Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

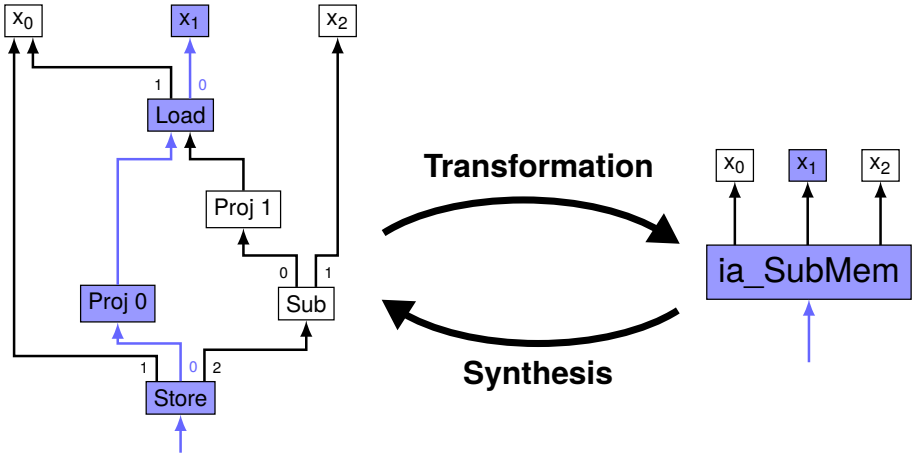
Lehrstuhl Prof. Dr.-Ing. Snelting

Synthesising Instruction Selection

Masterarbeit von

Andreas Fried

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 17. Februar 2016 – 8. August 2016

Zusammenfassung
Abstract

In dieser Arbeit wird ein Tool vorgestellt, das die Befehlsauswahlphase eines IR-basierten Compilers automatisch generieren kann. Aus formalen Spezifikationen der Zwischensprache des Compilers und der Ziel-Maschinensprache findet es mittels eines SMT-Solvers zu jeder Maschineninstruktion einen äquivalenten DAG in der Zwischensprache. In der Spezifikation kann insbesondere auch Speicherzugriff effizient dargestellt werden. Aus den so gewonnenen Zuordnungen wird dann eine Befehlsauswahlfunktion generiert, die in Firm integriert werden kann. Sie ist wie die bestehende Firm-Befehlsauswahl ein „greedy DAG-matcher“.

Die Synthese von 58 Maschinenbefehlen aus einer Zwischensprache mit 20 Befehlen dauert 2 Stunden. Die synthetisierte Befehlsauswahlfunktion kann die Mehrzahl der Befehle aus dem SPEC-CINT2000-Benchmark in Maschinensprache übersetzen.

We present a tool that can automatically synthesise the instruction selection phase of an IR-based compiler. Using formal specifications of intermediate representation and machine language instructions, we match machine language instructions to DAGs of IR instructions using an SMT solver. Our specification model includes an efficient representation of memory access. From the associations between IR and machine language, we generate a greedy DAG-matching instruction selector that can integrate into Firm.

Our tool can synthesise an instruction selector for 58 machine instructions from a set of 20 IR instructions in 2 hours. The synthesised instruction selector is able to handle the majority of instructions from the SPEC CINT2000 benchmark.

Contents

1. Introduction

The point of modern programming is not to convey orders to a computer, but ideas to a fellow human. For this reason, computer science has developed high-level programming languages, which are easier for humans to write and to understand.

Of course, a computer cannot execute a program in a high-level language without help. A translation program called a *compiler* turns the high-level program into a machine code program, which the computer can execute. In addition, the compiler should automatically improve (*optimise*) the program, so that the developer may concentrate on writing clear and understandable code, but still achieves good performance.

We can see from these requirements that developing a good compiler is a substantial undertaking. The major compilers GCC and LLVM each have hundreds of thousands of lines of code, and have been in development for more than 10 years. In fact, most of the code and development effort went into the optimisations, which are in principle the same in every compiler. Therefore, modern compilers are designed in such a way that the optimisations (*middle end*) can work together with any high-level language (*front end*) and any machine language (*back end*).

The middle end uses its own *intermediate language* to represent programs independently from the front end language, and more or less independently from the back end language (GCC's intermediate language is more backend-specific than LLVM's). This means that the front end has to translate the high-level language to the intermediate language, and the back end has to translate the intermediate language to the machine language. It is the second of these translation steps that this present work deals with.

In order to use the machine as efficiently as possible, the compiler should of course make use of all its capabilities, i.e. it should be able to use the whole machine language. This requires a lot of work, because today's machines (first and foremost Intel's) have very large *instruction sets*, and frequently extend them. A good compiler needs to keep up with this development.

To add a new machine instruction to a compiler, a developer has to do the following: Translate the new instruction to intermediate language, check if the translation overlaps with existing ones, write code to find the translation in an intermediate language program, and finally write code to generate an instance of the new instruction where its translation is found. With our work, we aim to relieve compiler developers of this tedious task.

In our approach, the developer just has to specify the behaviour (*semantics*) of each instruction in the intermediate and machine language in an abstract way. From these descriptions, we automatically generate the translation step from intermediate

language to machine language (called *instruction selection*). This works even when the same behaviour is specified in two different ways. For example, it is not obvious that the expressions $\sim(-x)$[1] and $x - 1$ are equivalent, but our program will match instructions even in these cases.

In our work, we use an *SMT solver*. This is a software tool that can automatically find the solutions to certain simple kinds of mathematical and logical problems. For example, we may ask it to find integers $x > 0$, $y > 0$, and $z > 5$ such that $x^2 + y^2 = z^2$, and the SMT solver will find one solution (it might be $x = 16$, $y = 12$, $z = 20$).

However, even with such a powerful tool at our disposal, there remains work to be done, which we present in this thesis. First, we look at some prerequisites at more detail in Chapter 2. Chapter 3 is the main part of our work; we develop our algorithm to synthesise instruction selection. In Chapter 4, we then evaluate the performance of our algorithm from different perspectives. Finally, we discuss our results and take a broader outlook in Chapter 5.

[1] "$\sim$" refers to the function that flips all bits in its argument

2. Basics

2.1. Compiler Design

At the most basic level, a compiler's task is to translate a program written in a source language to a semantically equivalent program in a target language. Besides that, the output program should be optimised, for example to run as quickly as possible, or to be as small as possible. The compiler therefore applies *optimisation passes* to the program when translating it, for example removing redundant code or simplifying computations.

A compiler usually includes a large number of optimisation algorithms, each of which requires considerable development effort. Compiler developers therefore want to share optimisations between compilers for different languages. Moreover, neither source nor target language are usually designed to be easily manipulated programmatically, so that a more convenient representation is required anyway.

For these reasons, most compilers are designed around an *intermediate representation* (IR), on which most parts of the compiler operate. We will discuss IR design in more detail in Section 2.1.1. A compiler having an IR is usually divided into three components, which process the source program in turn:

- First, the source-specific *front end* parses the source language program and checks it for errors. If the program is correct, the front end converts it to the compiler's IR.
- Next, the *middle end* applies the optimisations to the IR program in order to produce an efficient output program.
- The optimised IR program is finally passed to the target-specific *back end* of the compiler, where the IR is transformed into the target language, and some additional optimisations may be performed.

By way of the middle end, every front end can be combined with every back end. However, the compiler now requires two translation steps, from source language to IR, and from IR to target language. The latter transformation is the *instruction selection*, which we describe in Section 2.1.3.

From here on, we shall assume the most common case for compiler construction, namely that the source language is a high-level imperative programming language, and that the target language is an assembly language for a particular processor.

```
#include <stdlib.h>

void f(int a, int *b)
{
        int i = 2 * a;
        if (b != NULL) {
                *b -= i;
        }
}
```

(a) C source program

```
f:
        movl    8(%esp), %eax
        testl   %eax, %eax
        je      .LBB0_2
        movl    4(%esp), %ecx
        addl    %ecx, %ecx
        subl    %ecx, (%eax)
.LBB0_2:
        retl
```

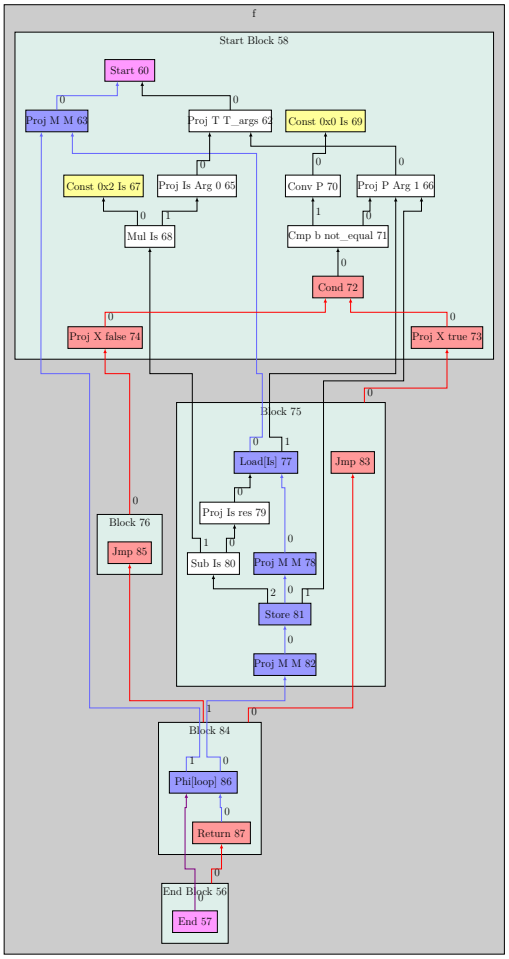
(b) x86 assembly produced by clang with optimisations enabled

```
define void @f(i32 %a, i32* %b) #0 {
  %1 = mul nsw i32 2, %a
  %2 = icmp ne i32* %b, null
  br i1 %2, label %3, label %6

; <label>:3
  %4 = load i32, i32* %b, align 4
  %5 = sub nsw i32 %4, %1
  store i32 %5, i32* %b, align 4
  br label %6

; <label>:6
  ret void
}
```

(c) LLVM bitcode after load-store optimisation



(d) FIRM graph before optimisation. See page 76 for a larger version.

Figure 2.1.: Comparison of high-level, assembler and intermediate languages. Both high-level and assembler language are more concise, but IRs have less complexity.

2.1.1. Intermediate Representations

We have already presented the reasons for using an IR. In order to motivate the design choices involved in an IR, we will first look at the typical structure of source and target languages (i.e. high-level and assembly languages), and see why they are unsuitable as IRs.

Throughout this section, we will refer to the example programs in Figure 2.1. Figure 2.3c shows a sample C program, the others show its translation into x86 assembly (Figure 2.1b), LLVM bitcode as an example for an IR (Figure 2.1c), and Firm (Figure 2.1d). We will discuss Firm in more detail in Section 2.1.2.

Programs in high-level languages are composed of nested statements and expressions. They are internally represented as an abstract syntax tree (AST), annotated with types and symbol resolutions. The AST can have tens of different node types, each with a different internal structure. For example, the grammar for C99 has 21 non-terminal symbols for expressions, 35 for declarations and 9 for statements. [1, Annex A]. This makes it inconvenient for the compiler to analyse and manipulate the AST. In addition, the language might have some abstract concepts, which are present in the AST, but hard for the optimisation algorithms to deal with.

On the other hand, assembly language programs have a simpler structure; they are lists of atomic[1] instructions. However, matters are complicated by hardware limitations: There are only a fixed number of registers available, and instructions might have implicit side effects (e.g. modifying flags). In addition, CISC processors often have instructions that combine several simpler instructions into one. Thus, the instruction set can become quite redundant. Again, such a language is not suited for analysis and manipulation by a compiler.

Having these problems in mind, we can now draw up the criteria for IR design.

Simple structure The structure of the IR should be simple and uniform, so that it may be easily handled by the compiler.

Small size There should be a small number of instructions with simple semantics in the IR to ease analyses in the middle end. However, it should be reasonably easy for the front end to produce IR from the source language.

Regularity IR instructions should have as few special cases as possible, and should not have implicit side effects.

Abstraction The IR should abstract from details of both the hardware and the source language.

Following these guidelines, IRs traditionally have the same structure as machine languages: They consist of sequences of atomic instructions. An IR's instructions are mostly simple: It usually contains instructions for every basic arithmetical and logical operation, and separate instructions for memory access.

[1] i.e. "indivisible", not "suitable for synchronisation"

On the other hand, it also contains some high-level instructions, which can be equivalent to a long sequence of machine instructions. For example, a function call is often represented as a single IR instruction in order to abstract from the particular hardware mechanisms in use.

Another important point of difference between IRs and machine languages is the treatment of registers. A processor has a limited register set, which the compiler's back end must carefully manage. This *register allocation* is a complex problem, and the middle end should not have to deal with it.

SSA form

Modern IRs use *static single assignment form* (SSA form) [2] to represent values that the high-level language would keep in local variables, and the machine language would keep in registers. In SSA form, each IR instruction *defines* some SSA values and *uses* SSA values previously defined. To use an SSA value, an instruction refers to the instruction that defined it earlier. Since SSA values are immutable, each assignment to a variable creates a new SSA value. The variable is therefore represented by a sequence of SSA values, with one SSA value for each assignment to it.

SSA form requires some overhead to deal with control flow. First, the program is divided into *basic blocks*. Simply put, a basic block is the sequence of instructions from one jump or jump target to the next, i.e. a set of instructions that are always executed together.

If a basic block B has multiple *predecessors*, i.e. basic blocks that jump to it, some values may depend on which predecessor actually executed before B. To express this dependency, SSA form uses *ϕ-functions* at the start of the basic block. The ϕ-function's takes the values from the predecessors as arguments and returns the value from the predecessor that actually executed. See Figure 2.2 for an example, whereby the value of `x3` depends on whether block A or block B executed previously.

In spite of this overhead, SSA form is easily constructible [3], and greatly simplifies later analyses, because there is no need to track the value of local variables.

2.1.2. The Firm IR

Even though IRs traditionally represent programs as a sequence of instructions, such total ordering is unnecessary. Instructions that do not use each other's results or have side effects can be reordered without changing the program's semantics. If the instructions are not fully ordered, the compiler can manipulate the program more easily.

Firm implements the idea of partially ordering instructions by representing programs as *dependency graphs*. In a dependency graph, each instruction becomes a node, and each dependency between instructions becomes an edge from the dependent instruction to the instruction it depends on.

```
if (condition())
/* block A */ {
  x = 1;
}
else
/* block B */ {
  x = 2;
}
return x;
```

(a) C program

```
if (condition())
/* block A */ {
  x1 = 1;
}
else
/* block B */ {
  x2 = 2;
}
x3 = φ(x1, x2);
return x3;
```

(b) Program in SSA form

Figure 2.2.: An example for the use of ϕ-functions. Part (a) shows a code snippet in C, part (b) shows the same code in SSA form. The value of x3 depends on previous execution. The ϕ-function selects the value form the predecessor block that executed.

Dependencies

Generally speaking, an instruction i depends on another instruction j, if i requires information from computed by j, and must therefore execute after j.

This includes the information, whether i should execute at all. In this case, i has a *control dependency* on j. For our work, *data dependencies* are more important though, and we shall now discuss these in more detail.

The simplest data dependency is the *define-use relationship*. The instruction j is in define-use relationship with the instruction i, if i uses a value that j has previously defined. This is the usual dependency between arithmetical instructions.

Besides that, instructions can also be data dependent through memory: If the instruction i reads a value from a memory location that j writes to (read-after-write), i and j must not be reordered to preserve the program's behaviour. The same is true for write-after-write and write-after-read dependencies.

Define-use relationships are explicit in all SSA-based IRs, but data dependencies through memory side-effects are only kept implicitly in the total order of the instructions. Firm makes all data dependencies explicit, and can therefore do away with ordering instructions totally.

A Firm instruction that has side-effects uses and defines an *M-value*, which represents the state of the outside world at a given position in the program. Thus, instructions with side-effects are ordered by a chain of M-values from one to the next.

The M-value also represents the state of memory (hence its name), and we will mostly need this aspect of it in our work below.

Ignoring loops, the dependency relation is a partial order, and the dependency graph is therefore a DAG. Only ϕ-instructions in loops may break the partial order

to use a value defined in the previous iteration. Since we will not be concerned with ϕ-instructions in our work, we may treat the dependency graphs as DAGs.

Modes

Each Firm node has a *mode*, which determines the kind of result it produces. It may also determine the kinds of arguments the node takes.

In contrast to types, which the programmer or the programming language define, modes are defined by the compiler. Firm has signed and unsigned integers 8, 16, 32, and 64 bits wide; boolean values; pointers; single- and double-precision floating point values; and special internal modes, namely M-values, values representing control flow, and tuples.

Compilers using Firm map primitive types of their source languages to the respective modes, and represent compound types as pointers.

The tuple mode T is required for nodes that define multiple values. For example, a Call node, which represents a procedure call, returns both the call's return value and the updated M-value in a tuple.

Nodes that use one of the values in a tuple do so by way of a *projection* node (called Proj). The Proj node uses the tuple value and defines the value of one of its elements.

Firm Nodes

Let us now take a closer look at how Firm implements these principles. A Firm node has the following components: Firstly, an *opcode* specifies the kind of instruction that the node represents, and the node's mode may specify a variant (e.g. floating-point or integer arithmetic).

The node's dependencies are stored in an array of pointers to the nodes it depends on. Because the dependencies of arithmetical instructions are their arguments, dependencies are also called arguments or inputs when talking about Firm nodes.

Finally, a node may have some opcode-specific *attributes*. For example, a node defining a constant value (a "Const" node) has this value as an attribute.

We shall now give an overview of the most important typed of Firm nodes. See Figure 2.3 for a display of some common nodes. The full list of available nodes can be found in the Firm API [4].

Arithmetical nodes All usual arithmetical and logical operations are available in Firm. The operations are not variadic, each instruction takes one or two arguments. Arithmetical instructions can have all integer modes, and floating-point modes where applicable. The input(s) must have the same mode as the output. A conversion instruction (Conv) is available to convert between modes.

Instructions that may fail (namely division and remainder) additionally depend on and produce an M-value to capture the side effect of them failing.

Memory-access nodes Only two kinds of nodes are relevant for our work, namely Load and Store. Both have the expected semantics. Even though Load nodes do not change the contents of memory, they still define a new M-value. This is required to encode write-after-read dependencies, and to model access to volatile memory.

Control flow FIRM models control flow by giving basic blocks (and therefore the nodes contained in them) a dependency on a control flow value produced by a jump node. There is a Jmp node for unconditional jumps and a Cond node, which takes a boolean value and produces a tuple of two control flow values, one of which is chosen by the argument.

The Cond node usually works together with a Cmp (compare) node, which compares two values and produces a boolean value as its result. The relation to use (for integral modes: $<$, $\leq$, $\neq$, $=$ $\geq$, $>$) is an attribute of the Cmp node.

2.1.3. Instruction Selection

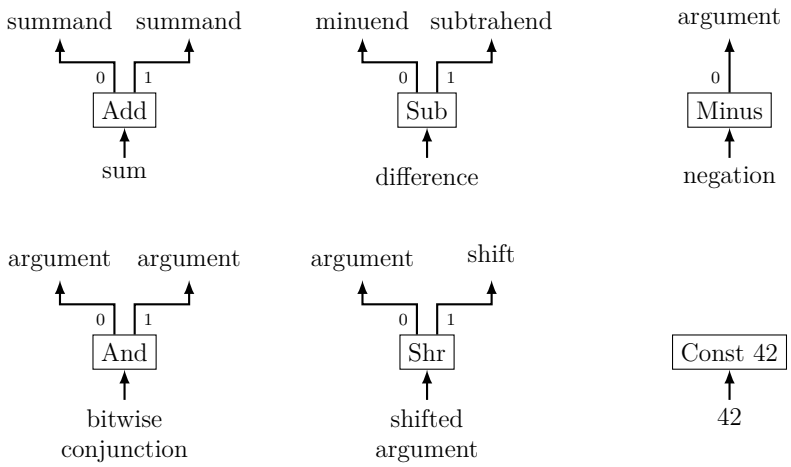
The compiler back end transforms IR to machine language in three main steps, of which instruction selection is the first. Its task is only to replace IR instructions with matching machine language instructions while keeping the program in graph form. In later phases, the instructions are fully ordered (*scheduling*), and hardware registers are assigned to the SSA values in the program (*register allocation*).

One machine instruction usually subsumes several IR instructions. Therefore, each machine instruction is associated with an IR *graph pattern* that it implements. A graph pattern is an incomplete IR graph, where some inputs are left unspecified (the pattern's arguments) and some nodes are marked as results. The unspecified inputs correspond to the inputs of the machine instruction, and the marked result nodes correspond to its results. If the machine instruction and its pattern are given the same inputs, they produce the same results. Therefore, the instruction selector can replace an occurrence of the pattern within the IR graph with the machine instruction.

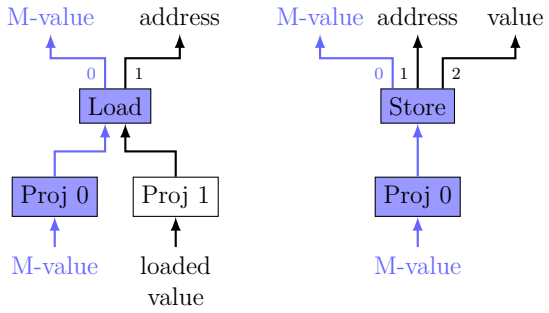
Additionally, each machine instruction is assigned a cost, for example the estimated number of cycles it takes to execute.

The task of instruction selection is then to cover the input IR graph with the patterns given by the available machine instructions, such that every node is covered by one pattern, and the edges within the patterns correspond to edges present in the graph. Also, in order to produce efficient machine code, the total cost of all instructions used should be minimal.

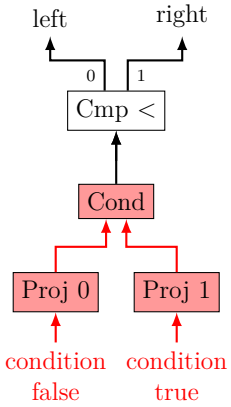
Unfortunately, this covering problem is only tractable if the IR graph is a tree. Even if we do not account for loops and restrict ourselves to IR DAGs, the problem becomes NP-complete [5]. Therefore, a large number of heuristics and formalisations have been developed. Blindell presents an extensive survey of instruction selection algorithms in [6].



(a) Examples for arithmetical and bitwise nodes.



(b) Memory access nodes with their Projs.



(c) A Cmp node and a Cond node as used for conditional jumps. In this case, the jump is taken if left $<$ right.

Figure 2.3.: Some common FIRM nodes. For the full list of available nodes, see the API [4].

We generate a greedy DAG-matcher, which recursively traverses the input dependency graph. At each node, it finds the largest pattern that has the node as a root, and replaces that pattern with the associated machine instruction.

However, our work can be applied to any instruction selector. Our task is to provide the associations between IR patterns and machine instructions. From this information, any kind of instruction selector can be built.

2.2. SMT

SMT (Satisfiability Modulo Theories) is an extension of boolean satisfiability (SAT). In it, the boolean variables of a SAT problem may be replaced by terms from one or more *theories*.

An SMT theory defines a sort (i.e. a type) or family of sorts together with its associated operations. The SMT-LIB project has standardised some SMT theories [7], which are used by the major SMT solvers. We now give an overview of the theories that we use in our work below.

First, SMT-LIB treats classic SAT as a theory named "Core". This theory defines the sort Bool with its values true and false and the usual logical operations.

The "Int" theory defines the sort Int of unbounded integers together with the usual arithmetic operations and the usual ordering.

The "FixedSizeBitVectors" theory defines a family of sorts: For each n, there is a sort BitVec_n representing a vector of n bits. The theory defines the usual logical operations on the bits, as well as operations to concatenate vectors and extract sub-vectors. More importantly, the theory also defines arithmetical operations, which interpret the bit-vectors as integers encoded in binary form. The FixedSizeBitVectors theory is therefore useful to represent CPU arithmetic.

Finally, the "ArraysEx" theory defines maps from one type to another. It also defines two operations to manipulate these maps: A value can be associated with another one by *storing* the association in the map, and the value associated with a given value can be *selected* from the map. These maps can be used to represent memory as a map from addresses to their contents.

In addition to theories, SMT problems also allow first-order quantifiers. As with SAT problems, free variables are implicitly existentially quantified, and the SMT solver needs to produce a valid assignment to them, called a *model*.

We use the SMT solver Z3 [8] in our work, which supports all these theories, and allows to mix them in one query.

In the algorithms below, we will use the following convention to represent SMT queries, with all quantifiers made explicit:

$(r, a) \leftarrow \text{SMTSolve}(\exists x.\phi(x))$

The function returns a pair, whereby the first element is either sat (satisfiable) or unsat (unsatisfiable), and the second element gives a model for the existentially quantified variables if the formula was satisfiable. We represent the model as a function from a variable's name to its assigned value.

For example, if we execute this query:

$(r, a) \leftarrow \text{SMTSolve}(\exists x.\, \exists y.\, \exists z.\, x * x + y * y = z * z)$,

we have $r = \mathsf{sat}$ and $a(x) = 3, a(y) = 4, a(z) = 5$ (or another Pythagorean triple).

2.2.1. Limitations

An SMT problem that may freely use quantifiers and the theories presented above can be undecidable, because both first-order logic [9] and non-linear arithmetic [10] are undecidable. Therefore, the admissible SMT problems may be restricted with a certain *logic*. For example, the logic might forbid non-linear arithmetic, the use of quantifiers, or unbounded integers (using bit vectors instead).

However, even if an SMT problem is decidable, it may still be impractically hard to solve. For example, a bit vector problem involving a universally quantified variable is decidable, but it may be necessary to consider all possible cases. When the bit vector becomes sufficiently large, this is clearly too inefficient.

Therefore, the problems given to an SMT solver should be as simple as possible, and should avoid universal quantification. However, a universal quantifier can sometimes be eliminated by breaking up one intractable SMT problem into several smaller ones.

2.2.2. CEGIS

CEGIS (Counterexample-guided inductive synthesis) can eliminate the universal quantifier in formulae of the structure $\exists x.\, \forall y.\, \phi(x, y)$, where $\phi(x, y)$ is a quantifier-free SMT formula.

In principle, the SMT solver now needs to consider all cases for y. However, a few cases are usually enough to check those formulae which occur in practice. If ϕ is sufficiently benign, there is a small set $\{y_1, \ldots, y_n\}$ such that

$$\exists x.\, \phi(x, y_1) \wedge \ldots \wedge \phi(x, y_n) \Rightarrow \forall y.\, \phi(x, y)$$

CEGIS (Algorithm 1) is an iterative heuristic to find this set of useful *test cases*. In one iteration, CEGIS first queries the SMT solver for an x^* that satisfies ϕ for all current test cases. This is called the *synthesis* step. Next, the algorithm queries for a y^* that does not satisfy $\phi(x^*, y^*)$ (*counterexample* step). If the solver finds a model for y^*, it is added to the set of test cases. Otherwise, the model for x^* satisfies ϕ for all y^*.

2.3. Related Work

2.3.1. Superoptimisers

The algorithm we present in Chapter 3 is closely related to those used in superoptimisation. Given a loop-free input program, the goal of superoptimisation is to find the shortest program equivalent to the input. Thus, a superoptimisation algorithm

Algorithm 1 CEGIS

```
1: procedure CEGIS(∃x.∀y.φ(x, y))
2:     Y ← ∅
3:     loop
4:         (r1, a1) ← SMTSolve(∃x*. ⋀_{yi∈Y} φ(x*, yi))
5:         if r1 = unsat then
6:             return (unsat, ∅)
7:         end if
8:         (r2, a2) ← SMTSolve(∃y*.¬φ(a1(x*), y*))
9:         if r2 = unsat then
10:            return (sat, {x ↦ a1(x*)})
11:        else
12:            Y ← Y ∪ a2(y*)
13:        end if
14:    end loop
15: end procedure
```

has two parts: it needs a way of generating possible programs, and it needs a way to check whether two programs are equivalent.

Superoptimisation was introduced by Massalin in 1987 [11] with the Superoptimizer. The Superoptimizer enumerates all programs as lists of machine instructions, and checks them for equivalence with the goal using a suite of random tests. A complete equivalence check was intractable at the time due to insufficient computing power.

In 2011, Gulwani et al. presented an algorithm which makes it possible to use SMT solvers for automated program synthesis [12]. They represent loop-free programs as a DAG of *components* and encode both the search and the verification in SMT queries. Thus, manual enumeration of all programs becomes unnecessary. This algorithm was used by Collingbourne to build the LLVM superoptimiser Souper [13].

2.3.2. Specifying Instruction Selectors

Since machine instructions often have multiple similar variants, and compiler developers want to avoid code duplication, some compilers contain a domain specific language to describe machine instructions concisely. For example, GCC contains "machine descriptions" [14], and LLVM uses the "TableGen" format [15]. Preprocessing tools then take these descriptions of the target instruction set, and expand them into source code (e.g. datatype definitions and parts of the back end).

These instruction specifications also contain an IR pattern for the instruction. However, this is a purely syntactic description. The generated instruction selector will only be able to match exactly the specified pattern. In contrast, we describe the instructions semantically, and can also find patterns that have a structure different from the specification.

2.3.3. Other Synthesis Approaches

Dias and Ramsey have already presented a tool to generate instruction selectors [16]. However, their synthesiser finds a sequence of machine instructions for each IR instruction, thus producing inefficient back end code. The authors claim that this can be remedied by subsequent peephole optimisations.

Heule et al. also worked on synthesis in the context of machine languages [17]. They developed a tool to find logical formulae for x64-64 instructions through inference from the instructions' behaviour in experiments.

3. Design and Implementation

In this chapter, we describe the design of the instruction selection generator, and of the algorithms it uses.

The instruction selection generator takes the semantic models of both IR and machine instructions, and produces code implementing an instruction selector that translates the IR to the machine language.

Algorithm 2 gives an overview of the process. The parameter I is the multiset of IR instructions, and M is the set of machine instructions. I must contain each IR instruction in sufficient number to build a pattern for any machine instruction.

For each machine instruction (the *goal instruction*), the *synthesiser* produces a set of patterns of IR instructions, which we collect in the set S. Then, the *code generator* creates an instruction selector based on this set.

Algorithm 2 Overview

```
1: procedure SelectionGenerator(I : {{Instruction}}, M : {Instruction})
2:     S ← {}                                          ▷ S : {(M × Pattern(I))}
3:     for each g ∈ M do
4:         {p_1, ..., p_n} ← Synthesise(I, g)          ▷ p_i : Pattern(I)
5:         S ← S ∪ {(g, p_1), ..., (g, p_n)}
6:     end for
7:     c ← GenerateCode(S)                             ▷ c : source code string
8:     return c
9: end procedure
```

As we have pointed out in Section 2.1.3, the results of the synthesiser can be used for any instruction selection algorithm. For compatibility with Firm, we generate a greedy DAG-matcher as used in the existing Firm back ends.

We present both steps in more detail below (Section 3.2 and Section 3.4). However, we first need to describe how we model instructions and their semantics (Section 3.1).

Notes on Notation

We will use the usual mathematical operators for both bit-vectors and integers. In addition, we use these bit-vector-specific functions:

- $v[n \dots m]$: The extraction of bits n down to m from the bit-vector v
- $v[b \mapsto x]$: A copy of bit-vector v with the bit at index b set to x

- $u \circ v$: The concatenation of bit-vectors u and v.

The indices are zero-based, with bit 0 being the least significant bit. For example, $0110[2 \ldots 1] = 11$, $0110[1 \mapsto 0] = 0100$, and $11 \circ 0100 = 110100$.

We also use square brackets to denote literal lists and access to list elements, as is common in programming. Lists use zero-based indexing, too.

Finally, we use let-bindings as known in functional programming to make our expressions more readable: We define "let $x = t$ in ϕ" to mean $(\lambda x.\, \phi)t$, where the abstraction captures unbound occurrences of x in ϕ.

3.1. Modelling Instructions

Our goal in this section is to construct a model of IR and machine instructions to represent them in SMT formulae. Our model is an extension of the one presented by Gulwani et al. [12].

Basically, an instruction takes some *arguments*, and computes some *results*. In addition, some instructions have *internal attributes*, which are chosen at compile time and fixed at run time. For example, a conditional branch instruction has the condition code as an internal attribute. Together, these form the instruction's *interface*.

The instruction declares the sorts of its interface in three lists S_a, S_i, and S_r for arguments, internal attributes and results respectively.

We also have to specify the instruction's *behaviour*. We do this through three functions P, Q, and V. Each of the functions takes three lists of SMT expressions v_a, v_i, and v_r, whereby the sorts of these expressions must be equal to those in S_a, S_i, and S_r respectively (i.e. $\text{sort}(v_a[n]) = S_a[n]$ for all elements). P and Q return SMT formulae, which specify the instruction's behaviour thus:

- $P(v_a, v_i, v_r)$ is the *precondition*. If this formula does not hold, the instruction's behaviour is undefined. In the synthesis, we may ignore test cases where the precondition does not hold.

- $Q(v_a, v_i, v_r)$ is the *postcondition*. If $P(v_a, v_i, v_r)$ holds, $Q(v_a, v_i, v_r)$ also holds. Its purpose is to define v_r in terms of v_a and v_i.

$V(v_a, v_i, v_r)$ is a list of SMT expressions, namely the list of *valid pointers* for this instruction. Depending on whether the instruction is the synthesis' goal or a candidate, we either assume or require that the pointers in $V(v_a, v_i, v_r)$ be valid.

Thus, as far as the synthesis algorithm is concerned, we can represent an instruction as a 6-tuple (S_a, S_i, S_r, P, Q, V). To denote one of the elements of instruction i, we write $S_a(i)$, $P(i)$ etc.

We now present some exemplary instruction definitions. In particular, we show how to model memory accesses.

3.1.1. Arithmetical instructions

Most arithmetical instructions are quite easy to define, because the bit-vector arithmetic defined in SMT-LIB is very similar to the usual arithmetic of a processor.

Thus, to represent an integral type of n bits, we choose an SMT bit vector of equal length (the sort BitVec_n). We do not distinguish between signed and unsigned types, but define separate instruction for signed and unsigned arithmetic in the few cases where this is necessary.

Then, defining most arithmetical instructions is straightforward. For example, this is the definition of a 32-bit subtraction instruction:

$$\begin{aligned} S_a &= [\mathsf{BitVec}_{32}, \mathsf{BitVec}_{32}] \\ S_i &= [] \\ S_r &= [\mathsf{BitVec}_{32}] \\ P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\ Q &= (v_a, v_i, v_r) \mapsto v_r[0] = v_a[0] - v_a[1] \\ V &= (v_a, v_i, v_r) \mapsto [] \\ \text{sub32} &= (S_a, S_i, S_r, P, Q, V) \end{aligned}$$

However, bit shift operations are problematic, because there are different ways to deal with an out-of-range shift amount. In C, the shift amount must be non-negative and less than the bit width, or the bit shift's behaviour is undefined. In the x86 assembly language and Firm, the shift amount is interpreted as unsigned and implicitly reduced modulo the bit width.

We can specify both behaviours in our semantics. The interface is the same for both:

$$\begin{aligned} S_a &= [\mathsf{BitVec}_{32}, \mathsf{BitVec}_{32}] \\ S_i &= [] \\ S_r &= [\mathsf{BitVec}_{32}] \\ V &= (v_a, v_i, v_r) \mapsto [] \\ \text{shr32} &= (S_a, S_i, S_r, P, Q, V) \end{aligned}$$

Then, to use the C semantics, we define

$$\begin{aligned} P &= (v_a, v_i, v_r) \mapsto 0 \leq v_a[1] < 32 \\ Q &= (v_a, v_i, v_r) \mapsto v_r[0] = v_a[0] >> v_a[1] \end{aligned}$$

and for the x86 and Firm semantics, we define

$$\begin{aligned} P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\ Q &= (v_a, v_i, v_r) \mapsto v_r[0] = v_a[0] >> (v_a[1] \bmod 32) \end{aligned}$$

We can see that the compiler may translate a C shift operation as a Firm shift operation (and, in turn, an x86 shift operation), because where the C operation is defined, the Firm/x86 operation has the same behaviour. However, we lose the freedom to exploit the undefinedness of the C operation when we translate it to Firm.

3.1.2. Memory Access

We model our memory access instructions like those in Firm: They use and define values representing the current state of memory.

When SMT is used to describe program semantics, the memory state is usually represented as an SMT array (i.e. an associative map) [18]: An address space of 32 bits with 8 bit words being addressed is presented as an array from BitVec_{32} to BitVec_8. The SMT-LIB Array theory then provides the functions select and store, which directly implement the usual load and store operations.

However, this approach proved to be too inefficient for our needs. In the counterexample step of our algorithm (see Section 3.2.2), we found that the SMT solver could not prove with acceptable efficiency that no counterexample memory state exists. We therefore need to find a more efficient model for memory state.

As we can see from the overview algorithm (Algorithm 2 on Page 21), we only consider one machine instruction at a time as our goal. Therefore, we can restrict our memory model to only represent those addresses that the goal instruction uses (its valid pointer list). If a synthesis candidate uses any other addresses, we can exclude the candidate.

We have tried to specify this restriction in the Array theory, but the SMT solver could not make use of it. Therefore, we need to model the memory ourselves.

The memory state needs to hold two pieces of information for each valid pointer: Of course, it must store the value located at that address. In addition, it must store an *access flag* that is set when the address is loaded from.

We need the access flag for the following reason: A load operation does not change the state of our representation, but can actually have side-effects on volatile memory. Moreover, Firm requires that a Load node produce a new M-value in order to encode write-after-read dependencies.

Therefore, we introduce an artificial change to the memory using the access flag. This way, the goal instruction and synthesis candidate are only deemed equivalent if they load from the same memory locations.

To generate an M-value sort $M(g)$ for a goal instruction g with the valid pointer list $V(g)$, we use a bit vector of size $|V(g)| \cdot (w+1)$, where w is the bit-width of the words being addressed.

This bit vector is laid out as follows: Bits $k \cdot (w+1)+1$ to $k \cdot (w+1)+w$ hold the value located at the address $V(g)[k]$. The extra bit $k \cdot (w+1)$ holds the access flag.

Defining load and store functions is then a matter of extracting and overwriting the right bits in the M-value. We also need not concern ourselves with handling

invalid pointers, since the synthesis algorithm will ensure that all pointers are valid (see Section 3.2.2).

First, we define two primitives "ld" and "st" to load and store one word, and then build up the memory access instructions from these. For the following definitions, we use these abbreviations:

$$\begin{aligned} l(k) &:= k \cdot (w+1) + 1 && \text{Lower bound of } k\text{-th word} \\ u(k) &:= k \cdot (w+1) + w && \text{Upper bound of } k\text{-th word} \\ f(k) &:= k \cdot (w+1) && \text{Flag for } k\text{-th word} \\ n &:= |V(g)| \end{aligned}$$

The function "ld" is parameterised by $V(g)$ (which we assume is externally defined), takes an M-value and an address to load from, and returns a new M-value and the value loaded:

$$\mathrm{ld}(m,a) = \begin{cases} (m[f(0) \mapsto 1], m[u(0) \dots l(0)]) & a = V(g)[0] \\ \vdots & \vdots \\ (m[f(k) \mapsto 1], m[u(k) \dots l(k)]) & a = V(g)[k] \\ \vdots & \vdots \\ (m[f(n-1) \mapsto 1], m[u(n-1) \dots l(n-1)]) & \text{otherwise} \end{cases}$$

Similarly, we define "st". It takes an M-value, an address and a value to store, and returns the updated M-value.

$$\mathrm{st}(m,a,v) = \begin{cases} m[n \cdot (w+1) \dots u(0)+1] \circ v \circ m[l(0)-1 \dots 0] & a = V(g)[0] \\ \vdots & \vdots \\ m[n \cdot (w+1) \dots u(k)+1] \circ v \circ m[l(k)-1 \dots 0] & a = V(g)[k] \\ \vdots & \vdots \\ m[n \cdot (w+1) \dots u(n-1)+1] \circ v \circ m[l(n-1)-1 \dots 0] & \text{otherwise} \end{cases}$$

In each definition's last case, we use the fact that we can assume the address to be valid. We therefore need not check against $V(g)[n-1]$. In the SMT solver's language, these definitions by cases become a chain of conditional assignments (called ite for if-then-else in SMT-LIB).

Having defined the load and store functions, we can now specify load and store instructions. For example, this is the specification of a 32-bit store instruction (again,

we assume that the goal instruction is globally defined):

$$
\begin{aligned}
S_a &= [M(g), \mathsf{BitVec}_{32}, \mathsf{BitVec}_{32}] && \textit{(M-value, address, value)}\\
S_i &= []\\
S_r &= [M(g)]\\
P &= (v_a, v_i, v_r) \mapsto \mathsf{true}\\
Q &= (v_a, v_i, v_r) \mapsto \text{let } m_0 = \mathrm{st}(v_a[0], v_a[1], v_a[2][7\ldots0]) \text{ in}\\
&\qquad \text{let } m_1 = \mathrm{st}(m_0, v_a[1] + 1, v_a[2][15\ldots8]) \text{ in}\\
&\qquad \text{let } m_2 = \mathrm{st}(m_1, v_a[1] + 2, v_a[2][23\ldots16]) \text{ in}\\
&\qquad \text{let } m_3 = \mathrm{st}(m_2, v_a[1] + 3, v_a[2][31\ldots24]) \text{ in}\\
&\quad v_r[0] = m_3\\
V &= (v_a, v_i, v_r) \mapsto [v_a[1], v_a[1] + 1, v_a[1] + 2, v_a[1] + 3]\\
\text{store32} &= (S_a, S_i, S_r, P, Q, V)
\end{aligned}
$$

Implementation Matters

For any instruction i, $V(i)$ can easily be generated from $Q(i)$: $V(i)$ is the set of all values occurring as second arguments to ld or st. This also ensures that $V(i)$ always stays consistent with $Q(i)$.

However, the definitions of ld and st require the memory type $M(g)$, and thus $V(g)$, to be known (where g is the goal instruction). Thus, if $Q(g)$ uses ld or st, we can only define it after $V(g)$ is known, but we would like to extract $V(g)$ from $Q(g)$ automatically.

In order to break this cycle, the instructions do not define P, Q, and V directly, but specify their semantics in one function that takes a `NodeEnv` as its argument. `NodeEnv` is an abstract type with two implementations. The first implementation (the "dummy implementation") records all uses of ld and st to collect V. The second implementation uses $V(g)$ to actually produce the SMT expressions for P and Q that specify the instruction.

3.1.3. Constants

There are two kinds of constants we have to deal with, which differ in the point at which their value is defined.

Firstly, *synthesis-time constants* have their value chosen during synthesis. That value is the same for all instances of the instruction. For example, the IR equivalent of the increment instruction $x_2 = \mathrm{inc}(x_1)$ is $x_2 = x_1 + 1$, where 1 is a synthesis-time constant.

We represent synthesis-time constants with IR instructions that take no arguments and produce a constant from their internal attribute. They are equivalent to the Const node in Firm in its different modes. For example, a 32-bit-wide constant is

defined as follows:

$$\begin{aligned} S_a &= [] \\ S_i &= [\mathsf{BitVec}_{32}] \\ S_r &= [\mathsf{BitVec}_{32}] \\ P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\ Q &= (v_a, v_i, v_r) \mapsto v_r[0] = v_i[0] \\ V &= (v_a, v_i, v_r) \mapsto [] \\ \text{const32} &= (S_a, S_i, S_r, P, Q, V) \end{aligned}$$

Secondly, there are *compile-time constants*. In assembly language, these occur as immediate arguments to instructions. Since it is very inefficient to generate separate rules for every possible immediate, the synthesis actually treats compile-time constants as arguments of the instruction that are marked as constant.

The synthesis then proceeds as normal, and it is up to the code generator to emit the tests required to ensure that the argument in question is in fact a compile-time constant.

3.1.4. Conditional Control Flow

Many IRs handle conditional control flow similarly, but still differ slightly (compare LLVM's `icmp` and `br` [19] to Firm's Cmp and Cond). Therefore, the following section is Firm-specific, but can be adapted to other IRs.

To represent control flow, we need to represent three additional modes:

- To represent boolean values, we use the built-in sort Bool
- To represent the possible relations for a comparison, we use a 3-bit bit vector. Bit 2 represents the relation "greater than", bit 1 represents "less than", and bit 0 represents "equal to". Relations are combined by settings multiple bits. For example, relation number 6 is "greater than or less than", i.e. "not equal to". The three bits correspond to the three bits in Firm's type `ir_relation` [4] that are used for integers (`ir_relation_unordered` is only used for floating-point comparisons).
- To represent control flow, we also use the sort Bool, whereby true represents a branch being taken.

With these new modes, we can represent the Cmp node, using an internal attribute for the relation. See the Firm API documentation for all possible relation values.

Signedness matters in the comparison instruction. Firm determines whether to use signed or unsigned comparison based on the arguments' modes; we define two variants of the instruction. Both definitions look exactly the same, only the meaning of the relational operators is changed.

$$
\begin{aligned}
S_a &= [\mathsf{BitVec}_{32}, \mathsf{BitVec}_{32}] \\
S_i &= [\mathsf{BitVec}_3] \\
S_r &= [\mathsf{Bool}] \\
P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\
Q &= (v_a, v_i, v_r) \mapsto v_r[0] = \begin{cases} \mathsf{false} & v_i[0] = 0 \\ v_a[0] = v_a[1] & v_i[0] = 1 \\ \quad\vdots & \quad\vdots \\ v_a[0] \neq v_a[1] & v_i[0] = 6 \\ \mathsf{true} & v_i[0] = 7 \end{cases} \\
V &= (v_a, v_i, v_r) \mapsto [] \\
\text{cmp32} &= (S_a, S_i, S_r, P, Q, V)
\end{aligned}
$$

When defining any control flow node, we must be careful to set exactly one of the control flow results to true, and the rest to false. Other than that, the definition of the Cond node is simple. It produces two control flow results. The first result is the branch taken if the input is false, the second result is taken if the input is true.

$$
\begin{aligned}
S_a &= [\mathsf{Bool}] \\
S_i &= [] \\
S_r &= [\mathsf{Bool}, \mathsf{Bool}] \\
P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\
Q &= (v_a, v_i, v_r) \mapsto v_r[0] = \neg v_a[0] \wedge v_r[1] = v_a[0] \\
V &= (v_a, v_i, v_r) \mapsto [] \\
\text{cond} &= (S_a, S_i, S_r, P, Q, V)
\end{aligned}
$$

3.1.5. Machine Instructions

We define machine instructions in the same way, but these may be more complex for CISC machines. For example, the following is the definition of a 32-bit-wide subtraction of a value from a location in memory (such as the x86 instruction `sub %eax, x`, where `x` is a global variable).

In comparison with the IR instruction store32, the order of M-value and address is reversed in the argument sequence. This is because we follow the interface set by the nodes in the Firm x86 back end.

$$
\begin{aligned}
S_a &= [\mathsf{BitVec}_{32}, M(g), \mathsf{BitVec}_{32}] && \textit{(address, M-value, subtrahend)}\\
S_i &= []\\
S_r &= [M(g)]\\
P &= (v_a, v_i, v_r) \mapsto \mathsf{true}\\
Q &= (v_a, v_i, v_r) \mapsto \text{let } (m_0, x_0) = \mathrm{ld}(v_a[1], v_a[0]) \text{ in}\\
&\qquad \text{let } (m_1, x_1) = \mathrm{ld}(m_0, v_a[0] + 1) \text{ in}\\
&\qquad \text{let } (m_2, x_2) = \mathrm{ld}(m_1, v_a[0] + 2) \text{ in}\\
&\qquad \text{let } (m_3, x_3) = \mathrm{ld}(m_2, v_a[0] + 3) \text{ in}\\
&\qquad \text{let } x = x_3 \circ x_2 \circ x_1 \circ x_0 \text{ in}\\
&\qquad \text{let } x' = x - v_a[2] \text{ in}\\
&\qquad \text{let } m_4 = \mathrm{st}(m_3, v_a[0], x'[7 \ldots 0]) \text{ in}\\
&\qquad \text{let } m_5 = \mathrm{st}(m_4, v_a[0] + 1, x'[15 \ldots 8]) \text{ in}\\
&\qquad \text{let } m_6 = \mathrm{st}(m_5, v_a[0] + 2, x'[23 \ldots 16]) \text{ in}\\
&\qquad \text{let } m_7 = \mathrm{st}(m_6, v_a[0] + 3, x'[31 \ldots 24]) \text{ in}\\
&\qquad v_r[0] = m_7\\
V &= (v_a, v_i, v_r) \mapsto [v_a[0], v_a[0] + 1, v_a[0] + 2, v_a[0] + 3]\\
\text{ia-submem32} &= (S_a, S_i, S_r, P, Q, V)
\end{aligned}
$$

Control Flow In Machine Instructions

Where control flow is concerned, machine architectures differ significantly. Our target architecture is x86, which has a generic comparison instruction, and encodes the relation to check for in the conditional jump. Other machines, such as MIPS, use an approach more like Firm's.

The x86 compare instruction ia-cmp32 sets the processor's *flags* [20], and the conditional jump instructions evaluate them [21]. We represent the arithmetic flags OF, SF, ZF, PF, and CF in a 5-bit bit vector.

The computation of SF, ZF, PF, and CF is straightforward. To compute OF, consider the following ($\oplus$ represents exclusive or): Let $s(x)$ be the sign bit of x, and let ci be the bit carried into the sign bit during the computation $a - b$. Then OF is defined as $ci \oplus \mathrm{CF}$, since CF is the carry out of the sign bit. On the other hand, we have $s(a - b) = s(a) \oplus s(b) \oplus ci$, because $-$ and $\oplus$ are the same function on single bits. From this follows $\mathrm{OF} = s(a - b) \oplus s(a) \oplus s(b) \oplus \mathrm{CF}$.

As with the IR comparison instruction, the relation to check (the *condition code*) is an internal attribute, but in this case one of the jump instruction. There are 16 possible condition codes for integers, which we represent in the same way as the type `x86_condition_code_t` in Firm's x86 back end.

We define ia-cmp32 and ia-jcc as follows, whereby the if-then-else function ite returns the second argument, if the first argument is true, and the third one otherwise.

$$
\begin{aligned}
S_a &= [\mathsf{BitVec}_{32}, \mathsf{BitVec}_{32}] \\
S_i &= [] \\
S_r &= [\mathsf{BitVec}_5] \\
P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\
Q &= (v_a, v_i, v_r) \mapsto \text{let } s = v_a[0] - v_a[1] \text{ in} \\
&\qquad \text{let } SF = s[31 \ldots 31] \text{ in} \\
&\qquad \text{let } ZF = \mathsf{ite}(s = 0, 1, 0) \text{ in} \\
&\qquad \text{let } PF = \bigoplus_{i=0}^{7} s[i \ldots i] \text{ in} \\
&\qquad \text{let } CF = \mathsf{ite}(v_a[0] <_{unsigned} v_a[1], 1, 0) \text{ in} \\
&\qquad \text{let } OF = SF \oplus v_a[0][31 \ldots 31] \oplus v_a[1][31 \ldots 31] \oplus CF \text{ in} \\
v_r[0] &= OF \circ SF \circ ZF \circ PF \circ CF \\
V &= (v_a, v_i, v_r) \mapsto [] \\
\text{ia-cmp32} &= (S_a, S_i, S_r, P, Q, V)
\end{aligned}
$$

$$
\begin{aligned}
S_a &= [\mathsf{BitVec}_5] \\
S_i &= [\mathsf{BitVec}_4] \\
S_r &= [\mathsf{Bool}, \mathsf{Bool}] \\
P &= (v_a, v_i, v_r) \mapsto \mathsf{true} \\
Q &= (v_a, v_i, v_r) \mapsto \text{let } OF = v_a[0][4] \text{ in} \\
&\qquad \text{let } SF = v_a[0][3] \text{ in} \\
&\qquad \text{let } ZF = v_a[0][2] \text{ in} \\
&\qquad \text{let } PF = v_a[0][1] \text{ in} \\
&\qquad \text{let } CF = v_a[0][0] \text{ in} \\
&\qquad\qquad \text{let } j = \begin{cases} OF = 1 & v_i[0] = 0 \\ OF = 0 & v_i[0] = 1 \\ \qquad\vdots & \qquad\vdots \\ ZF = 0 \wedge SF = OF & v_i[0] = 15 \end{cases} \text{ in} \\
v_r[0] &= \neg j \wedge v_r[1] = j \\
V &= (v_a, v_i, v_r) \mapsto [] \\
\text{ia-jcc} &= (S_a, S_i, S_r, P, Q, V)
\end{aligned}
$$

The difference between the way that the IR and the machine language handle control flow leads to a problem for the synthesis: From the overview algorithm

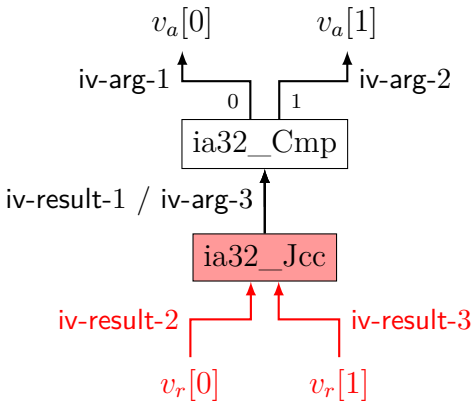


Figure 3.1.: A graph pattern consisting of a comparison and conditional jump instruction. We use this pattern as an example to construct a compound instruction.

(Algorithm 2 on Page 21), we can see that the synthesis searches for an IR pattern that implements a single instruction. However, the FIRM IR does not use flags, so no IR pattern can represent a machine instruction that uses or produces a flag value. To solve this problem, we introduce *compound instructions*.

3.1.6. Compound Instructions

A compound instruction is an instruction defined through a graph pattern. For example, the graph pattern shown in Figure 3.1 can define a compound instruction to represent the instructions `cmp x, y; jl label` as a unit.

To build an SMT formula for the precondition or postcondition of a compound instruction c (i.e. $P(c)$ or $Q(c)$), we first introduce fresh intermediate variables for every instruction in c's graph pattern. Then, we construct $P(i)$ or $Q(i)$ for every instruction i in the pattern. Finally, for each edge in the pattern, we assert that the values it connects should be equal.

Taking the compound instruction defined by the pattern in Figure 3.1 as an example, we get the following for $Q(c)$ (the condition code for "less than" is 12):

$$\begin{aligned} Q(c)(v_a, v_i, v_r) = {} & Q(\textit{ia-cmp32})([\text{iv-arg-1}, \text{iv-arg-2}], [], [\text{iv-result-1}]) \wedge \\ & Q(\textit{ia-jcc})([\text{iv-arg-3}], [12], [\text{iv-result-2}, \text{iv-result-3}] \wedge \\ & \text{iv-arg-1} = v_a[0] \wedge \\ & \text{iv-arg-2} = v_a[1] \wedge \\ & \text{iv-arg-3} = \text{iv-result-1} \wedge \\ & v_r[0] = \text{iv-result-2} \wedge \\ & v_r[1] = \text{iv-result-3} \end{aligned}$$

3.2. Synthesis Algorithm

Having defined the IR and machine instructions, we can now solve the main problem: Given a goal instruction g and the multiset of IR instructions I, we have to synthesise a graph pattern of IR instructions that implements g. Assuming that we have a function Q^+, which extends the postcondition-function Q from single instructions to patterns, we can express our query as follows:

$$\exists pattern.\, \exists V_i.\, \forall (V_a, V_r).\, Q^+(pattern)(V_a, V_i, V_r) \Rightarrow Q(g)(V_a, [], V_r)$$

This formula is simplified, because it ignores preconditions and pointer validity. We develop the full formula in Section 3.2.2.

In the following, we will assume that the goal instruction has no internal attributes. If we want to synthesise a goal instruction with internal attributes, we have to iterate over all possible assignments to them.

Of course, the concept of an IR graph pattern is not readily understandable to an SMT solver. Therefore, we first have to find an encoding of graph patterns that the SMT solver can use. We present this in Section 3.2.1.

Next, we use a CEGIS-like algorithm to search the set of IR graph patterns for those that implement the goal instruction. We present the search algorithm in Section 3.2.2, and discuss some optimisations in Section 3.3.

Throughout this section, we will use the ia-submem32 instruction as our example, which we will synthesise from a sub32 instruction, a load32 instruction, and a store32 instruction. Of course, in a real synthesis we would not know which IR instructions we need, and therefore would provide many more.

3.2.1. Pattern Representation

The IR pattern representation must support the following:

- The SMT solver must be able to enumerate IR patterns
- We need to express semantics for patterns in SMT formulae
- We need to assert that a pattern is well-formed

The representation we present has been developed in [12], where it is used in constructing a superoptimiser. We add support for multiple sorts and instructions with multiple results.

We construct our patterns from a set of *components*. There is one component for each element of I. For abbreviation, we will use the component as an argument to S_a, Q, etc. to mean the instruction associated with it.

To form a pattern, we must determine the order of components, the place where each component gets its input data from, and which values are the pattern's results.

Pattern Encoding

We encode the patterns using the concept of *locations*. A location is any place in the pattern where data is produced. Therefore, the set of locations is also the set of possible arguments for a component, and of possible results of the pattern.

In our IR patterns, we have two kinds of locations: Each argument to the whole pattern is a location, and each result of a component is a location.

We encode the locations by an integer index starting from 0. If our pattern has $|S_a(g)|$ arguments (g being the goal instruction), and the set of available components is C, the number of locations necessary is

$$N_l = |S_a(g)| + \sum_{c \in C} |S_r(c)|$$

A *location variable* is an SMT variable whose value is a location. In the SMT formulae, we use the Int sort for location variables, and constrain the values to the valid range: For every location variable l defined below, we add the constraint $0 \leq l \wedge l < N_l$. We call the formula containing all these constraints ϕ_{range}.

Given a component c, we need the following location variables to place this component in the pattern:

- c-loc is the location of the component itself. The component will define its results at the locations c-loc through c-loc + $(|S_r(c)| - 1)$
- c-arg-i-loc for each i from 0 to $|S_a(c)| - 1$ is the location of the value used as the i-th argument of the component.

In addition, we need to specify where the pattern's results are taken from:

- result-i-loc holds the location of the value to be used as the i-th result of the pattern, for each i from 0 to $|S_r(g)| - 1$.

During synthesis, it is the SMT solver's task to find an assignment to the location variables, which define an IR pattern for the goal instruction. To achieve this, we first need to constrain the arrangement of components to valid loop-free programs. We capture the conditions necessary for this in the *well-formedness constraint*.

Well-Formedness

The arrangement of components must fulfil these conditions to be a well-formed IR pattern:

- It must be loop-free, i.e. a DAG
- There must be exactly one pattern argument or component result assigned to each location.

- If a component's argument has the sort s, the location variable of that argument must refer to a location that is associated with a value of sort s.

We can achieve loop-freedom, if we require that all components take their arguments from lesser-valued locations. Thus, given a component c as above, we define its DAG-constraint as follows:

$$\phi_{DAG}(c) = \bigwedge_{i=0}^{|S_a(c)|-1} c\text{-arg-}i\text{-loc} < c\text{-loc}$$

Next, we must assure that there is exactly one value assigned to each location. SMT-LIB has a built-in predicate distinct, which holds if the values of all its arguments are distinct. We can efficiently express the required constraint using this predicate.

In particular, we need the following locations to be distinct: Firstly, all pattern arguments, and secondly all locations which are used as results by a component.

$$\phi_{distinct} = \text{distinct}\left(\{0,\ldots,|S_a(g)|-1\} \cup \bigcup_{c\in C} \{c\text{-loc},\ldots,c\text{-loc} + (|S_r(c)|-1)\}\right)$$

Finally, we must match sorts between definition and use. For this, let same(s) be the set of all locations (pattern arguments or component results) with sort s. Then, the i-th argument of the component c must refer to a location from same($S_a(c)[i]$):

$$\phi_{asort}(c) = \bigwedge_{i=0}^{|S_a(c)|-1} \left(\bigvee_{l\in \text{same}(S_a(c)[i])} c\text{-arg-}i\text{-loc} = l \right)$$

We also need a similar condition for the pattern's results (g is the goal instruction, which the pattern is to implement):

$$\phi_{rsort} = \bigwedge_{i=0}^{|S_r(g)|-1} \left(\bigvee_{l\in \text{same}(S_r(g)[i])} \text{result-}i\text{-loc} = l \right)$$

In conclusion, the well-formedness constraint for the set of components C is

$$\phi_{wf} = \phi_{range} \wedge \phi_{distinct} \wedge \bigwedge_{c\in C} (\phi_{DAG}(c) \wedge \phi_{asort}(c)) \wedge \phi_{rsort}$$

Example

In our example, the goal instruction is ia-submem32, and thus the pattern has three arguments with the sorts BitVec_{32}, M(ia-submem32), BitVec_{32}; and one result with the sort M(ia-submem32). Since the goal instruction has four valid pointers, we have M(ia-submem32) = BitVec_{36}.

The component set is $C = \{\text{sub32}, \text{load32}, \text{store32}\}$. In the SMT formula, we will name the components sub, load, and store. Since the load32 instruction has two results, we need seven locations in total.

All in all, we get the formula shown in Figure 3.2 for $\phi_{wf}(C)$. Following the definitions above, the formulae for ϕ_{asort} contain cases such as sub-arg-0-loc = sub-loc. These are of course impossible due to the DAG-constraint, and can be easily optimised, but we have rather kept the definition of ϕ_{asort} simpler.

$$
\begin{aligned}
\phi_{wf}(C) = \; & \mathrm{range}(\mathsf{sub\text{-}loc}) \wedge \mathrm{range}(\mathsf{sub\text{-}arg\text{-}0\text{-}loc}) \wedge \mathrm{range}(\mathsf{sub\text{-}arg\text{-}1\text{-}loc}) \wedge \\
& \mathrm{range}(\mathsf{load\text{-}loc}) \wedge \mathrm{range}(\mathsf{load\text{-}arg\text{-}0\text{-}loc}) \wedge \mathrm{range}(\mathsf{load\text{-}arg\text{-}1\text{-}loc}) \wedge \\
& \mathrm{range}(\mathsf{store\text{-}loc}) \wedge \mathrm{range}(\mathsf{store\text{-}arg\text{-}0\text{-}loc}) \wedge \mathrm{range}(\mathsf{store\text{-}arg\text{-}1\text{-}loc}) \wedge \\
& \quad \mathrm{range}(\mathsf{store\text{-}arg\text{-}2\text{-}loc}) \wedge \\
& \mathrm{range}(\mathsf{result\text{-}0\text{-}loc}) \wedge \\
& \mathsf{distinct}(0, 1, 2, \mathsf{sub\text{-}loc}, \mathsf{load\text{-}loc}, \mathsf{load\text{-}loc} + 1, \mathsf{store\text{-}loc}) \wedge \\
& \mathsf{sub\text{-}arg\text{-}0\text{-}loc} < \mathsf{sub\text{-}loc} \wedge \mathsf{sub\text{-}arg\text{-}1\text{-}loc} < \mathsf{sub\text{-}loc} \wedge \\
& \mathsf{load\text{-}arg\text{-}0\text{-}loc} < \mathsf{load\text{-}loc} \wedge \mathsf{load\text{-}arg\text{-}1\text{-}loc} < \mathsf{load\text{-}loc} \wedge \\
& \mathsf{store\text{-}arg\text{-}0\text{-}loc} < \mathsf{store\text{-}loc} \wedge \mathsf{store\text{-}arg\text{-}1\text{-}loc} < \mathsf{store\text{-}loc} \wedge \\
& \quad \mathsf{store\text{-}arg\text{-}2\text{-}loc} < \mathsf{store\text{-}loc} \wedge \\
& (\mathsf{sub\text{-}arg\text{-}0\text{-}loc} = 0 \vee \mathsf{sub\text{-}arg\text{-}0\text{-}loc} = 2 \vee \mathsf{sub\text{-}arg\text{-}0\text{-}loc} = \mathsf{sub\text{-}loc} \vee \\
& \quad \mathsf{sub\text{-}arg\text{-}0\text{-}loc} = \mathsf{load\text{-}loc} + 1) \wedge \\
& (\mathsf{sub\text{-}arg\text{-}1\text{-}loc} = 0 \vee \mathsf{sub\text{-}arg\text{-}1\text{-}loc} = 2 \vee \mathsf{sub\text{-}arg\text{-}1\text{-}loc} = \mathsf{sub\text{-}loc} \vee \\
& \quad \mathsf{sub\text{-}arg\text{-}1\text{-}loc} = \mathsf{load\text{-}loc} + 1) \wedge \\
& (\mathsf{load\text{-}arg\text{-}0\text{-}loc} = 1 \vee \mathsf{load\text{-}arg\text{-}0\text{-}loc} = \mathsf{load\text{-}loc} \vee \\
& \quad \mathsf{load\text{-}arg\text{-}0\text{-}loc} = \mathsf{store\text{-}loc}) \wedge \\
& (\mathsf{load\text{-}arg\text{-}1\text{-}loc} = 0 \vee \mathsf{load\text{-}arg\text{-}1\text{-}loc} = 2 \vee \mathsf{load\text{-}arg\text{-}1\text{-}loc} = \mathsf{sub\text{-}loc} \vee \\
& \quad \mathsf{load\text{-}arg\text{-}1\text{-}loc} = \mathsf{load\text{-}loc} + 1) \wedge \\
& (\mathsf{store\text{-}arg\text{-}0\text{-}loc} = 1 \vee \mathsf{store\text{-}arg\text{-}0\text{-}loc} = \mathsf{load\text{-}loc} \vee \\
& \quad \mathsf{store\text{-}arg\text{-}0\text{-}loc} = \mathsf{store\text{-}loc}) \wedge \\
& (\mathsf{store\text{-}arg\text{-}1\text{-}loc} = 0 \vee \mathsf{store\text{-}arg\text{-}1\text{-}loc} = 2 \vee \mathsf{store\text{-}arg\text{-}1\text{-}loc} = \mathsf{sub\text{-}loc} \vee \\
& \quad \mathsf{store\text{-}arg\text{-}1\text{-}loc} = \mathsf{load\text{-}loc} + 1) \wedge \\
& (\mathsf{store\text{-}arg\text{-}2\text{-}loc} = 0 \vee \mathsf{store\text{-}arg\text{-}2\text{-}loc} = 2 \vee \mathsf{store\text{-}arg\text{-}2\text{-}loc} = \mathsf{sub\text{-}loc} \vee \\
& \quad \mathsf{store\text{-}arg\text{-}2\text{-}loc} = \mathsf{load\text{-}loc} + 1) \wedge \\
& (\mathsf{result\text{-}0\text{-}loc} = 1 \vee \mathsf{result\text{-}0\text{-}loc} = \mathsf{load\text{-}loc} \vee \\
& \quad \mathsf{result\text{-}0\text{-}loc} = \mathsf{store\text{-}loc})
\end{aligned}
$$

Figure 3.2.: The full well-formedness constraint for the three components sub, load, and store. We abbreviate $0 \leq x \wedge x < 7$ with $\mathrm{range}(x)$.

Semantics of Patterns

As we have seen at the start of Section 3.2, we need to construct a formula that specifies the semantics of a pattern by combining the semantics of the pattern's nodes (i.e. their Q). Given a pattern made of the components C, SMT expressions for the pattern's arguments (V_a), and SMT expressions for its return values (V_r), we have to produce an SMT formula $Q^+(E, C, V_a, V_r)$ that constrains the elements of V_r to the results of C given the arguments V_a.

For each use of Q^+, we provide a set E of fresh variables for every argument and return value in C. For every $c \in C$, we have:

- E-c-arg-i with sort $S_a(c)[i]$ for each i from 0 to $|S_a(c)|$.
- E-c-result-i with sort $S_r(c)[i]$ for each i from 0 to $|S_r(c)|$.

In the implementation, we give each use of Q^+ an index, and prefix the variable names with that.

On the other hand, the values of internal attributes do not change with V_a, and we therefore assume that a variable c-internal-i with sort $S_i(c)[i]$ is globally declared for each i from 0 to $|S_i(c)|$. We call the set of all these variables V_i.

We must now connect the variables. There are two kinds of connections to encode: Firstly, the argument and result values of the same component must be connected as specified in the component's semantics: If we define for $c \in C$

$$\begin{aligned} v_a(E, c) &:= [E\text{-}c\text{-arg-}i]_{0 \le i \le |S_a(c)|} \\ v_i(c) &:= [c\text{-internal-}i]_{0 \le i \le |S_i(c)|} \\ v_r(E, c) &:= [E\text{-}c\text{-result-}i]_{0 \le i \le |S_r(c)|} \end{aligned}$$

we have

$$\phi_{compute}(E, C) = \bigwedge_{c \in C} Q(c)(v_a(E, c), v_i(c), v_r(E, c))$$

Secondly, we must connect the components to each other as prescribed by the location variables defined above. Each argument value E-c-arg-i has a corresponding location variable c-arg-i-loc, which refers to an argument of the pattern or a result of another component. We must now compare the value of c-arg-i-loc with the locations of the same sort, and set E-c-arg-i equal to the matching location's value. For the pattern arguments, we have:

$$\phi_a(E, C, V_a) = \bigwedge_{c \in C} \bigwedge_{i=0}^{|S_a(c)|-1} \bigwedge_{A \in \text{same}_a(S_a(c)[i])} c\text{-arg-}i\text{-loc} = A \implies E\text{-}c\text{-arg-}i = V_a[A]$$

where $\text{same}_a(s)$ is the set of all pattern arguments with sort s:

$$\text{same}_a(s) = \{A \mid 0 \le A < |S_a(g)| \wedge S_a(g)[A] = s\}$$

For the other component's results, we have:

$$\phi_c(E,C) = \bigwedge_{c,c' \in C} \bigwedge_{i=0}^{|S_a(c)|-1} \bigwedge_{r \in \text{same}_r(c', S_a(c)[i])} c\text{-arg-}i\text{-loc} = c'\text{-loc} + r \implies E\text{-}c\text{-arg-}i = E\text{-}c'\text{-result-}r$$

where $\text{same}_r(c,s)$ is the set of all results of c with sort s:

$$\text{same}_r(c,s) = \{r \mid 0 \leq r < |S_r(c)| \wedge S_r(c)[r] = s\}$$

In the same way, we connect the pattern's results (given by $V_r[i]$) to their respective locations (given by result-i-loc) in the formulae $\phi_{ar}(V_a, V_r)$ and $\phi_{cr}(E, C, V_r)$, namely:

$$\phi_{ar}(V_a, V_r) = \bigwedge_{i=0}^{|S_r(g)|-1} \bigwedge_{A \in \text{same}_a(S_r(g)[i])} \text{result-}i\text{-loc} = A \implies V_r[i] = V_a[A]$$

$$\phi_{cr}(E,C,V_r) = \bigwedge_{i=0}^{|S_r(g)|-1} \bigwedge_{c' \in C} \bigwedge_{r \in \text{same}_r(c', S_r(g)[i])} \text{result-}i\text{-loc} = c'\text{-loc} + r \implies V_r[i] = E\text{-}c'\text{-result-}r$$

All in all, we can now define $Q^+(E, C, V_a, V_r)$:

$$Q^+(E,C,V_a,V_r) = \phi_{compute}(E,C) \wedge \phi_a(E,C,V_a) \wedge \phi_c(E,C) \wedge \phi_{ar}(V_a,V_r) \wedge \phi_{cr}(E,C,V_r)$$

Example

Returning to our example to synthesise ia-submem32, we need three values for the pattern's arguments and one for its result. For the example, we choose the symbolic values $V_a = [\text{arg-0}, \text{arg-1}, \text{arg-2}]$ and $V_r = [\text{result-0}]$. The formula for $Q^+(E, C, V_a, V_r)$ is shown in Figure 3.3, but with the prefix for E omitted.

3.2.2. Search Algorithm

In the previous section, we have developed a representation of graph patterns which the SMT solver can work with. Using this representation, our simplified constraint now becomes

$$\exists L(C).\, \exists V_i.\, \forall (V_a, V_r).\, \exists E.\, \phi_{wf}(C) \wedge Q^+(E,C,V_a,V_r) \Rightarrow Q(g)(V_a, [], V_r)$$

where $L(C)$ is the set of all location variables for the components C.

We shall now extend this formula to include all necessary parts. First, the formula need only hold if the preconditions of all instructions in C hold. On the other hand,

$$
\begin{aligned}
&Q^{+}(E, C, V_a, V_r) = \\
&\quad Q(\mathrm{sub32})([\text{sub-arg-0}, \text{sub-arg-1}], [], [\text{sub-result-0}]) \wedge \\
&\quad Q(\mathrm{load32})([\text{load-arg-0}, \text{load-arg-1}], [], [\text{load-result-0}, \text{load-result-1}]) \wedge \\
&\quad Q(\mathrm{store32})([\text{store-arg-0}, \text{store-arg-1}, \text{store-arg-2}], [], [\text{store-result-0}]) \wedge \\
&\quad \text{sub-arg-0-loc} = 0 \Rightarrow \text{sub-arg-0} = \text{arg-0} \wedge \\
&\qquad \text{sub-arg-0-loc} = 2 \Rightarrow \text{sub-arg-0} = \text{arg-2} \wedge \\
&\qquad \text{sub-arg-0-loc} = \text{sub-loc} \Rightarrow \text{sub-arg-0} = \text{sub-result-0} \wedge \\
&\qquad \text{sub-arg-0-loc} = \text{load-loc} + 1 \Rightarrow \text{sub-arg-0} = \text{load-result-1} \wedge \\
&\quad \text{sub-arg-1-loc} = 0 \Rightarrow \text{sub-arg-1} = \text{arg-0} \wedge \\
&\qquad \text{sub-arg-1-loc} = 2 \Rightarrow \text{sub-arg-1} = \text{arg-2} \wedge \\
&\qquad \text{sub-arg-1-loc} = \text{sub-loc} \Rightarrow \text{sub-arg-1} = \text{sub-result-0} \wedge \\
&\qquad \text{sub-arg-1-loc} = \text{load-loc} + 1 \Rightarrow \text{sub-arg-1} = \text{load-result-1} \wedge \\
&\quad \text{load-arg-0-loc} = 1 \Rightarrow \text{load-arg-0} = \text{arg-1} \wedge \\
&\qquad \text{load-arg-0-loc} = \text{load-loc} \Rightarrow \text{load-arg-0} = \text{load-result-0} \wedge \\
&\qquad \text{load-arg-0-loc} = \text{store-loc} \Rightarrow \text{load-arg-0} = \text{store-result-0} \wedge \\
&\quad \text{load-arg-1-loc} = 0 \Rightarrow \text{load-arg-1} = \text{arg-0} \wedge \\
&\qquad \text{load-arg-1-loc} = 2 \Rightarrow \text{load-arg-1} = \text{arg-2} \wedge \\
&\qquad \text{load-arg-1-loc} = \text{sub-loc} \Rightarrow \text{load-arg-1} = \text{sub-result-0} \wedge \\
&\qquad \text{load-arg-1-loc} = \text{load-loc} + 1 \Rightarrow \text{load-arg-1} = \text{load-result-1} \wedge \\
&\quad \text{store-arg-0-loc} = 1 \Rightarrow \text{store-arg-0} = \text{arg-1} \wedge \\
&\qquad \text{store-arg-0-loc} = \text{load-loc} \Rightarrow \text{store-arg-0} = \text{load-result-0} \wedge \\
&\qquad \text{store-arg-0-loc} = \text{store-loc} \Rightarrow \text{store-arg-0} = \text{store-result-0} \wedge \\
&\quad \text{store-arg-1-loc} = 0 \Rightarrow \text{store-arg-1} = \text{arg-0} \wedge \\
&\qquad \text{store-arg-1-loc} = 2 \Rightarrow \text{store-arg-1} = \text{arg-2} \wedge \\
&\qquad \text{store-arg-1-loc} = \text{sub-loc} \Rightarrow \text{store-arg-1} = \text{sub-result-0} \wedge \\
&\qquad \text{store-arg-1-loc} = \text{load-loc} + 1 \Rightarrow \text{store-arg-1} = \text{load-result-1} \wedge \\
&\quad \text{store-arg-2-loc} = 0 \Rightarrow \text{store-arg-2} = \text{arg-0} \wedge \\
&\qquad \text{store-arg-2-loc} = 2 \Rightarrow \text{store-arg-2} = \text{arg-2} \wedge \\
&\qquad \text{store-arg-2-loc} = \text{sub-loc} \Rightarrow \text{store-arg-2} = \text{sub-result-0} \wedge \\
&\qquad \text{store-arg-2-loc} = \text{load-loc} + 1 \Rightarrow \text{store-arg-2} = \text{load-result-1} \wedge \\
&\quad \text{result-0-loc} = 1 \Rightarrow \text{result-0} = \text{arg-1} \wedge \\
&\qquad \text{result-0-loc} = \text{load-loc} \Rightarrow \text{result-0} = \text{load-result-0} \wedge \\
&\qquad \text{result-0-loc} = \text{store-loc} \Rightarrow \text{result-0} = \text{store-result-0}
\end{aligned}
$$

Figure 3.3.: Example semantics of the component set $C = \{\text{sub}, \text{load}, \text{store}\}$ with the symbolic arguments $[\text{arg-0}, \text{arg-1}, \text{arg-2}]$ and the result variable result-0.

it must ensure the precondition of the goal instruction. Thus, we have (eliding the quantifiers):

$$\phi_{wf}(C) \wedge Q^+(E, C, V_a, V_r) \wedge \mathrm{pre}(E, C) \Rightarrow \\ P(g)(V_a, [], V_r) \wedge Q(g)(V_a, [], V_r)$$

where $\mathrm{pre}(E, C)$ collects the preconditions of all components:

$$\mathrm{pre}(E, C) = \bigwedge_{c \in C} P(c)([E\text{-}c\text{-}\mathsf{arg\text{-}0}, \ldots], [c\text{-}\mathsf{internal\text{-}0}, \ldots], [E\text{-}c\text{-}\mathsf{result\text{-}0}, \ldots])$$

In addition, we must ensure that the instructions in C only access valid pointers. Since the set of valid pointers is defined as the set of pointers that g accesses, we do not need to add a constraint to the pointers of g. Including this, we arrive at the complete formula (again eliding quantifiers):

$$\phi_{wf}(C) \wedge Q^+(E, C, V_a, V_r) \wedge \mathrm{pre}(E, C) \Rightarrow \\ \mathrm{vp}(E, C) \wedge P(g)(V_a, [], V_r) \wedge Q(g)(V_a, [], V_r)$$

where $\mathrm{vp}(E, C)$ ensures that all pointers in C are valid:

$$\mathrm{vp}(E, C) = \bigwedge_{c \in C} \bigwedge_{p \in V'(E,c)} \left(\bigvee_{v \in V'(E,g)} p = v \right),$$

and where V' computes the list of valid pointers for E:

$$V'(E, c) = V(c)([E\text{-}c\text{-}\mathsf{arg\text{-}0}, \ldots], [c\text{-}\mathsf{internal\text{-}0}, \ldots], [E\text{-}c\text{-}\mathsf{result\text{-}0}, \ldots])$$

CEGIS

Our final formula is now simple enough to be handled by an SMT solver, but too complex to be solved in an acceptable time. However, it is suitable for a solution using CEGIS.

For CEGIS, we need two separate formulae: One to synthesise a graph pattern, and one to produce a counterexample test case for the pattern. A test case t is the list $V_a(t)$ of its arguments and the set $E(t)$ of variables required for Q^+. We will call the set of all test cases T.

To synthesise a graph pattern, we need to solve the following formula:

$$\begin{aligned} \phi_{synth}(C, T, g) = \exists L(C).\, \exists V_i.\ & \\ & \phi_{wf}(C) \wedge \\ & \bigwedge_{t \in T} \big(\exists E(t).\, \exists V_r.\, \mathrm{pre}(E(t), C) \Rightarrow \\ & \qquad \mathrm{vp}(E(t), C) \wedge Q^+(E(t), C, V_a(t), V_r) \wedge \\ & \qquad P(g)(V_a(t), [], V_r) \wedge Q(g)(V_a(t), [], V_r) \big) \end{aligned}$$

We still need to check for $\text{pre}(E(t), C)$, because the test cases might come from other solutions (i.e. other graph patterns) with different preconditions.

If ϕ_{synth} is satisfiable, solving it gives us a model for all location variables L^* and for the internal attributes V_i^* of the components in C.

Using the models, we can then verify that L^* and V_i^* are correct for all inputs, or obtain a new test case that was not covered previously. For this, we use the following formula:

$$\begin{aligned}\phi_{cex}(C, g, L^*, V_i^*) = \exists L(C).\, \exists V_a.\, \exists V_i.\, \exists V_r.\, \exists V_r'.\, \exists E.\\ L(C) = L^* \wedge V_i = V_i^* \wedge \text{pre}(E, C) \wedge\\ Q^+(E, C, V_a, V_r) \wedge Q(g)(V_a, [], V_r') \wedge\\ (\neg P(g)(V_a, [], V_r) \vee \neg \text{vp}(E, C) \vee V_r \neq V_r')\end{aligned}$$

As we can see from ϕ_{cex}, the test case V_a can be a counterexample in three different ways: It can fail to fulfil the goal's precondition, it can access an invalid pointer, or its results can be different from the goal's results.

In contrast to ϕ_{synth}, we demand that V_a fulfil $\text{pre}(E, C)$ rather than conditioning on it. Even though a V_a that does not fulfil $\text{pre}(E, C)$ is logically a valid counterexample for every possible C, it does not help at all in adding further constraints to ϕ_{synth}.

We can then use ϕ_{synth} and ϕ_{cex} in a CEGIS loop to synthesise a graph pattern that implements the goal instruction g. This is summarised in Algorithm 3.

Algorithm 3 Simple CEGIS

```
1:  procedure SimpleCEGIS(I : {{Instruction}}, g : Instruction)
2:      C ← MakeComponents(I)
3:      T ← ∅                                          ▷ T : {TestCase}
4:      loop
5:          (r1, a1) ← SMTSolve(φ_synth(C, g))
6:          if r1 = unsat then
7:              return []
8:          end if
9:          (r2, a2) ← SMTSolve(φ_cex(C, g, a1(L(C)), a1(Vi)))
10:         if r2 = unsat then
11:             gp ← MakeGraphPattern(a1(L(C)))
12:             return [gp]
13:         else
14:             T ← T ∪ {(a2(Va), fresh variables for E(t))}
15:         end if
16:     end loop
17: end procedure
```

Finding All Patterns

Many instructions have several different possible implementations: For example, the increment instruction $\mathrm{inc}(x)$ has the implementations $x + 1$, $1 + x$, $x - (-1)$, and $-\mathrm{not}(x)$. Since the instruction selector should recognise all these patterns, the synthesiser has to find them all.

To find multiple different patterns, we run the CEGIS algorithm repeatedly, adding an additional constraint to ϕ_{synth} that excludes those patterns from the solution that we have already found. If we have already found the patterns in F, where each element $(l_f, v_f) \in F$ consists of an assignment to the location variables l_f, and an assignment to the internal values v_f, ϕ_{synth} becomes:

$$\begin{aligned}\phi_{synth}(C, g, F) = \exists L(C).\, \exists V_i.\ & \\ & \phi_{wf}(C) \wedge \\ & \bigwedge_{t \in T} \Big(\exists E(t).\, \exists V_r.\, \mathrm{pre}(E, C) \Rightarrow \\ & \qquad \mathrm{vp}(E(t), C) \wedge Q^+(E(t), C, V_a(t), V_i, V_r) \wedge \\ & \qquad P(g)(V_a(t), [], V_r) \wedge Q(g)(V_a(t), [], V_r) \Big) \wedge \\ & \bigwedge_{(l_f, v_f) \in F} (L(C) \neq l_f \vee V_i \neq v_f)\end{aligned}$$

However, we now also find patterns in which we are not interested, because they would have been optimised earlier in the compiler. For example, we might find $\mathrm{not}(\mathrm{not}(x) + 1)$ for $\mathrm{inc}(x)$. Therefore, we add an additional constraint to ϕ_{synth} that sets an upper bound for the locations of the pattern's results, and thereby also for the number of components used in the pattern. Calling this upper bound ℓ (the length of the program), we are then interested in all patterns with the smallest possible ℓ.

Including this constraint then gives us the final version of ϕ_{synth}:

$$\begin{aligned}\phi_{synth}(C, g, F, \ell) = \exists L(C).\, \exists V_i.\ & \\ & \phi_{wf}(C) \wedge \\ & \bigwedge_{t \in T} \Big(\exists E(t).\, \exists V_r.\, \mathrm{pre}(E, C) \Rightarrow \\ & \qquad \mathrm{vp}(E(t), C) \wedge Q^+(E(t), C, V_a(t), V_i, V_r) \wedge \\ & \qquad P(g)(V_a(t), [], V_r) \wedge Q(g)(V_a(t), [], V_r) \Big) \wedge \\ & \bigwedge_{(l_f, v_f) \in F} (L(C) \neq l_f \vee V_i \neq v_f) \wedge \\ & \bigwedge_{i=0}^{|S_r(g)|-1} (\mathsf{result}\text{-}i\text{-}\mathsf{loc} \leq \ell)\end{aligned}$$

The final version of the whole CEGIS algorithm is shown in Algorithm 4. We iterate over ℓ rather than expressing our requirement to find the smallest possible

ℓ logically $(\exists \ell.\, \phi_{synth}(\ell) \wedge \neg \exists \ell'.\, \ell' < \ell \wedge \phi_{synth}(\ell'))$ to give the SMT solver an easier task.

Algorithm 4 CEGIS to find all results

```
1: procedure CEGISAllResults(I : {{Instruction}}, g : Instruction)
2:     C ← MakeComponents(I)
3:     T ← ∅                                                  ▷ T : {TestCase}
4:     F ← ∅                          ▷ F : {LocationVariables × InternalAttributes}
5:     R ← ∅                                                  ▷ R : {GraphPattern}
6:     ℓ ← 1
7:     while R = ∅ do
8:         loop
9:             (r1, a1) ← SMTSolve(φ_synth(C, g, F, ℓ))
10:            if r1 = unsat then
11:                Leave loop
12:            end if
13:            (r2, a2) ← SMTSolve(φ_cex(C, g, a1(L(C)), a1(Vi)))
14:            if r2 = unsat then
15:                F ← F ∪ (a1(L(C)), a1(Vi))
16:                gp ← MakeGraphPattern(a1(L(C)))
17:                R ← R ∪ {gp}
18:            else
19:                T ← T ∪ {(a2(Va), fresh variables E)}
20:            end if
21:        end loop
22:        ℓ ← ℓ + 1
23:    end whilereturn R
24: end procedure
```

Even if we are only interested in one solution, we still need to set an upper bound depending on the SMT solver used. While Gulwani et al. [12] found that Yices [22] always produced minimal patterns without the upper bound, Z3 often produced non-optimal patterns in our setup.

3.3. Optimisations

We have now fully presented the synthesis algorithm. However, this algorithm is still too inefficient. In this section, we will show some ways to improve its performance.

3.3.1. Distribution Of Work

The most important source of inefficiency is that we only have coarse-grained control over the amount of IR instructions used in the SMT queries. In our algorithm, the

multiset I must contain every IR instruction in sufficient number to construct a pattern for each machine instruction. However, every single machine instruction will only use a few of the instructions in I, making the SMT problem unnecessarily large and inefficient to solve.

We can therefore make the synthesis more efficient, if we move the iteration over the IR instructions to use from the SMT query to the driver procedure. There are three approaches to do this.

Iterating Over Multisets

In this approach, we take I to be a simple set and iterate over multisets of elements from I in order of increasing size. To enumerate the multisets, we order the elements of I arbitrarily and iterate over a sequence of indices as shown in algorithm 5.

We then run the CEGIS prodecure using each multiset generated by this iteration as the multiset of available IR instructions. When the CEGIS procedure first returns a solution, we continue to iterate over the remaining multisets of the current size. After that, we have found all patterns with minimal size.

Because the iteration proceeds in order of increasing size anyway, we do not need to constrain the maximum program length in ϕ_{synth} anymore, and we therefore also do not need to iterate over ℓ in the CEGIS procedure.

Algorithm 5 Iteration over all multisets

```
1: procedure MultisetIteration(I : {Instruction}, g : Instruction)
2:     I ← SetToSequence(I)
3:     ℓ ← 1
4:     R ← ∅
5:     while R = ∅ do
6:         K = [0, ..., 0]_{0≤*<ℓ}
7:         loop
8:             I' ← {{I[K[0]], ..., I[K[ℓ − 1]]}}
9:             R ← R ∪ CEGISAllResults(I', g)
10:            k ← max{k | K[k] < ℓ − 1}
11:            if no k exists then
12:                Leave loop
13:            end if
14:            K[k] ← K[k] + 1
15:            K[k + 1, ..., ℓ − 1] ← [K[k], ..., K[k]]
16:        end loop
17:        ℓ ← ℓ + 1
18:    end while
19: end procedure
```

Iterating over Sequences

This approach is similar to iteration over multisets, except that we now also proscribe the order of components in the pattern. We enumerate all sequences of elements from I as shown in Algorithm 6. In addition, we modify the CEGIS procedure to take a sequence instead of a multiset for I, and amend ϕ_{synth} with additional constraints to fix the location of each component:

$$\begin{aligned}\phi_{synth}(C, g, F, \ell) = \exists L(C).\, \exists V_i.\ & \\ & \phi_{wf}(C) \wedge \\ & \bigwedge_{t \in T} \Big(\exists E(t).\, \exists V_r.\, \mathrm{pre}(E, C) \Rightarrow \\ & \qquad \mathrm{vp}(E(t), C) \wedge Q^+(E(t), C, V_a(t), V_i, V_r) \wedge \\ & \qquad P(g)(V_a(t), [], V_r) \wedge Q(g)(V_a(t), [], V_r) \Big) \wedge \\ & \bigwedge_{(l_f, v_f) \in F} (L(C) \neq l_f \vee V_i \neq v_f) \wedge \\ & \bigwedge_{i=0}^{|C|-1} (C[i]\text{-}\mathsf{loc} = \Sigma_{k=0}^{i-1} |S_r(C[k])|)\end{aligned}$$

Because components can have multiple results, we need to sum the number of previous results to find the location of the i-th component. Again, we do not need the maximum program length constraint in the CEGIS procedure.

Iterating over Patterns

In this approach, the driver program generates all patterns and the SMT solver only has to check one specific pattern for equality with the goal. Buchwald has used this approach in the optimiser generator Optgen [23].

We have also evaluated this approach for our work, but did not find it worthwhile, because the iteration over patterns did not scale well (Optgen uses patterns of size up to 3, whereas we need size 5). Additionally, Optgen's approach is designed for an algorithm that works on many patterns at once, which is not our use case.

For these reasons, and because the implementation would differ significantly, we do not include an implementation of Optgen's iteration in our final program, and do not evaluate its performance.

All these optimisations replace a few large SMT queries (one CEGIS run) with more smaller and easier ones. If we have n instructions and want to generate patterns of length ℓ, we need the following number of iterations:

- To iterate over all multisets: $N_m(n, \ell) = \binom{n+\ell-1}{\ell}$
- To iterate over all sequences: $N_s(n, \ell) = n^\ell$

Algorithm 6 Iteration over all sequences

```
1: procedure SequenceIteration(I : {Instruction}, g : Instruction)
2:     I ← SetToSequence(I)
3:     ℓ ← 1
4:     R ← ∅
5:     while R = ∅ do
6:         K = [0, ..., 0]_{0≤∗<ℓ}
7:         loop
8:             I' ← {{I[K[0]], ..., I[K[ℓ − 1]]}}
9:             R ← R ∪ FixedPositionCEGISAllResults(I', g)
10:            k ← max{k | K[k] < ℓ − 1}
11:            if no k exists then
12:                Leave loop
13:            end if
14:            K[k] ← K[k] + 1
15:            K[k + 1, ..., ℓ − 1] ← [0, ..., 0]
16:        end loop
17:        ℓ ← ℓ + 1
18:    end while
19: end procedure
```

For example, if we have $n = 21$ IR instructions, and need patterns up to a size of $\ell = 5$, this amounts to 65 779 iterations for multisets and 4 288 305 for sequences to synthesise one machine instruction. On the other hand, the queries in the multiset iteration are more complex, because the SMT solver needs to assign the components' locations.

3.3.2. Simultaneous Search

In order to reduce the overhead from the large number of SMT queries, we can synthesise several machine instructions simultaneously if they have the same interface (i.e. the same S_a, S_i, and S_r). To do this, we collect the goal instructions in the sequence G, and add an SMT variable goal-index to let the SMT solver choose, for which instruction it synthesises a pattern. Thus, we extend ϕ_{synth} yet again to include

multiple goals and the choice between them.

$$\begin{aligned}\phi_{synth}(C, g, F, \ell) = \exists L(C).\, \exists V_i.\, \exists \textsf{goal-index}.\\ \phi_{wf}(C) \wedge 0 \leq \textsf{goal-index} < |G| \wedge\\ \bigwedge_{t \in T} \Big(\exists E(t).\, \exists V_r.\, \mathrm{pre}(E, C) \Rightarrow\\ \mathrm{vp}(E(t), C) \wedge Q^+(E(t), C, V_a(t), V_i, V_r) \wedge\\ \bigwedge_{i=0}^{|G|-1} (\textsf{goal-index} = i \Longrightarrow\\ P(G[i])(V_a(t), [], V_r) \wedge Q(G[i])(V_a(t), [], V_r))\Big) \wedge\\ \bigwedge_{(l_f, v_f) \in F} (L(C) \neq l_f \vee V_i \neq v_f) \wedge\\ \bigwedge_{i=0}^{|S_r(g)|-1} (\textsf{result-}i\textsf{-loc} \leq \ell)\end{aligned}$$

ϕ_{cex} remains unchanged, because we have already decided on the instruction to synthesise at this point.

The CEGIS algorithm with simultaneous search is shown in Algorithm 7. For each ℓ, we collect the instructions which have an implementation of length ℓ in the set D. When we have found all possible patterns of length ℓ, we remove D from G, so that no more patterns for these instructions are generated.

3.3.3. Multithreading

Our algorithm naturally lends itself to multithreading, because the syntheses of different goals (or, with simultaneous search, sets of goals) are completely independent.

Our implementation uses a fixed pool of threads with a shared list of goals left to synthesise, and a shared list of synthesis results. Threads need only acquire a mutex for a very short time to update these lists, when they have finished a synthesis.

We evaluate the success of all optimisations in Section 4.1.

3.4. Generating the Instruction Selector

Our task is now to implement the function GenerateCode from Algorithm 2 on page 21. Since the implementation of this function is tightly coupled to the target compiler, we present the concrete implementation for x86 on Firm here. In order to describe the code generator, we need to define some new data types:

Graph Patterns

First, we need a more precise definition of graph patterns (Section 2.1.3). A graph pattern is a DAG, where each node is of one of these three kinds:

Algorithm 7 CEGIS with simultaneous search

```
1: procedure SimultaneousCEGISAllResults(I : {{Instruction}}, G : {Instruction} )
2:     G ← SetToSequence(G)
3:     C ← MakeComponents(I)
4:     T ← ∅                                                    ▷ T : {TestCase}
5:     F ← ∅
   ▷ patterns already found – F : {LocationVariables × InternalAttributes}
6:     R ← ∅                                                    ▷ results – R : {GraphPattern}
7:     ℓ ← 1
8:     while G ≠ ∅ do
9:         D ← ∅                ▷ instructions found in this round – D : {Instruction}
10:        loop
11:            (r_1, a_1) ← SMTSolve(φ_synth(C, g, F, ℓ))
12:            if r_1 = unsat then
13:                Leave loop
14:            end if
15:            g ← G[a_1(goal-index)]
16:            (r_2, a_2) ← SMTSolve(φ_cex(C, g, a_1(L(C)), a_1(V_i)))
17:            if r_2 = unsat then
18:                F ← F ∪ {(a_1(L(C)), a_1(V_i))}
19:                gp ← MakeGraphPattern(a_1(L(C)))
20:                R ← R ∪ {(g, gp)}
21:                D ← D ∪ {g}
22:            else
23:                T ← T ∪ {(a_2(V_a), fresh variables E)}
24:            end if
25:        end loop
26:        ℓ ← ℓ + 1
27:        G ← G \ D
28:    end while
29:    return R
30: end procedure
```

- An *instruction node* contains a reference to an instruction and a matching list of internal values. In addition, it has a list of edges pointing to other nodes, whose results are the instruction's arguments.
- A *variable node* represents one of the inputs of the whole pattern. It contains the variable's index in the pattern's argument array, and information on whether the variable has to be a compile-time constant.
- A *projection node* selects one result from an instruction that defines multiple results. We include it in our patterns to make them isomorphic to the matching Firm graphs.

We call an instruction node the *root* of a graph pattern, if all other nodes, except for the root's projection nodes, are reachable from the root. If a pattern p has a root, we call it *rooted*, and write root(p) to refer to the root.

Given a node n in a graph pattern, we use the following functions:

- $v_i(n)$ refers to the list of internal values, if n is an instruction node
- args(n) refers to the list of the n's arguments
- users(n) refers to the list of nodes that have an edge to n

Instruction Data

In the code generator, we need to be able to recognise Firm nodes as instances of certain IR instructions and construct Firm back end nodes. Therefore, we extend each instruction i with two additional attributes that specify the necessary code:

- The predicate generator function $g_p(i)$ takes an expression e and a list of expressions e_i. It returns an expression that evaluates to `true`, if and only if e is an instance of the instruction. e evaluates to a Firm node, and the elements of e_i evaluate to integers that are its internal attributes.
- The constructor generator function $g_c(i)$ takes two lists of expressions e_a and e_i, and an l-value v. It returns a statement that constructs an instance of the instruction in v. The elements of e_a evaluate to Firm nodes that are the instruction's arguments, and the elements of e_i evaluate to integers that are its internal attributes.

For example, the IR instruction sub32 has

$g_p(\text{sub32})(e) =$

```
is_Sub(e) && (get_irn_mode(e) == mode_Is ||
get_irn_mode(e) == mode_Iu || get_irn_mode(e) == mode_P)
```

and the back end instruction ia-submem32 has

g_c(ia-submem32)$(e_a, e_i, v) =$
    v `= new_bd_ia32_SubMem(dbgi, block,` $e_a[0]$`, noreg_GP,` $e_a[1]$`,` $e_a[2]$`);`
    `set_ia32_ls_mode(`v`, mode_Iu);`

The constructor generator g_c may assume that variables called `block` and `dbgi` are in scope, and contain references to the current basic block and debug information respectively. Since these names are frequently used all over Firm, we saw no use in abstracting over them.

Implementation Matters

Because closures are relatively heavyweight objects in C++, we instead represent g_p and g_c with template strings. In our implementation, g_c(ia32-submem) is actually the string

```
$node = new_bd_ia32_SubMem(dbgi, block, $arg0,
                           noreg_GP, $arg1, $arg2);
set_ia32_ls_mode($node, mode_Iu);
```

The "$" characters in that string mark the variables to be replaced.

3.4.1. Overview

We continue our example to generate a subtraction from a value in memory (in Firm parlance, an `ia32_SubMem` node). The pattern that implements this node is shown in Figure 3.4. The numbers annotated at the edges are the indices in the argument array.

The structure of Firm graphs imposes a limitation to the instruction selector we generate: Firm nodes only have pointers to their arguments; traversing patterns from definitions to users requires complicated use of the *out-edges*. In addition, the existing Firm instruction selector also assumes that instruction selection happens recursively from End node towards Start node.

Since we aim to be compatible with the existing instruction selector, we require that all IR patterns be rooted. This way, we can match the pattern when we encounter the root and replace it with the associated machine instruction. All of this does not require traversing the out-edges.

In Algorithm 8, we give an overview of the code generator. First, we sort the elements of S by the following rule:

$$(m_1, p_1) <_{match} (m_2, p_2) :\Leftrightarrow |p_1| > |p_2| \vee (|p_1| = |p_2| \wedge |vars(p_1)| < |vars(p_2)|$$

This ensures that we check for large patterns first, which produce one (presumably efficient) machine instruction from many IR instructions. When patterns have the same size, we prefer the pattern with fewer variables, because we assume that the

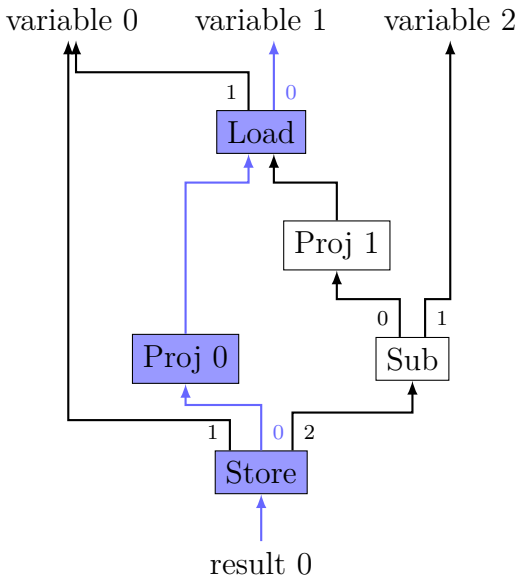


Figure 3.4.: An IR graph pattern that implements a subtraction from a value in memory (`sub %eax, x` in x86; `ia32_SubMem` in Firm).

target machine instruction will have a simpler encoding if it has access to fewer variables.

Then, we emit a function for each pair of pattern and machine instruction. The first part of the function checks that the node we analyse is in fact the root of the pattern. If that is the case, the function constructs an instance of the machine instruction with the arguments taken from the pattern's arguments, after they have themselves been transformed into machine instructions.

Finally, we emit a function that checks every pattern in turn and returns the first match.

3.4.2. Matching IR Patterns

The predicate generator g_p of an instruction does not need to check for the correct arguments, which would require a recursive function with access to the other instructions' g_p. Instead, we can walk the IR graph pattern independent from the types of instructions contained within.

To check whether the pattern p matches the Firm graph rooted at r, we traverse p in pre-order. During the traversal, the variable f holds the expression that evaluates to the equivalent Firm node relative to r. At each node, we have four cases to consider:

- At an instruction node with instruction i, we generate a condition to check for $g_p(i)(f)$. If the predicate does not hold, we abort the match.

Algorithm 8 Overview of the code generator

```
1: procedure GenerateCode(S : {(M × Pattern(I))})
2:                                  ▷ M: machine instructions, I: IR instructions
3:     S ← SortByMatchOrder(S)
4:     for each (m, i) ∈ S do
5:         Emit code: ir_node *transform_i(ir_node *node, ir_node *block,
6:                                         dbg_info *dbgi) {
7:         for each v ∈ [0, ..., |S_a(m)| − 1] do
8:             Emit code:      ir_node *var_v = NULL;
9:         end for
10:        EmitMatchPattern(i, node)
11:        for each v ∈ [0, ..., |S_a(m)| − 1] do
12:            Emit code:      var_v = be_transform_node(var_v);
13:        end for
14:        Emit code:      ir_node *new_node = NULL;
15:                        g_c(m)(new_node, [var_0, ..., var_|S_a(m)| − 1], [])
16:                        return new_node;
17:                    }
18:    end for
19:    Emit code: ir_node *transform_all(ir_node *node) {
20:                   ir_node  *new_node = NULL;
21:                   ir_node  *block    = be_transform_node(
22:                                            get_nodes_block(node));
23:                   dbg_info *dbgi     = get_irn_dbg_info(node);
24:    for each (m, i) ∈ S do
25:        Emit code:      new_node = transform_i(node, block, dbgi);
26:                        if (new_node != NULL) return node;
27:    end for
28:    Emit code: }
29: end procedure
```

Also, if the node is not the pattern's root, we demand that the FIRM node has as many users as the node in the pattern.

- At a projection node, we check for a Proj node on the FIRM side. If there is none, we abort the match.
- When we first encounter a variable node, we accept any node on the FIRM side, and save it as the argument for the back end node. If we encounter the variable node again, we check that the node on the FIRM side is equal to the value we saved.
- An immediate node is handled like a variable node, with the additional constraint, that FIRM must be able to construct an immediate from the value. The x86 back end provides the function `x86_match_immediate` for this purpose.

Algorithm 9 summarises the description above. We have omitted the case for immediate values, since it is very similar to the case for variables.

Example

Returning to our example instruction ia-submem32, we can finally present the part of the instruction selector that matches its pattern and constructs the back end node. The function is shown in Figure 3.5. Because of its length, we have abridged the checks for the correct modes of each instruction node. Each "..." in the listing stands for the following expression:

```
get_irn_mode(node) == mode_Is ||
get_irn_mode(node) == mode_Iu ||
get_irn_mode(node) == mode_P
```

Algorithm 9 Matching a pattern p against the FIRM graph rooted at r

```
1: procedure EmitMatchNode(n : Node, f : Code)
2:     if n is an instruction node with instruction i then
3:         Emit code: if (!g_p(i)(f, InternalsToCode(v_i(n)))) {
4:                         return NULL;
5:                     }
6:         if n is not the pattern's root then
7:             Emit code: if (|users(n)| != get_irn_n_edges(f)) {
8:                             return NULL;
9:                         }
10:        end if
11:        for each i ∈ [0, ..., |args(n)| − 1] do
12:            EmitMatchNode(args(n)[i], get_irn_n(f, i))
13:        end for
14:    else if n is a projection node with projection number pn then
15:        Emit code: if (!is_Proj(f) || get_Proj_num(f) != pn) {
16:                        return NULL;
17:                    }
18:        EmitMatchNode(args(n)[0], get_Proj_pred(f))
19:    else if n is a variable node with variable index i then
20:        Emit code: if (var_i == NULL) {
21:                        var_i = f;
22:                    } else if (var_i != f) {
23:                        return NULL;
24:                    }
25:    end if
26: end procedure
27: procedure EmitMatchPattern(p : Pattern, r : Code)
28:     EmitMatchNode(root(p), r)
29: end procedure
```

```
static ir_node *transform_f50(ir_node *node, ir_node *block, dbg_info *dbgi)
{
  (void)block;
  (void)dbgi;
  x86_imm32_t tmp_imm;
  (void)tmp_imm;
  ir_node *var0 = NULL;
  ir_node *var1 = NULL;
  ir_node *var2 = NULL;

  if (!(is_Store(node) && ... )) {
    return NULL;
  }
  if (!(is_Proj(get_irn_n(node, 0)) && get_Proj_num(get_irn_n(node, 0)) == 0 &&
        get_irn_n_edges(get_irn_n(node, 0)) == 1)) {
    return NULL;
  }
  if (!(is_Load(get_Proj_pred(get_irn_n(node, 0))) && ... &&
        get_irn_n_edges(get_Proj_pred(get_irn_n(node, 0))) == 2)) {
    return NULL;
  }
  if (var1 == NULL || var1 == get_irn_n(get_Proj_pred(get_irn_n(node, 0)), 0)) {
    var1 = get_irn_n(get_Proj_pred(get_irn_n(node, 0)), 0);
  } else return NULL;

  if (var0 == NULL || var0 == get_irn_n(get_Proj_pred(get_irn_n(node, 0)), 1)) {
    var0 = get_irn_n(get_Proj_pred(get_irn_n(node, 0)), 1);
  } else return NULL;

  if (var0 == NULL || var0 == get_irn_n(node, 1)) {
    var0 = get_irn_n(node, 1);
  } else return NULL;

  if (!(is_Sub(get_irn_n(node, 2)) && ... &&
        get_irn_n_edges(get_irn_n(node, 2)) == 1)) {
    return NULL;
  }
  if (!(is_Proj(get_irn_n(get_irn_n(node, 2), 0)) &&
        get_Proj_num(get_irn_n(get_irn_n(node, 2), 0)) == 1 &&
        get_irn_n_edges(get_irn_n(get_irn_n(node, 2), 0)) == 1)) {
    return NULL;
  }
  if (!(is_Load(get_Proj_pred(get_irn_n(get_irn_n(node, 2), 0))) && ... &&
        get_irn_n_edges(get_Proj_pred(get_irn_n(get_irn_n(node, 2), 0))) == 2)) {
    return NULL;
  }
  if (var1 == NULL ||
      var1 == get_irn_n(get_Proj_pred(get_irn_n(get_irn_n(node, 2), 0)), 0)) {
    var1 = get_irn_n(get_Proj_pred(get_irn_n(get_irn_n(node, 2), 0)), 0);
  } else return NULL;

  if (var0 == NULL ||
      var0 == get_irn_n(get_Proj_pred(get_irn_n(get_irn_n(node, 2), 0)), 1)) {
    var0 = get_irn_n(get_Proj_pred(get_irn_n(get_irn_n(node, 2), 0)), 1);
  } else return NULL;

  if (var2 == NULL ||
      var2 == get_irn_n(get_irn_n(node, 2), 1)) {
    var2 = get_irn_n(get_irn_n(node, 2), 1);
  } else return NULL;

  ir_node *new_node = NULL;
  new_node = new_bd_ia32_SubMem
          (dbgi, block,
           is_ia32_Immediate(var0) ? var0 : be_transform_node(var0),
           noreg_GP,
           is_ia32_Immediate(var1) ? var1 : be_transform_node(var1),
           is_ia32_Immediate(var2) ? var2 : be_transform_node(var2));
  set_ia32_ls_mode(new_node, get_irn_mode(get_Store_value(node)));
  return new_node;
}
```

Figure 3.5.: The matcher function that matches an ia-submem32 instruction. Checks for the correct modes of instruction nodes have been abridged.

4. Evaluation

To evaluate the success of our work, we will consider each of the following questions in turn:

- Does the synthesis run with acceptable performance? What is the effect of the optimisations from Section 3.3?
- Which kinds of IR nodes can the synthesised instruction selector handle?
- How much of the existing instruction selector can be replaced with our work?
- Does our approach have advantages over specification languages (e.g. TableGen)?
- How good is the machine code that the synthesised instruction selector produces?
- What is the performance of the synthesised instruction selector when compiling?

4.1. Synthesis Performance

First of all, we must set ourselves a more concrete goal: Synthesising a new instruction selector only needs to be done when the IR or the machine language change, or after a specification bug has been found. While repairing a bug, a full synthesis is not necessary. Therefore, our goal is that the full synthesis should run "over night", i.e. in at most 12 hours.

For the performance benchmarks, we use the following set of IR instructions as available components: Add, And, Cmp (signed/unsigned), Cond, Const, Eor, Jmp, Load, Minus, Mux, Mul, Mulh (signed/unsigned), Not, Or, Shl, Shr, Shrs, Store, Sub. All instructions, as well as the machine instructions we synthesise in this section, use 32-bit arithmetic.

We ran all except the multithreading benchmarks on an Intel Core i5-4200U processor with 12 GB of RAM. Since this is only a dual-core processor, we ran the multithreading benchmarks on a quad-core Intel Core i5-750 with 20 GB of RAM.

4.1.1. Optimisations

We have seen very quickly that the non-iterative CEGIS algorithm does not perform well at all as the number of available IR instructions grows. With 18 IR instructions

to choose from (the set above without Load and Store), even the synthesis of the most simple machine instructions did not finish within 90 minutes. This clearly does not meet our goal. Thus, we definitely require optimisations.

Search Strategies

First, we compare the two remaining search strategies, namely iterating over multisets and iterating over sequences (see Section 3.3.1). With each strategy, we synthesised the following machine instructions, which take from 1 to 5 IR instructions to represent:

IR size	benchmark sample
1	subtract (sub)
2	decrement (dec)
2	compare and jump if greater (cmp, jg)
3	subtract from memory (sub (memory))
4	subtract from memory with indexed addressing (sub (memory, indexed))
4	decrement value in memory (dec (memory))
4	bitwise rotate left (rol)
5	decrement value in memory with indexed addressing (dec (memory, indexed))

The results of this benchmark can be found in Table 4.1. We can see that iteration over multisets performs better than iteration over sequences, especially when the patterns (and therefore the search space) get large. While we could iterate over all multisets of size 5 out of 20 IR instructions in about 20 minutes, iterating over all sequences did not finish within 6 hours.

In addition, we can see that the Rol instruction is harder to synthesise than the other nodes with 4 IR instructions. This is due to two reasons: Firstly, the computation that this node performs is more complicated than a read-modify-write operation. Secondly, the other nodes all access memory, and the SMT solver can quickly rule out any combination of IR nodes that does not contain a Load or Store node.

Simultaneous Search and Multithreading

Having found a search strategy, we now evaluate the usefulness of simultaneous search (see Section 3.3.2). To do this, we need a larger set of goal instructions, because the simultaneous search is intended to increase performance if multiple instructions have the same interface.

We therefore use our full set of 32-bit instructions as the goal set. Many of the instructions have three variants: Simple register-register operation, destination operand in memory, and destination operand in memory with indexed addressing. These instructions are marked with an asterisk in the following list.

Goal instruction	IR size	Multiset	Sequence
sub	1	0.26 s	0.27 s
dec	2	2.02 s	3.26 s
cmp, jg	2	1.78 s	2.93 s
sub (memory)	3	17.70 s	93.70 s
sub (memory, indexed)	4	182.78 s	2 206.06 s
dec (memory)	4	125.53 s	2 051.89 s
rol	4	1 232.58 s	3 941.31 s
dec (memory, indexed)	5	1 128.15 s	*

Table 4.1.: Performance comparison of iteration over multisets and iteration over sequences. Iteration over sequences did not finish within 6 hours (21 600 s) for decrement in memory with indexed addressing.

Number of threads	Not simultaneous	Simultaneous
1	10 729 s	9 401 s
2	6 541 s	5 067 s
3	5 232 s	4 872 s
4	5 041 s	4 472 s

Table 4.2.: Performance comparison with and without simultaneous search, using one to four threads.

These are the instructions we use as goals for the benchmark (see [21] for detailed descriptions): lea/add*, and*, or*, xor*, sub*, inc*, dec*, not*, shl*, shr*, sar*, rol, ror, andn, btc, btr, bts, blsi, blsr. In addition, we add a compound instruction consisting of a compare and each of the following condition jumps: jb, jae, je, jne, jbe, ja, js, jns, jl, jge, jle, jg.

We also test the benefit of multithreading in this section. Because simultaneous search merges several syntheses into one, there are fewer, but larger packets of work for the threads, so that simultaneous search may have a negative impact on multithreading.

Therefore, we run the synthesis of the instruction set above on one to four threads, and with and without simultaneous search. The results can be found in Table 4.2.

We can conclude from this that simultaneous search and multithreading do indeed have a positive effect on performance. However, the speedup due to multithreading is markedly sublinear. This is because there is often one long synthesis left to do at the end, which cannot be done in parallel to others.

Benchmark	Synthesis	Call etc.	Phi etc.	Fallback	Ratio
164.gzip	9 570	911	2 004	5 442	0.637
175.vpr	27 244	4 281	3 819	14 310	0.656
176.gcc	63 367	10 008	6 420	16 020	0.798
181.mcf	3 451	168	454	802	0.811
186.crafty	15 367	844	1 833	5 882	0.723
197.parser	44 406	5 683	7 715	20 884	0.680
253.perlbmk	25 897	2 957	3 052	6 907	0.789
254.gap	156 300	18 219	25 228	82 833	0.654
255.vortex	108 084	23 869	12 707	74 234	0.593
256.bzip2	10 853	1 330	1 580	3 766	0.742
300.twolf	55 723	4 112	8 091	20 527	0.731
Total	520 262	72 382	72 903	251 607	0.674

Table 4.3.: Number of nodes transformed by the synthesised instruction selector ("Synthesis") and the existing instruction selector ("Fallback") for each benchmark. Call nodes and their Proj nodes are counted as "Call etc."; Phi nodes and Sync nodes are counted as "Phi etc". "Ratio" gives the ratio between "Synthesis", and "Synthesis" plus "Fallback" (i.e. all nodes except Calls and Phis).

4.2. IR Coverage

For this and the following section, we need a set of source code files to use as benchmarks. Since we do not synthesise any floating-point instructions, these should be purely integer programs. We therefore chose the SPEC CINT2000 integer benchmarks [24].

We instrumented the compiler to log each call to `be_transform_node` (see [4]). The log contains the kind of IR node to transform, whether the transformation was done by our instruction selector or fell back to the existing one, and which rule of our instruction selector has matched.

We exclude two kinds of Firm nodes from our evaluation, because they are impossible to specify: First, we cannot specify Call nodes with their complicated calling conventions, and second, Phi and Sync nodes are variadic. Although their function is simple, we cannot express variadicity in our specifications.

The results are shown in Table 4.3. Our synthesised instruction selector can transform between 59.3 % and 81.1 % of all nodes in the individual benchmarks, and 67.4 % of all nodes in total.

We should also look at those nodes that the synthesised instruction selector should be able to transform, but failed. To do this, we collected all those nodes from all benchmarks, and grouped them by their type. Table 4.4 on Page 59 shows the ten most common node types that we could not transform, and how often they occurred.

There are three groups of node types that we have trouble transforming:

- Load nodes that load from a smaller value than 32 bit. See Section 5.2.1 for further discussion of this topic.
- Cond nodes whose condition does not match one of the "compare and jump" patterns. To match these nodes, we need to add more machine instructions that produce flags, such as the "test" instruction. This can be done with little effort.
- Constants and addresses (Const, Address, Member). These nodes may contain symbolic values, which are only later resolved by the linker. We cannot capture these in our specifications.

Rank	Node	Count
1	Proj Load	40 560
2	Proj Cond	24 838
3	Const	22 544
4	Conv	20 559
5	Load	18 137
6	Proj Proj	17 419
7	Cmp	13 516
8	Cond	12 419
9	Address	11 862
10	Member	10 448

Table 4.4.: Most frequent node types that the synthesised instruction selector could not transform. If the node is a Proj, its predecessor is also given.

4.3. Replacing the Instruction Selector

In this section, we would like to find out how much of the existing instruction selector we can replace with the synthesised instruction selector. To do this, we compile the benchmarks twice, and record the code coverage of the existing instruction selector each time.

First, we compile with the synthesised instruction selector deactivated. This gives us the amount of code in the existing instruction selector that is needed to compile the benchmark. This amount of code is our baseline.

Then, we compile the benchmark with the synthesised instruction selector activated. If the synthesised instruction selector cannot transform an IR node, it falls back to the existing instruction selector. If the synthesised instruction selector can fully take over one task (e.g. transforming additions) from the existing instruction selector, a

Benchmark	Synthesised selector off	Synthesised selector on	Delta	Delta / off
164.gzip	51.93 %	48.68 %	3.25 %	6.26 %
175.vpr	55.75 %	49.98 %	5.77 %	10.35 %
176.gcc	64.60 %	60.71 %	3.89 %	6.02 %
181.mcf	41.14 %	30.53 %	10.61 %	25.79 %
186.crafty	57.39 %	52.85 %	4.54 %	7.91 %
197.parser	47.15 %	41.15 %	6.00 %	12.73 %
253.perlbmk	61.41 %	57.92 %	3.49 %	5.68 %
254.gap	47.83 %	43.94 %	3.89 %	8.13 %
255.vortex	58.24 %	56.14 %	2.10 %	3.61 %
256.bzip2	51.60 %	47.54 %	4.06 %	7.87 %
300.twolf	55.55 %	52.03 %	3.52 %	6.34 %

Table 4.5.: Comparison of code used in the existing instruction selector, without the synthesised instruction selector, and with the synthesised instruction selector. Numbers give the percentage of lines of code executed at least once. "Delta" is the absolute amount of code that the synthesised instruction selector could replace, "Delta / off" is the amount of code the instruction selector could replace relative to the total amount of code used without it.

part of the existing code will not be used anymore. This difference in code coverage is a measure for the amount of work that the synthesised instruction selector can take over.

The coverage results are shown in Table 4.5. Unfortunately, even though we can synthesise a large portion of the benchmarks' nodes (see Section 4.2), there are still special cases for which we need to fall back to the existing instruction selector. Therefore, the difference in coverage is quite small for all benchmarks.

In addition, the Firm code is of course well-engineered and avoids code duplication: Code for integral and floating-point types is shared in the instruction selector wherever possible. Therefore, if a benchmark contains e.g. a floating-point addition, the code to transform an addition is still executed in the existing instruction selector, even though we could transform all integral additions with the synthesised instruction selector.

4.4. Specification and Synthesis

Many of the syntheses are straightforward. After all, there is no way to implement an addition, but with another addition. This level of specifying rules for instruction selection is also possible in existing specification languages such as TableGen.

However, our approach should be more flexible for complex instructions that

blsi x	$x \wedge -x$
blsr x	$x \wedge (x + (-1))$ $x + (x \vee -x)$ $x \wedge (x - 1)$ $x \wedge (x \oplus -x)$ $x \oplus (x \wedge -x)$ $x \wedge \sim(-x)$ $x - (x \wedge -x)$ $-x \oplus (x \vee -x)$ $(x \vee -x) - (-x)$
andn x, y	$y \oplus (x \wedge y)$ $\sim x \wedge y$ $y - (x \wedge y)$ $x \oplus (x \vee y)$
btc x, y	$x \oplus (1 << y)$
bts x, y	$x \vee (1 << y)$
btr x, y	$x \wedge (-1 + (-1 << y))$ $x \oplus (x \wedge (1 << y))$ $x \wedge (x \oplus (1 << y))$ $x \wedge \sim(1 << y)$ $x - (x \wedge (1 << y))$ $(x \vee (1 << y)) \oplus (1 << y)$ $(x \vee (1 << y)) - (1 << y)$

Table 4.6.: Implementations found for the BMI and BMI2 instructions. $\wedge$ and $\vee$ are bitwise operations [21], $\sim$ is bitwise negation, $\oplus$ is exclusive or.

do not have direct equivalents in the IR. We have chosen the Intel extension "Bit Manipulation Instructions" (BMI and BMI2) as a benchmark.

We have specified each instruction naively in terms of what happens to the individual bits. For example, the instruction "blsr" (reset lowest set bit) is specified by a definition in 32 cases to search for the lowest set bit.

Our synthesiser can still generate efficient IR implementations for these instructions, which we show in Table 4.6 on Page 61. To keep the table shorter, we have omitted variants in which only the arguments of a commutative operation are swapped. The operators $\wedge$ and $\vee$ represent the bitwise rather than logical operations; $\sim$ represents bitwise negation, and $\oplus$ represents exclusive or.

Benchmark	Instruction count old	Instruction count new	Increase	Execution time old	Execution time new	Slowdown
164.gzip	11 959	14 956	1.25	0.484 s	0.515 s	1.07
175.vpr	41 834	51 565	1.23	0.009 s	0.009 s	1.00
176.gcc	569 632	71 2405	1.25	0.361 s	0.405 s	1.12
181.mcf	3 123	4 470	1.43	0.050 s	0.058 s	1.17
197.parser	53 889	66 664	1.24	0.766 s	0.875 s	1.14
253.perlbmk	241 041	294 117	1.22	0.239 s	0.248 s	1.04
254.gap	186 480	223 119	1.20	0.211 s	0.232 s	1.10
255.vortex	193 187	218 172	1.13	0.045 s	0.045 s	1.00
256.bzip2	12 741	15 531	1.22	0.663 s	0.663 s	1.10
300.twolf	55 689	72 133	1.30	0.060 s	0.064 s	1.07

Table 4.7.: Quality of code generated by the existing ("old") and the synthesised instruction selector ("new"). "Instruction count" is the number of machine language instructions in the compiled binary, "Execution time" is the average runtime of 100 "isok" checks.

4.5. Code Quality

We now want to compare the machine code that our instruction selector generates with what the existing instruction selector generates.

For this, build the SPEC CINT2000 benchmarks with and without the synthesised instruction selector activated. Then, we compare the number of instructions in the resulting binary, and the execution time of the binary. To measure execution time, we average over 100 runs of the "isok" check.

We have to skip the benchmark "186.crafty", because the compiler miscompiled it into an infinite loop with or without the synthesised instruction selector. In addition, we exclude the tests "sleep.t" and "time.t" from the benchmark "253.perlbmk", because they run idle for four seconds.

See Table 4.7 for the results. The execution time results for the benchmarks "175.vpr" and "255.vortex" are unusable, because they ran for too short a time. We can see that the code produced by the synthesised instruction selector is 24.7 % larger on average and takes 10.1 % longer to execute. The coefficient of variation of the execution times is at most 3.4 %, except for "175.vpr" (9.2 %) and "255.vortex" (43.3 %).

This degradation is due to the fact that the synthesised instruction selector cannot match the IR patterns of x86 instructions that load one of their operands from memory (e.g. `add (%esp), %eax`). We have no problems finding a pattern for this type of instruction, but this pattern is not rooted (see Section 3.4.1). Therefore, the code generator cannot create a matching function for this pattern. The existing instruction selector contains a workaround for this special case, which we cannot replicate in the synthesised instruction selector.

Benchmark	old	new	Slowdown
164.gzip	120 ms	260 ms	2.16
175.vpr	385 ms	768 ms	1.99
176.gcc	6 175 ms	11 982 ms	1.94
181.mcf	31 ms	78 ms	2.49
186.crafty	434 ms	869 ms	2.00
197.parser	560 ms	1 200 ms	2.14
253.perlbmk	2 640 ms	5 537 ms	2.10
254.gap	1 983 ms	4 131 ms	2.08
255.vortex	2 070 ms	3 810 ms	1.84
256.bzip2	127 ms	256 ms	2.02
300.twolf	577 ms	1 211 ms	2.10

Table 4.8.: Performance of the synthesised instruction selector ("new") compared to the existing one ("old"). Times have been reported by the FIRM timing utility for the "codegen" step.

4.6. Performance of the Instruction Selector

Finally, we evaluate the performance of the synthesised instruction selector itself. FIRM already has a timing ultility that can report the time taken for the different compilation steps. The step called "codegen" measures the time spent in the instruction selector.

Table 4.8 shows the results of our measurements. We can see that the synthesised instruction selector takes about twice as long as the existing instruction selector. However, we did not focus on building an efficient instruction selector in our work, so these values are acceptable. In particular, the slowdown does not increase with program size.

5. Conclusion

In this work, we have constructed a synthesiser that produces an instruction selector from specifications of a compiler's intermediate representation and a machine language. This instruction selector is able to transform a majority of the code of the SPEC CINT2000 benchmarks.

Like the old instruction selector in Firm, our instruction selector is a greedy DAG matcher. Still, it produces noticeably slower code. This is due to the fact that the old instruction selector contains a workaround to overcome its architectural limitations in an important special case, namely instructions which load one operand from memory. Since we generate our instruction selector without global knowledge of all patterns and their interactions, we cannot insert such a workaround.

The synthesis itself has good performance. This means that we can add more instructions to be synthesised without performance problems. However, adding new IR instructions means an exponentially larger search space, and we cannot be sure how many more IR instructions the synthesis can support with acceptable performance.

Of course, our work is not definite. There are several areas where improvement is still possible.

5.1. Limitations

First, in some cases we are restricted by the available technology in SMT solving or by the interfaces we have to conform to.

5.1.1. Floating-Point Arithmetic

We did not cover floating-point arithmetic at all in our work, because there is no efficient way to use it in an SMT query at present. The SMT-LIB project has defined a theory [7, 25], and the SMT solver Z3 does already support floating-point arithmetic [26], but uses a "bit-blaster" to implement it.

A bit-blaster takes an SMT query and, without further optimisation, translates it to a SAT query, which it then solves. Because any knowledge of the underlying arithmetic is lost in this process, the performance is not acceptable for our needs.

5.1.2. Non-Rooted Patterns

We have already seen in Section 4.5 that our inability to work with non-rooted patterns is a problem for code generation. This is due to the need for compatibility

with the pre-existing Firm instruction selector. If we were to design an instruction selector from scratch, we could avoid this problem.

For example, we could separate the matching of patterns and the actual transformation into two phases. This way, the matcher can correct earlier "mistakes" that it made when it encountered a pattern through a non-root node.

5.2. Further Work

There are also a number of items where our program could be extended without the need for external advances.

5.2.1. Other Bit Widths

We only generate 32-bit-wide instructions, because we would have to specify many more IR instructions otherwise and slow down synthesis performance. In any case, there should be no need for most instructions to synthesise them for 32 bits and smaller bit widths separately, because they have the same behaviour.

For example, a 32-bit-wide addition can also implement a 16- or 8-bit-wide addition. Even if the higher bits of the input are unknown, the relevant bits of the output will be correct. However, this is not true for all instructions. We may not transform a 16-bit-wide right shift to a 32-bit-wide right shift, because in this case the unknown higher bits would be shifted into the result.

To conclude, we have not found a model that can both exploit the possibilities for cases like addition, and still give correct results for cases like right-shifts. The usual approach, for example used in Souper [13], is to provide all input data in sign-extended or zero-extended form. We cannot do this, because in our case the higher bits are arbitrary.

5.2.2. More Efficient Selection

The instruction selector as described in Section 3.4 is quite inefficient, because it simply tries every known pattern in turn for every input IR node. We could speed up this search in two ways:

Firstly, we could use a lookup table to quickly associate the type of the root node with just a few possible patterns. This saves us the iteration over all the others.

Secondly, if patterns overlap, we can factor out their common part and only match it once. If the common part does not match, we can rule out several patterns at once.

We did not implement these optimisations, because we concentrated on the synthesis.

5.3. Outlook

In future, we expect machine languages to become more dynamic as instruction sets are extended more frequently, and in application specific ways. In addition, research is currently done on "reconfigurable computing", which combines general-purpose processors with components (e.g. FPGAs) that can be configured at runtime to support the current application. [27]

We hope that our work may be of use in this area: By combining a hand-written instruction selector for the general-purpose processor, and a synthesised instruction selector for the new instruction set extension or the application-specific instructions, compilers can rapidly adapt to new instruction sets, but not lose the possibility of hand-optimising the instruction selector used for the majority of the program.

Bibliography

[1] ISO, *ISO/IEC International Standard 9899:1999: Programming languages – C*. International Organization for Standardization, Geneva, Switzerland, 1999.

[2] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pp. 12–27, ACM, 1988.

[3] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," in *Compiler Construction* (R. Jhala and K. Bosschere, eds.), vol. 7791 of *Lecture Notes in Computer Science*, pp. 102–122, Springer, 2013.

[4] "libfirm API (latest version)." `http://pp.ipd.kit.edu/firm/api_latest/`. Retrieved: 20 Jul. 2016.

[5] D. R. Koes and S. C. Goldstein, "Near-optimal instruction selection on DAGs," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'08)*, (Washington, DC, USA), IEEE Computer Society, 2008.

[6] G. H. Blindell, "Survey on instruction selection: An extensive and modern literature review," *CoRR*, vol. abs/1306.4898, 2013.

[7] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB) – Theories." `http://smtlib.cs.uiowa.edu/theories.shtml`. Retrieved: 01 Aug. 2016.

[8] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[9] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[10] Y. V. Matiyasevich, "Diophantine representation of enumerable predicates," *Mathematical notes of the Academy of Sciences of the USSR*, vol. 12, no. 1, pp. 501–504, 1972.

[11] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, (Los Alamitos, CA, USA), pp. 122–126, IEEE Computer Society Press, 1987.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 62–73, ACM, 2011.

[13] P. Collingbourne, J. Regehr, R. Sasnauskas, *et al.*, "Souper, a superoptimizer for LLVM IR." `https://github.com/google/souper`. Retrieved: 20 Jul. 2016.

[14] "GCC machine description documentation." `https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html`. Retrieved: 20 Jul. 2016.

[15] "TableGen documentation." `http://llvm.org/docs/TableGen/`. Retrieved: 20 Jul. 2016.

[16] J. Dias and N. Ramsey, "Automatically generating instruction selectors using declarative machine descriptions," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, (New York, NY, USA), pp. 403–416, ACM, 2010.

[17] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," in *Programming Language Design and Implementation (PLDI)*, ACM, June 2016.

[18] C. Sinz, S. Falke, and F. Merz, "A precise memory model for low-level bounded model checking," in *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2010.

[19] "LLVM language reference – instruction reference." `http://llvm.org/docs/LangRef.html#instruction-reference`. Retrieved: 20 Jul. 2016.

[20] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 1. December 2015.

[21] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 2. December 2015.

[22] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)* (A. Biere and R. Bloem, eds.), vol. 8559 of *Lecture Notes in Computer Science*, pp. 737–744, Springer, July 2014.

[23] S. Buchwald, "Optgen: A generator for local optimizations," in *Compiler Construction* (B. Franke, ed.), Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2015.

[24] Standard Performance Evaluation Corporation, "CINT2000 (Integer Component of SPEC CPU2000)." `https://www.spec.org/cpu2000/CINT2000/`. Retrieved: 01 Aug. 2016.

[25] P. Rümmer and T. Wahl, "An SMT-LIB theory of binary floating-point arithmetic," in *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[26] Microsoft Corporation, "Z3 Theorem Prover – Release notes." `https://github.com/Z3Prover/z3/blob/master/RELEASE_NOTES`. Retrieved: 01 Aug. 2016.

[27] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, pp. 18:1–18:28, May 2011.

Erklärung

Hiermit erkläre ich, Andreas Fried, dass ich die vorliegende Masterarbeit 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

A. Appendix

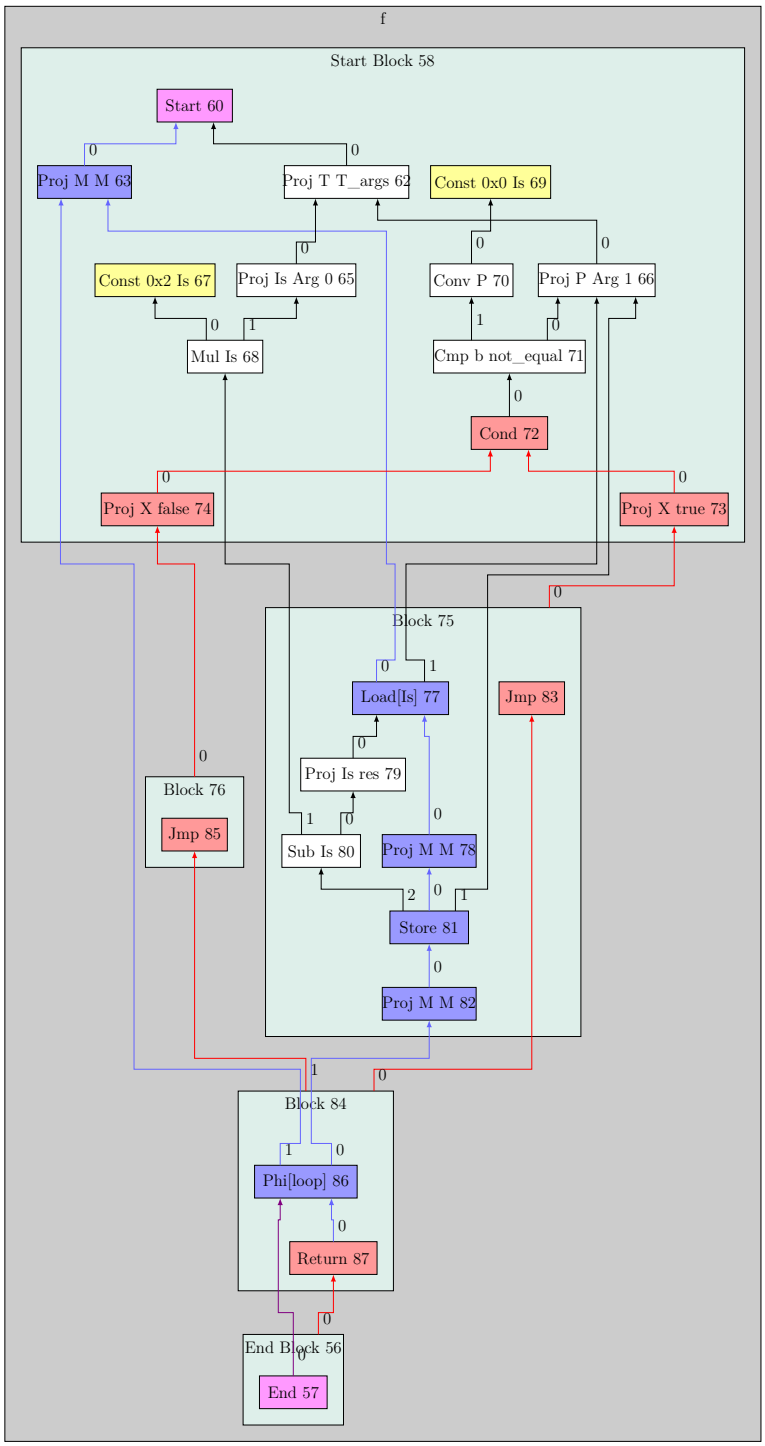


Figure A.1.: FIRM graph from Figure 2.1