite System (BURS) in 1988 [27]. BURS improves on BUPM by allowing rules with variables. This helps to express abstract laws such as commutativity more concisely.

Tree matching algorithms have solid theoretical foundations and can be fast, but in reality program graphs are rarely adequatly representable by simple trees. In order to adapt tree matching to program DAGs, we can split the program DAG into subgraphs that are trees and match these parts individually. With this strategy, however, the produced covers are not optimal. Furthermore, patterns are often DAGs as well. Multi-output instructions cannot be handled at all.

3.3 DAG matching

SSA graphs can be laid out in such a way that each loop includes at least one ϕ node. During instruction selection we can ignore these ϕ nodes because no pattern can include ϕ functions. This makes each SSA graph effectively a program DAG. Program DAG matching can therefore be regarded the supreme discipline of instruction selection.

The earliest attempt to perform instruction selection with DAGs was done by Aho, Johnson, and Ullman in 1976 [28]. Their scheme assumes that each IR node maps to one target machine node, however. This assumption circumvents the pattern matching problem completely. Their solution for the pattern selection problem is to use a straightforward greedy technique, which produces good enough code in most situations.

For pattern matching, early attempts to handle program DAGs were done by Ertl [29]. His DBURG system can only handle tree-shaped patterns and perform instruction selection for each basic block separately, however.

Garey and Johnson have shown that matching trees on DAGs is NP-complete [30]. In contrast to tree matching, which can be solved optimally in linear time, DAG matching must therefore employ heuristics that lead to suboptimal solutions.

Most DAG matching techniques transform the program DAG into trees, perform (optimal) tree matching, and then reassemble the original DAG. There are two ways

to split DAGs: edge splitting and node duplication. A near-optimal technique has been introduced by Koes and Goldstein in 2008 [31]. Their solution is based on the one by Ertl. They use both edge splitting and node duplication but claim to always choose the one technique which is the least detrimental to code quality in every situation. Like Ertl's design, however, they can only handle tree-shaped patterns.

Eckstein et al. [32] were the first to use the abstract optimization problem PBQP as a framework for an instruction selection algorithm. Their technique could only handle tree-shaped patterns, however. Ebner et al. [33] extended the technique to incorporate arbitrary DAG patterns. However, both employ transformations that could lead to situations where the problem instance no longer has a solution even though the original instance did. Buchwald and Zwinkau [13] identify atomical completeness and compositionality as two properties of pattern sets that are required for their PBQP-based instruction selectors to always succeed. Atomical completeness is vital to the algorithm we use in this work as well.

4 Proposed solution

In order to meet all the requirements stated in the previous chapter, we propose a two-phase instruction selector that translates any given IR program graph into a target graph of equal semantics. The two phases neatly correspond to pattern matching and pattern selection as a general framework. In the pattern matching phase we annotate each node in the program graph with the set of matching patterns. Afterwards we can use these annotations in the pattern selection phase in order to produce a target graph. For the pattern selection phase we employ a simple greedy approach, which we explain later. The emphasis of our work is on the pattern matching phase. We will now motivate the idea of our pattern matching algorithm, explain its traits, and point out some difficulties.

4.1 Motivation

In order to motivate our algorithm design, we start with a straightforward but slow linear algorithm and analyze its shortcomings. This simple algorithm is works by trying out every pattern at every node and annotating accordingly. We can perform this linear search with the pattern set shown in Figure 4.1 and the program graph shown in Figure 4.2. The labels a, b, and c are placeholders.

Let us assume we want to annotate the node marked with a green x in Figure 4.2. We first check if pattern I matches at the marked node in the program graph. We perform this check by comparing the labels at each node in depth-first order. Variable argument nodes can match any type of node. We see that pattern I matches because all the labels match. Pattern II doesn't match, because the labels at the root are different. We now move on the node marked with a blue y. Again, we try if pattern I matches but we soon come to the conclusion that it doesn't because of the label mismatch at the root. In order to check if pattern II matches, we compare all the labels in a depth-first order again and arrive at the conclusion that it is indeed a match. However, we had to move deep into the program graph in order to arrive at this conclusion and we performed a lot of the same checks we did earlier when we were testing the green marked node for a match of pattern I. We call this the *duplicate work problem*.

Similar patterns make us perform similar checks which leads to a lot of duplicate work. This duplicate work is non-essential. A fast algorithm would perform as little accidental (non-essential) work as possible.[1] We will try to extract the similarities

[1] The distinction of essential vs. accidental complexity originated in software engineering research [34].

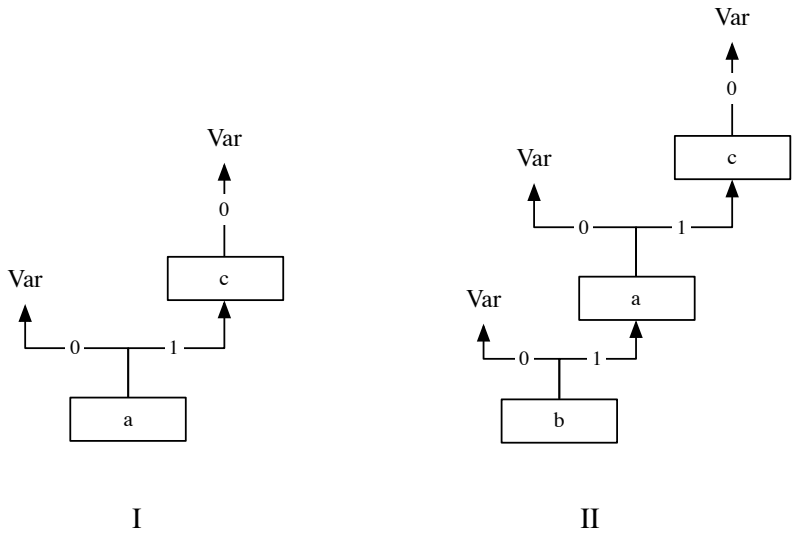


Figure 4.1: A simple pattern set with two patterns. The pattern on the left is included as a subpattern in the pattern on the right.

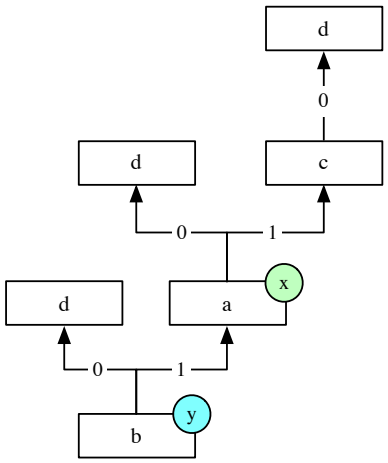


Figure 4.2: A simple program graph.

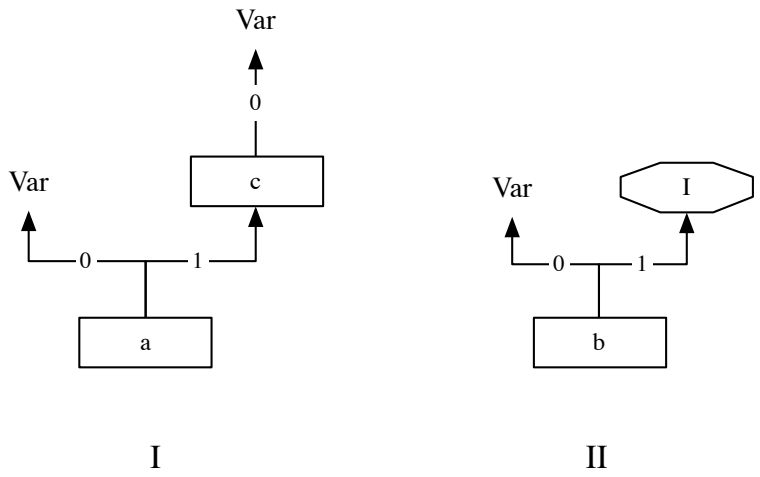


Figure 4.3: A factorized pattern set. The pattern on the right has a reference to the pattern on the left (visualized by the octagon shape) as a subpattern.

and perform checks for them separately. We call this technique *decomposition* or *factorization* because it is a similar approach to how we would simplify a term in a (semi-)ring for example: $a \times b + a \times c = a \times (b + c)$. If we calculate the term in its more expanded form on the left we do more work than in the factorized version on the right. In an analogous way, we try to avoid duplicate work in our pattern matching checks. An ideal matching algorithm would check for the equal subpattern in patterns I and II just once. Figure 4.3 shows one possible way to factorize patterns I and II. Note that sub patterns need names in order to be referenced in other patterns. In this case we can reuse the names from the original patterns but in general we might have to invent new names.

If we now perform the matching algorithm with this decomposed pattern set, we avoid unnecessary duplicate work by reusing work done earlier. The matching process is the same for pattern I at the green marked node, but when we now want to find a match for the orange marked node, we can reuse this information and do not have to dig deep into the program graph again. We simply check if the subpattern I of pattern II has been matched previously at the green x node. We now need to traverse the nodes in the program graph in a topological order because annotating nodes in the program graph might depend on earlier work done above.

The simple linear depth-first-search approach to pattern matching has another problem. More than 80% of the patterns in our test pattern set are multi-rooted. There is no straightforward way to handle multi-rooted patterns with this linear approach. Let us have a look at the situation in Figure 4.4. When we are at the b node in the program graph, we can say that the pattern might potentially be a match regarding the current *downward view* but we have no idea if the other root with the label c is to be found anywhere. One awkward solution to find this other root is to go to the node a and then traverse all incoming edges in the reverse direction. However, while the number of outgoing edges is bounded and small, the number of

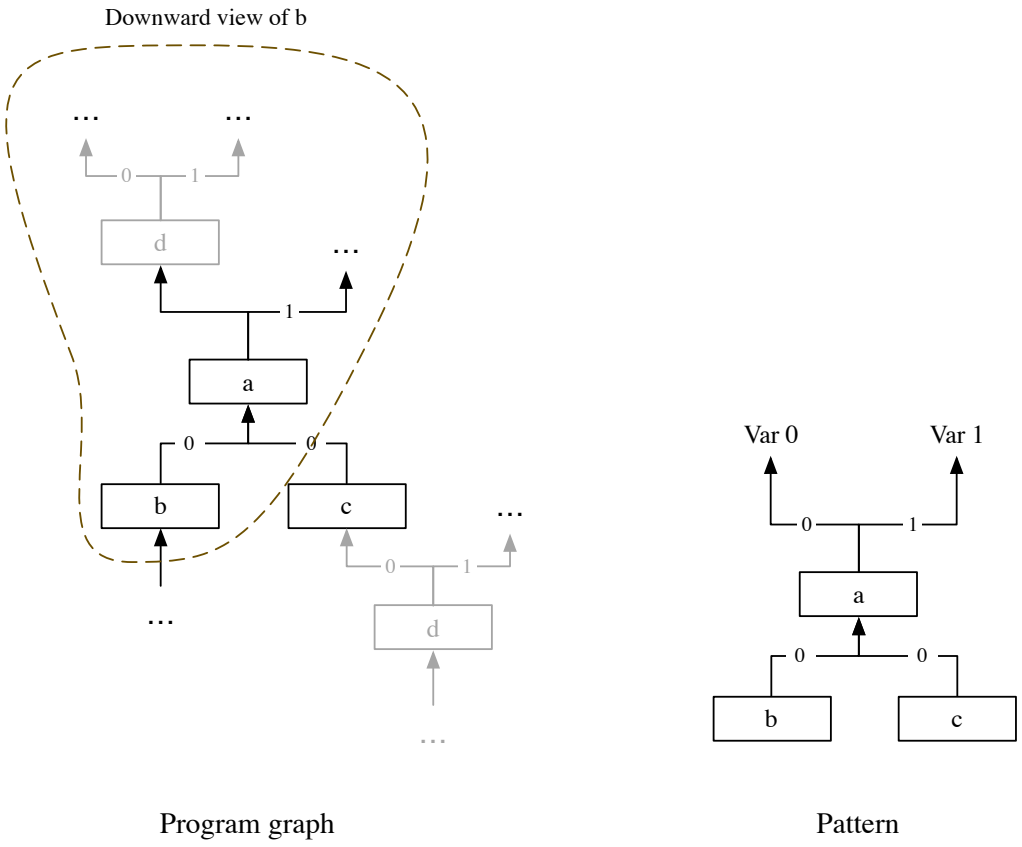


Figure 4.4: A program graph and a multi-rooted pattern. From the point of view of the b node in the program graph, there is no easy way to tell if the pattern matches. The downward view of b does not include the other root c of the pattern.

incoming edges is unbounded. Traversing all incoming edges might be very costly. We call this the *reverse-edge-direction problem*.

4.2 Overview

We introduce *pattern decomposition* and a *bottom-up annotation algorithm* in order to tackle the duplicate work problem and the reverse-edge-direction problem respectively. Pattern decomposition happens entirely at generation time. Our generator is written in Haskell, so we present the ideas of pattern decomposition in functional style pseudo code. We assume our pseudo code to be evaluated lazily. The generator uses the result of the pattern decomposition to generate C code for the bottom-up annotation algorithm. The annotation algorithm itself runs at compile time and is given a program graph as input. We present the idea of the annotation algorithm in imperative style pseudo code similar to C.

The following snippet shows how the parts are connected. A `decompose` function

decomposes the pattern set into a set of intermediate patterns called *flat patterns*. In `genCode` we take these flat patterns and generate the corresponding C code.

```
gen :: {Rule} -> String
gen = genCode . decompose

decompose :: {Rule} -> {FlatPattern}

genCode :: {FlatPattern} -> String
```

We will explain how the latter two functions work in detail over the course of this chapter.

The remainder of this chapter is organized as follows. First, in Section 4.3, we explain the pattern selection phase and the process of translating an annotated program graph into a semantically equivalent target graph. We therefore learn what groundwork the pattern selection phase needs. We explain next how the pattern matching phase provides this groundwork. In Section 4.4 we explain how decomposition and the bottom-up annotation algorithm work in the context of tree patterns. This helps to understand the general idea of both concepts in a less complex environment. In Section 4.5 we develop these ideas to accomodate single-rooted DAG patterns. In Section 4.6 we incorporate bulletin boards in order to accomodate double-rooted DAG patterns. In Section 4.7 we expand this technique to arbitrary DAG patterns. In Section 4.8 we point out some nuances.

4.3 Pattern selection and program graph translation

In the pattern selection phase we are given an annotated program graph that we have to translate to a semantically equivalent target graph. As we have seen in Section 2.5 pattern selection corresponds to program graph tiling.

Our solution to the pattern selection problem is a straightforward greedy approach. libFirm calls the following function with the root of the program graph.

```
ir_node *be_transform_node(ir_node *node) {
  ir_node *new_node = NULL;
  if (AUTOTRANSFORM_ENABLED) {
    new_node = autotransform(node);
  }
  if (new_node == NULL) {
    // call the handwritten instruction selector
    new_node = handtransform(node);
  }
  return new_node;
}
```

If the autotransform extension is enabled, the function calls `autotransform`, which is provided by our generator. In some situations (**Phi** or **Sync** nodes for example) our generated instruction selector is not able to find a match. In this case the `autotransform` function returns NULL. The handwritten instruction selector then takes over and performs a transformation.

```
ir_node *autotransform(ir_node *node) {
  annotation **as = get_annotations(node);
  for (annotation a : as) {
    transform_info_t *tinfo = get_transform_info(a);
    ir_node *tnode = transform(node, tinfo);
    if (tnode) {
      return tnode;
    }
  }

  return NULL;
}
```

The autotransform function just scans through the annotations, which are sorted best-first, and tries to perform the corresponding transformations. The `transform` function does the heavy lifting.

```
ir_node *transform(ir_node *node, transform_info_t *tinfo) {
  // Check for overlap with the current partial program graph
     tiling
  if (has_overlap(node, tinfo)) {
    return NULL;
  }

  ir_node *input_0 = get_input_0(tinfo);
  ir_node *input_1 = get_input_1(tinfo);
  ir_node *input_2 = get_input_2(tinfo);
  ir_node *input_3 = get_input_3(tinfo);
  ir_node *input_4 = get_input_4(tinfo);

  inputs = [];
  // Recursively transform the inputs
  for (input *i : get_inputs(tinfo)) {
    inputs[get_index(i)] = be_transform_node(get_node(i));
  }

  result = tinfo->constructor(tinfo, inputs);

  ir_node *new_node;
  for (ir_node *r : get_roots(tinfo)) {
    ir_node *new_r = get_transformed_root(result, r);
    be_set_transformed_node(r, new_r);
```

```
    if (r == node) {
      new_node = new_r;
    }
  }

  return new_node;
}
```

We first check if the selected transformation annotation overlaps with the current partial graph tiling. This check can simply be implemented by iterating over all nodes that are covered by the match and check if they were transformed before. The details of this check are of little importance.

Now we call `be_transform_node` recursively for each of the inputs. Next, we call a node constructor provided by the annotation. The result holds a transformed node for each of the root nodes mentioned in the annotation. One of these roots is the node `node` we want to transform. We return this transformed root in the end.

This recursive scheme starts at the root of the program graph, greedily selects the best pattern match and then recursively calls itself with the inputs of that match. We have seen that an annotation consists of a set of input nodes, a set of root nodes, and a constructor function pointer. We will now see how our pattern matcher provides these annotations.

4.4 Tree matching

We will now walk through the innards of our pattern matcher by starting with a slimmed down version that only works with tree patterns.

4.4.1 Pattern decomposition

Patterns are defined as labelled graphs. We represent these graphs with the help of adjacency lists.

```
data Pattern = Pattern {
  vertices   :: {Vertex}
  argsMap    :: Map Vertex (Map Index Vertex)
  labels     :: Map Vertex Label
  targetInfo :: TargetInfo
}
```

The field `vertices` holds the set of all nodes. The field `argsMap` holds a map of indexed adjacency lists for each node. In order to get the nth operand of node v in a pattern p we have to perform two lookups:

```
let am = argsMap p
let os = lookup(am, v)
let y  = lookup(os, n)
```

The field `labels` holds a label for each node. The field `targetInfo` holds information needed for the translation of the pattern to its target equivalent.

```
data TargetInfo = TargetInfo {
  targetGraph :: TargetGraph
  rootMap     :: Map Vertex Vertex
  inputMap    :: Map Vertex Vertex
  constructor :: String
  priority    :: Int
}
```

The fields `rootMap` and `inputMap` correspond to the maps we explained in Figure 2.5. They express how nodes in the target graph are mapped to nodes in the pattern. The `priority` field holds a value that is used later in the pattern selection phase. Patterns that are higher prioritized are preferably selected over other patterns.

In the decomposition phase we want to decompose patterns to smaller patterns that make life easier. Since our pattern set is atomically complete, we can take decomposition to the extreme and decompose all patterns down to the atomic parts. We call these atomic parts *flat patterns*. A flat pattern is a tree of height 0 (a single **Const** node) or 1 (most other cases).

```
data FlatPattern = FP {
  flatTree   :: FlatTree
  targetInfo :: Maybe TargetInfo
}

data FlatTree = FT {
  rootLabel :: Label
  children  :: Map Int FlatTreeNode
}

data FlatTreeNode = FTRef FlatPatternId | FTVar | FTImm
```

Figure 4.5 shows an example of a flat pattern decomposition. As we can see, every node other than the inputs correspond to one flat pattern.

A flat pattern holds a description of its structure as a *flat tree*. A flat tree is simply a labelled root and a set of indexed children. The children are either immediate or variable inputs or refer to other flat patterns. If a flat pattern corresponds to a root of the original pattern, it holds the original target info of the pattern in the field

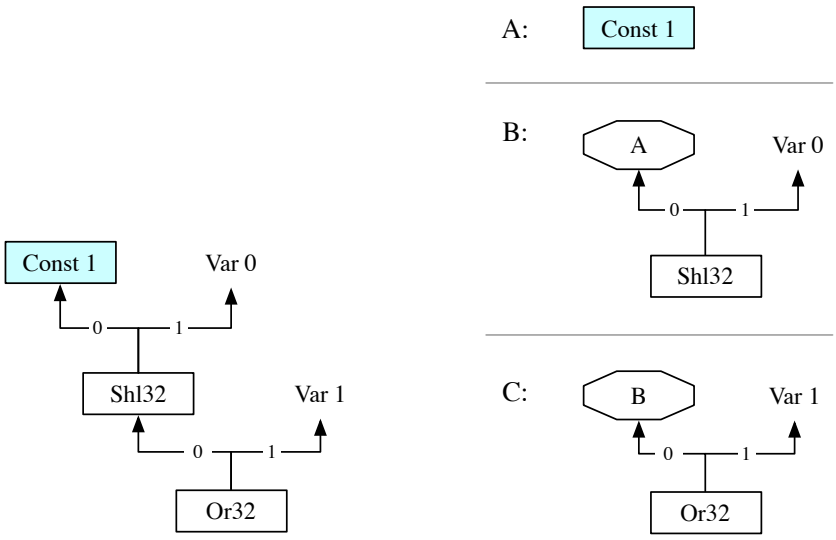


Figure 4.5: A pattern (left) and its decomposition (right). Flat patterns are trees of height one. The leaves are variable or immediate inputs or references to other flat patterns. References are denotes by the octagon shapes.

`targetInfo`. If the flat pattern does not correspond to a root, the `targetInfo` field is set to `Nothing`.

As we alluded to earlier, every flat pattern needs an id to enable us to refer to it from other flat patterns. The id encodes the structure of the flat pattern. We want two flat patterns a and b to have the same id iff whenever a matches, b also matches and whenever b matches, a also matches. This means that when we obtain a flat pattern from decomposing one pattern and we obtain another flat pattern from decomposing another pattern and both flat patterns have the same id, we can just keep one of the two flat patterns and discard the other. Calculating flat pattern ids is an intricate matter. We defer this discussion to Section 4.8.1.

Flat patterns are patterns in the sense of the definition in Section 2.4. We can therefore use the term *match* for flat patterns as well.

The function `decompose` is straightforward.

```
decompose p = lefts (map (mkFlatPattern p) (vertices p))

mkFlatPattern :: Pattern -> Vertex -> Either FlatPattern ImmVarFlag
data ImmVarFlag = ImmFlag | VarFlag
```

The `lefts` function takes a collection of `Either a b`, discards all the `Right` values and unpacks the `Left` values. The `mkFlatPattern` function performs the main work.

```
mkFlatPattern p v = case lookup v (labels p) of
  Var ... -> Right VarFlag
```

```
  Imm ... -> Right ImmFlag
  lbl -> Left (FlatPattern {
    flatTree = ft
    targetInfo = if isRoot p v
                   then Just (targetInfo p)
                   else Nothing
    } where ft = FlatTree {
      rootLabel = lbl
      children = Map.map (mkFTChild p) (lookup v (argMap p))
    }
```

Variable and immediate input nodes are mapped to the respective flag. For non-input nodes we forward the target info and form a flat tree. For the flat tree we translate each of the node's children with `mkFTChild`.

```
mkFTChild :: Pattern -> Vertex -> FlatTreeNode
mkFTChild p v = case mkFlatPattern p v of
  Right VarFlag -> FTVar
  Right ImmFlag -> FTImm
  Left fp -> FTRef (flatPatternId fp)
```

As we can see, we call `mkFlatPattern` recursively. The recursion will terminate because with each step we move closer to the leaves of the pattern and patterns do not contain cycles. At the leaves the recursion stops. We can memoize the results of these calls in order to make the entire computation faster.

This is the basis of what decomposition is about. The main difficulty lies in the computation of flat pattern ids (`flatPatternId`). We cover this topic in Section 4.8.1.

4.4.2 Bottom-up annotation algorithm

After decomposition we generate code for the bottom-up annotation algorithm. We will first explain how the algorithm works and then how we generate its code. At compile time we are given a program graph. Our goal is to annotate each node with the set of all patterns that match at this node.

We visit the nodes in topological order.

```
void annotateAll(ir_graph *irg) {
  walk_topological(irg, annotate);
}
```

This means that the `annotate()` function is called with each node and if we make sure to annotate the given node correctly, we can be certain to have annotated all of the node's operands before. We visit the nodes moving from leaves to roots, or

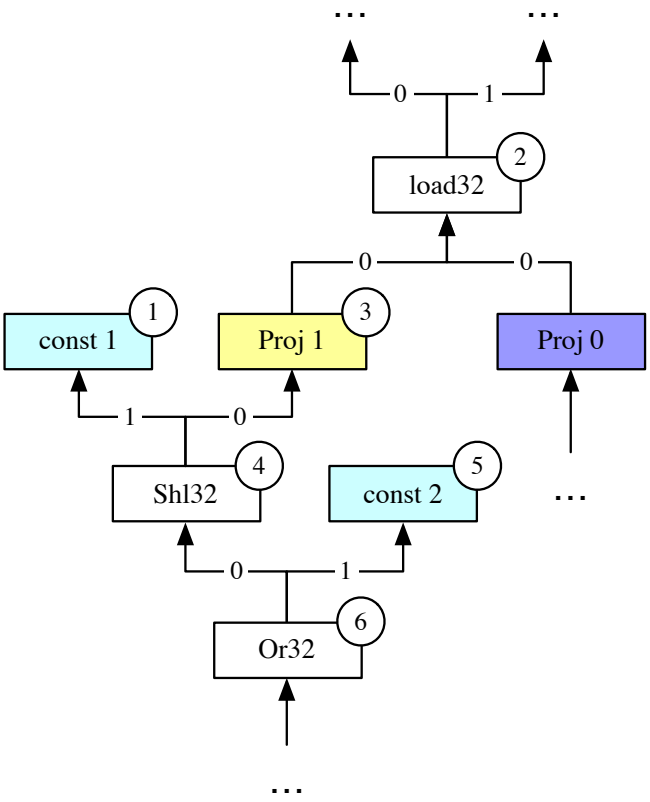


Figure 4.6: An excerpt of a program graph.

bottom-up[2]. The function is best explained with the help of a small example. Assume that the rule set we were given solely consisted of rule R in Figure 2.4. We decompose the rule as shown in Figure 4.5 which means we obtain three different flat patterns: A, B, and C. A part of the program graph that we want to annotate is shown in Figure 4.6. The nodes in the program graph are annotated with numbers. Assume we already annotated all nodes above and including node 2. We choose a topological visitation order of 1, 3, 4, 5, 6 for the other nodes.

The code for `annotate()` takes care of the three flat patterns A, B, and C and will look like this:

```
void annotate(ir_node *node) {
  if (label(node) == Const
      && const_value(node) == 1) {
    addAnnotation(node, A);
  }

  if (label(node) == Shl32
```

[2]The term *bottom-up* is confusing since leaves are drawn at the top in this thesis. When we say bottom-up, we always mean going from the leaves towards the roots.

```
      && hasAnnotation(operand(node, 0), A)) {
    addAnnotation(node, B);
  }

  if (label(node) == Or32
      && hasAnnotation(operand(node, 0), B)) {
    addAnnotation(node, C);
    handle_C(node);
  }
}

void handle_C(ir_node *node) {
  target_info_t *ti = new_target_info();
  set_inputs(ti, <pointers to the inputs>);
  set_constructor(ti, &c_orShift);
  set_roots(ti, node);
  addPatternAnnotation(node, ti);
}
```

We see that for each flat pattern we generate an if statement. The condition contains predicates for the root label and in some cases the operands' annotations. If the statement succeeds, we annotate that we found a flat pattern match. In the case of flat pattern C, we also annotate that we found a pattern match (inside `handle_C`), because flat pattern C corresponds to the root of the original pattern (we say that C is a *root flat pattern*). For a pattern match to be of use to us later, we need to store some additional information in a `target_info_t` structure. The structure holds pointers to the pattern's inputs and roots and a constructor function. There is one constructor function for each target instruction. In our generator we just look up these constructor function pointers from an external data map.

We chose to use a separate `handle_C` function to wrap the `addPatternAnnotation` call. These functions will do more heavy lifting once we adapt the algorithm to multi-rooted patterns.

4.4.3 Generating the bottom-up annotation code

Generating this code consists of two parts: generating the main `annotate()` function and generating each of the `handle_X()` functions.

```
genCode :: {FlatPattern} -> String
genCode fps = concat (map genAnnotate_X fps)
              ++ genAnnotate fps
```

When we only handle single-rooted patterns, the `handle_X()` functions are simple.

```
genAnnotate_X :: FlatPattern -> String
genAnnotate_X fp
 = "void annotate_$(id fp)(ir_node *node) {"
   "target_info_t *ti = new_target_info();"
   "set_inputs(ti, $(<make pointers to the inputs>));"
   "set_constructor(ti, &$(constructor (targetInfo fp)));"
   "set_roots(ti, node);"
   "addPatternAnnotation(node, ti);"
   "}"
```

For the main `annotate()` function we first gather a map of predicates. In this map we associate each flat pattern with the set of predicates we need to check in order to make sure that the flat pattern matches at the given node.

```
fpPredicatesMap :: {FlatPattern} -> Map FlatPattern {Predicate}
fpPredicatesMap fps = Map.fromList
                        (map (\fp -> (fp, predicates fp)) fps)

predicates :: FlatPattern -> {Predicate}
predicates fp = {labelPredicate fp}
                `union` operandsPredicates fp
                `union` specialPredicates fp

labelPredicate fp
 = case (rootLabel (flatTree fp)) of
     Const X -> "label(node) == Const && const_value(node) == $X"
     Shl32   -> "label(node) == Shl32"
     Or32    -> "label(node) == Or32"
     ...     -> ...

operandsPredicates fp = set (map operandPredicate
                                 (children (flatTree fp)))

operandPredicate :: (Int, FlatTreeNode) -> Predicate
operandPredicate (idx, FTVar) = "true"
operandPredicate (idx, FTImm) = "is_immediate(node)"
operandPredicate (idx, FTRef id) = "hasAnnotation(get_irn_n(node,
   $idx), $id)"

specialPredicate = ...
```

The set of predicates for a given flat pattern consists of a label predicate, a set of operand predicates and a set of special predicates. The label predicate checks if the root label of the given flat tree matches at the current node in the program graph. The operand predicates check that for each of the operands we find annotations for the referenced flat patterns or immediate nodes if we need an immediate input. The special predicates take care of further details. Arithmetic nodes need a predicate to

check that the node has an integer mode, for example. These special predicates are very tied to the way FIRM works. We get the set of special predicates as a map from a source external to our generator. The details of these predicates are irrelevant to the understanding of our algorithms.

With the help of the flat pattern predicates map we can now generate the checks for flat pattern matches.

```
genAnnotate :: {FlatPattern} -> String
genAnnotate fps
 = let fpPreds = fpPredicatesMap fps in
     "void annotate(ir_node *node) {\n" ++
     concat (map check fpPreds) ++
     "}"

check :: (FlatPattern, {Predicate}) -> String
check (fp, preds)
 = "if ($(intercalate " && " preds)) {\n" ++
   "  addAnnotation(node, $(id fp));\n" ++
   if isRootFlatPattern fp
     then "handle_$(id fp)(node)"
     else ""
   ++ "}"
```

We simply lay out all the checks in a linear fashion. For each check we open a single if statement and concatenate all predicates to form a conjunction. In Section 4.8.2 we will see how we can improve on this linear layout. The general approach of collecting predicates and laying them out in if statements stays the same, however.

4.4.4 Avoiding useless matches

Recall how we discussed that some matches can never be part of a valid program graph tiling. In Section 2.5.2 we called such matches *useless*. It is easy to avoid collecting useless matches with one more predicate for each flat pattern. Matches are rendered useless if there are additional edges going from outside a match to one of its inner nodes. We can detect useless matches by keeping track of the number of incoming edges. Iterating over incoming edges is expensive but obtaining the in degree is cheap.

In decomposition we have to calculate the in degree for all the nodes in a pattern and store this information in the flat patterns.

```
data FlatPattern = FlatPattern {
  ...
  inDegree :: Int
}
```

```
mkFlatPattern p v = case lookup v (labels p) of
  ...
  lbl -> Left (FlatPattern {
    ...
    inDegree = patternInDegree p v
  })

patternInDegree :: Pattern -> Vertex -> Int
patternInDegree p v = length (filter (pointsTo p v) (vertices p))

pointsTo :: Pattern -> Vertex -> Vertex -> Bool
pointsTo p to from = let args = lookup from (argsMap p) in
                       any (map f args)
                       where
                         f (idx, v) = v == to
```

With this information we can add one more predicate to the set of predicates for each flat pattern.

```
predicates fp = ...
                `union` {inDegreePredicate fp}

inDegreePredicate fp = if root fp
                         then "true"
                         else "get_irn_n_edges(node) == $(inDegree
                            fp)"
```

This way we make sure to not match any useless patterns. For flat patterns that correspond to roots in the original pattern we must not restrict the in degree because roots are at the edge of a match and can therefore be used as inputs any number of times.

4.5 Single-rooted DAG matching

We will now adapt our solution to single-rooted DAG patterns. Such patterns already fit the `Pattern` data type definition due to our choice to use adjacency lists to model edges.

We can view a DAG as a tree and a set of *branch convergences*. A branch convergence is a set of paths. If we start at the root and follow each of these paths we must end up at the same node in all cases. We call this node a *confluence node*. Figure 4.7 shows an example DAG and its representation as a tree with a branch convergence. Since outgoing edges are indexed, we can describe paths by a list of indices.

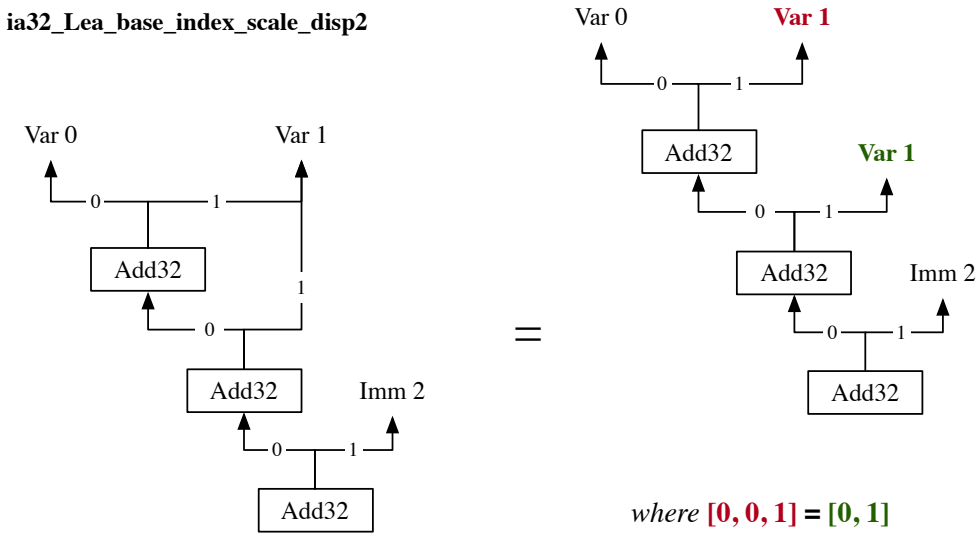


Figure 4.7: A single-rooted DAG pattern and its tree representation. For the tree representation we need to specify an additional branch convergence. A branch convergence is a set of paths that have to lead to the same node starting at the root.

```
type Convergence = {Path}
type Path = [Index]
```

We save these convergences in the flat pattern data structure that corresponds to the root of the original pattern.

```
data FlatPattern = FlatPattern {
  ...
  convergences :: Maybe {Convergence}
}
```

We can find all branch convergences during pattern decomposition.

```
mkFlatPattern p v
 = case lookup v (labels p) of
     ...
     lbl -> Left (FlatPattern {
       ...
       convergences = if isRoot p v
                      then Just (findConvergences p v)
                      else Nothing
     } ...
```

The `findConvergences p v` function works by iterating over all nodes w in the pattern p and enumerating all paths from v to w. If there is more than one such path we have found a convergence.

Now during the bottom-up annotation algorithm we have to check if the convergences occur in the program graph as well. We simply add this check as one more predicate.

```
predicates fp = ...
                `union` convergencePredicates fp

convergencePredicates :: FlatPattern -> {Predicate}
convergencePredicates fp
 = case convergences fp of
     Nothing -> {}
     Just cs -> map convergencePredicate cs

convergencePredicate :: Convergence -> Predicate
convergencePredicate paths
 = "same("
   ++ intercalate ", " (map followPath paths)
   ++ ")"

followPath :: Path -> String
followPath = followPath' . reverse

followPath' :: Path -> String
followPath' [] = error
followPath' [idx] = "get_irn_n(node, $idx)"
followPath' idx:idxs = "get_irn_n($(followPath idxs), $idx)"
```

The `same` function is a variadic helper function that checks if all of the arguments are pointers to the same node.

Adding predicates for the convergences is all it takes to adapt our solution to single-rooted DAG patterns.

4.6 Double-rooted DAG matching

We now want to add DAG patterns with two roots to our pattern set. An example pattern is shown in Figure 4.8. Let's try to employ the bottom-up annotation algorithm as described up until this point in the example program graph shown in Figure 4.9. Suppose we arrive at the Add32 node on the left. So far in the bottom-up annotation algorithm our view was solely directed downwards into the program graph. We therefore come to the conclusion that the ia32_Add_disp pattern matches at the Add32 node. This conclusion is premature, however. The match we found is only a partial match. We are still lacking information about the Proj 0 root: Is such

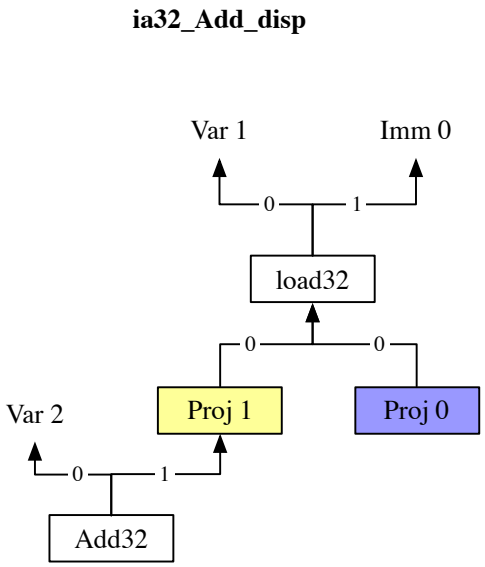


Figure 4.8: An ia32_Add_disp pattern with two roots. The two roots are a consequence of the load32 memory instruction.

a root to be found in the program graph and if so, where? Suppose we move on with the algorithm and arrive at the Proj 0 node. Again, we come to the conclusion that the pattern matches at this node. Now we are in the same situation as before, however. We only found a partial match and do not know where the other root is to be found or if it even exists at all. From a bird's eye view we see that the two partial matches fit together, but so far we have no means to use this perspective in our pattern matcher.

When we are at the **Add32** node we could search for the **Proj 0** root node explicitly. We first move down to the **load32** node, which is a meeting point node in respect to both roots. We now have to iterate over all incoming edges of this meeting point. For each incoming edge we have to check if the node at the start of the edge is a **Proj 0** node. This does not sound like much extra work, but iterating over incoming edges can be costly, because their number is unbounded. Let τ denote the average number of incoming edges at each node in an average program graph. Moving down to the meeting point node and then looking for **Proj 0** nodes in this manner therefore takes $\tau + 2$ steps on average.[3]

Even if τ is small and the cost of this method is therefore acceptable in this specific situation, there are situations in which the cost grows significantly. We are not guaranteed to arrive at the **Add32** node before we arrive at the **Proj 0** node. Suppose we arrive at the **Proj 0** node first. Now, if we want to search for **Add32** root nodes explicitly, we have to move down to the **load32** node and then go up two levels. On average, this takes $1 + \tau^2$ steps. The cost grows exponentially with the

[3]We assume that we want to find all occurrences of **Proj 0** nodes. In practice, we only need to find one **Proj 0** node.

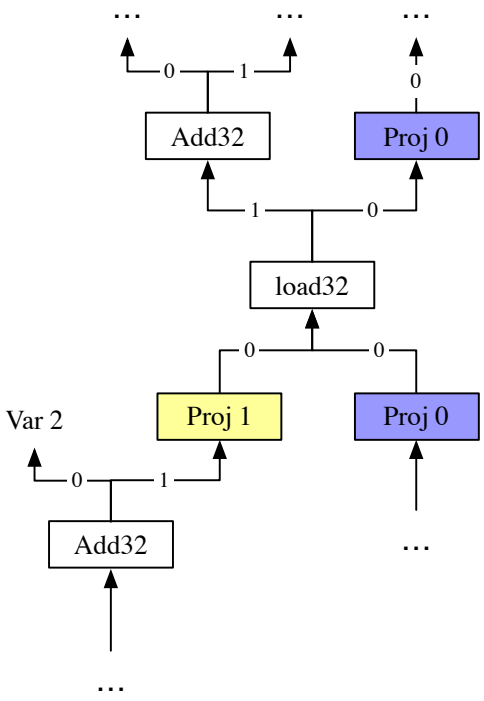


Figure 4.9: An excerpt of a program graph containing an instance of a double-rooted pattern.

distance between the roots and the meeting point.

The in-degree check, which we introduced earlier, helps to keep τ fairly small (depending on the pattern set), because we immediately reject matches with edges pointing from outside the pattern to its inner nodes. Iterating over incoming edges might therefore still be a viable option in practice, at least when it comes down to performance. There is another, non performance-related, problem in this situation, however. Starting at the **Proj 0** root, we can only directly access two of the three inputs of the pattern. The **Var 2** input can only be reached by the **Add32** root. If we look for the **Add32** node explicitly, we must then also look for this missing input node. Explicitly searching for roots and inputs quickly becomes unwieldy. We therefore reject this approach for implementation complexity reasons.

4.6.1 Virtual bulletin boards

The source of our problem is that we arrive at the two roots of the same pattern at different points in time. We have to asynchronously exchange information between these two events. Instead of looking for the roots explicitly, we use a method people use in the real world in order to exchange information: bulletin boards. Bulletin boards make asynchronous communication possible between complete strangers. We

want to replicate those characteristics of bulletin boards for our purposes[4].

The following listing shows the C `struct` definition of a virtual bulletin board. The primary purpose of our bulletin boards is to acquaint two compatible roots with one another via the `root_0` and `root_1` fields. The *pid* field holds a unique identifier of the pattern. It allows for different pattern matching passes to find the bulletin board they are concerned with.

```
typedef struct {
  long pid;

  ir_node *root_0; // gather all roots
  ir_node *root_1;

  ir_node *input_0; // gather all inputs
  ir_node *input_1;
  ir_node *input_2;
  ir_node *input_3;
  ir_node *input_4;
} bb_t;
```

The secondary purpose of bulletin boards is to gather all the additional information necessary to perform a pattern translation. The *input_x* fields gather all inputs to the pattern.

Where do we set up these bulletin boards? Meeting points, such as the **load32** node in the example above, make up ideal information exchanges. A meeting point is a node that is accessible from two or more roots in a pattern. We call a program graph node that we entrust with holding a bulletin board a *hub*.

A single hub can hold many bulletin boards. A bulletin board is concerned with a single pattern id only. In this way, independent pattern matching operations do not interfere with each other because they communicate via independent bulletin boards.

4.6.2 Extending the bottom-up annotation algorithm

Recall that we split the matching process into a main `annotate()` function and `handle_X()` functions in the case of root flat patterns. So far, the latter did not do much work, but now we can use these functions to handle the bulletin board extension. At the point when we call `handle_X` we have only found a partial match. The purpose of these handler functions is to find full matches.

Definition (Partial match)**.** *Given a program graph $G = (V_G, E_G, \gamma, \delta)$, a multi-rooted pattern $P = (V_P, E_P, \sigma, e)$, and a node $v \in V_G$. Let r be one of the roots in the pattern and let V' be the set of nodes in P that we can reach by traversing P*

[4]We are essentially describing asynchronous unbounded channels as described in the concurrency programming literature (e.g. Reppy [35]). Since we are not dealing with true concurrency, we give this concept a new name to avoid confusion.

starting at r. Let P' be the subgraph of P induced by V'. We say that P is a partial match in G at v, if P' matches in G at v.

Let A be the flat pattern id of the **Add32** root and let B be the flat pattern id of the **Proj 0** node. We want the annotation function for A to look as follows.

```
void handle_A(ir_node *node) {
  // Get the hub
  ir_node *hub = get_irn_n(get_irn_n(node, 1), 0);

  // Get the bulletin board
  bb_t *bb = get_bulletin_board(hub, <ia32_Add_disp pattern id>);

  // Enter node as root_0
  bb->root_0 = node;

  // Enter all known inputs
  bb->input_0 = <the IMM_0 node>;
  bb->input_1 = <the VAR_1 node>;
  bb->input_2 = <the VAR_2 node>;

  // Check if we know everything
  if (bb->root_0
      && bb->root_1) {
    target_info_t *ti = new_target_info();
    add_inputs(ti, <pointers to the inputs>);
    set_constructor(ti, &c_ia32_Add_disp);
    set_roots(ti, bb->root_0, bb->root_1);

    // Add annotation for both roots
    addPatternAnnotation(bb->root_0, ti);
    addPatternAnnotation(bb->root_1, ti);
  }
}
```

And similarly we want the annotation function for B to look like this:

```
void handle_B(ir_node *node) {
  // Get the hub
  ir_node *hub = get_irn_n(get_irn_n(node, 1), 0);

  // Get the bulletin board
  bb_t *bb = get_bulletin_board(hub, <ia32_Add_disp pattern id>);

  // Enter node as root_1
  bb->root_1 = node;

  // Enter all known inputs
  bb->input_0 = <the IMM_0 node>;
```

```
  bb->input_1 = <the VAR_1 node>;

  // Check if we know everything
  if (bb->root_0
      && bb->root_1
      && bb->input_0
      && bb->input_1
      && bb->input_2) {
    target_info_t *ti = new_target_info();
    add_inputs(ti, <pointers to the inputs>);
    set_constructor(ti, &c_ia32_Add_disp);
    set_roots(ti, bb->root_0, bb->root_1);

    // Add annotation for both roots
    addPatternAnnotation(bb->root_0, ti);
    addPatternAnnotation(bb->root_1, ti);
  }
}
```

We first get the hub and the bulletin board corresponding to the current pattern. Since we only get to the `handle_X` functions as root nodes, we can enter `node` as a root in the bulletin board. In the case of flat pattern A we enter the node as `root_0` and in the case of flat pattern B we enter the node as `root_1`. This arrangement is arbitrary, but it has to be consistent. We just need to know of both roots in the end. Next, we gather all the inputs we can access from the current root node. In the case of flat pattern A — the one that corresponds to the **Add32** node in the original pattern — we know of all three inputs. In the case of flat pattern B, however, we only know the identity of two out of three inputs. In the end we check if we know everything we need for a pattern annotation. If this is the case, we build a `target_info_t` struct as before and annotate both roots.

Assume we are working with a program graph where the ia32_Add_disp pattern above matches. We want to find this match with the bottom-up annotation algorithm. Assume we first arrive at the **Proj 0** node that is the right root of the pattern. We learn that the pattern partially matches. We therefore enter the `handle_B` function. There we get the bulletin board from the hub at the **load32** node and enter the **Proj 0** node as `root_1`. We also enter the location of the two inputs. Next, we check if the bulletin board knows of the location of both roots and all three inputs. That is not the case, we are lacking `root_0` and `input_2`. We continue the matching process. Eventually we arrive at the **Add32** node that is the left root of the pattern in the program graph. We learn that the pattern partially matches. We therefore enter the `handle_A` function. We get the bulletin board from the hub at the **load32** node and enter the **Add32** node as `root_0`. We can also enter the location of all three inputs. We now check if the bulletin board knows the location of all roots and inputs. Along with the info provided by the earlier pass, the information in the bulletin board is now complete, so we annotate the pattern match at both the **Add32** and the **Proj 0** nodes.

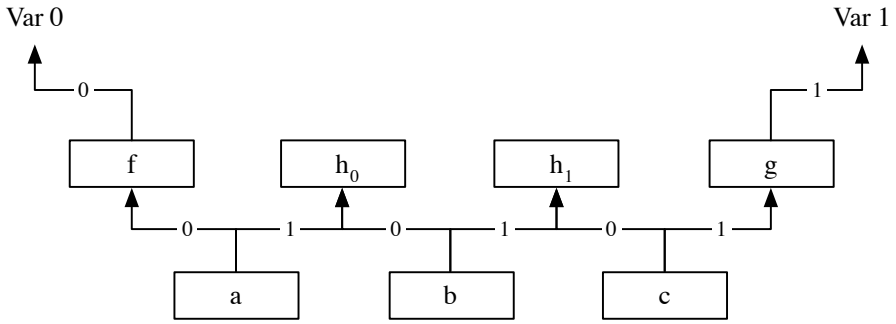


Figure 4.10: A pattern with three roots and two inputs. No two roots taken together can access all inputs. This makes gathering inputs very intricate.

4.7 Multi-rooted DAG matching

With the introduction of bulletin boards for double-rooted patterns, we laid the groundwork for matching arbitrary multi-rooted patterns. There are, however, some minor annoyances that can occur with three or more roots per pattern. For instance, gathering all the roots and inputs for DAG patterns with more than two roots can be complicated. So far, with double-rooted DAG patterns, information that was missing for one root could always be provided by the other root. With some triple-rooted patterns, no two roots taken together can access all inputs. An artificial example of such a pattern is shown in Figure 4.10. Assume we want to find a full match of this pattern in a program graph where this pattern occurs. Assume we arrive at the root with the label **b** first. At **b** we can access neither of the two inputs, so we only enter our own root location in the bulletin boards at the hubs h_0 and h_1. Assume we arrive at root **a** next. We can access **Var 0** and the hub h_0. From h_0 we obtain the location of the root **b** but nothing more. We enter our own root location and the location of the input in the hub h_0. We do not know where root **c** and the input **Var 1** are to be found, so we continue. Assume we arrive at root **c** next. We can access **Var 1** and the hub h_1. From h_1 we obtain the location of the root **b** but nothing more. We enter our own root location and the location of the input in the hub h_1. We are lacking the location of root **a** and **Var 0** so we still have not found a full match. Unfortunately we have already visited all the roots of the pattern. The pattern will therefore never be found. Due to the order of visitation we failed to propagate all the necessary information properly.

We only have limited influence on the order of visitation of program graph nodes. Even if we could fully control the visitation order, there are situations in which no visitation order will lead us to finding all patterns with the scheme above. The ideal visitation order for one pattern might conflict with the ideal visitation order for another pattern. Manipulating the visitation order is therefore no solution to the problem of incomplete propagation of information. Instead, we adapt the bulletin boards to accomodate multi-rooted patterns properly.

We have each bulletin board hold a list of subscribers. An interested root node

can register a callback function that triggers whenever the bulletin board changes. In the example above, root **b** subscribes to changes in the bulletin boards at h_0 and h_1. Now, when root **a** adds the location of **Var 0** and **a** to the bulletin board at h_0, the respective callback function is called. The callback function propagates the new information from hub h_0 to hub h_1. Finally, when it is **c**'s turn, hub h_1 holds all information necessary to enable **c** to decide that we found a full pattern match. If we have every root node subscribe to all accessible hubs, we are guaranteed that for every visitation order, all information eventually propagates to a root that can decide whether a full match is found.

To implement this scheme, we adapt our bulletin boards. The new version is shown in the listing below. In addition to the fields for roots and inputs, we now have a set of callbacks and subscription keys. The callbacks will be the `handle_X` functions. For root **b** the `handle_B` function is shown in Figure 4.11.

```
typedef struct {
  long pid;

  void (**callbacks)(ir_node *); // interested subscribers
  ir_node *subscribers[];        // subscription keys

  ir_node *root_0;               // gather all roots
  ir_node *root_1;
  ir_node *root_2;

  ir_node *input_0;              // gather all inputs
  ir_node *input_1;
  ir_node *input_2;
  ir_node *input_3;
  ir_node *input_4;
} bb_t;
```

Root **b** can access two hubs, so we first get the two respective bulletin boards. Next, we gather all information locally. We combine the information from the two hubs and we add our own root location as `root_1`. We can now check if this information is complete. If it is, we build a `target_info_t` package as before and annotate all the roots accordingly. We have successfully found a full match so we return from the function. If we do not have complete information yet, we subscribe to both bulletin boards for further updates. The callback function is the `handle_B` function itself. We use `node` as the subscription key to unsubscribe later. In any case, afterwards we propagate information to the two bulletin boards. The `add_info` function adds the given information to the bulletin board and returns `true` if any new information has actually been added or `false` if the bulletin board already had all the given information. If we added new information, we notify all subscribers to the bulletin board. The `notify` function takes a bulletin board and a function pointer. It iterates through the list of subscribers of the bulletin board and calls the callbacks. If one

of the callbacks is equal to the function pointer in the second argument, the call is skipped. This means we notify all subscribers *other than ourselves*, else this scheme would recur endlessly.

We use the pattern in Figure 4.10 to illustrate the behaviour of these new handler functions. The `handle_A` and `handle_C` functions look similar to the `handle_B` function in Figure 4.11. Assume we are arriving at a **b** node in a program graph where this pattern matches. As before we find a partial match so we enter the `handle_B` function. We access the bulletin boards at the hubs h_0 and h_1 and combine their information with the location of `root_1`. This information is not complete yet, so we subscribe to both bulletin boards. We also put the combined information into both bulletin boards at the hubs. Since we added information at each bulletin board, we notify all subscribers other than ourselves. At this point there are no other subscribers.

Assume that we arrive at the **a** node in the program graph at a later point. Again we take the information from the only accessible hub at h_0 and combine it with our own information, which is the location of `root_0` and `input_0`. The information is still not complete, so we subscribe to the bulletin board at h_0. We also put the new information in the bulletin board and therefore notify the list of subscribers. The only other subscriber is `handle_B`. We call it.

When we enter `handle_B` a second time, there is new information in the bulletin board at hub h_0. Our information is still not complete, however. We call `subscribe` on both bulletin boards again, but this has no effect, because we were already subscribed and subscription is idempotent. Now, we call `add_info(bb_0, ...)` but bulletin board 0 already knows everything we want to tell it. The call therefore returns `false` and we do not notify the subscribers. Bulletin board 1, however, gets new information from us, so we notify its subscribers of which there are none. Bulletin board 1 now holds information for all fields except `root_2` and `input_1`.

We finally arrive at the **c** node in the program graph. We gather information from the bulletin board h_1 and add the location of `root_2` and `input_1`. The information is now complete, so we build a `transform_info_t` package and perform an annotation at all three roots. Our scheme was able to match this intricate triple-rooted pattern.

4.8 Details

In this section we explain some more details that we skipped over during the previous sections.

4.8.1 Calculating flat pattern IDs

Every flat pattern that we generate needs an identifier so we can reference the flat pattern in other flat patterns. The id encodes the structure of the flat pattern such that two flat patterns a and b shall have the same id iff for every program graph whenever a matches, b also matches and vice versa.

```
void handle_B(ir_node *node) {
  // Get the hubs
  ir_node *hub_0 = get_irn_n(node, 0);
  ir_node *hub_1 = get_irn_n(node, 1);

  // Get the bulletin boards
  bb_t *bb_0 = get_bulletin_board(hub_0, <id>);
  bb_t *bb_1 = get_bulletin_board(hub_1, <id>);

  // Gather all information locally
  ir_node *root_0 = bb_0->root_0 || bb_1->root_0;
  ir_node *root_1 = node;
  ir_node *root_2 = bb_0->root_2 || bb_1->root_2;
  ir_node *input_0 = bb_0->input_0 || bb_1->input_0;
  ir_node *input_1 = bb_0->input_1 || bb_1->input_1;

  // Check if a full match is found
  if (root_0 && root_1 && root_2 && input_0 && input_1) {
    target_info_t *ti = new_target_info();
    add_inputs(ti, input_0, input_1);
    set_constructor(ti, &<constructor>);
    set_roots(ti, root_0, root_1, root_2);
    addPatternAnnotation(root_0, tinfo);
    addPatternAnnotation(root_1, tinfo);
    addPatternAnnotation(root_2, tinfo);

    // Unsubscribe from all bulletin boards
    unsubscribe(bb_0, node);
    unsubscribe(bb_1, node);
    return;
  } else {
    // Subscribe to all bulletin boards
    subscribe(bb_0, node, &handle_B);
    subscribe(bb_1, node, &handle_b);
  }

  // Propagate new information
  if (add_info(bb_0, root_0, root_1, root_2, input_0, input_1)) {
    // bb_0 changed, notify subscribers
    notify(bb_0, &handle_B);
  }
  if (add_info(bb_1, root_0, root_1, root_2, input_0, input_1)) {
    // bb_1 changed, notify subscribers
    notify(bb_1 &handle_B);
  }
}
```

Figure 4.11: The handle_B function for a multi-rooted pattern. We can access two hubs. We get the corresponding bulletin boards and merge their information. We also add the root location of the currently handled root node. If all necessary information is collected, we construct a `target_info_t` structure as before and we unsubscribe from all bulletin boards. If our information is not yet complete, we subscribe to all bulletin boards for further updates. We then propagate our merged information to the two bulletin boards.

```
flatPatternId :: FlatPattern -> FlatPatternId
```

Recall the final algebraic data type definition of flat patterns and flat trees:

```
data FlatPattern = FP {
  flatTree :: FlatTree
  targetInfo :: Maybe TargetInfo
  inDegree :: Int
  convergences :: Maybe {Convergence}
}

data FlatTree = FT {
  rootLabel :: Label
  children  :: Map Int FlatTreeNode
}

data FlatTreeNode = FTRef FlatPatternId | FTVar | FTImm
```

It is obvious that the structure of the flat pattern must include the fields `flatTree`, `inDegree`, and `convergences`. If a and b differ in any of these three fields, they are not interchangable. We use a cryptographic hash function to generate an identifier from these fields. The `targetInfo` has to be handled with more care.

```
flatPatternId fp = hash (flatTree fp,
                         inDegree fp,
                         convergences fp,
                         relevant (targetInfo fp)
  where ...
```

Not all details of the `targetInfo` are relevant to the creation of a flat pattern id. We use the function `relevant` to carve out the relevant parts. First up, the `targetInfo` field is only set for root flat patterns. A root flat pattern is never interchangeable with a non-root flat pattern.

```
relevant Nothing = Nothing
relevant (Just ti) = Just (carve ti)
-- or: relevant = fmap carve
```

Recall the definition of the `TargetInfo` type:

```
data TargetInfo = TargetInfo {
```

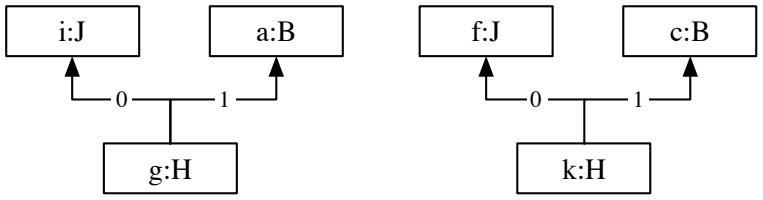


Figure 4.12: Two isomorphic patterns. The patterns only differ in the vertex names. Each vertex is annotated in the scheme x : Y where x is the vertex identifier and Y is the label.

```
  targetGraph :: TargetGraph
  rootMap :: Map Vertex Vertex
  inputMap :: Map Vertex Vertex
  constructor :: String
  priority :: Int
}
```

The `carve` function must forward the fields `priority` and `constructor` because no two flat patterns that differ in these fields are interchangeable.

```
carve ti = (priority ti,
            constructor ti,
            ...)
```

The other fields of the target info include data of the type `Vertex`, which are vertex identifiers. The `TargetGraph` type uses these identifiers internally to express the structure of the graph. The `rootMap` and `inputMap` fields map IR pattern vertex identifiers to target pattern vertex identifiers. So far when we drew graphs, we omitted these identifiers and only showed the node labels. The vertex identifiers are internal implementation details. They never reach the generated output. Nevertheless, internally we can have two patterns that are exactly the same but use different vertex identifiers. Figure 4.12 shows two such patterns. Each vertex is annotated in the scheme $x : Y$ where x is the vertex identifier and Y is the label. We can clearly see that the two graphs are isomorphic.

The `carve` function has to include the structure of the target info without using the specific vertex identifiers. The `TargetGraph` includes such vertex identifiers. We first reassign these identifers by traversing the target graph. Since all our target graphs have a very simple structure, this is an easy exercise and consistently assigns the same vertex identifiers for isomorphic target patterns. This translation of old vertex identifiers (`Vertex`) to new vertex identifiers (`NewVertex`) is expressed in the function `vertexTranslate`. We omit showing the implementation because it includes a lot of details that are irrelevant to the understanding.

```
vertexTranslate ti :: Vertex -> NewVertex
```

We translate the vertices in the range of the `inputMap` and `rootMap` accordingly.

```
    rootMap' ti  = mapValues vertexTranslate (rootMap ti)
    inputMap' ti = mapValues vertexTranslate (inputMap ti)
```

For the identifiers in the domain of these maps we use a different technique. Recall that each flat pattern corresponds to a vertex in the original IR pattern. Flat patterns with a target info correspond to roots. Each root is mapped to a target root via `rootMap`. Instead of including the entire root map, we are only interested in the target vertex of the currently processed root.

```
targetRoot ti = lookup r (rootMap' ti)
r = originalVertex fp
```

Assume that the `originalVertex` function can lookup the original vertex identifier of the corresponding flat pattern from a global data store. In practice, since Haskell is pure, we have to carry this vertex through explicitly.[5]

We disregard the other roots of the pattern entirely. These other roots are irrelevant to the structure of the current flat pattern.

Handling the `inputMap` is a bit more intricate. First, we want to disregard all inputs that are inaccessible from the current root. Next, we translate each vertex identifier in the domain of the map by replacing it with the path from the current root to the input.

```
-- mapMaybeKeys :: Map a b -> (a -> Maybe c) -> Map c b
inputMap'' = mapMaybeKeys f inputMap'
  where f :: Vertex -> Maybe Path
        f v = path r v
```

The `targetRoot` and `inputMap'` are now completely devoid of vertex identifiers but still encode the entire target info structure relevant to the current root. The following listing shows the complete process in summary.

```
flatPatternId fp = hash ((flatTree fp),
                         (inDegree fp),
                         (convergences fp),
```

[5]Or implicitely in a Reader monad

```
                        (relevant (targetInfo fp))
  where
    relevant = fmap carve
    carve ti = (priority ti,
                constructor ti,
                targetRoot ti,
                inputMap'' ti)
    rootMap' ti  = mapValues vertexTranslate (rootMap ti)
    inputMap' ti = mapValues vertexTranslate (inputMap ti)
    vertexTranslate v = ...
    targetRoot ti = lookup r (rootMap' ti)
    r = originalVertex fp
    inputMap'' ti = mapMaybeKeys f (inputMap' ti)
      where f v = path r v
```

4.8.2 Layouting matching conditions

The main `annotate` function has the responsibility of finding partial matches by matching flat patterns. We did this by first collecting a set of predicates for each flat pattern. We then checked for these predicates in an `if` condition.

In Section 4.4.3 the main `annotate` function looked like this:

```
genAnnotate :: {FlatPattern} -> String
genAnnotate fps
 = let fpPreds = fpPredicatesMap fps in
     "void annotate(ir_node *node) {\n" ++
     concat (map check fpPreds) ++
     "}"

check :: (FlatPattern, {Predicate}) -> String
check (fp, preds)
 = "if ($(intercalate " && " preds)) {\n" ++
   "  addAnnotation(node, $(id fp));\n" ++
   if isRootFlatPattern fp
     then "handle_$(id fp)(node)"
     else ""
   ++ "}"
```

Figure 4.13 shows an example of a flat pattern and its decomposition into flat patterns. The above code generates the following C code for these flat patterns.

```
void annotate(ir_node *node) {
  if (label(node) == Add
      && is_immediate(get_irn_n(node, 0))
      && get_irn_n_edges(node) == 1) {
    addAnnotation(node, A);
```

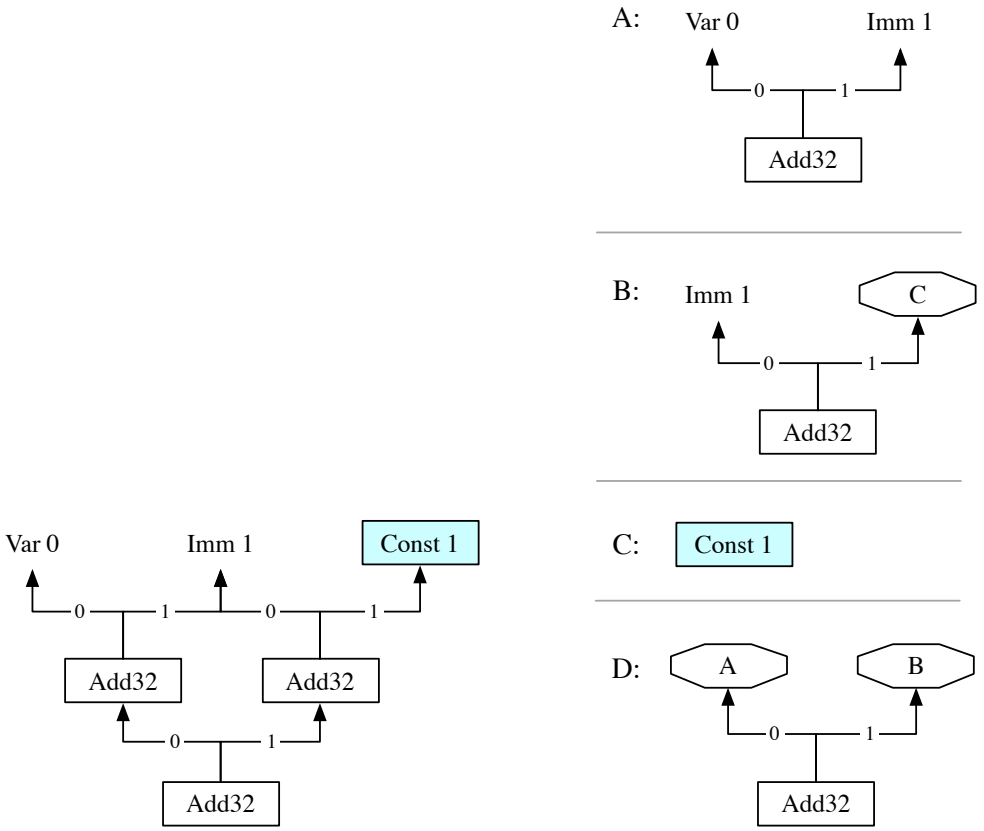


Figure 4.13: A pattern (left) and its decomposition into three flat patterns (right).

```
  }

  if (label(node) == Add
      && is_immediate(get_irn_n(node, 0))
      && get_irn_n_edges(node) == 1
      && hasAnnotation(get_irn_n(node, 1), C)) {
    addAnnotation(node, B);
  }

  if (label(node) == Const 1) {
    addAnnotation(node, C);
  }

  if (label(node) == Add
      && ...) {
    addAnnotation(node, D);
    handle_D(node);
  }
}
```

As we can see the `if` blocks are laid out linearly. This is wasteful because there are some overlapping predicates among the different if blocks. A better layout would be the following.

```
void annotate(ir_node *node) {
  if (label(node) == Add) {
    if (is_immediate(get_irn_n(node, 0))
        && get_irn_n_edges(node) == 1) {
      addAnnotation(node, A);

      if (hasAnnotation(get_irn_n(node, 1), C)) {
        addAnnotation(node, B);
      }
    }
    if (...) {
      addAnnotation(node, D);
      handle_D(node);
    }
  }

  if (label(node) == Const 1) {
    addAnnotation(node, C);
  }
}
```

Compiler optimization passes might find such cases to a certain degree but to make sure we consistently get the best result we will lay out the conditions ourselves.

All predicates divide the set of flat patterns into two parts: the flat patterns

that need this predicate to get matched and the flat patterns that do not need this predicate. Some predicates apply to more matching checks than others. In the example above, the `label(node)== Add` predicate is the one that applies to most checks. We call this the *most discriminating predicate*. We always want to check for the most discriminating predicate first. We rewrite the `genAnnotate` function to follow this idea.

```
genAnnotate :: {FlatPattern} -> String
genAnnotate fps
 = let fpPreds = fpPredicatesMap fps in
     "void annotate(ir_node *node) {\n" ++
     annotateInner {} fpPreds ++
     "}"

annotateInner :: Map FlatPattern {Predicate} -> {Predicate} -> {
   FlatPattern} -> String
annotateInner m ps fps = case mostDiscriminating ps m fps of
  Nothing -> annotateInnerLinear fps
  Just (pred, l, r) -> "if ($(pred)) {"
                    ++ annotateInner m (pred:ps) l
                    ++ "}"
                    ++ annotateInner m (pred:ps) r
```

The new `annotateInner` function takes the predicates map, a set of predicates, and a set of flat patterns. The second argument are all predicates that we already checked and that apply to all flat patterns in the third argument. Initially the first set is empty and the second holds all flat patterns.

The `annotateInner` function first works out the most discriminating predicate out of the remaining predicates (those still in the predicates map `m` but not in `ps`). If such a predicate does not exist because all remaining predicates either apply to all or none of the remaining flat patterns, we lay out the remaining flat patterns linearly via the `annotateInnerLinear` function. Otherwise, if a most discriminating predicate does exist, we get a partition of the set of flat patterns. The variable `l` holds the flat patterns the predicate `pred` applies to. We lay out the check for `pred` accordingly. Inside the body of the if, we recursively call `annotateInner` but only with the remaining flat patterns. We also add `pred` to the set of predicates we already checked for. Outside the if we do the same with the flat patterns `r`.

This approach benefits from our decision to collect all the predicates in a separate step. This way, we do not care where the predicates come from or what their specific semantics are. We can treat them very abstractly.

5 Evaluation

There are three categories in which we evaluate our implementation. The first category describes the performance of the generator itself (Section 5.1), the second category describes the performance of the generated compiler (Section 5.2), and the third category describes the performance of the compiled programs (Section 5.3).

All tests are performed on an Intel Core i7-6700 3.4 GHz with 64GB of RAM. We test with a rule set of 60.623 rules.

5.1 Generation time performance

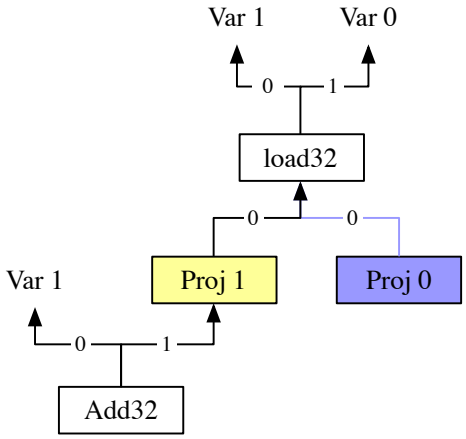
We measure the wall clock time it takes our generator to produce an instruction selector from the complete rule set of 60.623 patterns. On our test machine this process takes 8m 11s. In an industry scenario, this process would only happen when the input rule set changed or when the generator itself was updated. The generation time is therefore of minor significance. The up front workload is dominated by the rule set synthesis and the compilation of the generated C output files. These processes both take time in the order of hours.

5.2 Compile time performance

We now measure the quality and performance of our produced compiler in two ways. First we determine if our generated compiler is able to find all patterns that we gave the generator as input in the rule set. We use the prior work by Buchwald et al. [1] to generate one C file per pattern. The C code mirrors the exact structure of the pattern. Figure 5.1 shows an example of a pattern and the corresponding C code. The instruction selector now has to find the pattern match for each C file. We check if the match was found by analyzing the produced assembly code.

This test relies on our input set being prioritized correctly. Often large patterns contain smaller patterns as subgraphs. If we assign a higher priority to the smaller pattern, our instruction selector will always match the smaller pattern. There is no way for us to generate a test for the larger pattern. Unfortunately our input rule set was not prioritized correctly at the time of this test. Still, in all failing cases we were able to manually tweak the input set priorities and get the correct behaviour.

Our second test measures the performance of the generated compilers. We run the SPEC CINT2000 benchmarks with the handwritten instruction selector and with our generated instruction selector and measure the time spent (1) in the code generation




```
uintptr_t ia32_Add_base(char *var0, uintptr_t var2)
{
      asm volatile("" : : "r" (var0), "r" (var2) : "memory");
      uintptr_t Var94724482045424 = var2;
      char *Var94724483990272 = var0;
      uintptr_t load3294724480884720_1 = *(uintptr_t *)Var94724483990272;
      uintptr_t Proj94724483854112 = load3294724480884720_1;
      uintptr_t Add3294724482208688 = Var94724482045424 + Proj94724483854112;
      return Add3294724482208688;
}
```

Figure 5.1: A ia32_Add_base pattern (above) and its C representation (below). The C code mirrors the exact structure of the pattern.

phase and (2) overall. We then calculate the average slowdown factor, the ratio of compilation time with our generated instruction selector divided by compilation time with the handwritten instruction selector, grouped by benchmark. Table 5.2 shows the results. The average slowdown is 310.45 with very high standard deviations. Our instruction selector is therefore 310 times slower on average than the handwritten version. In comparison the simple linear generated instruction selector is 1000 times slower than the handwritten version. Overall, cparser with our instruction selector is 10 times slower than cparser with the handwritten instruction selector.

	Codegen only		Overall	
Benchmark	Avg ratio	σ	Avg ratio	σ
164.gzip	277.792	76.556	8.1299	3.7513
175.vpr	271.307	103.851	7.7198	2.1757
176.gcc	303.730	92.822	9.1820	3.6161
181.mcf	362.365	72.470	12.9803	3.4852
186.crafty	323.130	126.645	9.7552	2.7449
197.parser	304.647	85.533	10.1134	3.1775
253.perlbmk	316.537	106.287	9.7404	3.3420
254.gap	323.412	88.503	9.3625	2.4376
255.vortex	311.253	69.637	13.4666	3.8990
256.bzip2	286.856	87.137	7.7579	0.8208
300.twolf	341.991	110.223	9.7199	3.0365
Total	310.450	93.321	10.2029	3.6650

Figure 5.2: Result of the compilation performance tests. We measured the code generation time for each compilation unit with the handwritten instruction selector and our generated instruction selector. We then took the ratio of these two measurements and calculated average and standard deviation values for each benchmark. We compare the measured times for the code generation phase only (Codegen only) and for the entire compiler (Overall). As we can see, the standard deviations are quite high. The overall average ratio is 310.45 for codegen only and 10.2 overall, which means that our instruction selector is about 310 times slower than the handwritten instruction selector and this leads to the entire compilation process taking 10 times as long.

5.3 Quality of the produced code

We finally measure the quality of the code produced by the generated compilers. This comes down to running the SPEC 2000 benchmark suite with our generated

compilers. Since our input rule set only contains integer rules, we disable the floating point arithmetic benchmarks. We used an iteration count of 20. Table 5.3 shows the results of this test. As expected, the compilers we generate come very close to the performance of the reference compiler. In theory, our instruction selector is capable of finding more patterns, so we would expect our solution to produce better code. In practice, however, this does not hold true. Handwritten implementations still have the edge over our generated instruction selectors.

Benchmark	Autotransform	σ_{Auto}	Handwritten	σ_{Hand}	Auto/Hand
164.gzip	56.23 s	0.46 s	55.45 s	0.02 s	101.42%
175.vpr	43.13 s	0.21 s	41.38 s	0.21 s	104.25%
176.gcc	19.34 s	0.02 s	18.38 s	0.03 s	105.20%
181.mcf	20.69 s	0.09 s	20.14 s	0.10 s	102.74%
186.crafty	24.90 s	0.08 s	24.81 s	0.04 s	100.33%
197.parser	54.29 s	0.07 s	52.18 s	0.07 s	104.05%
253.perlbmk	40.68 s	0.03 s	40.93 s	0.03 s	99.39%
254.gap	23.14 s	0.07 s	22.19 s	0.14 s	104.30%
255.vortex	40.29 s	0.16 s	36.89 s	0.17 s	109.23%
256.bzip2	43.21 s	0.14 s	42.64 s	0.12 s	101.35%
300.twolf	59.43 s	0.21 s	58.10 s	0.18 s	102.29%
					Mean: 103.11%

Figure 5.3: Result of the runtime performance tests. We ran each benchmark once compiled with our generated instruction selector (Autotransform) and the handwritten instruction selector (Handwritten). The ratio of the two values in the last column is a measure of the quality of our approach.

6 Future work

In this chapter we provide three angles of attack to improve the performance of our generated compilers.

6.1 Better bulletin boards

In Section 4.7 we introduced a new technique to handle multi-rooted patterns in our bottom-up annotation algorithm. We defined special nodes that act as hubs during the pattern matching phase. Hubs hold bulletin boards, which are information exchanges for communication between the different roots of a multi-rooted pattern. The roots of each pattern communicate through a distinct bulletin board separate from the bulletin boards of all other patterns. This turns out to be a bit of a waste of space and time in some cases. Consider the two patterns in Figure 6.1.

We can see that the downwards view of the **Proj 0** roots is the same in both patterns (subgraphs I and II). In a program graph at a node where the subgraph I matches, II also matches and vice versa. We would therefore put two distinct bulletin boards at the hub at the **load32** node. One such bulletin board would be sufficient, though, because both hold the same information anyway. Address mode patterns like these are very common in our IA-32 pattern set. Limiting this waste of space and effort might prove worthwile.

One idea to mitigate the problem is to make our virtual bulletin boards act more like bulletin boards in the real world. Let us assume for now that all of the multi-rooted patterns have exactly two roots. Our goal is to aquaint these pairs of complementary roots with one another. We can designate one root as an *offer poster* and the other one as a *search request poster*. An offer only consists of a description of the structure of the offer poster's downwards view in the pattern. For the two patterns in Figure 6.1 we would ideally designate the **Proj 0** roots as the offer posters and the **Add32** and **Sub32** roots as searchers.

Consider the program graph excerpt shown in Figure 6.2. Assume we arrive at node v first. We know that the downwards view of the **Proj 0** roots matches at v. Let S denote the structure of this downwards view. We now note down the information that node v provides structure S at the **load32** hub. Later we arrive at node w. At the hub at the **load32** node we post a search request for the structure S because the **Sub32** root wants to find a match of the *ia32_Sub_base* pattern and S is the structure of its complementary root. We find out about node v this way and can finalize the pattern match.

The order of arrival does not matter. If we arrive at w first, we post the same

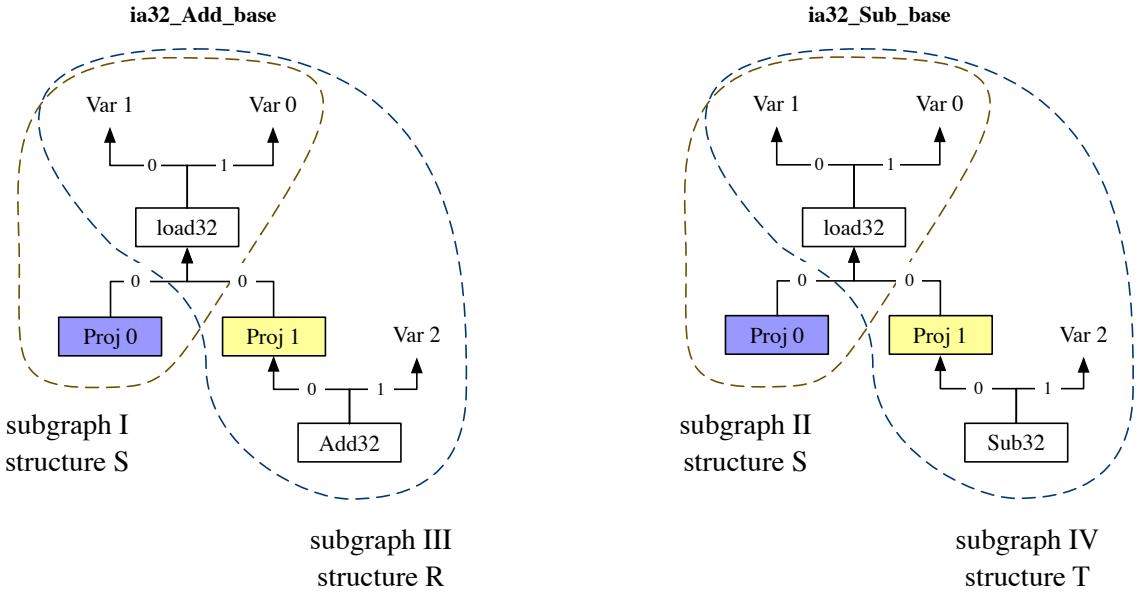


Figure 6.1: Two similar multi-rooted patterns. The downwards view of both Proj 0 roots is the same.

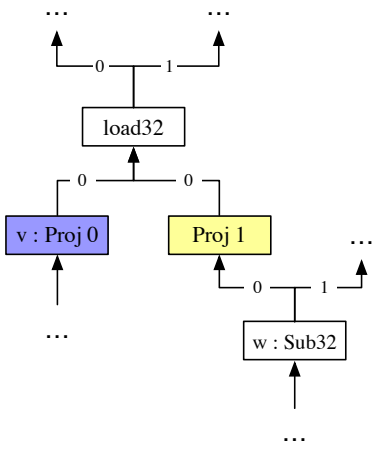


Figure 6.2: An excerpt of a program graph. The *ia32_Sub_base* pattern matches.

search request for S at the hub. The hub stores this request alongside a callback function provided by the *Sub*32 root. The search request informally says "I am interested in S. You can call me at ..." Later when we post an offer for S, the hub will recognize that the **Sub32** node was looking for such an offer and calls the corresponding callback function.

With this scheme each hub holds exactly one bulletin board. The bulletin boards store search requests and offers in a hash table with the structures (S above) as keys for fast lookup.

There is one question left to answer with this scheme, however. In the example above we designated the **Proj 0** nodes as offer posters and the **Add32** and **Sub32** nodes as searchers. Obviously for each pattern both roles have to be filled for the scheme to work. This still leaves us with a potentially large set of possible assignments of roles. If the **Add32** and **Sub32** nodes were posting offers instead, the **Proj 0** root in the *ia32_Add_base* pattern would have to post a search request for structure R and the **Proj 0** root in the *ia32_Sub_base* pattern would have to post a search request for structure T. The two cannot be collapsed any more. In a program graph where we find a partial match of the subgraphs I and II, we therefore have to post two search requests. We thereby essentially make the same effort as before when we had two distinct bulletin boards. Assigning roles becomes an important part of this scheme. Most assignments are not as straightforward to optimize such as in the little example above.

Solving this puzzle is a fun exercise. It might prove only sparsely profitable, however, since in our overall framework, handling bulletin boards only takes up a minor fraction of compute time.

6.2 Better matching condition layouting

In Section 4.8.2 we tried to lay out the predicates for flat pattern matches such that the resulting code is as efficient as possible. We are given a set of predicates and a map Predicate $\mapsto$ {Flat Pattern}, which denotes the set of flat patterns that a given predicate applies to. The best layout we came up with was a tree structure. In practice, we handled the root label predicates slightly differently, however. Instead of checking for each label in a spearate `if`-block and falling through if the label does not match, we used a single `switch` statement. This has two benefits. First, a `switch` can be translated to an efficient jump table by the compiler. Second, if one `case` block matches, we know that none of the other `case` blocks can match, because a flat pattern can only ever have a single unique root label. We can therefore `break` out of the switch statement immediately without falling through to all the other `case` blocks.

We are allowed to take these shortcuts because there is an exclusion relation between the different root label matching predicates:

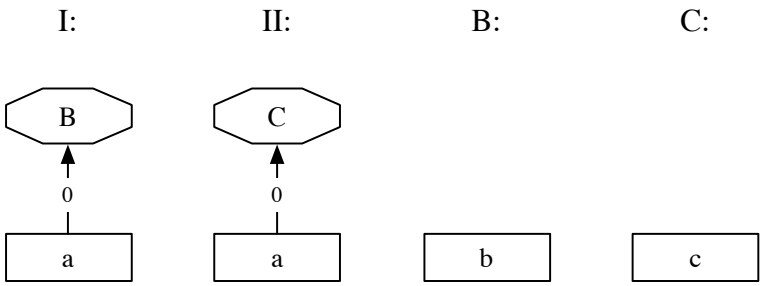


Figure 6.3: A set of flat patterns with exclusions.

$$
\begin{aligned}
&\text{label(node)} == \text{Add32} \implies \neg\text{label(node)} == \text{Sub32} \\
\wedge\ &\text{label(node)} == \text{Add32} \implies \neg\text{label(node)} == \text{Or32} \\
\wedge\ &\text{label(node)} == \text{Add32} \implies \neg\text{label(node)} == \text{Proj} \\
\wedge\ &\ldots
\end{aligned}
$$

We can find such exclusions between other predicates as well. Consider the flat pattern set in Figure 6.3. With the layouting method developped so far we would generate the following matching code for these flat patterns:

```
switch(label(node)) {
  case a:
    if (contains_pattern_id(get_irn_n(node, 0), B)) {
      add_pattern(node, I)
    }
    if (contains_pattern_id(get_irn_n(node, 0), C)) {
      add_pattern(node, II)
    }
    break;
  case b:
    add_pattern(node, B)
    break;
  case c:
    add_pattern(node, C)
    break;
}
```

Inside the `case a` block, we check for the flat pattern matches B and C. If the first check succeeds, we annotate accordingly and fall through to the second check. Falling-through is unnecessary here, however. We know that if B was annotated at a node, C can never be annotated at the same node, because each node only has one label and the root labels of the flat patterns B and C are in conflict.

$$\begin{aligned} & \text{contains\_pattern\_id}(\text{get\_irn\_n}(\text{node}, 0), B) \\ \Longrightarrow\ & \neg\text{contains\_pattern\_id}(\text{get\_irn\_n}(\text{node}, 0), C) \end{aligned}$$

There are a lot of these exclusions to be found in our flat pattern set and this layouted code has a big impact on performance. Investigating further into this topic might therefore prove fruitful.

6.3 Parameterized labels

Assume you have a large pattern set of size n. Decomposition yields a large flat pattern set of size m. Before we started out, we assumed that when we add another pattern to the large pattern set, most of the corresponding flat patterns will already occur in the large flat pattern set. Therefore, the relationship between n and m should be non-linear, because m only counts unique flat patterns. In reality, $n \approx 3m$ on average regardless of the sizes.

Our expectations were based on an implicit assumption that turned out to be wrong. We assumed patterns to be similar at the leaves and in the trunk and dissimilar at the roots. Figure 6.4 shows how two patterns that only differ at the root get decomposed. As we can see the two flat pattern sets overlap substantially, which means the combined flat pattern set is small. Our solution works well with such patterns that are similar near the leaves but dissimilar near the roots.

Figure 6.5 paints a different picture. The two patterns shown in this figure are similar at the roots and in the trunk and dissimilar at just one leaf. Because of the way that we use flat pattern ids and references, this small change propagates through a large part of the pattern. This leads to a lot of new and unique flat patterns. Our solution does not work well with patterns that are similar near the roots but dissimilar near the leaves.

It turns out that in our IA-32 pattern set these small dissimilarities are evenly distributed over all levels of depth in the patterns. Patterns that are similar near the roots but dissimilar near the leaves are therefore quite common, especially because we treat **Const 0** as a totally distinct label from **Const 1** and **Const 2**, etc. One way to mitigate this problem is to parameterize the patterns and therefore also the flat patterns. We essentially extract the dissimilarity at some leaves and put it in the roots. When we check the roots for a match, we now have to additionally check if the **Const** node further down in the program graph is a **Const 1** or a **Const -1** node. This additional check involves only little overhead compared to the complexity we save by making all other flat patterns collapse. Figure 6.6 shows an example of the idea.

Const nodes lend themselves naturally to this idea, but we might be able to parameterize other node types or even entire subgraphs as well. This idea needs further investigation. We think that out of the three ideas that we present in

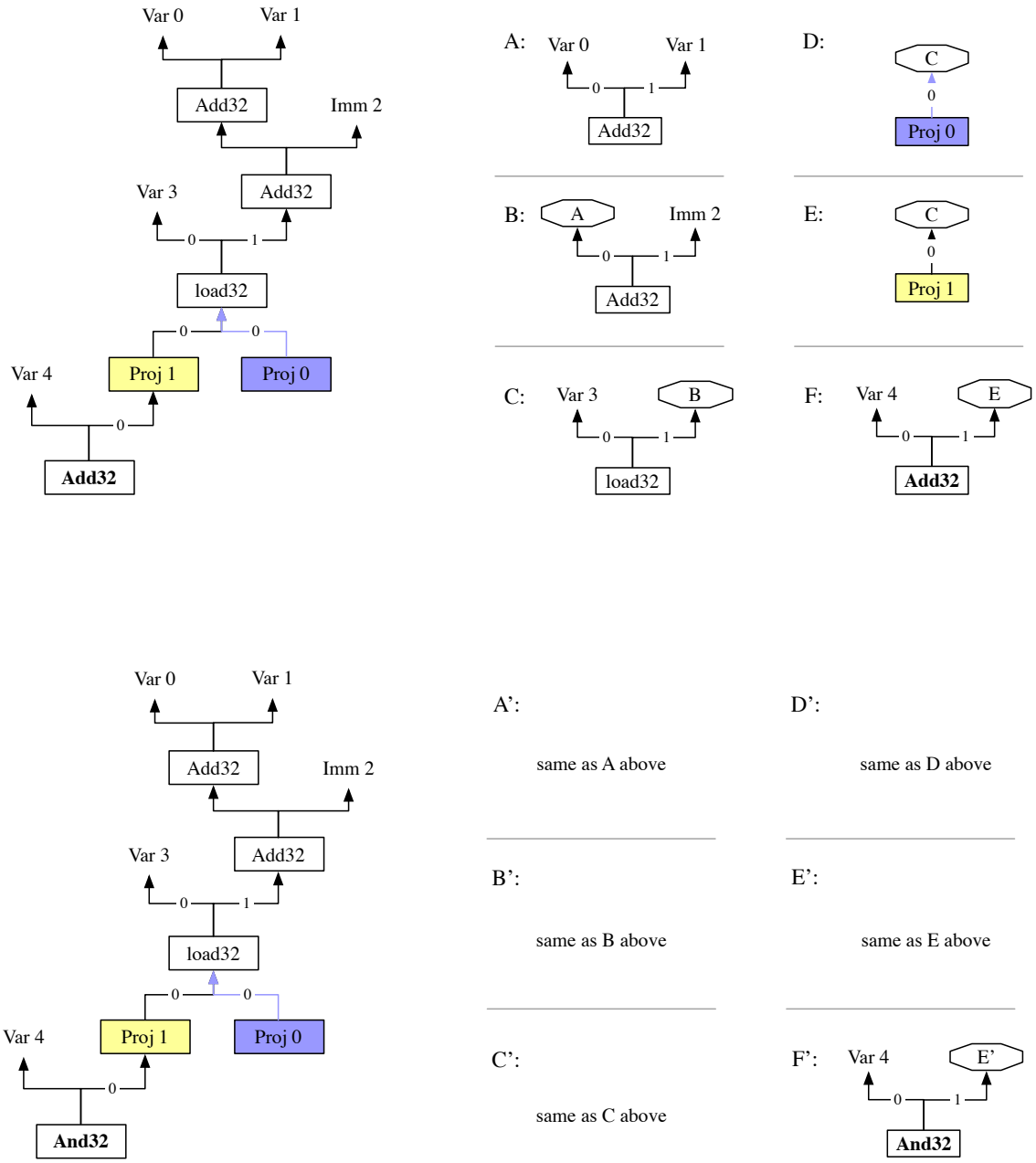


Figure 6.4: Two patterns (left) and their corresponding flat pattern sets after decomposition (right). The two patterns only differ in one root label: **Add32** vs. **And32**. A lot of flat patterns therefore overlap. With the addition of the pattern below, we add only one flat pattern to the total set of unique flat patterns.

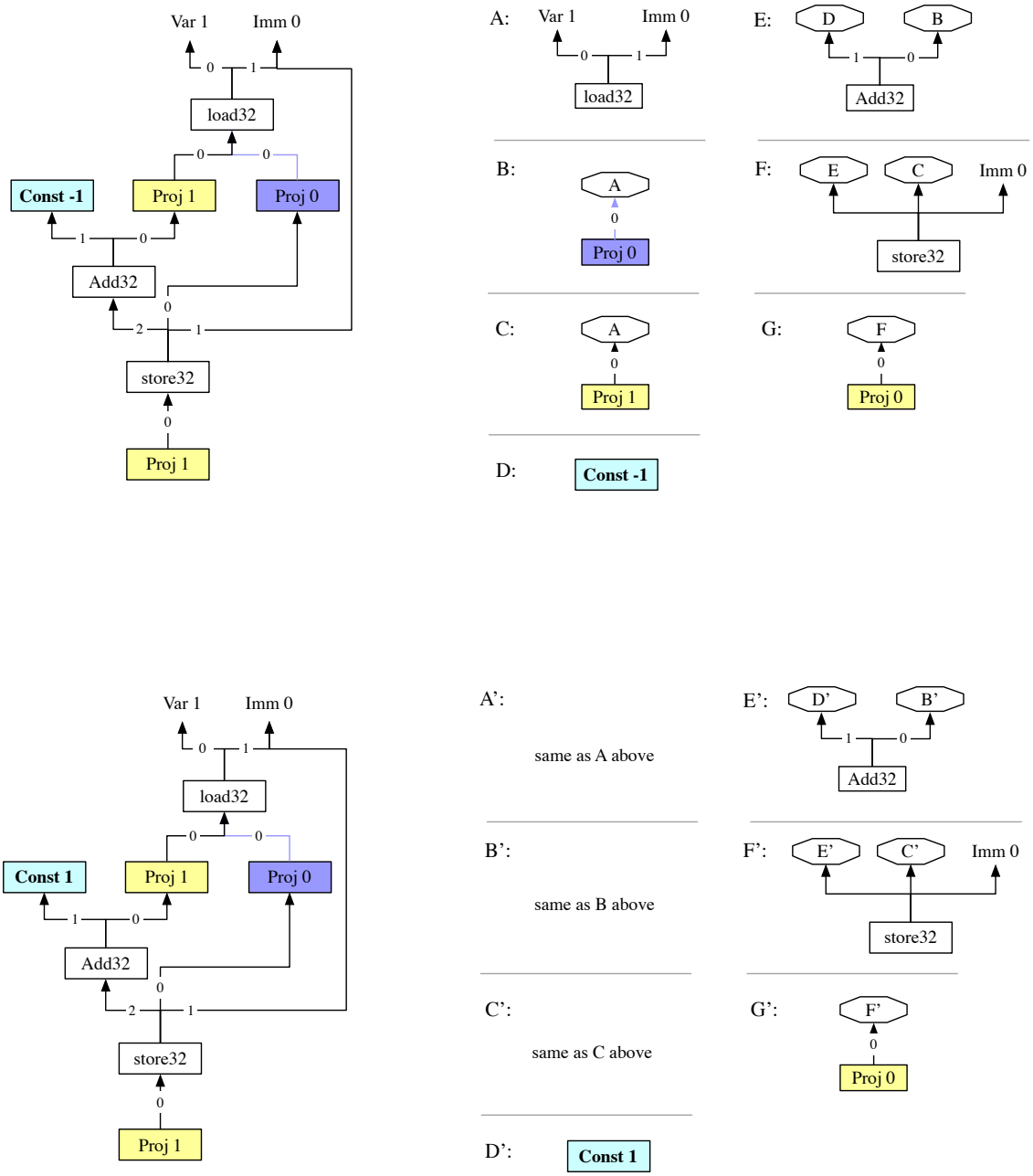


Figure 6.5: Two patterns (left) and their corresponding flat pattern sets after decomposition (right). The two patterns only differ in one leaf: **Const 1** vs. **Const -1**.

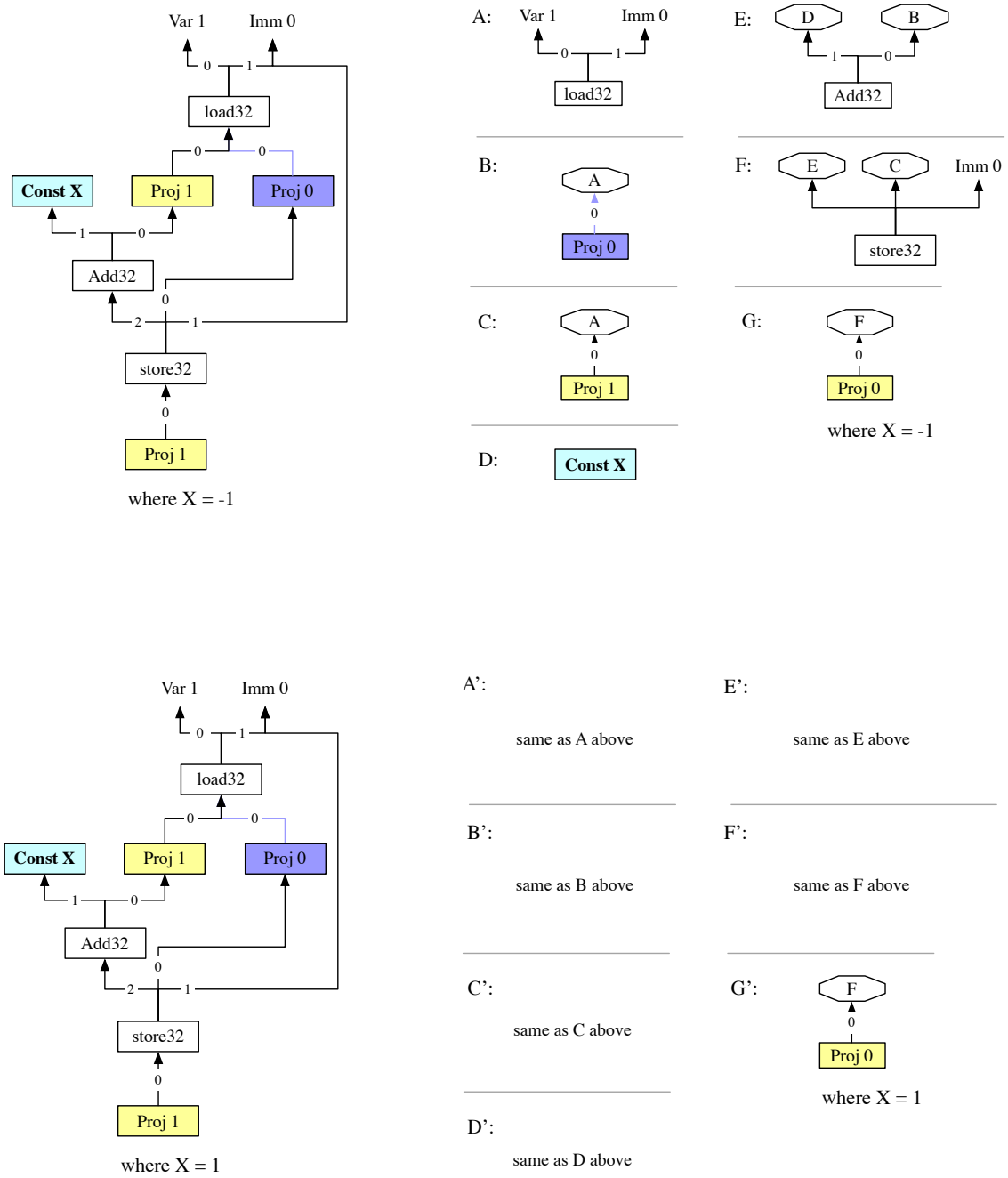


Figure 6.6: Two patterns (left) and their corresponding flat pattern sets after decomposition (right). The two patterns only differ in the **Const** leaf. We parameterized this leaf (**Const X**). The parameter has to be additionally checked during pattern matching.

this chapter, (flat) pattern parameterization has the greatest chance of having a meaningful impact on performance.

7 Conclusion

Our solution turned out to be a lot slower than we expected. During development we were testing the solution with debug build settings. With these settings our generated instruction selector was only about 15 times slower than the handwritten solution. The handwritten solution benefits heavily from the optimized build settings, but our solution does not, so the factor climbs up to 300 when compiled with optimizations. It might be fruitful to start investigation by analysing why our implementation is comparatively fast when compiled without optimizations but comparatively slow when compiled with optimizations. We did not knowingly implement optimizations by hand.

One explanation for this discrepancy might be that our algorithm is fast in theory but suffers from poor cache use in practice. We often take huge jumps in our algorithm, rendering the code cache fairly useless. Even the linear solution fares better in this regard. We are uncertain if this problem can even be entirely solved. We still have to wade through more than 60.000 patterns for each node in the program graph. Making this process perfectly cache-friendly might be impossible. Still, blaming poor cache use is only a hypothesis for now. Further investigation is necessary to confirm or deny this hypothesis.

Another explanation for our poor performance is that our algorithm does not fit the structure of our pattern set very well. As we alluded to in Chapter 6, our algorithm is implicitely based on the assumption that patterns are similar in the trunk and towards the leaves with differences in the roots. This turned out not to be true. Our decomposition algorithm therefore produces a lot of similar flat patterns that make the bottom-up annotation algorithm perform more work than necessary. We put high hopes in an algorithm that solves this problem, for example by transforming some differences at the leaves into differences at the roots. We introduce an idea for such an improvement in Section 6.3. Another reason for the bad performance of our solution is that we have to refrain from making any assumptions that are too restrictive. When we write an instruction selector for the IA-32 architecture by hand, we see that addressing modes play an important role in the instruction set. We can optimize our handwritten solution to cope well with these special cases. In constrast, our instruction selection generator cannot make any of these narrow assumptions, because we want our generator to work for any conceivable architecture.

To our own defence we have to mention that our generated instruction selector is capable of finding a lot more patterns than the handwritten instruction selector. The comparison is therefore not entirely fair. For a better comparison we would have to feed our generator only with those patterns that the handwritten solution is capable of matching as well. We do not expect to come close to the performance of

the handwritten instruction selector, but a lot closer than the factor of 300.

Even if it is not possible to make the algorithm significantly faster, our efforts were not in vain. The generated instruction selector is not fast relative to the handwritten solution, but it might be fast enough in absolute terms. Pattern decomposition and the bottom-up annotation algorithm can be employed to generate an instruction selector for any rule set. We did not make any assumptions about the shape and form of the patterns or the rule set in general. Especially our work on the treatment of arbitrary multi-rooted patterns might be useful independent of the underlying matching implementation.

Bibliography

[1] S. Buchwald, A. Fried, and S. Hack, "Synthesizing an instruction selection rule library from semantic specifications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), pp. 300–313, ACM, 2018.

[2] "GCC." `https://gcc.gnu.org`.

[3] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pp. 75–, IEEE Computer Society, 2004.

[4] "ISO/IEC 9899:1999, programming languages – C," standard, International Organization for Standardization, Geneva, CH, 1999.

[5] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*. No. 253669-033US, December 2009.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[7] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[8] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization*, (New York, NY, USA), pp. 20–24, ACM, 1970.

[9] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, (New York, NY, USA), pp. 194–206, ACM, 1973.

[10] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 181–210, Apr. 1991.

[11] "FIRM API documentation." `https://pp.ipd.kit.edu/firm/Documentation.html`.

[12] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, (New York, NY, USA), pp. 151–158, ACM, 1971.

[13] 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, (New York, NY, USA), pp. 31–40, ACM, 2010.

[14] "clang." `https://clang.llvm.org`.

[15] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[16] "ISO/IEC 14977:1996, Extended BNF," standard, International Organization for Standardization, Geneva, CH, 1996.

[17] G. H. Blindell, *Instruction Selection*. Springer International Publishing, 2016.

[18] M. Elson and S. T. Rake, "Code-generation technique for large-language compilers," *IBM Systems Journal*, vol. 9, no. 3, pp. 166–188, 1970.

[19] T. R. Wilcox, "Generating machine code for high-level programming languages," 1971.

[20] A. Snyder, "A portable compiler for the language c," tech. rep., Cambridge, MA, USA, 1975.

[21] W. M. McKeeman, "Peephole optimization," *Commun. ACM*, vol. 8, pp. 443–444, July 1965.

[22] J. W. Davidson and C. W. Fraser, "Code selection through object code optimization," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 505–526, Oct. 1984.

[23] R. Cattell, "A survey and critique of some models of code generation," 01 1977.

[24] M. Ganapathi, C. N. Fischer, and J. L. Hennessy, "Retargetable compiler code generation," *ACM Comput. Surv.*, vol. 14, pp. 573–592, Dec. 1982.

[25] R. S. Glanville and S. L. Graham, "A new method for compiler code generation," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, (New York, NY, USA), pp. 231–254, ACM, 1978.

[26] C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *J. ACM*, vol. 29, pp. 68–95, Jan. 1982.

[27] E. Pelegri-Llopart and S. Graham, "Optimal code generation for expression trees: An application of burs (bottom-up rewrite systems) theory," p. 18, 01 1988.

[28] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *J. ACM*, vol. 24, pp. 146–160, Jan. 1977.

[29] M. A. Ertl, "Optimal code selection in dags," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, (New York, NY, USA), pp. 242–249, ACM, 1999.

[30] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[31] D. R. Koes and S. C. Goldstein, "Near-optimal instruction selection on dags," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, (New York, NY, USA), pp. 45–54, ACM, 2008.

[32] E. Eckstein, O. König, and B. Scholz, "Code instruction selection based on ssa-graphs," in *Software and Compilers for Embedded Systems* (A. Krall, ed.), (Berlin, Heidelberg), pp. 49–65, Springer Berlin Heidelberg, 2003.

[33] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec, "Generalized instruction selection using ssa-graphs," *SIGPLAN Not.*, vol. 43, pp. 31–40, June 2008.

[34] F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, pp. 10–19, Apr. 1987.

[35] J. H. Reppy, *Concurrent Programming in ML*. New York, NY, USA: Cambridge University Press, 1st ed., 2007.

Erklärung

Hiermit erkläre ich, Markus Schlegel, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift