Towards Universal Code Generator Generation

Timothy Richards, Edward K. Walters II, J. Eliot B. Moss, Trek Palmer, and Charles C. Weems
Department of Computer Science, University of Massachusetts Amherst

Abstract

One of the most difficult tasks a compiler writer faces is the construction of the code generator. The code generator is that part of the compiler that translates compiler intermediate representation (IR) into instructions for a target machine. Unfortunately, implementing a code generator "by hand" is a difficult, time consuming, and error prone task. The details of both the IR and target instruction set must be carefully considered in order to generate correct and efficient code. This, in turn, requires an expert in both the compiler internals as well as the target machine. Even an expert, however, can produce a code generator that is difficult to verify and debug.

In this paper we present a universal approach for automating the construction of correct code generators. In particular, we show that both the compiler IR and target instruction set semantics can be described by a machine description language and leveraged by a heuristic search procedure to derive code generator patterns. We then utilize formal methods to determine if the IR and target sequence pairs that make up these patterns are semantically equivalent.

1. Introduction

The construction of efficient code generators for modern compilers and modern architectures is a formidable task. Not only must a code generator implement high-level programming language constructs in the target instruction set correctly, but it must also use the most efficient instructions to perform the job. As such, the implementor must be well versed in the compiler internals, the semantics of the compiler IR, and be an expert in the target architecture. Unfortunately, even with an expert it is still time consuming and error-prone to build a code generator.

What is required is a method for generating the code generator component of a compiler automatically. In addition, the solution must be *universal* with respect to the compiler framework as well as the target machine. That is, a given solution must not depend on the particulars of any compiler or architecture. Furthermore, the correctness of the code generator (and thus the correctness of the generated code) is crucial and the approach must be practical for widespread adoption. In this work, we present a practical solution that enables code generator construction to occur automatically, correctly, and independent of the compiler and target processor.

To be compiler and target independent requires a lingua franca that serves as a bridge between the semantics of the compiler intermediate representation (IR) and that of the target machine. More specifically, we need a common language that allows us to specify accurately the meaning of any machine (real or virtual) and the details of its instruction set. We use the CoGenT Instruction Set Language (CISL) [14, 16, 15] to accomplish this task.

Our approach uses the CISL language to describe the memory structures and instruction set semantics of both the compiler IR as well as the target. These semantic descriptions are then leveraged by a heuristic search procedure to "guess" efficiently *potential* code generator patterns (that is, a sequence of target instructions that possibly implement the semantics of the given IR). Given a *potential* code generator pattern we apply a set of test vectors to further filter out incorrect patterns and increase our confidence level. Those patterns that pass the set of test vectors are then subject to a verification stage that attempts to prove rigorously that the target sequence correctly implements the semantics of the IR. Correct patterns may then be used to generate the code generator component for the compiler.

We organize the rest of the paper as follows: Section 2 provides a brief overview of our approach; Section 3 describes the universal code generator generation technique in detail; Section 4 describes related work; and Section 5 concludes.

2. Overview

The goal of our universal code generator generator approach can be precisely formulated as follows: given a semantic description of a compiler IR and target machine, *find* target instruction sequences that *correctly* implement the source IR. Considering the large number of possible operations available to an instruction set and available addressing modes this might appear to be a lofty goal at first glance. Our approach, however, illustrates how we can quickly reduce the number of possible target sequences we must consider to make this feasible.

In searching for target instruction sequences that imple-

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

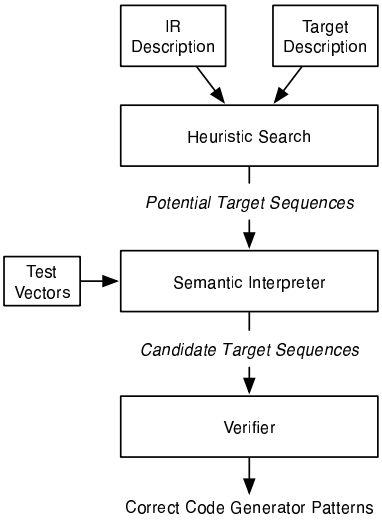


Figure 1. Universal CGG

ment a particular IR instruction we need to make a distinction between those sequences that are clearly incorrect (i.e., do not implement the compiler IR exactly) and those that are possibly correct. In most cases, it is easy to determine if a target sequence does not perform the same operation as an IR instruction. For example, an IR operation that performs an addition is clearly not the same as a target instruction that performs a multiplication (operator comparison).

These observations form the basis for our heuristic search procedure whose primary goal is to reduce the number of possible target sequences quickly. To make these observations automatically requires information about the semantics of the instructions themselves. As it turns out, a CISL semantic description of an instruction provides a wealth of instruction properties that can be transcribed into heuristics that can be used by our search procedure. We describe these features in more detail in Section 3.3.

Our approach uses a "generate-and-test" methodology. That is, we use heuristic search to generate quickly possible target sequences based on simple properties of an instruction. This produces a set of target sequences that are *similar* to the given IR operation. To further eliminate the number of incorrect sequences we use a test phase to interpret the semantic descriptions of the IR and target sequence on a set of test vectors. A comparison of the resulting simulated states is used to separate out those target sequences that failed to produce an identical output state. Naturally, to do this correctly requires one to specify an equivalence between the IR and target memories. Issues related to interpretation and memory mapping are outlined in Section 3.4.

The above procedures indicate how we might find possible code generator patterns quickly. In particular, they serve as a fast procedure for filtering out the obviously bad sequences. We are concerned, however, with *correct* code generator patterns. A correct code generator pattern is one for which we can guarantee that the target sequence implements identically the semantics of the given IR instruction. That is, using rigorous methods, we can prove that they are indeed equivalent. This is accomplished by a verification step that uses techniques such as boolean satisfiability and term rewriting systems to guarantee that the source IR and target sequence are semantically equivalent. An overview of our approach is illustrated in Figure 1.

3. Universal Code Generator Generation

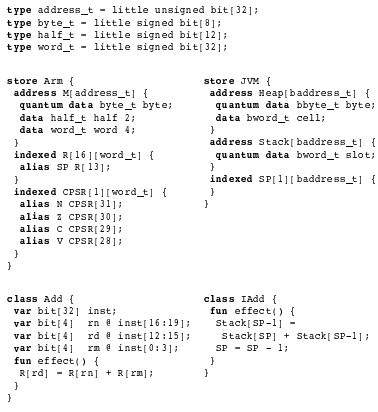
In this section we describe the details of universal code generator generation. We begin by describing how we model instruction semantics in the CISL language. We then show how the heuristic search procedure leverages these semantic descriptions to generate target instruction sequences. Next, we illustrate how target sequences are filtered by a set of test vectors using semantic interpretation to produce a refined set of possibly correct candidates. Lastly, we outline our techniques for verifying semantic correctness between the IR and target sequence.

3.1. Modeling Instruction Semantics

Automatically generating a code generator and other machine oriented software requires a clear definition of the machine. This definition must be rigorous enough to be manipulated by a computer program, yet flexible enough to allow us to define a wide variety of machines. This includes the definition not only of processors such as the x86 [10] or PowerPC [13] but also virtual machines such as those based on a compiler's IR. Extending our notion of a model to incorporate both is not only novel but also allows us to remain compiler and architecture agnostic, a crucial requirement for the success of a universal CGG system.

Many machine formalisms have been devised since the advent of the processor. Most have grown out of needs of particular applications. For this work, the requirements include definitions that allow us to generate code generators. Although our formalism also supports automatic generation of other systems related tools, such as functional simulators and assemblers [18], here we focus on aspects related to code generator generation.

Because most (if not all) elements of a machine can be described in terms of bits, CISL provides as its fundamental data type the *bit array*. A bit array can be used to model such things as an instruction word as well as structures such as main system memory, register files, and condition codes. In addition, bit arrays can take on attributes that modify how they are interpreted. In particular, bit arrays may be specified as being *little-* or *big-endian* as well as *signed* or *unsigned*. This not



```
type address_t = little unsigned bit[32];
type byte_t = little signed bit[8];
type half_t = little signed bit[12];
type word_t = little signed bit[32];

store Arm {
 address M[address_t] {
  quantum data byte_t byte;
  data half_t half 2;
  data word_t word 4;
 }
 indexed R[16][word_t] {
  alias SP R[13];
 }
 indexed CPSR[1][word_t] {
  alias N CPSR[31];
  alias Z CPSR[30];
  alias C CPSR[29];
  alias V CPSR[28];
 }
}

class Add {
 var bit[32] inst;
 var bit[4]  rn @ inst[16:19];
 var bit[4]  rd @ inst[12:15];
 var bit[4]  rm @ inst[0:3];
 fun effect() {
  R[rd] = R[rn] + R[rm];
 }
}
```

```
store JVM {
 address Heap[baddress_t] {
  quantum data bbyte_t byte;
  data bword_t cell;
 }
 address Stack[baddress_t] {
  quantum data bword_t slot;
 }
 indexed SP[1][baddress_t] {
 }
}

class IAdd {
 fun effect() {
  Stack[SP-1] =
   Stack[SP] + Stack[SP-1];
  SP = SP - 1;
 }
}
```

Figure 2. CISL Description Example

only makes it easier to transcribe directly from an architecture's manual, but is important when comparing storage structures between IR and target machine.

In Figure 2 we show how we might specify the storage locations and instruction semantics for the ARM [19] and Java Virtual Machine (JVM) [11]. We use the JVM as an example of a compiler IR as its semantics have been widely published and can be thought of as an intermediate form that can be compiled to real target machines (as it is often the case in modern JIT-enabled JVMs). Although CISL provides modern programming features such as classes, inheritance, and mixins [16, 15, 18], we highlight only those aspects important for this work.

Because certain bit structures (such as an array of 32 bits) show up frequently in a description, CISL allows one to define named types. One may then use these named types as a short-hand throughout the rest of a description. The example shows how we might specify named types for common bit arrays such as *bytes* (`byte_t`) and *words* (`word_t`). To conserve space, we have omitted the big-endian versions (those named types that are prefixed with a `b`) that are used by the JVM description.

In addition to named types, we must be able to specify the storage locations that are available to a machine. Typically, one encounters two kinds of memories, *indexed* and *addressed*. An indexed memory is one that is small and can be accessed directly, such as a register file. In our example for the ARM, the register file, `R`, is an array of 16 elements of type `word_t`. It is also possible to define indexed memories that consist of only one location as we do for the JVM stack pointer (SP) register. Indexed memories also allow for the definition of named locations or *aliases* such as SP (stack pointer). Aliases may also be specified with attributes such as *reserved* or *general*. These attributes aid the search procedure in determining which target locations can be used for general purpose use (i.e., locations for storing intermediate results). An alias defaults to being reserved unless specified otherwise.

An addressed memory is one that is significantly larger than an indexed memory and is accessed using an address. Because these kinds of memories often return varying data lengths (i.e., byte or word) CISL allows for the definition of *data selectors*. A data selector indicates the type of the data that can be accessed as well as if there are any alignment constraints. In addition, a *quantum* data selector must be specified to indicate the smallest accessible unit of data (other selectors must be aggregations of the quantum). For example, as is the case for the ARM, main memory is byte-oriented so the quantum is a byte, but it is also possible to access 32-bit words of data so a selector is provided as well.

As CISL is a class-based language, instructions are specified using classes. In our example, we show the definitions of the ARM (left) and JVM (right) **`Add`** and **`IAdd`** instruction respectively. The ARM register-based add instruction requires register operands that are encoded in the instruction word. These operands are then used to add the values that are stored in the indicated registers and store the result. Operands are specified in CISL using *variables* and *references*. A variable defines an actual bit array and a reference refers to a subarray that is within an actual bit array. For the ARM add, the instruction word **`inst`** is defined as a little-endian, unsigned, bit array of length 32 (we omitted **`unsigned`** and **`little`** to conserve space). The destination register index, **`rd`**, is declared as a reference to a 4-bit sub-array starting at position 12 in **`inst`**. The other register indexes are specified similarly.

Once the memories and any instruction operands are defined, we are in a position to specify the semantics of the instruction. The semantic "code" for an instruction is declared in a special method named **`effect`**. The JVM specification describes the *iadd* instruction as popping the top two values off the operand stack, adding those values, and then storing the result back on the stack. Because the JVM specification deliberately leaves out implementation details, we must choose something concrete. In our example, we implement the stack (**`Stack`**) as an addressed memory that can be indexed by our single element indexed memory **`SP`** (stack pointer). Because no operands are required by this instruction the effect method contained in the class **`IAdd`** simply uses these memory structures to implement the semantics as described by the JVM specification. The implementation of the semantics for the ARM add is straight-forward and is given in the effect method contained in the class **`Add`**.

3.2. Memory Mapping

The operators of our language have well defined meanings that allow us to match operator to operator. The memory configuration, however, can come in many shapes and sizes. As such, mapping from a source machine's memory onto a target can also take several different forms. As such, matching instructions requires a *memory mapping*.

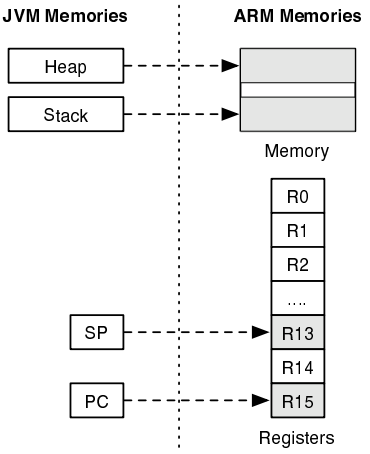


Figure 3. JVM to ARM Memory Mapping

Consider the case of generating a code generator for the JVM that targets the ARM architecture. The JVM has a memory for the stack and a memory for the heap, and the two are disjoint. The ARM, however, has only a single flat memory space and a set of general purpose registers. Mapping instructions from the JVM to the ARM is thus complicated by the fact that values may be retrieved from different locations yet the semantics of these instructions have the same overall effect. In the simplest case, we could designate two disjoint regions of ARM memory to be reserved for the JVM heap and stack. In addition, we can specify the JVM stack pointer, SP, to be mapped onto register R13 on the ARM (in fact, this is an ARM convention). Alternatively, we could reserve a set of n registers that correspond to the top n locations on the stack for faster access. When the number of values on the stack is greater than n, those values must be *spilled* to memory at some well known location. Figure 3 illustrates some of the memory mappings that are necessary for the simple case.

The distinguishing feature is that the same overall effect (an addition) is performed on the same values yet the locations where those values reside and how they are retrieved are different. Thus, before we can determine the sequence of instructions on the ARM that is semantically equivalent to the JVM *iadd* instruction we must first view the semantic description of that *iadd* in terms of the memories of the ARM. In other words, we must have a correspondence or mapping between the specific locations used by the JVM *iadd* instruction and a subset of locations found on the ARM.

To solve this problem, we require one to specify a mapping between the source IR and target machine store. That is, which memory locations defined by the target machine implement the memory of the source IR. An explicit mapping allows a degree of flexibility for the implementor of a code generator. For example, one could specify a simple mapping quickly to start producing a working code generator immediately, whereas a more complicated mapping would require more effort but yield more efficient generated code (as can be imagined by the two JVM to ARM example mappings). Producing a mapping automatically would be useful and raises interesting questions regarding store equivalence; we leave that to future work.

Currently, we assume a simple memory mapping between source IR and the target machine (i.e., JVM stack pointer is equivalent to the register 13 on the ARM). More sophisticated mappings will likely require additional support by the memory mapping framework. This, however, does not preclude us from generating good code generators.

3.3. Finding Potential Sequences

In order for a code generator to be useful it must be complete. That is, for each IR operation we must have a corresponding sequence of target instructions that implement it. The first step towards determining those target sequences is a *fast* heuristic search procedure. This is accomplished by exploiting instruction *features* that we can easily extract from our CISL language and use to guide the search quickly towards likely target sequences. The first phase of our approach follows a "generate-and-test" methodology. That is, we find possible target sequences quickly (the generation) and test both to determine if the sequence implements the source IR. We describe the interpretation step in Section 3.4. In this section we focus on the instruction features and how they are used by the search procedure to "guess" target sequences.

Searching for target instructions is essentially a matching problem. Given a source instruction sequence we must find target instructions that match semantically. Matching semantics, however, is not necessarily a straightforward task. For example, a source and target instruction may accomplish the same effect, but may do so by entirely different means. In addition, a target instruction may achieve only part of the semantic effect of the source. In this case, the source may match several target instructions that collectively provide the intended effect. Furthermore, once we have found a target candidate sequence, it is entirely possible that it includes extraneous effects (i.e., setting of condition codes) or that we must use specific memory locations (i.e., x86 segment registers). We must consider all of these details if we are to achieve our goal.

As such, we need a representation of an instruction that is amenable to our search procedure. In particular, we view an instruction as possessing a set of *features*. We then use these features as a heuristic input to locate target instructions more efficiently. The original Superoptimizer [12] relied on a very simplistic representation of an instruction: a string of bits. Viewing an instruction is this manner generated many meaningless sequences. Perhaps such a representation is sufficient for generating peephole optimizers in which the source and target are the same machine. In our case, however, the

source and target may be entirely different. In fact, the kind of machines we are often dealing with have no bit-level representation at all (i.e., compiler IR).

Fortunately, the search procedure has access to a rich semantic description of each instruction on both the source and target machine. It leverages these descriptions to extract key features of the instruction to prune the search space significantly. More specifically, we abstract the semantic descriptions into a parameterized model that enables us to match instructions systematically.

At the most fundamental level one can view an instruction as a sequence of statements parameterized by a set of variables. Each of these variables corresponds to either an input or an output parameter of the instruction. For example, an instruction may perform some operation on an immediate field, register, or memory location and store its result in some destination location. Using this information alone is a significant improvement with respect to prior efforts in that we can match target instructions based on their input and output similarities. Thus, a source instruction taking a register and a 16-bit immediate field as input would most likely match a target instruction that also takes a register and 16-bit immediate field as input. Naturally, there may be several constraints involved such as allowable registers and the size of the immediate fields.

Despite this improvement, however, matching on just inputs and outputs will yield a substantial number of mismatches. Consider the fact that most arithmetic instructions on a RISC style machine operate on registers and place the result in some destination register. Thus, to improve our results we must also consider the set of operators a particular instruction uses. Given this operator information for a source instruction, we can sort the target instructions by operator usage. Those target instructions that are considered most likely to match the source IR reside at the front of the list. We can then sort by input and output parameters to increase further the likelihood of quick matching.

These abstractions significantly improve the likelihood of matching target instructions, but we can do better. It is not uncommon for a CISC instruction (even some RISC instructions such as those found on the ARM) to require several operators to describe the complete effect. This is especially true when computing address locations. These computations, however, are really just a by-product of the overall intended effect of the instruction. For example, The ARM *add* instruction can perform an addition on the value contained in two registers. One of the registers can be optionally shifted. If we were to consider the *shift* and the *add* operation to be equivalent in weight the overall intended effect could be lost. To alleviate this problem we sort operators based on their relative importance within a semantic description.

We can easily determine the importance of an operator based on its location in the *abstract syntax tree* (AST) representation produced by the CISL compiler. Those operators

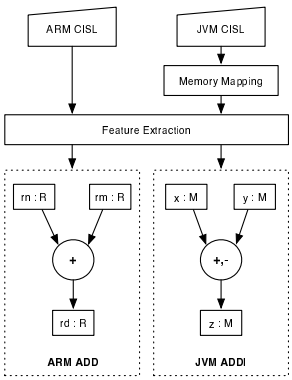


Figure 4. ARM/JVM Feature Extraction

appearing closer to the leaves of the tree are typically performing computations for operators near the top of the tree. This corresponds to addressing mode computations that are merely preparing values for use by the actual instruction operation. Thus, an operator appearing near the top of the tree is ranked higher than one appearing lower.

In Figure 4 we show how we represent the features of the ARM *add* and JVM *iadd* instructions. Boxes represent the input and output locations, circles correspond to the set of operators, and edges indicate the flow of data. For example, the ARM *add* has two inputs from the indexed memory **`R`** and an output into the indexed memory **`R`**. The only operator that is used in this case is the CISL symbol for addition: '+'. We apply a memory mapping to the JVM *addi* instruction before extracting its features. The mapping used corresponds to our simple mapping described earlier. It translates the JVM **`Stack`** and **`SP`** memories into the **`M`** and **`R`** memories on the ARM. The operators used by *iadd* are both addition, '+', and subtraction, '-'.

The search procedure uses these features to make fast informed choices about the target sequence, not to represent the semantics exactly. Input, output, and operator features isolate target sequences that are *related* to the IR operation. Thus, the search space is dramatically reduced by considering only target instructions that match feature-wise. In particular, we sort target instruction features by operands and operators based on the IR operation. We then attempt to match the features of the IR with those of the target instructions. The goal is to cover each of the IR features with one or more of the target instruction features. We refer to the target instructions whose features cover the IR operation a *potential* sequence. A potential sequence is then tested by the semantic interpreter discussed in the next section.

3.4. Semantic Interpretation

The goal of the semantic interpreter test is to increase our confidence that the target sequence is performing the same func-

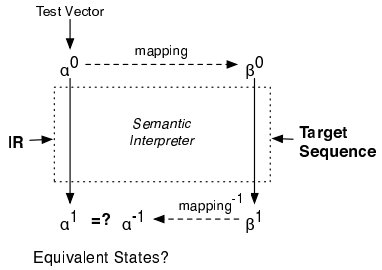


Figure 5. Semantic Interpreter

tion as the IR operation. We call a sequence that passes the interpretation test a *candidate* for verification. The verifier component is described in Section 3.5. In this section, we describe how instruction semantics are interpreted and how those sequences that fail interpretation are subject to refinement and re-submitted to the search procedure.

The basic goal of semantic interpretation is to "simulate" both the semantics of the IR and the target sequence and directly compare the resulting simulated memory state. To make this possible, we must be concerned with what values we initialize simulated memory prior to interpretation, how we compare the resulting memory state for sameness, and how we actually interpret the semantics.

Before interpretation proceeds, we require a useful representation of the instructions and memories. In particular, it must be possible to initialize the instruction semantics with specific constant values that it operates on (i.e., registers and immediate fields). In terms of CISL, this means that we are able to "bind" CISL variables to the particular values they take on. In addition, we need to be able to initialize simulated memory with legitimate values. These parameterized abstractions allow the IR and target semantics to operate on meaningful inputs and produce sensible outputs; ultimately providing a basis for comparison.

Because CISL has a well defined semantics it can be interpreted just like a typical programming language. Unlike other languages, however, it operates at a much lower level. That is, variables and references can take on arbitrary bit widths, signedness, and endianness that may not necessary map directly to the underlying types of the host language of the interpreter. In addition, we might require a different representation for the values (i.e., something other than two's-complement). We don't expect this to be a common occurrence (that is, changes in the fundamental representation of values), however, currently we have been able to isolate these details into components that can be overridden in a straightforward manner.

Because we are also concerned with comparing resulting states, we require a method for translating from one simulated state representation to another. For example, if we simulate an IR instruction and a target sequence how do we compare their resulting effects on memory? This is related to the problem of mapping memories for our heuristic search procedure in Section 3.3. In fact, we utilize the same mapping specification to translate an initialized IR state into a corresponding target machine state representation prior to interpretation. After interpretation, we apply an inverse mapping to translate from the resulting target state into the state representation of the IR.

Figure 5 illustrates clearly our interpretation technique. An initial IR state, α^0, is initialized with a particular test vector. The IR state α^0 is then translated (using the memory map) into a corresponding initial state for the target, β^0. The IR and target semantics are then "executed" by the interpreter using the particular state representation to record any side-effects. After simulating both semantics the resulting states, α^1 and β^1 are produced. In order to make our final comparison we translate the target state back into its original IR state representation, α^{-1}. We can then compare α^1 and α^{-1} to determine if they are the same. If α^1 and α^{-1} are the same across the entire set of test vectors the target sequence becomes a *candidate* and sent to the verifier; otherwise we may be able *repair* the potential target sequence to allow it to pass interpretation.

Currently, we apply three repair procedures to a failed sequence: *load/store insertion*, *operand permutation*, and *operator permutation*. load/store insertion remedies the case when the values contained in the resulting states are the same but reside in different locations. For this, we add data movement instructions to place results into the same locations. Operand permutation is applied when output values are the logical or arithmetic inverse of those produced by the IR. The attempted correction is to simply swap the operands of an instruction. Operator permutation is used when the IR and target states partially match. Here we substitute operations by an algebraically similar instruction.

A "repaired" sequence is then re-submitted to the interpreter for testing. The repair procedures are applied again if the new sequence fails. This process continues for some specified number of times or until no repair procedures apply. If the sequence fails interpretation and repair it is discarded and the next potential sequence is examined. If the sequence passes each test vector after repair it can then be passed to the verifier as a candidate. The verifier is described in the next section.

3.5. Verifier

The goal of our generate-and-test phase is to find target sequences quickly that "do the same thing" as the given IR operation. The result is a code generator pattern that we believe is correct, but not guaranteed. In this sense, we are generating automatically patterns that are just as good as a carefully hand crafted code generator. Although automation is a significant improvement over manual methods we desire code generator patterns that are provably correct. We generate correct code generator patterns using a final verification step that uses for-

mal methods to guarantee that the IR semantics is equivalent to the target sequence semantics over all inputs. We currently focus on two techniques, term rewriting systems and boolean satisfiability, to achieve this goal.

A term rewriting system uses a set of axioms or rules that define semantic preserving transformations on terms. A term, in our case, corresponds to the AST representation of the semantics of an instruction or instruction sequence. An application of an axiom on a term yields a new semantically equivalent term. For example, applying the rule $x - y \rightarrow -y + x$ (where x and y are variables) to the term **`R[rn]-imm`**, would yield the equivalent term **`-imm+R[rn]`**.

We determine the equivalence between terms, and thus between IR and target sequences, by repeated application of rules that define the laws of arithmetic and boolean operations, and rules related to machine semantics. If the resulting terms (after rule application) of the IR and target sequences are syntactically equivalent they have the same semantic meaning. To guarantee this property, however, we must ensure that the term rewriting system is both *terminating* and *confluent*. That is, the repeated application of rules will not diverge and that the order of application will not result in different terms. Fortunately, we can apply a well known procedure called *completion* [1] to generate a terminating and confluent term rewriting system from a given set of axioms. Naturally, it is possible that the application of the completion procedure itself may never terminate. The rule sets that we have currently explored, however, have been successful.

An alternative approach leverages a SAT solver to determine semantic equivalence. To do this, requires our semantic language to be formulated as boolean expressions on an initial state. In particular, we must translate the input description language into a boolean expression that can be used as input to a SAT solver. In the end, we wish to ask the question "Can any state variable have a different value in the state reached by executing the sequence of source instruction semantics and in the state reached by executing the sequence of target instruction semantics, starting both from the same initial (unspecified) state?".

By leaving the initial state unbound in the translated SAT formula, the problem then becomes "$\exists$ an initial state that allows any of the state variables to obtain a different value after executing the two instruction sequences?". To a certain degree this is similar to the semantic interpreter from Section 3.4 but over all possible input values rather than a chosen set of test vectors. The crux of the problem is describing transformations of possibly very large states in an essentially "state-less" way, and such that the end formula is in a form suitable for SAT, namely existentially quantified over values of boolean variables.

4. Related Work

Glanville [9] uses LR parsing techniques to generate a code generator from a grammar-like description of the code generator patterns. To improve this approach Ganapathi et al. [8] uses attribute grammars to increase the amount of semantic information during IR parsing. Unfortunately, both approaches depend on the IR being closely related to the target machine (i.e., same operations).

Bottom-up rewrite system techniques (BURS) [4, 6, 7, 17] use tree-rewriting given a set of code generator patterns. Although the approach is independent of the IR and target, the pattern specification is dependent on both. As such, it can not be reused by another compiler suite.

Bansal and Aiken [2] introduce techniques for automatically generating peephole superoptimizers. They improve on superoptimization techniques originally introduced by Massalin [12]. Although there goal is a same machine code generator generator their approach is similar to ours. They attempt to find more efficient instructions quickly and prove equivalence using SAT solver techniques. Unlike our approach, however, they do not have a representation of the instruction semantics available to improve search and the boolean formulae are hand written.

Fraser [5] uses a production system to generate code generators automatically. Knowledge capturing code generator isses are used to derive code generator patterns. The approach uses the IR to encode rules and is thus compiler dependent. In addition, correctness of the generated patterns can not be verified.

Cattell [3] uses a formal approach to guarantee correct code generator patterns. An set of axioms are used to rewrite descriptions of IR semantics into equivalent target semantic trees using means-ends analysis. His description language is based off the compiler IR making the technique dependent on his compiler framework. In addition, the technique failed to recognize values residing in two different locations (i.e., memory and register), thus leading to patterns containing redundant operations (i.e., unnecessary loads).

5. Conclusion

We presented an approach towards universal code generator generation. In particular, we demonstrated a novel technique that leverages a common semantic language describing precisely the semantics of the compiler IR and the target instructions. The semantic descriptions are used to drive a generate-and-test approach for finding code generator patterns. In addition, we described how the same semantic descriptions can be used to verify the correctness of those patterns to ensure that the generated code generator will produce correct code. Unlike prior efforts, the use of our common language allows our approach to be both compiler and target independent.

At present, we have prototyped several aspects of our approach to determine the feasibility of the system. This includes a simplified form of the generate-and-test procedure, a term rewriting based verifier for a subset of the semantic language, and a SAT library for translating our semantic language into boolean formulae. In addition, we have generated a functional simulator for a subset of the ARM instructions from a semantic description, thus increasing our confidence that semantic interpretation is possible.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[2] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 394–403, New York, NY, USA, 2006. ACM Press.

[3] R. G. G. Cattell. *Formalization and automatic derivation of code generators.* PhD thesis, 1978.

[4] H. Emmelmann, F.-W. Schröer, and L. Landwehr. Beg: a generation for efficient back ends. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 227–237, New York, NY, USA, 1989. ACM Press.

[5] C. W. Fraser. *Automatic generation of code generators.* PhD thesis, 1977.

[6] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.

[7] C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[8] M. Ganapathi and C. N. Fischer. Description-driven code generation using attribute grammars. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–119, New York, NY, USA, 1982. ACM Press.

[9] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254, New York, NY, USA, 1978. ACM Press.

[10] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Vols 1–3*, 2003.

[11] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] H. Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[13] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification For A New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

[14] J. E. B. Moss, B. Moss, C. C. Weems, and T. Richards. The CoGenT Project: Co-generating Compilers and Simulators for Dynamically Compiled Languages. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 210.2, Washington, DC, USA, 2003. IEEE Computer Society.

[15] J. E. B. Moss, T. Palmer, T. Richards, I. Edward K. Walters, and C. C. Weems. CISL: A Class-based Machine Description Language for Co-generation of Compilers and Simulators. *Int. J. Parallel Program.*, 33(2):231–246, 2005.

[16] J. E. B. Moss, T. S. Palmer, T. Richards, E. K. W. II, and C. C. Weems. CMDL: A Class-based Machine Description Language for Co-generation of Compilers and Simulators. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, 2004.

[17] T. Proebsting. Burg, iburg, wburg, gburg: so many trees to rewrite, so little time (invited talk). In *RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 53–54, New York, NY, USA, 2002. ACM Press.

[18] T. D. Richards, E. K. W. II, T. Palmer, J. E. B. Moss, and C. C. Weems. A unified framework for the automatic generation of system tools and components. Technical report, University of Massachusetts Amherst, 2007.

[19] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.