Rewriting Strategies for Instruction Selection

Martin Bravenboer
Eelco Visser

www.stratego-language.org

Technical Report UU-CS-2002-021
Institute of Information and Computing Sciences
Utrecht University

April 2002

This technical report is a preprint of:
M. Bravenboer and E. Visser. Rewriting Strategies for Instruction Selection.
To appear in S. Tison (editor) *Rewriting Techniques and Applications (RTA'02)*. Lecture Notes in Computer Science. Springer-Verlag, Copenhagen, Denmark, June 2002. (© Springer-Verlag)

Copyright © 2002 Martin Bravenboer and Eelco Visser

Address:
Institute of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
email: visser@acm.org
`http://www.cs.uu.nl/~visser/`

Rewriting Strategies for Instruction Selection

Martin Bravenboer
Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. http://www.cs.uu.nl/∼visser, mbravenb@cs.uu.nl, visser@acm.org

Abstract. Instruction selection (mapping IR trees to machine instructions) can be expressed by means of rewrite rules. Typically, such sets of rewrite rules are highly ambiguous. Therefore, standard rewriting engines based on fixed, exhaustive strategies are not appropriate for the execution of instruction selection. Code generator generators use special purpose implementations employing dynamic programming. In this paper we show how rewriting strategies for instruction selection can be encoded concisely in Stratego, a language for program transformation based on the paradigm of programmable rewriting strategies. This embedding obviates the need for a language dedicated to code generation, and makes it easy to combine code generation with other optimizations.

1 Introduction

Code generation is the phase in the compilation of high-level programs in which an intermediate representation (IR) of a program is translated to a list of machine instructions. Code generation is usually divided into instruction selection and register allocation. During instruction selection the nodes of IR expression trees are associated with machine instructions.

The process of instruction selection can be formulated as a rewriting problem. First the IR tree is rewritten to an instruction tree, in which parts (tiles) of the original tree are replaced by instruction identifiers. This instruction tree is then flattened into a list of instructions (code emission) during which (temporary) registers are introduced to hold intermediate values. Typically, the set of rewrite rules for rewriting the IR tree is highly ambiguous, i.e., there are many instruction sequences that implement the computation specified by the tree, some of which are more efficient than others. The instruction sequence obtained *depends on the rewriting strategy* used to apply the rules.

Standard rewriting engines such as ASF+SDF [4] provide a fixed strategy for exhaustively applying rewrite rules, e.g., innermost normalization. This is appropriate for many applications, e.g., algebraic simplifications on expression trees, but not for obtaining the most efficient code sequence for an instruction selection problem.

For the construction of compiler back-ends, code generator generators such as TWIG [1], BEG [5], and BURG [8] use tree pattern matching [6] and dynamic

programming [2] to compute all possible matches in parallel and then choose the rewrite sequences with the lowest cost. These systems use the same basic strategy, although there are some differences between them, mainly in the choice of pattern match algorithm and the amount of precomputation done at generator generation time. Although some other tree manipulation operations can be formulated in these systems, they are essentially specialized for code generation. Other compilation tasks need to be defined in a different language.

In this paper we show how rewriting strategies for instruction selection can be encoded concisely in Stratego, a language for program transformation based on the paradigm of programmable rewriting strategies [10,11]. The explicit specification of the rewriting strategy obviates the need for a special purpose language for the specification of instruction selection, allows the programmer to switch to a different strategy, and makes it easier to combine code generation with other optimizations such as algebraic simplification, peephole optimization, and inlining.

In Section 2 we illustrate the instruction selection problem by means of a small intermediate representation, and a RISC-like instruction set, define instruction selection rules, and discuss covering a tree with rules. In Section 3 we explore several strategies for applying the RISC selection rules, including global backtracking to generate all possibilities, innermost rewriting, and greedy top-down (maximal munch). It turns out that no dynamic programming is needed in the case of simple instruction sets. In Section 4 we turn to the problem of code generation for complex instruction set (CISC) machines, which is illustrated with a small instruction set and corresponding rewrite rules. In Section 5 we introduce a specification of a rewriting strategy based on dynamic programming in the style of [1] to compute the cheapest rewrite sequence given costs associated with rewrite rules. The generic specification of dynamic programming is very concise, it fits in half a page, and provides a nice illustration of the use of scoped dynamic rewrite rules [9].

2 Instruction Selection

In this section we describe instruction selection as a rewriting problem. First we describe a simple intermediate representation and a subset of an instruction set for a RISC-like machine. Then we describe the problem of code generation for this RISC machine as a set of rewrite rules. The result of rewriting an intermediate representation tree with these rules is a tile tree which is turned into a list of instructions by *code emission*, a transformation which flattens a tree. In Section 4 we will consider code generation for a CISC-like instruction set.

2.1 Intermediate Representation

An intermediate representation (IR) is a low level, but target independent representation for programs. Constructors of IR represent basic computational operations such as adding two values, fetching a value from memory, moving a

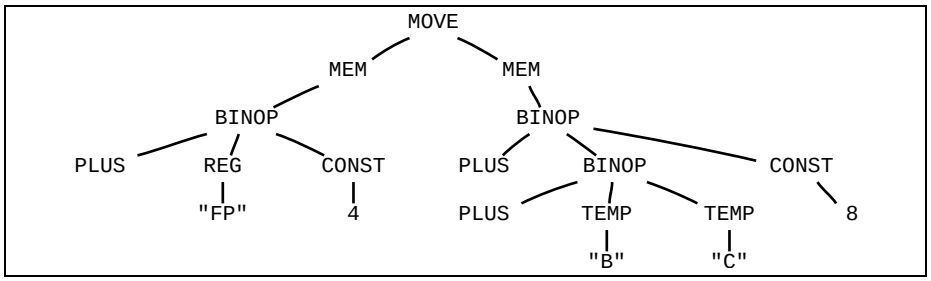
```
module IR
imports Operators
signature
  sorts Exp Stm
  constructors
    BINOP : BinOp * Exp * Exp -> Exp // arithmetic operation
    CONST : Int -> Exp               // integer constant
    REG   : String -> Exp            // machine register
    TEMP  : String -> Exp            // symbolic (temporary) register
    MEM   : Exp -> Exp               // dereference memory address
    MOVE  : Exp * Exp -> Stm         // move between memory a/o registers
```

Fig. 1. Signature of an intermediate representation

value from one place to another, or jumping to another instruction. These operations are often more atomic than actual machine instructions, i.e., a machine instruction involves several IR constructs. This makes it possible to translate intermediate representations to different architectures. Figure 1 gives the signature of a simple intermediate representation format. We have omitted constructors for forming whole programs; for the purpose of instruction selection we are only interested in expressions and simple statements. Figure 2 shows an example IR expression tree in textual and corresponding graphical form. The expression denotes moving a value from memory at the address at offset 8 from the sum of the `B` and `C` registers to memory at offset 4 from the `FP` register.

2.2 Instruction Set

Code generation consists of mapping IR trees to sequences of machine instructions. The complexity of this task depends on the complexity of the instruction set of the target architecture. Reduced Instruction Set (RISC) architectures provide simple instructions that perform a single action, while Complex Instruction Set (CISC) architectures provide complex instructions that perform many actions. For example loading and storing values from and to memory is expressed




```
MOVE(MEM(BINOP(PLUS,REG("FP"),CONST(4))),
     MEM(BINOP(PLUS,BINOP(PLUS,TEMP("B"),TEMP("C")),CONST(8))))
```

Fig. 2. Example IR expression tree

```
module RISC
signature
  sorts Instr Reg
  constructors
    TEMP    : String -> Reg             // temporary register
    REG     : String -> Reg             // machine register
    R       : Reg -> Reg                // result register
    ADD     : Reg * Reg * Reg -> Instr  // add registers
    ADDI    : Reg * Reg * Int  -> Instr // add register and immediate
    LOADC   : Reg * Int -> Instr        // load constant
    LOAD    : Reg * Reg -> Instr        // load memory into register
    LOADI   : Reg * Reg * Int -> Instr  // load memory into register
    MOVER   : Reg * Reg -> Instr        // move register to register
    STORE   : Reg * Reg -> Instr        // store register in memory
    STOREI  : Reg * Reg * Int -> Instr  // store register in memory
```

Fig. 3. Constructors for a subset of the instructions of RISC machine.x

by separate instructions on RISC machines, while CISC instruction sets allow many instructions to fetch their operands from memory and store their results back to memory.

We will first consider code generation for the small RISC-like instruction set of Figure 3. The instructions consist of arithmetic operations working on registers and storing their result in a register (only addition for the example), load and store operations for moving values from and to memory, and an operation for loading constants and one for moving values between registers. Several instructions have an 'immediate' variant in which one of the operands is a constant. In load and store operations these immediate values indicate an offset from the address register. For example, `LOADI(REG("a"), REG("b"), 8)` means loading the value at offset 8 from the address in register `b`. The sequence of instructions in Figure 4 implements the example IR tree in Figure 2.

2.3 Selecting Instructions with Rewrite Rules

The process of instruction selection can be divided into two phases. First, determine for each tree node which instruction to use. Then linearize the tree into a sequence of instructions (code emission). The connection between machine instructions and intermediate representation can be defined in terms of rewrite rules. Figure 5 defines rules mapping IR tree patterns to RISC instructions. To translate a complete IR tree to machine instructions we now have to find an

```
[ADD(TEMP("b"),TEMP("B"),TEMP("C")),
 LOADI(TEMP("a"),TEMP("b"),8),
 STOREI(TEMP("a"),REG("FP"),4)]
```

```
ADD b B C
LOADI a b 8
STOREI a FP 4
```

Fig. 4. An instruction sequence in abstract and concrete syntax.

```
module IR-to-RISC-Rules
imports IR RISC
rules
  Select-ADD :
    BINOP(PLUS, e1, e2) -> ADD(R(TEMP(<new>)), e1, e2)
  Select-ADDI :
    BINOP(PLUS, e, CONST(i)) -> ADDI(R(TEMP(<new>)), e, i)
  Select-LOADC :
    CONST(i) -> LOADC(R(TEMP(<new>)), i)
  Select-MOVER :
    MOVE(r@<REG(id) + TEMP(id)>, e) -> MOVER(r, e)
  Select-STORE :
    MOVE(MEM(e1), e2) -> STORE(e2, e1)
  Select-STOREI :
    MOVE(MEM(BINOP(PLUS, e1, CONST(i))), e2) -> STOREI(e2, e1, i)
  Select-LOAD :
    MEM(e) -> LOAD(R(TEMP(<new>)), e)
  Select-LOADI :
    MEM(BINOP(PLUS, e, CONST(i))) -> LOADI(R(TEMP(<new>)), e, i)
```

Fig. 5. Instruction selection rules mapping IR trees to instructions.

application of rules to nodes of the tree such that each node of the tree is *covered* by some pattern; except for 'leaf' nodes such as `REG`, `TEMP` and constants. Figure 6 shows a possible covering for the example IR tree of Figure 2 together with the resulting instruction tree.

In an expression tree, results from subexpressions are passed implicitly up in the tree. When breaking a tree into a sequence of instructions this is no longer the case; intermediate values should be stored into registers or in memory. We will assume here that they can be stored in registers. In the rewrite rules in Figure 5 passing of intermediate values is achieved by generating a new temporary register at the position of the destination register to hold the result of the instruction. For example, rule `Select-ADD` generates the term `ADD(R(TEMP(<new>)), e1, e2)`. The `R(_)` constructor indicates that this argument contains the result of the instruction. The `new` primitive generates a new unique name.

2.4 Code Emission

After covering the IR tree with instructions, the program is still in tree shape. *Code emission* linearizes the tree into a list of instructions. For example, for the tiling in Figure 6 we get the code sequence of Figure 4. A specification of a code emission strategy based on the generic tree flattening strategy `postorder-collect` is given in Appendix A.

3 Strategies for Instruction Selection

In the previous section we have seen that rewrite rules can be used for expressing the mapping from intermediate representation trees to lists of machine instruc-

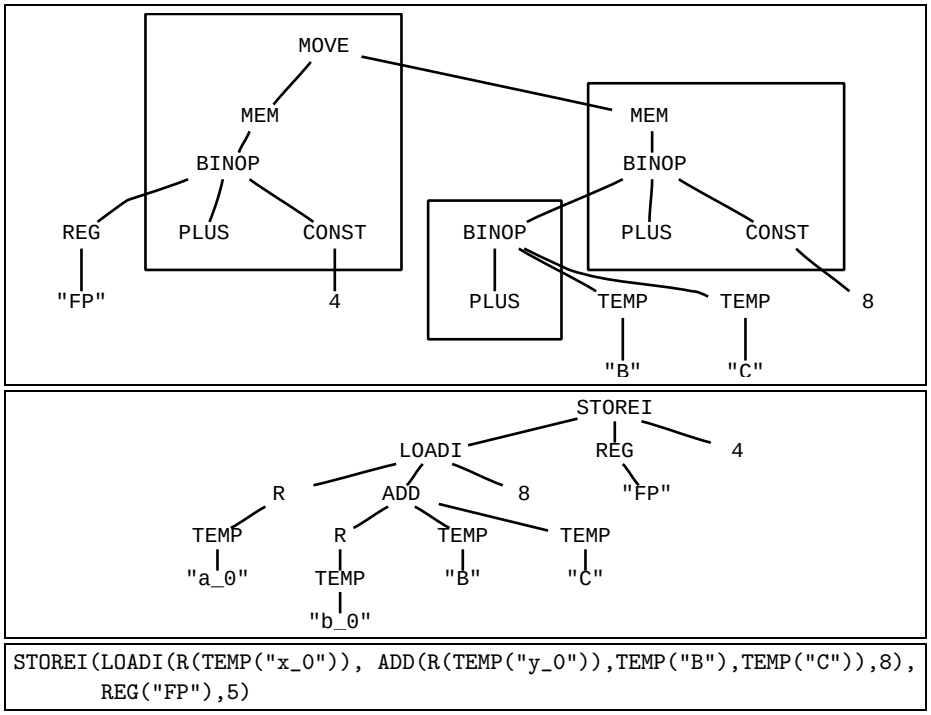


Fig. 6. Optimum tiling of the example tree and the corresponding instruction tree with temporary intermediate result registers.

tions. However, the set of rewrite rules in Figure 5 is highly ambiguous, i.e., there are many rewritings possible. For example, Figure 7 shows all 9 possible tilings for the example expression in Figure 2 together with three of the code sequences pretty-printed. This kind of ambiguity is typical for instruction selection rules. Therefore, the instruction sequence finally obtained *depends on the rewriting strategy* used to order the application of selection rules. In this section we will examine several strategies and their applicability to instruction selection.

3.1 Intermezzo: Rewriting Strategies

Thusfar we have considered algebraic signatures and standard rewrite rules. In a normal rewriting system a term over a signature is normalized with respect to the set of rewrite rules. In Stratego the rewriting strategy is not fixed, but can be specified in a language of strategy combinators. A strategy is a term transformation that may fail. Rewrite rules are basic strategies. Examples of basic combinators are sequential composition, non-deterministic choice, deterministic choice, and a number of generic term traversal operators. Using these combinators a wide variety of rewriting strategies can be defined. The Stratego library provides a large number of generic strategies built using these combinators. In the rest of this paper we will define several compound strategies. The strategy

```
STORE(LOAD(a,ADD(b,ADD(c,B,C),LOADC(d,8))),ADD(e,FP,LOADC(f,4)))
STORE(LOAD(a,ADD(b,ADD(c,B,C),LOADC(d,8))),ADDI(g,FP,4))
STORE(LOAD(a,ADDI(h,ADD(i,B,C),8)),ADD(j,FP,LOADC(k,4)))
STORE(LOAD(a,ADDI(h,ADD(i,B,C),8)),ADDI(l,FP,4))
STORE(LOADI(m,ADD(n,B,C),8),ADD(o,FP,LOADC(p,4)))
STORE(LOADI(m,ADD(n,B,C),8),ADDI(q,FP,4))
STOREI(LOAD(r,ADD(s,ADD(t,B,C),LOADC(u,8))),FP,4)
STOREI(LOAD(r,ADDI(v,ADD(w,B,C),8)),FP,4)
STOREI(LOADI(x,ADD(y,B,C),8),FP,4)
```

```
ADD   c B C
LOADC d 8
ADD   b c d
LOAD  a b
LOADC f 4
ADD   e FP f
STORE a e
```

```
ADD   n B C
LOADI m n 8
LOADC p 4
ADD   o FP p
STORE m o
```

```
ADD    y B C
LOADI  x y 8
STOREI x FP 4
```

Fig. 7. All possible instruction selections for the example IR tree and three pretty-printed code sequences

elements in these definitions will be explained when needed. For a full account we refer to the literature; good starting points are [10,9,11].

3.2 Exhaustive Application

The traditional way to interpret a set of rewrite rules is by applying them exhaustively to a subject term. The following strategy takes this approach:

```
innermost-tilings =
  innermost(Select-ADD + Select-ADDI + Select-STORE + Select-LOAD +
            Select-STOREI + Select-LOADC + Select-LOADI + Select-MOVER)
```

The *local non-deterministic choice operator* `(s1 + s2)` combines two strategies `s1` and `s2` by trying either of them. If that fails the other is tried. If one of the branches has succeeded the choice is committed, i.e., no backtracking to this choice is done if the continuation fails.

The innermost strategy takes a transformation, e.g., a choice of rules, and applies them exhaustively to a subject term starting with the inner terms. This is expressed generically as

```
innermost(s) = all(innermost(s)); try(s; innermost(s))
```

The generic traversal combinator `all(s)` applies the transformation `s` to each direct subterm of a constructor application. Thus, `<all(s)>C(t1,...,tn)` denotes `C(<s>t1,...,<s>tn)`. The first part of the strategy normalizes all direct subterms of a node. After that (`;` is sequential composition), the strategy tries to apply the transformation `s`. If that succeeds the result, e.g., the right-hand side of the term, is again normalized. The combinator `try(s)` is defined as `try(s)`

`= (s <+ id)`. That is, try to apply `s`, but if that fails do nothing, i.e., use the identity strategy `id`. Thus, if `s` fails the subject term is in normal form.

The innermost strategy is not adequate for instruction selection, however. It produces the worst result, e.g., the first result in Figure 7. This is caused by the fact that reducible subexpressions of a pattern are reduced before the (more complex) pattern itself is applied. Furthermore, the exhaustive application is overkill since instruction selection rules do not need exhaustive application; each node needs to be rewritten at most once.

3.3 All Rewritings

Another approach is to generate all possible results. This is achieved by the following strategy:

```
all-tilings =
  bagof(topdown(try(Select-ADD ++ Select-ADDI ++ Select-STORE
                    ++ Select-STOREI ++ Select-LOADC
                    ++ Select-LOAD ++ Select-LOADI ++ Select-MOVER)))
```

In constrast to the `+` operator used above, the *global non-deterministic choice* operator `(s1 ++ s2)` is a non-committing choice operator. This means that after one of the branches has succesfully terminated, but the continuation fails, the strategy backtracks to the other branch. Formally, we have that `(s1 ++ s2); s3` is equal to `(s1; s3) ++ (s2; s3)`, which does not hold for `+`. This property is used by the `bagof(s)` operator to produce the list of all possible results by forcing a backtrack to the last choice point. The topdown strategy, defined as

```
topdown(s) = s; all(topdown(s))
```

traverses a tree in pre-order, applying a transformation `s` to a node before transforming its subterms using the generic traversal operator `all`. The list of all results in Figure 7 was produced using this strategy.

After generating the list of all possible results we could define a filter that selects the best solution. As a specification this is interesting, as an implementation it is not feasible due to the combinatorial explosion; for each node all possible rewrites for its subnodes need to be considered.

3.4 Maximal Munch

If each rule corresponds to a machine instruction, the best solution is usually the shortest sequence of instructions. In other words, we want to maximize the tree tilings selected. The following strategy takes this approach:

```
maximal-munch =
  topdown(try((Select-ADDI <+ Select-ADD) + Select-LOADC
              + (Select-STOREI <+ Select-STORE)
              + (Select-LOADI <+ Select-LOAD) + Select-MOVER))
```

Starting at the root of the expression tree, the strategy tries to apply one of the selection rules. Instead of combining all rules with the non-deterministic choice operator `+`, some pairs of rules are combined using the *deterministic choice* operator `(s1 <+ s2)`, which always tries its left argument first. In this way the `maximal-munch` strategy gives priority to patterns that take the largest bite, e.g., preferring `Select-ADDI` over `Select-ADD`. In fact, this simple strategy, combining pre-order top-down traversal with rule priority, is adequate for simple RISC-like instruction sets.

3.5 Pattern Matching

As can be seen from the examples above, programmable strategies allow the combination of rewrite rules into different strategies. This flexibility in combining rules does not restrict the efficient implementation of pattern matching. When combining rules in a choice, as in the examples above, the rules are not tried one by one. Instead a pattern match optimization compiles the rule selection into a matching automaton which folds left-hand sides with the same root constructor into a single condition (and recursively for subpatterns).

4 Complex Instruction Sets

In RISC instruction sets complex memory addressing modes are restricted to load and store operations. In complex instruction set (CISC) machines, many instructions can use the full range of addressing modes to read their operands directly from memory using base and index registers and constant offsets from these. Some machines also have heterogeneous register sets such that not all instructions can work with all registers, requiring values to be moved between registers.

For such machines maximal munch is not adequate. Code generators in production compilers use a dynamic programming approach to compute the code sequence with the least cost. To illustrate rewriting with dynamic programming we define the small CISC-like instruction set in Figure 8. In contrast to the RISC instruction set there are no special load and store operations. Instead each instruction can use all addressing modes and thus load and store values

```
module CISC
signature
  sorts Reg Addr Instr
  constructors
    TEMP  : String -> Reg       IMM   : Int -> Addr
    REG   : String -> Reg       REGA  : Reg -> Addr
    NONE  : Reg                 ADDR  : Reg * Reg * Int * Int -> Addr
    R     : Reg -> Reg          ADD   : Addr * Addr * Addr -> Instr
                                MOVEM : Addr * Addr -> Instr
```

Fig. 8. Constructors for a subset of the instructions of a CISC machine

```
MOVEM 4[FP] 8[B,C]  # C1=1 C2=1 C3=0 C4=0 C5=0 C6=0 C7=0 cost=1

ADD r B C           # C1=1 C2=1 C3=0 C4=0 C5=0 C6=0 C7=2 cost=2
MOVEM 4[FP] 8[r]

ADD   n FP $4       # C1=1 C2=1 C3=1 C4=1 C5=3 C6=3 C7=4 cost=6
ADD   e B C
ADD   l e $8
MOVEM [n] [l]
```

Fig. 9. Lowest cost selections based on different cost assignments

to memory. The `ADDR(r1,r2,scale,off)` addressing mode indicates a value in memory at address `r1 + (r2*scale) + off`, where `r1` is a base register, `r2` an index register, and `scale` and `off` constants.

Figure 10 defines instruction selection rules for this instruction set from the intermediate representation of Figure 1. Since each instruction can use all addressing modes, it is not desirable to define rules for each instruction parsing each addressing mode. Instead addressing modes are selected separately. To ensure that tiles connect, each rule indicates the *mode* of its tile and the expected modes of the leaves of the tile using the constructor `l(mode,tile)`.

Again, this rule set is ambiguous. To indicate which rewrite should be used, costs are assigned to rewrite rules corresponding to the operational cost of selected instruction. Figure 11 assigns symbolic costs `C1` to `C7` to several of the patterns of Figure 10. The goal now becomes to select the tree with lowest overall cost. Figure 9 shows several lowest cost selections for different cost assignments to these symbolic costs. This illustrates that instruction selection can no longer be achieved by a fixed strategy.

5 Dynamic Programming

Maximal munch does not work since chain rules such as `Select-LOAD` can be applied indefinitly. A straightforward solution would be to generate all solutions, compute their cost, and take the cheapest solution. However, in Section 3 we already saw that this leads to a combinatorial explosion in which subresults are computed many times. In the case of the rules of Figure 10, the situation is worse, since rules `Select-REGA` and `Select-LOAD` together lead to infinitly many possibile rewrites.

The dynamic programming approach of [2,1] is similar to the `all-tilings` strategy of Section 2, but instead of computing all possible rewrites in sequence, we first compute all possible matches in one bottom-up pass over the tree, tabulating the cheapest match for each node. This information can then be used to perform the optimal rewrite sequence. Figure 13 presents the specification of a generic dynamic programming strategy. Figure 12 shows an instantiation of this strategy with the rules and costs of Figures 10 and 11, and a default cost of 0.

```
module IR-to-CISC-Rules
imports CISC IR Dynamic-Programming
signature
  constructors
    a : Mode  reg : Mode  imm : Mode  s : Mode
rules
  Select-MOVEM :
    MOVE(e1, e2) -> l(s, MOVEM(l(a, e1), l(a, e2)))
  Select-ADDM :
    MOVE(e1, BINOP(PLUS, e2, e3)) -> l(s,ADD(l(a,e1), l(a,e2), l(a,e3)))
  Select-ADD :
    BINOP(PLUS,e1,e2) -> l(reg,ADD(R(REGA(TEMP(<new>))),l(a,e1),l(a,e2)))
  Select-imm :
    CONST(i) -> l(imm, i)
  Select-reg :
    REG(r) -> l(reg, REG(r))
  Select-reg' :
    TEMP(r) -> l(reg, TEMP(r))
  Select-ABS :
    MEM(i) -> l(a, ADDR(NONE,NONE,1,l(imm, i)))
  Select-BASE :
    MEM(r) -> l(a, ADDR(l(reg, r),NONE,1,0))
  Select-BASEIMM :
    MEM(BINOP(PLUS, e1, e2)) -> l(a, ADDR(l(reg,e1), NONE, 1, l(imm,e2)))
  Select-BASEINDEX :
    MEM(BINOP(PLUS, e1, e2)) -> l(a, ADDR(l(reg,e1), l(reg,e2), 1, 0))
  Select-BASEINDEXIMM :
    MEM(BINOP(PLUS, BINOP(PLUS, e1, e2), i)) ->
    l(a, ADDR(l(reg, e1), l(reg, e2), 1, l(imm, i)))
  Select-IMM :
    r -> l(a, IMM(l(imm, r)))
  Select-REGA :
    r -> l(a, REGA(l(reg, r)))
  Select-LOAD :
    x -> l(reg, MOVEM(R(REGA(TEMP(<new>))), l(a, x)))
```

Fig. 10. Instruction selection rules mapping IR trees to CISC instructions.

```
module CISC-Costs
imports CISC
rules
  CiscCost : ADD(x, y, z)        -> C1
  CiscCost : MOVEM(x, y)         -> C2
  CiscCost : ADDR(NONE, y, 1, 0) -> C3 where <not(NONE)> y
  CiscCost : ADDR(x, NONE, 1, 0) -> C4 where <not(NONE)> x
  CiscCost : ADDR(NONE, y, 1, i) -> C5 where <not(NONE)> y; <not(0)> i
  CiscCost : ADDR(x, NONE, 1, i) -> C6 where <not(NONE)> x; <not(0)> i
  CiscCost : ADDR(x, y, 1, i)    -> C7 where <not(NONE)> x; <not(NONE)> y
```

Fig. 11. Instruction costs

5.1 Optimum Tiling

The dynamic programming strategy is implemented by `optimum-tiling`, which is parameterized with a cost computation `cost`, a set of normal rules `rs`, and a set of chain rules `is`. The strategy consists of a bottomup traversal followed by a topdown traversal. During the bottomup traversal, the strategy `match-tile` finds out which rules are applicable and which one is the cheapest. This is expressed by defining a dynamic rewrite rule `BestTile` for each subtree `t`, which rewrites `t` tagged with a mode `m` as `l(m,t)` to the corresponding instruction tree. These dynamic rules are then applied in a topdown traversal over the tree. In other words, this corresponds to `maximal-munch` where the rule to be used is determined dynamically instead of statically and can be different at each node.

5.2 Match Tile

Strategy `match-tile` first creates a pair of the subject term and the application of the rules to the subject term. If one of the rules succeeds, it is registered by `register-match`. Then `fail` forces backtracking to another rule, until no more rules match. After applying the normal rules, the closure of the chain rules is computed by repeating the same procedure until no more chain rules apply. The repetition is necessary since one chain rule application can enable another one. This process terminates since `register-match` only succeeds if the cost of the match is lower than the cost of all previous matches.

5.3 Register Match

`Register-match` is the core of the dynamic programming strategy. It gets a pair of a subject `tree` and the result of applying a rule to it, which should be a `tile` tagged with a `mode`. `ComputeCost` returns the cumulative cost of the tile and the costs of the trees at the leafs of the tile. If this cost is less than the `BestCost` so far for the `tree` in this `mode`, the match is registered by generating new *dynamic rules* for `BestCost` and `BestTile` rewriting the `l(mode,tree)` term to the computed cost and the given tile. This overrides any previously generated rules for `l(mode,tree)`. Thus, the best result per mode is computed for each node in a bottomup fashion; suboptimal results are discarded.

Dynamic rewrite rules [9] are just like ordinary rules, except that they inherit the values of any of their variables that are bound in their definition context. Thus the definitions of the rule `BestCost` inherits from its context the values of the variables `mode`, `tree`, and `cost`. A rule specific for these values is generated, which may exist next to other `BestCost` rules, except if the values for `mode` and `tree` are the same. In that case the old rule is overridden.

5.4 Compute Cost

The cost of a match is determined by adding the cost of the tile itself as provided by the parameter strategy `cost` and the `BestCosts` of the leafs of the tile.

```
module IR-to-CISC
imports lib IR-to-CISC-Rules CISC-Costs Dynamic-Programming
strategies
  cisc-select =
    optimum-tiling(CiscCost <+ !0, cisc-tiles, cisc-injections, !s)

  cisc-tiles =
    Select-ADDM ++ Select-MOVEM ++ Select-ADD ++ Select-imm
    ++ Select-reg ++ Select-reg' ++ Select-ABS ++ Select-BASE
    ++ Select-BASEINDEX ++ Select-BASEIMM ++ Select-BASEINDEXIMM

  cisc-injections =
    Select-LOAD + Select-IMM + Select-REGA
```

Fig. 12. Instantiation of the dynamic programming algorithm

```
module Dynamic-Programming
imports lib dynamic-rules
signature
  constructors
    l : Mode * a -> a
strategies
  optimum-tiling(cost, rs, is, rootmode) =
    {| BestTile, BestCost :
       bottomup(match-tile(cost, rs, is));
       !l(<rootmode>, <id>);
       topdown(try(BestTile))
    |}

  match-tile(cost, rs, is) =
    try(   test(!(<id>, <rs>); register-match(cost); fail));
    repeat(test(!(<id>, <is>); register-match(cost)))

  register-match(cost) =
    ?(tree, l(mode, tile));
    where(
      <ComputeCost(cost)> tile => cost
    ; <lt-inf> (cost, <BestCost <+ !Infinite> l(mode, tree))
    ; rules(
        BestCost : l(mode, tree) -> cost
        BestTile : l(mode, tree) -> tile
      )
    )
  ComputeCost(cost) = where(cost => tile-cost)
    ; collect(?l(_,_)); foldr(!tile-cost, add, BestCost)
```

Fig. 13. Dynamic programming algorithm for computing an optimum tiling.

The leafs are obtained by collecting all outermost subterms matching `l(_,_)` from the tile. For example, the tile `ADDR(l(reg,e1), NONE, 1, l(imm,e2))` has cost `C6` and leafs `[l(reg,e1),l(imm,e2)]`. Thus the fold over this list produces `<add>(<BestCost>l(reg,e1), <add>(<BestCost>l(imm,e2), C6))`. If `BestCost` fails for some leaf of a tile, this indicates that no tile was found for the corresponding tree with the mode needed by this tile.

6 Discussion

The instruction selection techniques in this paper were developed as part of a project building a complete Tiger compiler [3] using rewriting techniques. The generic strategies are part of the Stratego Standard Library, which contains many more generic strategies. The examples in this paper are written in Stratego and have been tested with release 0.7 of the Stratego Compiler[1].

The maximal munch algorithm was directly inspired by [3]. However, our separation of selection rules and code emission makes the rules independent of the strategy to be used. The dynamic programming algorithm is based on the ideas of [2,1]. These ideas have been used in several systems including TWIG [1], BEG [5], and BURG [8]. The main differences between these systems is the pattern matching algorithm used and the amount of computation done at generator generation time. BURG [8] is based on bottom-up rewriting (BURS) theory and performs all cost computations at generator generation time, whereas our implementation does all cost computations at code generation time. Furthermore, BURG uses bottom-up pattern matching [6], an efficient matching algorithm that does not reexamine subterms. This is achieved by normalizing patterns to shallow patterns and introducing new modes.

We have not yet performed comparative benchmarks, but our implementation will undoubtely be much slower than the highly optimized BURG code generators. For a list of IR trees of 9200 nodes, the `maximal-munch` strategy took 0.03 seconds to produce a (RISC) instruction tree of 11,100 nodes (2100 instructions), while the `optimum-tiling` dynamic programming strategy took 1.96 seconds to produce a (CISC) instruction tree of 15,000 nodes (1000 instructions). We expect to improve the performance considerably by applying specialization and fusion in the style of [7], where an optimization for specialization of the generic `innermost` strategy is presented. A possible optimization is the normalization of rule left-hand sides in order to achieve better pattern matching.

7 Conclusion

In this paper we have shown the complete code of *several* instruction selection algorithms. Creating a complete code generator only requires more rules, not more implementation. This is clear evidence of the expressivity of the specification of program transformations with rewrite rules and rewriting strategies. Instead of

[1] `http://www.stratego-language.org`

hardwiring the strategy in the rewrite engine, the programmer can define the most appropriate strategy for the task at hand. For example, one could start with a simple maximal-munch strategy for a code generator and only switch to dynamic programming when it is really needed. Furthermore, programmable strategies enable the combination of several transformations each using different strategies. Much of the expressivity of strategies is due to the ability of capturing generic schemas with generic traversal operators. Finally, the incorporation of dynamic programming techniques provides new opportunities for the specification of other kinds of transformations, and poses new opportunities for the optimization of strategies. In particular, it would be interesting to generalize tabulation techniques from code generators to more general transformation strategies.

References

1. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree pattern matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.
2. A. V. Aho and S. C. Johnson. Optimimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1976.
3. A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
4. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, Singapore, September 1996.
5. H. Emmelmann, F.-W. Schroer, and R. Landwehr. BEG - a generator for efficient back ends. In *ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI'89)*, pages 227–237. ACM, July 1989.
6. C. H. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.
7. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
8. T. A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.
9. E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.
10. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
11. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

A Code Emission

After covering the IR tree with instructions, the program is still in tree shape. *Code emission* linearizes the tree into a list of instructions. For example, for the tiling in Figure 6 we get the code sequence of Figure 4.

Code emission is simply a matter of putting the nodes of the instruction tree in post-order. That is, collect each node and put the nodes collected from the direct subterms before it, and replace direct subterms with result register in which they leave their value. Figure 14 presents the complete specification of code emission for RISC instructions.

A.1 Emit

The `Emit` strategy recognizes which subtrees correspond to instructions by means of the non-deterministic choice (the operator `_+_`) between the instruction patterns. The instructions are recognized and transformed by means of congruence operators. For each n-ary constructor `C` declared in a signature, a corresponding *congruence operator* `C(s1,...,sn)` is defined, which transforms a `C(t1,...,tn)` term into `C(<s1>t1,...,<sn>tn)`, i.e., applying the `si` strategies to the corresponding subterms of the constructor application. The congruence fails when applied to term constructed with any other constructor.

The `reg` transformation is applied to the direct subterms of an instruction to replace a complete instruction tree with just the register in which it leaves its result. The definition of `reg` leaves `REG` and `TEMP` nodes alone. An `R` node is reduced to its argument using a match and project operator. A match and project operator `?C(...,<s>,...)` matches the subject term against the pattern, and at the position of `<s>` the transformation `s` is applied. If this succeeds the result term is the result of the transformation. For example, `?R(<id>)` applied to `R(TEMP("x_0"))` is `TEMP("x_0")`. Any other node is expected to be an instruction with an `R(_)` term as one of its argument; the register containing the intermediate result. The construction `op#(xs)` generically deconstructs a term into its constructor `op` and the list of its direct subterms `xs`. Put together, the strategy `LOADI(reg,reg,id)` turns the instruction tree

```
module RISC-Code-Emission
imports RISC
strategies
  emit-code = postorder-collect(Emit)

  Emit = ADD(reg, reg, reg) + ADDI(reg, reg, id)
    + LOADC(reg, id) + LOAD(reg, reg) + LOADI(reg, reg, id)
    + STORE(reg, reg) + STOREI(reg, reg, id)

  reg = REG(id) + TEMP(id) + ?R(<id>) <+ ?op#(<fetch(?R(r)); !r>)
```

Fig. 14. Code emission rules and strategy

```
LOADI(R(TEMP("x_0")),ADD(R(TEMP("y_0")),TEMP("B"),TEMP("C")),8)
```

into the instruction

```
LOADI(TEMP("x_0"),TEMP("y_0"),8)
```

A.2 Post-Order Collect

The `emit-code` strategy flattens a tree by collecting all subterms for which `Emit` succeeds and putting them in post-order. This tree flattening process can be expressed as a generic transformation strategy for flattening arbitrary tree structures. The strategy `postorder-collect` is part of the Stratego Standard Library and is defined as

```
postorder-collect(s) = postorder-collect(s, ![])
postorder-collect(s, acc) =
  where((s => y; ![y | <acc>] <+ acc) => ys);
  crush(!ys, \ (x, xs) -> <postorder-collect(s, !xs)> x \ )
```

The first line defines `postorder-collect/1` in terms of `postorder-collect/2` passing as extra parameter the transformation that builds the empty list, which is the way to pass a term to a strategy in Stratego. The second definition tries to apply the parameter strategy `s` to the subject tree. If that succeeds it inserts it in the accumulation list `acc`, otherwise nothing is added to the list. Subsequently the subterms of the subject tree are collected recursively by means of a `crush`. In general, `crush(zero, plus)` generically deconstructs a constructor application `C(t1,...,tn)` into `<plus>(t1,...<plus>(tn, <zero>))`, i.e., performs a fold over the direct subterms of the node.