Efficient instruction selection using attribute grammars

Rik van de Ven,
Utrecht University

Version
November 7, 2002

Abstract

Modern day compilers are designed in such a way they can be easily retargeted to different machines. A common way to model such a retargetable compiler is by seperating the compiler into a language dependent front-end and a machine dependent back-end. The front-end translates an intermediate program into an intermediate presentation. The back-end generates machines code for some machine from this intermediate program. The back-end is usually generated by feeding a machine description into some generic back-end. This is done at compile-compile time, making it possible to partially evaluate parts of the back-end.

An important part of the back-end is the instruction selection problem. The goal of instruction selection is to find the optimal set of instruction which can be used to represent the entire intermediate program. In this thesis we examine how the instruction selection problem can be solved using attribute grammars. We start by introducing a basic and inefficient tree matching method. This method is improved until we get an efficient compiler. Besides some small partial evaluations their are two major techniques introduced to achieve our goal.

Looking at the instruction selection problem using attribute grammars it is obvious we have a lot of attributes for every nonterminal. For every node the attributes depend merely on a small subset of the attributes of its children. By modelling the tree matcher in the right way this dependencies can be found by analyzing the attribute grammar. By doing so we can introduce a specialized version of the attribute grammar which holds only those attributes needed. Such an analysis can not only be used for the tree matcher, but for any attribute grammar having not needed attributes at some point. The analysis is modelled in such a way that it can be used for any attribute grammar.

Although removing the attributes creates a more efficient tree matcher, there are still a lot of calculations necessary at every node. Inspecting the tree matcher shows this calculations only depend on two things: the operators node and the costs for every storage class at the children. The number of operators is limited. The number of costs for every storage class can be limited by using normalized costs instead of absolute costs. When both these are limited we can partially evaluate the attribute grammar for the tree matcher and construct an automaton for every possible operator and set of costs per storage class. When we have such an automaton the tree matching at compile time is no more then a table lookup, taking only constant time at every node of the input program. This gives us an efficient tree matcher constructed using attribute grammars.

Contents

Chapter 1

A common model for retargetable compilers

1.1 Preface

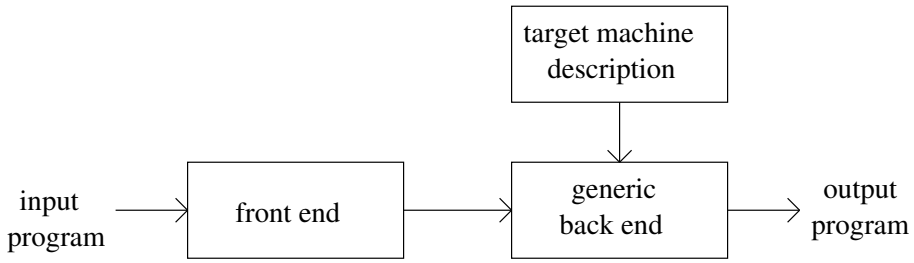
The subject of designing efficient compilers which are able to produce efficient program code has been studied for a long period of time. Since higher level programming languages became more popular, people got interested in the possibility to compile one program to different machines. Most higher level languages are machine independent. Hence it is possible to write a compiler for such a language for many machines. If we code each compiler by hand, we are doing a lot of unnecessary work. The compilers share identical optimizations and possibly identical bugs. Obviously a lot of effort has been made to write compilers in such a way, that they can be easily adapted to compile for some new machine. When a compiler is designed that way, we speak of a retargetable compiler.

There already has been done much work in the field of retargetable compilers. Our goal is to show how a part of the compiler can be implemented using attribute grammars. We will only focus on a small part of a compiler. To clarify which part this is, we will first need an understanding of the different parts of a compiler.

1.2 Structure of a compiler

Basically a compiler just reads an input program and transforms it into some lower level program. A C-compiler for Intel x86 based machines will read any C-program and deliver x86-assembler as result. Of course this process can be seen as a couple of loose tasks. First of all a compiler must be able to parse input C-programs. It must be able to select and schedule machine instructions and it must allocate memory and registers. In most cases we want our compiler to perform some optimizations too. We want the compiler to select those instructions which will result in fast program execution. This is just an example; we can optimize on many aspects of the program. We can for instance be not so much interested in program execution speed, but more in program size. Then a different way of optimization is necessary. Retargetable compilers take a machine description in addition to the input program. This is a description of the machine we are compiling to.

Figure 1.1: Structure of a retargetable compiler



In most cases this model is expanded a little bit. Since most parts of the compiler are machine independent (for instance parsing of a C-program) most compilers are split into a front end and a back end. The front end is source language dependent but completely machine independent. The back end is source language independent and handles all the machine dependent tasks (see figure refcompstruct). In this case, it suffices to only feed the machine description into the back end. The front end parses the program and performs any other tasks which can be done without knowledge about our target machine. This results in a program in some intermediate representation. The back end then takes this intermediate program and produces the machine dependent code for it. This gives us another advantage. We can write front end code for different languages. Making them all result in programs in the same intermediate representation we can use the same back end for all these different programming languages. In this way we get a compiler which can be used to map many different program languages to many different target machines. All optimization techniques embedded in the back end will be available for all the languages, which saves lots of repetitive work. In most cases the front end performs any possible machine independent optimizations.

We generate machine specific back ends by feeding machine descriptions into a generic back end. The machine descriptions are known when we are designing the compiler. This raises the question if it is possible to exploit the fact that the compiler uses identical calculations at some part. It is interesting to search for such identical calculations. If we find them it may be possible to create a more efficient back end by partially evaluating the back end. A common approach is to use a back end generator. This generator reads some machine description and delivers a machine specific back end. We only need to generate this back end once for every machine description. This can be done at compile-compile time (being the compilation time of the compiler itself, not at the time the compiler compiles some input program). When we do this at compile-compile time, we get a faster compiler. It also makes it possible to exploit the fact that large parts of the back end may be partially evaluated. For this thesis we are only studying how to generate a small part of the back end of a compiler. The goal is not to build a complete working compiler, we are just examining how one solves a small part of the whole problem. For further reading about complete compiler generation see [4] and for a complete implementation of a C compiler written in C, read the book on compiler design by Hanson and Fraser[5].

1.3 The back end in detail

The back end is the part of the compiler which generates code for a certain machine. This code generation process consists of three main subproblems. These problems are:

- instruction selection: the mapping of the intermediate representation onto the instructions and addressing modes of the target machine
- register allocation: the mapping of the intermediate values and variables occuring in the intermediate representation onto the registers (and memory locations) of the target machine
- instruction scheduling: the ordering of the target machine instructions

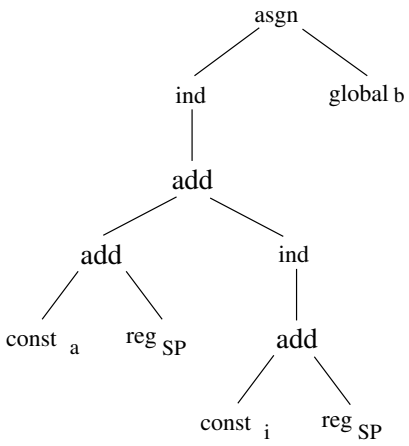
We are looking for a compiler which gives us machine code optimized on some variable aspect. The question arises how we can get optimal results. Looking at the subproblems separately, we can see optimality is interdependent. One might find an optimal solution for the instruction selection, but it might be that the restricted number of available registers introduces spilling[1], in the end yielding a not optimal total solution. For a truly optimal solution for the code generation problem it is necessary to solve all problems together. This however will make writing fast compilers difficult or maybe even impossible. Therefore we will not search for optimal solutions, but solutions which are considered close enough to optimal. When we speak of optimal, we do not mean we have found the optimal solution for the whole problem. Nevertheless we will speak of optimality quite often, and by doing so, we just state we have found a solution which is optimal for one of the subproblems. In practice we see that by finding an optimal solution for one of the subproblems, we often get a pretty good solution for the whole problem. This is to be expected, if we look at the fact that for instance the Intel x86 processor has a large set of instructions and lots of registers available. If we solve instruction selection well for this machine, we get a good complete solution. A common approach for solving the complete problem is to first do the instruction selection assuming we have an infinite number of registers. After this we use an algorithm called graph coloring to handle the register allocation. This might introduce some overhead because we do not have enough registers available. But on machines having a reasonable number of registers this approach yields good overall results[3].

1.4 Instruction selection

There are many ways to solve the instruction selection problem. We are searching for an approach expressed using attribute grammars. One of the common methods for the instruction selection problem is based on tree matching. If we are going to use attribute grammars a method using trees is an obvious choice. The tree matching (or tree covering) approach for the instruction selection problem has been studied thoroughly. A couple of articles about treematching worth studying are written by Emmelman et. al. [7], by Fraser, Hanson and Proebstring [6] and by Aho, Ganapathi and Tjiang [1]. We will use the examples and ideas

[1] to obtain an empty register some register is temporarily stored in memory and later (when needed) read from memory back into the register

Figure 1.2: Example intermediate-code tree.



presented in the article by Aho, Ganapathi and Tjiang [1] to clearify the process of treematching.

As mentioned instruction selection is the process of mapping a program in the intermediate representation onto the intructions and addressing modes of the target machine. The intermediate program is a tree. Let us look at an example of how we can represent a program in an intermediate tree. The program consists of an assignment statement `a[i] := b`. Both `a` and `i` are local variables stored on the stack. There runtime addresses are given as offsets, $\texttt{const}_\texttt{a}$ and $\texttt{const}_\texttt{i}$, from a stack pointer stored in register `SP`. Figure 1.2 shows the tree for this assignment.

The address of the first element of the array `a` is found by adding the value $\texttt{const}_\texttt{a}$ to the contents of register `SP`; the value of `i` is in the location obtained by adding the value $\texttt{const}_\texttt{i}$ to the contents of register `SP`. In the tree, the `ind` operator makes its argument a memory address.

For code generation, the target machine instructions can be represented by tree-rewriting rules. A rule consists of a replacement node, a tree template, a const and an action. The target code can be found by repeatedly finding subtrees in the intermediate tree that match templates and rewriting the matched subtrees by the corresponding replacement nodes. This results in a sequence of subtrees. This is called a *cover* of the intermediate tree. The target code is emitted by the actions associated with the rules used in this cover. The total cost is the sum of the costs of the covering rules.

The templates we match the intermediate tree on are a set of tree-rewriting rules. All these rules combined describe a complete machines instruction set. We refer to this set as the machine description. A machine descriptions is a set of tree-rewriting rules. A single tree-writing rule is of the form: $\mathit{replacement} \leftarrow \mathit{templatecost} = \mathit{instruction}$ where

1. *replacement* is a single node,
2. *template* is a tree,
3. *cost* is the cost representation for this rewriting rule,

Figure 1.3: Example machine description

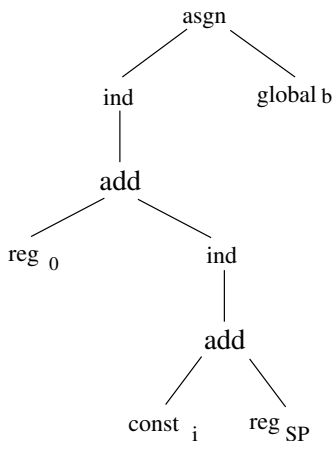
	replacement	template	cost	instruction
(1)	reg_i	const_c	2	MOV #c, Ri
(2)	reg_i	mem_a	2	MOV a, Ri
(3)	statmt	ASGN(mem_a,reg_i)	2 + cost.reg_i	MOV Ri, a
(4)	statmt	ASGN(IND(reg_i),global_b)	2 + cost.reg_i	MOV b, * Ri
(5)	reg_i	IND(ADD(const_c,reg_j))	2 + cost.reg_j	MOV c(Rj), Ri
(6)	reg_i	ADD(reg_i,IND(ADD(const_c,reg_j)))	2 + cost.reg_i + cost.reg_j	ADD c(Rj), Ri
(7)	reg_i	ADD(reg_i,reg_j)	1 + cost.reg_i + cost.reg_j	ADD Rj, Ri
(8)	reg_i			

4. *instruction* is a code fragment in machine dependent instructions.

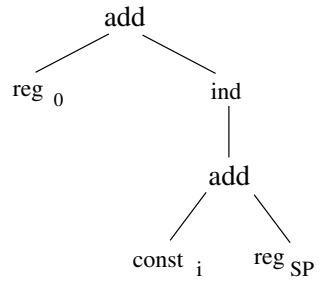
Some of these rules may be only applicable when a certain condition is satisfied. For example, a constant may be required to fall in a certain range. Figure 1.3 presents a machine descriptions for some target machine. The last rewrite-rule can only be applied when the constant has value 1.

The actual instruction selection is done by treematching subtrees from the input program on templates from the machine description. In a depth first traversal we search for a replacement tree at every node. The best match for every match is determined by comparing costs. When we apply a replacement we must add the associated instruction to our solution. In one bottom up traversal we find a solution for the instruction selection problem for a certain input program. If at some point there is another larger match possible at a higher level we delay the decision until we reach that higher level. When we arrive at that level we can again use the costs to decide which rule must be applied.

To illustrate, let us apply the above procedure using the machine description from figure 1.3 applied to the intermediate-tree from figure 1.2. The template of the first rule matches the leftmost leaf of the IR tree with $i = 0$ and $c =$ a. Applying this rule changes the left-most leaf from const_a to reg_0. The associated instruction for this rewriting step is `MOV #a, R0`. We can now apply the template of the seventh rule with $i = 0$ and $j =$ SP. We rewrite the leftmost subtree with root labeled *ADD* into reg_0, the associated instruction for this step is `ADD SP, R0`. After these steps, the resulting intermediate tree looks like:



Now we have arrived at a point where it is possible to apply more then one rule. Examining the lowest subtree IND(PLUS(const$_i$,reg$_S P$)) we immediately notice we can apply rule (5) to reduce the subtree. But there is another rule which can possible be applied at a higher level, being rule (6). Whether this is possible is not clear at this time, so we decide not to rewrite anything yet and move up one level. Here we find rule (6) is indeed applicable. By doing so we can reduce the following subtree:



We can reduce it to a single node reg$_0$ with the associated instruction `ADD i (SP), R0`. We decide to apply the larger match. The resulting complete tree becomes: ASGN(IND(reg$_0$),global$_b$). We can reduce this to a single node with rule(4). The associated instruction is `MOV b, * R0`. Since we have derived a single node the treematching process is finished. Combining all the instructions associated with the combined rules we get the result of the instruction selection:

```
MOV #a, R0
ADD SP, R0
ADD i (SP), R0
MOV b, * R0
```

This is the basic idea behind instruction selection using tree matching. In the following chapter we will look at how this looks in practice, using attribute grammars.

Chapter 2

A simple tree matcher using attribute grammars

2.1 Overview

In this chapter we describe how to implement a tree matcher as described in the previous chapter. We will describe the tree matcher using attribute grammars[1] and the Haskell language. Our goal in this chapter is not to produce an efficient tree matcher. We just want to illustrate the working of a tree matcher. In later chapters we demonstrate faster approaches. To understand the ideas exploited there a good understanding of the techniques explained in this chapter is important.

2.2 The needed data

As described in section 1.4, we are going to match some machine instructions on an intermediate representation of a program. We want to be able to describe this process in some way. Therefore we immediately need some way of representing the machines instuctions and the intermediate program. Some decisions may seem awkward at this point. They will become clear in later chapters.

2.2.1 The intermediate representation

We will use attribute grammars to describe our tree matcher. Therefore our intermediate program must be a tree. Type inferencing is machine independent, we assume this is handled by the front end. If this is the case our intermediate program will be typed correctly and all necessary additional program statements needed for the program being typed correctly are already part of the intermediate tree. For the time being we will assume the intermediate tree only contains nullary, unary and binary operators. Most operators follow this assumption. If we encounter operators having more than two arguments, it is always possible to rewrite them to a binary form. We will show this later.

[1] we use Utrecht University's AG system, see [2]

Now consider a very simple example of a 'program':

```
reg_1 = x + 3
```

If x is located at memory location 8, the intermediate representation looks something like this:

```
ASGNI4 (REG 1) (ADDI4 (MEM 8) (CONST 3))
```

Basically, we just have a nonterminal containing an operator, possibly some arguments and possibly a value. Described using an attribute grammer this may look like:

```
DATA Tree | Nil
          | Nullary op    : Operator
                    value : Value
          | Unary   op    : Operator
                    left  : Tree
                    value : Value
          | Binary  op    : Operator
                    left  : Tree
                    right : Tree
                    value : Value
```

The operator is just a datatype having a constructor for every operator and no other arguments. The value needs to be able to hold all possible values. We can have an alternative for integers and an alternative for strings. Quit often there is no value at all. Since we do not want to get lost in needless details, we assume the value is always an integer. To handle the cases where there is no value at all, we make it type $Maybe\ Int$.

We now have a way to denote programs in intermediate form without having to worry to much about details. The solution may seem a bit simple and incomplete but it is a solution capable of representing everything needed. We do simplify a lot; for instance, function calls are completely left out of the intermediate representation. This is for a reason. We are investigating instruction selection inside procedures, not on a global level. Therefore all details which are of no concern here are left out, since they will make things unnecessarily complicated. Needless to say all techniques described can be implemented using a complete intermediate representation. This would draw attention to details which are not interesting for the way the instruction selection works.

2.2.2 Representing machine instructions

We can represent machine instructions in any way which suits us. A machine description is no more than a list of instructions. For each instruction we need to specify the result it yields and the arguments it needs. Since we are looking for some form of optimality, we must be able to assign a cost to an instruction. We will refer to the result of an instruction as a 'storage class'. This might not be the best term for it. But it is a term which excludes confusion about what we mean. Now consider a very simple (purely illustrative) machine. It is called 'Simple'. The complete instruction set for the Simple machine is shown in figure 2.2.2 It is made out of a few components. Every line represents an available instruction. It

Figure 2.1: Machine description for the *Simple* machine

```
statmt: ASGN(dst,plus1) 2
statmt: ASGN(dst,plus2) 6
statmt: ASGN(dst,plus3) 2
regx  : MEM(addr)       3
regx  : CONST           6
regx  : CONST           2
dst   : addr
src   : regx
src   : MEM(addr)
addr  : regx
addr  : CONST           2
regx  : REG
plus2 : PLUS(src,con1)
plus3 : PLUS(src,con2)
con1  : CONST
con2  : CONST
plus1 : PLUS(src,regx)
```

starts with a storage class which is the result of applying the instruction. Then its operator (if any) is given, followed by its arguments. At the end of the line the cost of the instruction is specified. Instructions which have no costs assigned are considered to have zero cost.

For our tree matcher this machine is represented as a list of rules. Each instruction (or rule) specifies its result and its right hand side. We add a function with returns the associated costs for each rule. The right hand side of a rule is made out of an operator and its arguments. We introduce a special operator called 'Noop' which will be used for instructions having no operators. Unlike the other operators we do not refer to it in uppercase. This is to distinguish it from the others. The 'Noop' operator is not supposed to be part of intermediate programs. We introduce a special storage class for operators having no arguments, called 'Nil'. A machine description for the Simple machine is given in figure 2.2. The example is based on an example found in [4]. We now have gathered all the material needed to express a simple tree matcher. But first we will provide some more details about the above.

Chainrules

In the Simple machine descriptions there are some instructions which have no operators at all. The collection of all the rules having no operators will be called 'the chainrules'. Most machines have a couple of chainrules. These are just some basic operations. We can for instance read a value out of a register and temporarily store it in memory. This is quite a common operation. The chainrules are those rules which have the 'Noop' operator. Earlier we mentioned that an intermediate program will not use this operator. The rules corresponding to this operator will therefore never be selected. But the instructions for this rules may be used at all times so we must pay special attention to the chainrules since chainrules can provide us with better results for the instruction selection problem.

Figure 2.2: Simple machine description in Haskell

```
type Rulenr = Int
type Rule = (Rulenr, Storage, Rhs)
type Rules = [Rule]
data Rhs = Rhs Operator Storage Storage

data Operator = Noop | ASGN | CONST | MEM | PLUS | REG deriving (Eq)

data Storage = Nil | Addr | Con1 | Con2 | Dst
                | Plus1 | Plus2 | Plus3
                | Regx | Src | Statmt deriving (Eq)

rules :: Rules
rules = [(1, Statmt, (Rhs ASGN Dst Plus1)),
         (2, Statmt, (Rhs ASGN Dst Plus2)),
         (3, Statmt, (Rhs ASGN Dst Plus3)),
         (4, Regx, (Rhs MEM Addr Nil)),
         (5, Regx, (Rhs CONST Nil Nil)),
         (6, Regx, (Rhs CONST Nil Nil)),
         (7, Dst, (Rhs Noop Addr Nil)),
         (8, Src, (Rhs Noop Regx Nil)),
         (9, Src, (Rhs MEM Addr Nil)),
         (10, Addr, (Rhs Noop Regx Nil)),
         (11, Addr, (Rhs CONST Nil Nil)),
         (12, Regx, (Rhs REG Nil Nil)),
         (13, Plus2, (Rhs PLUS Src Con1)),
         (14, Plus3, (Rhs PLUS Src Con2)),
         (15, Con1, (Rhs CONST Nil Nil)),
         (16, Con2, (Rhs CONST Nil Nil)),
         (17, Plus1, (Rhs PLUS Src Regx))]

rulecost :: Rulenr -> Int
rulecost 1 = 2
rulecost 2 = 6
rulecost 3 = 2
rulecost 4 = 3
rulecost 5 = 6
rulecost 6 = 2
rulecost 11 = 2
rulecost _ = 0
```

Transforming rules

So far we have introduced machine description with two important restrictions. Every rule contains at most one operator and every operator is at most binary. This limitation may easily be worked around. If we want to use an operator having three children, it can easily be rewritten to a binary form using tree translations. Imagine we have an operator *TripleOp* hich takes three arguments. It can be rewritten to a binary operator *TripleOp* with as first argument the first argument of the original operator. The second argument must be a special storage class *Sttripleop*. We add an extra rule which takes the second and third arguments of the original operator and results in the new storage class. Assume the definition for *TripleOp* was `regx : TRIPLEOP(dst,plus1,regx)`. This will give us the following rules:

```
rules = [...,
         (88, Regx, (Rhs TRIPLEOP Dst Sttripleop)),
         (89, Sttripleop, (Rhs Noop Plus1 Regx)),
        ...]
```

It is easy to rewrite any intermediate program to this form.

This still leaves us with the limitation of one operator per rule. When we introduced the ideas behind treematching we had an example (1.3) which gained a lot by applying a rule with more than one operator. An example of such a rule is `regx : PLUS (regx,MEM(addr))`. This rule states that if we add something to a register and the second argument is put in memory from an address, there is a special instruction which can probably do the addition without first reading the memory value into a register for the second argument. We can again use tree-rewriting to handle these kind of instructions. For the above instruction we simple add two rules:

```
rules = [...,
         (103, Regx, (Rhs PLUS Regx En1)),
         (104, En1, (Rhs MEM Addr Nil)),
        ...]
```

This can easily be done automatically. By doing so we can now handle rules with more than two arguments and more than one operator. It introduces some overhead but we now can use the fact that for the tree matching we only need to consider rules with at most one operator which is at most binary. This will be used in later chapters.

Machine code

When looking at a complete back-end it is important to have the machine code belonging to each instruction. Without it is impossible to generate any code at all. This code is part of the above machine descriptions. For instance, a part of the machine description for Intel's x86 family of processors will look like this:

```
stmt: ASGNI1(addr,rc)  "mov byte ptr \%0,\%1\n"   1
stmt: ASGNI2(addr,rc)  "mov word ptr \%0,\%1\n"   1
stmt: ASGNI4(addr,rc)  "mov dword ptr \%0,\%1\n"  1
```

The part between the double quotes represents the x86 machine code. The part `%0` must be replaced by the generated code associated with the first child (`addr` in this example). The part `%1` must be replaced with the next childs code.

In the rest of this thesis the code is ignored, because it is of no concern to us. Register allocation is not considered, so it is impossible to produce meaningfull code. The actual code for the selected instructions is generateded after register allocation has been done. For the techniques of instruction selection the actual machine code is not important and will only distract us from what we are trying to achieve: selecting the right instructions. In the next section we will use the machine descriptions and the intermediate program to model a simple tree matcher. Normally we will transform the intermediate tree in some other tree (which is labeled with machine instructions at every node). For this thesis we are only interested in the instructions, so a list of instructions will suffice. The tree matcher will be a semantic function on the intermediate tree having a list of selected instructions as its result.

2.3 The basics

It is time to look at the basics of the tree matcher. We will use the machine description as described in 2.2.2 and 2.2. To get a feeling about how to model a tree matcher, we first make a strange assumption. For now, we assume the instructions a machine offers are mutually exclusive. Chainrules are out of the picture too. By these assumption the machine description has become an ambiguous grammar. This is not realistic and these assumptions will be dropped soon. But for the time being they suit our needs in expressing the basic idea of a tree matcher.

Having a mutually exclusive instruction set, we know we can only apply one rule in every node. Which rule will depend on the nodes operator and the results of its children. Since nullary operators do not have any children, we can do nothing there but apply the rule corresponding to the operator in the node (the mutual exclusiveness implies there is only one rule satisfying the operator). Combining this with our intermediate tree described in 2.2.1 we get a very simplystic tree matcher. In one bottom up traversal all instructions are selected. We start with the nullary operators, which are the easiest:[2]

```
{
rule_cop :: Operator -> Rule -> Bool
rule_cop op1 (_, _, Rhs op2 _ _) = (op1 == op2)

rule_opget :: Operator -> Rules
rule_opget op = filter (rule_cop op) rules
}

ATTR Tree [ | | deriv : {[Rulenr]} stor : Storage ]
SEM Tree
   | Nullary loc.rule       = head (rule_opget @op)
             lhs.deriv      = list (fst3 @loc.rule)
             lhs.(_,stor,_) = @loc.rule
```

For the unary operators it is a little bit more complicated. Every unary operator can have multiple rules, mutual exclusivity only implies there is only one rule to be applied for every possible childs storage class. The derivation becomes the selected rule, followed by the deriviation of its child. Rule selection remains quite simple:

[2] The functions $fst3$ and $snd3$ will be used everywhere. They work like the normal fst and snd functions, but the integer states on what kind of tuples they operate. For example, the function $foth6$ will take the fourth element of a six-tuple.

```
SEM Tree
   | Unary loc.rule       = head $ filter (\(_,_,(Rhs _ st _)) -> st == @left.stor) (rule_opget @op)
          loc.deriv      = list (fst3 @loc.rule)
          lhs.deriv      = @loc.deriv ++ @left.deriv
          lhs.(_,stor,_) = @loc.rule
```

For the binary operators the approach is similar but now we match on both childrens non-terminals:

```
SEM Tree
   | Binary loc.rule       = head $ filter (\(_,_,(Rhs _ stl str)) -> stl == @left.stor &&
                                                                      str == @right.stor) (rule_opget @op)
            loc.deriv      = list (fst3 @loc.rule)
            lhs.deriv      = @loc.deriv ++ @left.deriv ++ @right.deriv
            lhs.(_,stor,_) = @loc.rule
```

Allthough not very realistic nor useful, we now possess a basic tree matcher. At the root of our tree the attribute *deriv* holds the set of rules for expressing the intermediate tree for a given instruction set. This is used as a framework which will be expanded to make it more and more realistic. The mutual exlusivity assumption must be removed as soon as possible. But what will be the result of this? To start with, since we stated only one rule can be selected, we know every node can yield only one storage class. By dropping the exclusivity this no longer holds. Every node can yield different storage classes, each having its own rule. We have to pass a list containing pairs of storage classes and rules. By doing so, it is no longer possible to build a deriviation in one single run. All possible storage classes at every node can be calculated in one bottom up traversal. After that, by selecting a storage class at the root, it is possible to select the correct rule for each node in one top down traversal of the tree. So we derive a tree matcher having:

- a synthesized attribute holding a list of storage classes and associated rules
- an inherited attribute specifying which storage class a parent node wants

Combining these two gives us the deriviation.

Dropping the mutual exclusiveness means more than this. Not only can different rules give us different storage classes, it is also possible that we have different rules yielding the same storage class. Which rule should we apply? We can just randomly select one that suits our needs. But we are looking for an optimal selection of the instruction selection problem. Optimality means that we have chosen the *cheapest* rule in parts of the tree where multiple solutions are possible. So this is the part where we start looking at the costs. It is a little more complicated than simply selecting the cheapest rule. This is because the cost of a derivation is made out of two components. The cost of selecting a certain rule in a certain node and the cost of deriving its corresponding storage class for the childs nodes. We can for example have a unary operator with two matching rules both resulting in a certain storage class $s1$. The first rule has cost 1 and the second rule has cost 4. But the first rule wants its child to present its result in storage class $s2$, while the second rule takes storage class $s3$. It is easy to see that when we can get a value in $s2$ at the child for cost 5 and in $s3$ for free, the second rule is the rule to apply.

To make these selections possible it is necessary to pass the cost parameter combined with every storage class and rule. Our synthesized storage attribute becomes a list of storage classes with associated rules and costs. Allthough the chainrules have not yet been examined, the tree matcher is starting to get quite realistic. All the above expansions have made the matching process a bit more complicated, but it still is quite straightforward.

Types

Besides the types already introduced in 2.2 the following types are needed for passing data:

```
type Deriv = [Rulenr]
type Derivs = [Deriv]
type LabelValues = [LabelValue]
data LabelValue  = Lv Storage Rulenr Int
```

Rule functions

To make life a bit easier, we introduce a set of functions for retrieval and comparison on rules:

```
{
rhs_op :: Rhs -> Operator
rhs_op (Rhs op _ _) = op
rhs_lstor :: Rhs -> Storage
rhs_lstor (Rhs _ l _) = l
rhs_rstor :: Rhs -> Storage
rhs_rstor (Rhs _ _ r) = r

rule_rhs :: Rule -> Rhs
rule_rhs = thd3

rule_nr :: Rule -> Rulenr
rule_nr = fst3
rule_stor :: Rule -> Storage
rule_stor = snd3
rule_op :: Rule -> Operator
rule_op = rhs_op . rule_rhs

rule_lstor :: Rule -> Storage
rule_lstor = rhs_lstor . rule_rhs

rule_rstor :: Rule -> Storage
rule_rstor = rhs_rstor . rule_rhs

rule_cop :: Operator -> Rule -> Bool
rule_cop op rule = op == (rule_op rule)
rule_cnr :: Rulenr -> Rule -> Bool
rule_cnr rnr rule = rnr == (rule_nr rule)

rule_opget :: Operator -> Rules
rule_opget op = filter (rule_cop op) rules
```

```
rule_get :: Rulenr -> Rule
rule_get nr = head $ filter (rule_cnr nr) rules

deriv_get :: Storage -> LabelValues -> Deriv
deriv_get st [] = [0]
deriv_get st1 ((Lv st2 rls _):xs) | st1 == st2 = rls
                                  | otherwise = deriv_get st1 xs
}
```

These functions will be used everywhere when addressing to rule information. If the structure of the machine description changes, it is not necessary to adapt the whole tree matcher. Adjusting the above functions to suit the changes will suffice.

Operator application

As stated above, the application of the operators has become a little bit more complicated. Instead of a storage class per child, we have a list of storage classes per child. Second we have to take costs into account. We introduce a Haskell function *lv_build* which given an operator and the labelvalues of a nodes children returns the new labelvalue for the current node:

```
{
lv_build :: Operator -> LabelValues -> LabelValues -> LabelValues
lv_build op ll lr = nub $ sort $ (lv_applyop op ll lr)

lv_applyop :: Operator -> LabelValues -> LabelValues -> LabelValues
lv_applyop op left right =
    [ (Lv xst [xnr] ((rulecost xnr) + (lv_cost left xlstor) + (lv_cost right xrstor)))
    | (xnr, xnt, (Rhs xopt xlstor xrstor)) <- (rule_opget op)
    ]

lv_cost :: LabelValues -> Storage -> Int
lv_cost [] ri = infCost
lv_cost ((Lv nt fri sri):xs) n
     | nt==n = sri
     | otherwise = lv_cost xs n

infCost :: Int
infCost  = 99999999 --some number representing 'infinite' cost
}
```

There still is nothing to complicated going on. Every rule which can possibly be applied is applied. This results in a list of —labelvalue elements—. Because we want the cheapest way to derive every storageclass, the list is first sorted and then —nubbed— to remove any double derivation per storage class. This requires an instance of equality and ordering on labelvalues. It must be in such a way that a labelvalue's equality is only determined by its storageclass. For the ordering we examine the costs, so the labelvalues get ordered by ascending costs. We get:

```
instance Eq LabelValue where (==)    = eqlv
eqlv :: LabelValue -> LabelValue -> Bool
eqlv (Lv nt1 _ _) (Lv nt2 _ _) = (nt1==nt2)
```

```
instance Ord LabelValue where compare = complv
complv :: LabelValue -> LabelValue -> Ordering
complv (Lv nt1 rls1 cst1) (Lv nt2 rls2 cst2)
                                  | nt1 < nt2 = LT
                                  | nt1 > nt2 = GT
                                  | cst1 < cst2 = LT
                                  | cst1 > cst2 = GT
                                  | length rls1 < length rls2 = LT
                                  | length rls1 > length rls2 = GT
                                  | otherwise = EQ
```

To complete things one more thing is added. Currently underivable storage classes are represented using infinite costs. Since we now look at absolute costs, the cost integer will soon overflow. If we have a tree ten levels deep and a certain storage class has infinite costs in all nodes, the addition of these costs will result in an error. But the absolute cost are not really interesting. All we need to know is which storage class is the cheapest, and what the difference is between that class and the others. If a certain node is inspected and its left child has three storage classes: st1, st2 and st3 derivable at costs 10, 14, 17, not all that information is needed to decide what to do in the current node. All we need to know is that st1 is the cheapest, st2 is 4 more expensive and st3 is again 3 more expensive. So we can normalize the costs, giving us costs 0, 4, 7. This is all the information needed. By normalizing everywhere, we limit the cost at a certain level, eliminating the risk of cost integer overflow. We add a function norm_lv and apply it to the result of lv_build.

Tree matching

We now have all the groundwork to model the actual treematching. Extending the tree matcher constructed for mutual exclusive instruction sets with the new operator application results in a very basic tree matcher, see figure 2.3. All future tree matchers are derived from this basic form. The tree matching process must be clear at this point. Despite the fact that we have not arrived at a complete tree matcher for actual machines, we do have the basic framework to produce those. In fact, we only need two minor supplements to derive a working tree matcher for for example the Intel x86 family of processors[3]. These two additions are the use of chainrules and taking conditional instructions into account. We will look into these now, but they do not require any changes to our treematching framework. We can simply modify some of the functions described above, without changing the used grammar or approach. For now the tree matcher as described in figure 2.3 needs the following functions to work:

```
lv_build :: Operator -> LabelValues -> LabelValues -> LabelValues
lv_build op ll lr = lv_norm $ nub $ sort $ (lv_applyop op ll lr)

lv_norm :: LabelValues -> LabelValues
lv_norm lval = map (lv_min . minimum $ map (\(Lv _ _ c) -> c) lval) lval

lv_min :: Int -> LabelValue -> LabelValue
lv_min min (Lv nt rl cst) = Lv nt rl (cst-min)
```

[3] As a basis for the x86 machines description we use a machine description used by LCC described in [5]

Figure 2.3: A basic tree matcher

```
ATTR Tree [ | | labval : LabelValues ]
SEM Tree
   | Nullary loc.labval = lv_build @op [] [] @var
   | Unary   loc.labval = lv_build @op @left.labval [] @var
   | Binary  loc.labval = lv_build @op @left.labval @right.labval @var

ATTR Tree [ derivstor : Storage | | ]
SEM Tree
   | Unary  left.derivstor  = rule_lstor (rule_get $ last @deriv)
   | Binary left.derivstor  = rule_lstor (rule_get $ last @deriv)
            right.derivstor = rule_rstor (rule_get $ last @deriv)

ATTR Tree [ | | deriv : Deriv ]
SEM Tree
   | Nullary loc.deriv = deriv_get @lhs.derivstor @labval
   | Unary   loc.deriv = deriv_get @lhs.derivstor @labval
             lhs.deriv = @deriv ++ @left.deriv
   | Binary  loc.deriv = deriv_get @lhs.derivstor @labval
             lhs.deriv = @deriv ++ @left.deriv ++ @right.deriv
```

2.4 Chainrules

As mentioned in the beginning of this chapter, there is a set of special instructions which have no operator and can therefore always be applied. We called these rules the *chainrules*. Although playing a small part in the concept of treematching, the chainrules do play a big role in the result of the tree matcher. Chainrules can often be used to move values from one storage class to some other. Let us look at an example. The following is a subset of some machine description:

```
rules :: Rules
rules = [(1, s1, (Rhs OP1 s2 Nil)),
         (2, s1, (Rhs OP1 s3 Nil)),
         (3, s2, (Rhs Noop s3 Nil))]

rulecost :: Rulenr -> Int
rulecost 1 = 0
rulecost 2 = 3
rulecost 3 = 1
```

If in a certain unary node having operator $OP1$ and a child giving a labelvalue containing only the storage class $s3$ then we can only apply rule number 2. It is easy to see that there is a cheaper way of deriving storage class $s1$. We can first use rule 3. By doing so, we can apply rule 1 to derive $s1$. But by doing so we have a total cost $rulecost(3) + rulecost(1)$ which is 1. So we have derived the same result with smaller costs. Needless to say we want our tree matcher to be able to apply these chainrules. We can decide to apply chainrules before investigating labelvalues (apply chainrules on synthesized labelvalues) or after the application of an operator. The result is exactly the same. We choose to apply chainrules after application

of an operator. Therefore we can always assume we can not improve a synthesized labelvalue by applying chainrules, since all possible chainrules have been applied.

We need a function which given a certain labelvalue applies all possible chainrules and delivers the resulting possibly improved labelvalue. When applying a chainrule, we must keep in mind that by introducing a new storage class, it can become possible to apply a chainrule investigated in an earlier stage. Therefore we must start the chainrule application all over when a chainrule yields an improvement. We must iterate until no further improvements can be established. Since we have decided to apply chainrules after operator application, adding the chainrule function and changing the operation a slight bit will suffice. Besides that we need to change the type $LabelValue$ since an operator no longer applies just one rule, but possibly a list of rules.

```
{
data LabelValue  = Lv Storage [Rulenr] Int

rule_chain :: Rules
rule_chain = rule_opget Noop

lv_build :: Operator -> LabelValues -> LabelValues -> LabelValues
lv_build op ll lr = lv_norm $ nub $ sort $ lv_chain rule_chain (lv_applyop op ll lr,[])

lv_chain :: [Rule] -> (LabelValues,LabelValues) -> LabelValues
lv_chain rls ([],lv) = lv
lv_chain rls ((i:lv), vlv) = lv_chain rls (lv_chainwalk rls i (lv,vlv))

lv_chainwalk :: [Rule] -> LabelValue -> (LabelValues, LabelValues) -> (LabelValues, LabelValues)
lv_chainwalk [] i (lv,vlv) = (lv,(i:vlv))
lv_chainwalk (r:rl) i@(Lv ir _ _) (lv,vlv)
     | rule_lstor r == ir    = lv_chainwalk rl i (lv_exitchain (lv_applychain r i) lv (vlv) [])
     | otherwise             = lv_chainwalk rl i (lv,vlv)

lv_exitchain :: LabelValue -> LabelValues -> LabelValues -> LabelValues -> (LabelValues, LabelValues)
lv_exitchain lv lvl [] tmp      = (lv:lvl, tmp)
lv_exitchain lv@(Lv st1 _ c1) lvl (l@(Lv st2 _ c2):lvr) tmp
                | not (st1 == st2) = lv_exitchain lv lvl lvr (l:tmp)
                | c1 < c2          = (lv:lvl, lvr++tmp)
                | otherwise        = (lvl, (l:lvr)++tmp)

lv_applychain :: Rule -> LabelValue -> LabelValue
lv_applychain (rnr, rst, (Rhs op stl _)) (Lv st rls rc)
        | not (op == Noop && stl == st) = error "illegal operation in lv_applychain"
        | otherwise = (Lv rst (rnr:rls) (rc + rulecost rnr))
}
```

2.5 Conditional treematching

The only thing left to add before the tree matcher can be used for actual instruction sets is the ability to handle conditional instructions. Until now we assumed every instruction can always be applied. In reality this is not the case. Most machines have instructions which can only be used when a certain condition holds. An obvious example of such an instruction is

a special instruction for adding values. If we add a value to a register, and this value is the constant 1 we can use a special instruction which simply raises the value of the register by one. The instruction for this special case is probably cheaper then the instruction which is able to add any integer from a large range to a register.

The tree matching techniques introduced so far can easily be adapted to handle conditions. We expand a machine description with some extra information, a function which given a rule number and a tree, returns a boolean stating whether the tree holds the condition associated with the given rule. Assume we have two types of condition functions. They are called 'range' and 'memop'. We will look to the functions themselves later, but for now their types will suffice:

```
range :: Tree -> Int -> Int -> Bool
memop :: Tree -> Bool
```

With these functions, the conditions simply be assigned to rules using one pattern matched condition function:

```
condition :: Rulenr -> Tree -> Bool
condition 2 tree = memop tree
condition 5 tree = range tree 0 31
condition 6 tree = range tree 1 1
condition _ _    = True
```

This way we can assign many sorts of conditions to different rules. Unconditional rules always evaluate to $True$. To let our tree matcher use conditions, we only need one slight addition to the operator application function. The function was:

```
lv_applyop :: Operator -> LabelValues -> LabelValues -> LabelValues
lv_applyop op left right =
           [ (Lv xst [xnr] ((rulecost xnr) + (lv_cost left xlstor) + (lv_cost right xrstor)))
           | (xnr, xnt, (Rhs xopt xlstor xrstor)) <- (rule_opget op)
           ]
```

To add condition evaluation it is changed into:

```
lv_applyop :: Operator -> Tree -> LabelValues -> LabelValues -> LabelValues
lv_applyop op tr left right =
           [ (Lv xst [xnr] ((rulecost xnr) + (lv_cost left xlstor) + (lv_cost right xrstor)))
           | (xnr, xnt, (Rhs xopt xlstor xrstor)) <- (rule_opget op),
           condition xnr tr
           ]
```

Since its type has changed, the function call changes too. It wants the intermediate tree for the current node as its parameter. The attribute grammar for tree matching becomes:

```
ATTR Tree [ | | labval : LabelValues tree : Tree ]
SEM Tree
   | Nullary loc.tree   = Tree_Nullary @op @value
             loc.labval = lv_build @op @loc.tree [] []
   | Unary   loc.tree   = Tree_Unary @op @left.tree @value
             loc.labval = lv_build @op @loc.tree @left.labval []
```

```
   | Binary  loc.tree   = Tree_Binary @op @left.tree @right.tree @value
           loc.labval = lv_build @op @loc.tree @left.labval @right.labval

ATTR Tree [ derivstor : Storage | | ]
SEM Root
   | Root   tr.derivstor = @stor
SEM Tree
   | Unary  left.derivstor  = rule_lstor (rule_get $ last @loc.deriv)
   | Binary left.derivstor  = rule_lstor (rule_get $ last @loc.deriv)
            right.derivstor = rule_rstor (rule_get $ last @loc.deriv)

ATTR Root Tree [ | | deriv : Deriv ]
SEM Root
   | Root    lhs.deriv = @tr.deriv
SEM Tree
   | Nullary loc.deriv = deriv_get @lhs.derivstor @labval
   | Unary   loc.deriv = deriv_get @lhs.derivstor @labval
             lhs.deriv = @deriv ++ @left.deriv
   | Binary  loc.deriv = deriv_get @lhs.derivstor @labval
             lhs.deriv = @deriv ++ @left.deriv ++ @right.deriv
```

We already gave the types of two condition functions, *range* and *memop*. In fact, these two functions are the only two condition types found in the x86 instruction set. Therefore we will now give the code for these two functions. By doing so, distributed through this chapter we have provided all necessary components of a complete tree matcher for the Intel x86 family of processors. The implementation of *memop* also demonstrates why we want the condition function to have a tree as an argument instead of only the value associated with a certain node.

```
{
range :: Value -> Int -> Int -> Bool
range Nothing _ _ = False
range (Just v) l u = (l <= v && v <= u)

memop :: Tree -> Tree -> Bool
memop tree1 tree2 = let left2 = treeLeft tree2
                    in (hasindirop left2) && sameTree tree1 (treeLeft left2)

treeLeft :: Tree -> Tree
treeLeft Tree_Nil              = Tree_Nil
treeLeft (Tree_Nullary _ _)    = Tree_Nil
treeLeft (Tree_Unary _ l _)    = l
treeLeft (Tree_Binary _ l _ _) = l
sameTree :: Tree -> Tree -> Bool
sameTree Tree_Nil Tree_Nil
     = True
sameTree (Tree_Nullary op1 v1) (Tree_Nullary op2 v2)
     = op1 == op2 && v1 == v2
sameTree (Tree_Unary op1 l1 v1) (Tree_Unary op2 l2 v2)
     = op1 == op2 && v1 == v2 && sameTree l1 l2
sameTree (Tree_Binary op1 l1 r1 v1) (Tree_Binary op2 l2 r2 v2)
     = op1 == op2 && v1 == v2 && sameTree l1 l2 && sameTree r1 r2
sameTree _ _ = False
```

```
hasindirop :: Tree -> Bool
hasindirop (Tree_Nullary op _)    = (take 5 (show op)) == "INDIR"
hasindirop (Tree_Unary op _ _)    = (take 5 (show op)) == "INDIR"
hasindirop (Tree_Binary op _ _ _) = (take 5 (show op)) == "INDIR"
hasindirop _ = False
}
```

Chapter 3

A more efficient approach to treematching

3.1 Overview

The previous chapter layed the groundwork to solve the instruction selection problem. It is still an inefficient approach. A lot of calculations are done at every node. The number of nodes in a program will likely be quite large. The calculations depend on the operators and storage classes. If it is possible to exploit the fact that the number of operators and storage classes is fixed, the calculation at every node can be done a lot faster. In this chapter the amount of work done at each node is reduced. In other words, the operator application function will be simplified. Because that is the most time consuming function used at every node. The techniques to reduce the amount of work will be introduced step by step throughout this chapter.

We will use the fact that the number of operators and storage classes are fixed; in some way this is evaluating a machine description. It must be easy to retarget the compiler so all used optimizations must be performed automatically on a machine description. Optimizations done by hand are of no interest to us. This chapter concerns the ideas behind possible optimizations, it is not our goal to give exact code specifications.

The machine description can be evaluated at the time we transform it from the readible form to the Haskell form used by the tree matcher. So, like in the previous chapter, the inpus is a machine description of the following form:

```
stmt: ASGNI1(addr,rc)  "mov byte ptr \%0,\%1\n"   1
stmt: ASGNI2(addr,rc)  "mov word ptr \%0,\%1\n"   1
stmt: ASGNI4(addr,rc)  "mov dword ptr \%0,\%1\n"  1
```

This time, instead of simply parsing it and delivering the set of rules needed for the tree matcher, we analyze it after parsing. How it is analyzed and how the results of the analysis are used is what this chapter is about. We will start with the most obvious analyses, and step by step get into less obvious onces. Slowly moving further away from the tree matcher given in the previous chapter, towards a more efficient solution.

Figure 3.1: Simple machine description

```
statmt: ASGN(dst,plus1) 2
statmt: ASGN(dst,plus2) 6
statmt: ASGN(dst,plus3) 2
regx  : MEM(addr)       3
regx  : CONST           6
regx  : CONST           2
dst   : addr
src   : regx
src   : MEM(addr)
addr  : regx
addr  : CONST           2
regx  : REG
plus2 : PLUS(src,con1)
plus3 : PLUS(src,con2)
con1  : CONST
con2  : CONST
plus1 : PLUS(src,regx)
```

3.2 Selecting rules per operator

One of the most obvious improvements is the selection of rules per operator. In every node we inspect, we require the set of rules available for the operator of this node. We get these rules by filtering all the rules. We have a limited set of operators and we know these after parsing a machine description. Therefore we can in advance apply this filter once for every operator. We remove the function rule_opget and replace it by a pattern matched instance of op_get. The result of this function is just a list of rules. Remember the simple machine description as given in figure 3.2. Ofcourse we will deliver the rules as stated in the previous chapter. But we add some extra information, found by analyzing the rules. It is easy to see we can find this without any extra information. Everything can be done automatically. We get:

```
rules :: Rules
rules = [(1, Statmt, (Rhs ASGN Dst Plus1)),
         (2, Statmt, (Rhs ASGN Dst Plus2)),
         (3, Statmt, (Rhs ASGN Dst Plus3)),
         (4, Regx, (Rhs MEM Addr Nil)),
         (5, Regx, (Rhs CONST Nil Nil)),
         (6, Regx, (Rhs CONST Nil Nil)),
         (7, Dst, (Rhs Noop Addr Nil)),
         (8, Src, (Rhs Noop Regx Nil)),
         (9, Src, (Rhs MEM Addr Nil)),
         (10, Addr, (Rhs Noop Regx Nil)),
         (11, Addr, (Rhs CONST Nil Nil)),
         (12, Regx, (Rhs REG Nil Nil)),
         (13, Plus2, (Rhs PLUS Src Con1)),
         (14, Plus3, (Rhs PLUS Src Con2)),
         (15, Con1, (Rhs CONST Nil Nil)),
         (16, Con2, (Rhs CONST Nil Nil)),
```

```
         (17, Plus1, (Rhs PLUS Src Regx))]

rule_opget ASGN = [(1, Statmt, (Rhs ASGN Dst Plus1)),
                   (2, Statmt, (Rhs ASGN Dst Plus2)),
                   (3, Statmt, (Rhs ASGN Dst Plus3))]
rule_opget MEM  = [(4, Regx, (Rhs MEM Addr Nil)),
                   (9, Src, (Rhs MEM Addr Nil))]
rule_opget CONST = [(5, Regx, (Rhs CONST Nil Nil)),
                    (6, Regx, (Rhs CONST Nil Nil)),
                    (11, Addr, (Rhs CONST Nil Nil)),
                    (15, Con1, (Rhs CONST Nil Nil)),
                    (16, Con2, (Rhs CONST Nil Nil))]
rule_opget REG = [(12, Regx, (Rhs REG Nil Nil))]
rule_opget PLUS = [(13, Plus2, (Rhs PLUS Src Con1)),
                   (14, Plus3, (Rhs PLUS Src Con2)),
                   (17, Plus1, (Rhs PLUS Src Regx))]
rule_opget Noop = [(7, Dst, (Rhs Noop Addr Nil)),
                   (8, Src, (Rhs Noop Regx Nil)),
                   (10, Addr, (Rhs Noop Regx Nil))]
```

3.3 Inspecting the chainrules

When looking at the operator application at each node we notice the application of the operator itself now takes a fixed number of steps (the length of the associated rule list for each operator). The chainrule application however is a recursive function. With every new derivable storage class rises the necessity to inspect all the chainrules from the start. This is a time consuming process. It will help a lot if we can change this into a fixed amount of work. This is possible, but the solution is not as straightforward as the one applied for the operators above. When looking at the chainrules, we investigate every derivable storage class. Newly introduced ones will later be inspected too. But when applying chainrules to a labelvalue holding different storage classes, the storage classes do not interfere which each other. Assume we have a function *nubs* which sorts a labelvalue and removes all double occuring storage classes. We assume the *nub* part of the function behaves in such a way that every first occurence of a value is kept, the rest is removed. Therefore we are for sure that if we first sort the list and then nub it, the remaining list holds all the cheapest solutions.

Furthermore we have a function *chain* which given a labelvalue calculates the derivable labelvalue when using chainrules. If we have a labelvalue containing two elements, $s1$ and $s2$, the following holds:

$$nubs(chain([s1, s2])) = nubs((chain\ [s1]) + +(chain\ [s2])) \tag{3.1}$$

This makes it possible to build a new chained labelvalue separately from each element of the original labelvalue. We can calculate the chain value for a single element in advance. When we investigate a machine description we know all available storage classes and also the set of chainrules. We simply call the original chainrule function for every storage class, with cost parameter zero. This gives us for every storage class a list of all possible derivable storage classes for a certain cost. Remember we stated the cost to start with is zero. Therefore, if a certain storage class occurs while treematching, for cost 3, we can simply use the precalculated list and add the cost 3 to every element. When treematching we are not looking at one storage

class but at a labelvalue holding many. So we first look at the chaining for every element of this labelvalue and then use (3.1) to combine the results. For the simple machine description used in the previous section this gives us[1]:

```
lv_chain :: LabelValues -> LabelValues
lv_chain lv = nub $ sort $ concat $ map lab_val lv

lab_val :: LabelValue -> LabelValues
lab_val (Lv nt rls cst) = map (lab_apply rls cst) (lab_list nt)
        where lab_apply r1 c1 (Lv n r2 c2) = Lv n (r2++r1) (c1+c2)

lab_list Dst = [(Lv Dst [] 0)]
lab_list Plus1 = [(Lv Plus1 [] 0)]
lab_list Statmt = [(Lv Statmt [] 0)]
lab_list Plus2 = [(Lv Plus2 [] 0)]
lab_list Plus3 = [(Lv Plus3 [] 0)]
lab_list Addr = [(Lv Dst [7] 0), (Lv Addr [] 0)]
lab_list Regx = [(Lv Src [8] 0), (Lv Dst [7,10] 0), (Lv Addr [10] 0), (Lv Regx [] 0)]
lab_list Src = [(Lv Src [] 0)]
lab_list Con1 = [(Lv Con1 [] 0)]
lab_list Con2 = [(Lv Con2 [] 0)]
```

Needless to say, we have derived much faster chainrule application, since all recursion has been removed. Combined with the faster operator rule selection we have derived a much faster tree matcher than we had in the previous chapter. No strange things were done, in fact, we have just been partially evaluating a machine description. Allthough the new tree matcher is a lot faster, there still is lots of needless work done at every node while matching.

3.4 Even further: exploiting Haskell's lazyness

There is still a lot of calculation overhead going on. In every node we inspect, we calculate all possible derivable storage classes. But in most cases the parent of a certain node will only expect a small subset of all possible storage classes. If we know this subset at the node itself, we will have to do less work. An obvious choice is to simply add an inherited attribute containting the list of accepted storage classes at the parent. We then use this list to calculate only the needed storage classes. But in practice, calculating this storage list and passing it through introduces so much overhead that we do not win by this approach. But why do we want to pass this list? Is Haskell's lazy evaluation not supposed to take care of this, when a parent never requests a certain storage class it must never be calculated. But unfortunately it does not work that way. This is because we pass a list of storage classes. This is a list build up using chainrule application. This chainrule application requires the whole list to be calculated. So though we will never use certain elements of this list, the whole list will always be calculated. We somehow want to get rid of this list. This must be possible, since the maximum length of the list is the number of available storage classes (which is ofcourse available while evaluating a machine description). Instead of passing a list, we can just add a single synthesized attribute for every storage class. With the simple machine description this will lead to:

[1] since all chain rules have cost zero in the simple machine description all chaining is free, hence the zero in each labelvalues element. In reality it is more likely that every storage class can only be derived at a certain cost more than zero.

```
ATTR Tree [ | | dst : LabelValue
              plus1 : LabelValue
              plus2 : LabelValue
              plus3 : LabelValue
              stamt : LabelValue
              addr : LabelValue
              regx : LabelValue
              src : LabelValue
              con1 : LabelValue
              con2 : LabelValue ]
```

But what do we win by this? If we only replace the list attribute for a list of seperate attributes we gain nothing. Part of our tree matcher will look something like this:

```
SEM Tree
   | Unary   loc.labval = lv_build @op @loc.tree [@left.dst,...,@left.con2] []
            lhs.dst    = fromLabval Dst @loc.labval
```

Here $fromLabval$ has type $Storage \rightarrow LabelValues \rightarrow LabelValue$. This is just a more complicated way of doing things without gaining anything. It demonstrates a lot of extra work that must be done. We want to state in the attribute grammar which storage classes are required from our children. Which storage classes we need depends on the operator of a node. To be able to model the requirements we therefore evaluate the operator *into* the treegrammar. Given a machine description and the following grammar, we can model a specialized version of the grammar. The grammar was:

```
DATA Tree | Nil
          | Nullary op    : Operator
                    value : Value
          | Unary   op    : Operator
                    left  : Tree
                    value : Value
          | Binary  op    : Operator
                    left  : Tree
                    right : Tree
                    value : Value
```

Since the amount of operators is fixed and their type (nullary, unary or binary) is known at evaluation time, we can push the operator into the grammar. For the simple machine description from figure 3.2 we derive the specialized grammar:

```
DATA STree | Nil
           | CONST value : Value
           | REG   value : Value
           | MEM   left  : Tree
                   value : Value
           | PLUS  left  : Tree
                   right : Tree
                   value : Value
           | ASGN  left  : Tree
                   right : Tree
                   value : Value
```

Needless to say, this transformation can easily be done automatically. Programs in the original intermediate tree form can easily be transformed into this new tree shape in one bottom up traversal of the tree. What do we gain by this transformation? If we only apply this transformation, again, nothing. But the new grammar has a major advantage, operators are hard coded into the grammar. We already knew which rules belong to each operator. Now we are going to 'fold' the operator application function into the grammar. First, we forget about the chainrules and conditions for a while. Then with the rule information for each operator we can quite easily generate for for instance the operator PLUS:

```
{
lv_cost :: LabelValue -> Int
lv_cost (Lv _ _ cst) = cst

lv_rules :: LabelValue -> Int
lv_rules :: (Lv _ rls _) = rls
}
SEM STree
    | PLUS loc.plus1 = [(Lv Plus1 [17] (lv_cost @left.src) (lv_cost @right.regx))]
           loc.plus2 = [(Lv Plus2 [13] (lv_cost @left.src) (lv_cost @right.con1))]
           loc.plus3 = [(Lv Plus3 [14] (lv_cost @left.src) (lv_cost @right.con2))]
    | ...
```

The list construction may seem odd, but we can have multiple rules for one storage class. The list is necessary. Condition evaluation can easily be applied now. We can just evaluate the conditions in the way we used to, but now we want a function which result in the empty list when the condition fails, or in the element when the condition holds. So we get:

```
{
condCheck :: Rulenr -> Tree -> LabelValue
condCheck rn tr lv | condition rn tr = [lv]
                   | otherwise       = []
}
SEM Tree
   | PLUS loc.plus1 = condCheck 17 @loc.tree
                         (Lv Plus1 [17] (lv_cost @left.src) (lv_cost @right.regx))
          loc.plus2 = condCheck 13 @loc.tree
                         (Lv Plus2 [13] (lv_cost @left.src) (lv_cost @right.con1))
          loc.plus3 = condCheck 14 @loc.tree
                         (Lv Plus3 [14] (lv_cost @left.src) (lv_cost @right.con2))
   | ...
```

Functionally, we are almost back to the original tree matcher. We still need to add the chainrules. Remember, we already evaluated the chainrules to:

```
lab_list Dst = [(Lv Dst [] 0)]
lab_list Plus1 = [(Lv Plus1 [] 0)]
lab_list Statmt = [(Lv Statmt [] 0)]
lab_list Plus2 = [(Lv Plus2 [] 0)]
lab_list Plus3 = [(Lv Plus3 [] 0)]
lab_list Addr = [(Lv Dst [7] 0), (Lv Addr [] 0)]
lab_list Regx = [(Lv Src [8] 0), (Lv Dst [7,10] 0), (Lv Addr [10] 0), (Lv Regx [] 0)]
```

```
lab_list Src = [(Lv Src [] 0)]
lab_list Con1 = [(Lv Con1 [] 0)]
lab_list Con2 = [(Lv Con2 [] 0)]
```

We can use this information, to evaluate the reverse of this. Now, we give a storage class and the result is a list of derivable storage classes. This collection of lists can be used to automatically construct a function which given a storage class returns *from which storage class it can be derived* instead of which storage classes are derivable from it. This is a quite simple process. Say we are looking at the storage class *Dst*. We simple call lab_list for each storage class. If the result of this contains *Dst* then we can conclude that *Dst* is derivable from the the current storage class. When can automatically evaluate this new function from the old chainrule function. The new function can be added directly into the attribute grammar. Chainrules must be applied before passing the results to parents, so we get:

```
{
chain_app :: [Rulenr] -> Int -> LabelValues -> LabelValues
chain_app nt1 rls1 cst1 lvs = map (\(Lv nt2 rls2 cst2) -> (Lv nt1 rls1++rls2 cst1+cst2)) lvs
}
SEM Tree
   | PLUS loc.plus1 = condCheck 17 @loc.tree
                        (Lv Plus1 [17] (lv_cost @left.src) (lv_cost @right.regx))
          loc.plus2 = condCheck 13 @loc.tree
                        (Lv Plus2 [13] (lv_cost @left.src) (lv_cost @right.con1))
          loc.plus3 = condCheck 14 @loc.tree
                        (Lv Plus3 [14] (lv_cost @left.src) (lv_cost @right.con2))
          loc. .... = []
          lhs.dst   = @loc.dst ++
                        (chain_app Dst [7] 0 @loc.addr) ++
                        (chain_app Dst [7, 10] 0 @loc.regx)
          lhs. .... = @loc. .... ++ chain_app ....
   | ...
```

We have derived a tree matcher with all the functionality from the basic tree matcher. In this version however, the dependency between storage classes has been made implicit. Therefor lazy evaluation will now do its job and make sure a lot of values will no longer be calculated. However, these values do still use some space and introduce some calculation overhead. Allthough they are no longer calculated they are allocated. This are a lot of values for every node. And since a program holds a lot of nodes, we still have lots of unnecessary storage use. In the next chapter we will look into that. But for now we will first expand the above a bit more. We will derive a more elegant solution and give the complete solution for the simple machine description.

3.4.1 Putting it all together

Before giving the complete code for a tree matcher, there are a few things that must be noted. First of all, normalization has been removed. Not because we do not like the effect of it anymore, but because it is no longer possible. If we normalize over the costs of all derivable storage classes, lazy evaluation will no longer do its job. Since each cost depends on all costs of the other storage classes, we always need to calculate everything. We started this more complicated way of treematching because we wanted to exploit Haskell's laziness, so we have

to forget about normalization. Since only costs for derivable storage classes are used the cost sum will remain quite small. To make sure it does not grow too large in very large programs we substract 1 from the cost at every node. So we apply some sort of normalization, but now by a constant factor.

We will give the complete tree matcher for the simple machine description. All changes to the tree matcher can be seen as an instance of partial evaluation on the original tree matcher given in the previous chapter. Since all semantic functions for the operators look like each other and they are quite long, only one operator function is given.

The original and new datatype, and the function for converting the original datatype to the specialized form:

```
DATA Tree | Nil
          | Nullary op    : Operator
                    value : Value
          | Unary   op    : Operator
                    left  : Tree
                    value : Value
          | Binary  op    : Operator
                    left  : Tree
                    right : Tree
                    value : Value

DATA STree | Nil
           | CONST value : Value
           | REG   value : Value
           | MEM   left  : STree
                   value : Value
           | PLUS  left  : STree
                   right : STree
                   value : Value
           | ASGN  left  : STree
                   right : STree
                   value : Value

ATTR Tree [ | | stree : STree ]
SEM Tree
   | Nil lhs.stree     = STree_Nil
   | Nullary lhs.stree = bn_stree @op @value
   | Unary lhs.stree   = bu_stree @op @left.stree @value
   | Binary lhs.stree  = bb_stree @op @left.stree @right.stree @value

{
bn_stree CONST val      = STree_CONST val
bn_stree REG val        = STree_REG val
bu_stree MEM tl val     = STree_MEM tl val
bb_stree PLUS tl tr val = STree_PLUS tl tr val
bb_stree ASGN tl tr val = STree_ASGN tl tr val
}
```

All of the above can be generated starting from the original datatype and the list of storage types and operators.

The condition evaluation can remain unchanged. We use the synthesized attribuut 'tree' in the same way we used it in previous versions of the tree matcher. This tree is built in a slightly different way:

```
SEM STree
   | Nil   loc.tree = Tree_Nil
   | CONST loc.tree = Tree_Nullary CONST @value
   | REG   loc.tree = Tree_Nullary REG @value
   | MEM   loc.tree = Tree_Unary MEM @left.tree @value
   | PLUS  loc.tree = Tree_Binary PLUS @left.tree @right.tree @value
   | ASGN  loc.tree = Tree_Binary ASGN @left.tree @right.tree @value
```

Now we need the attributes for every storage class. Using these attributes of the children, we build the local rule information. The list of rules contained in ruleinf is not the local list of rules, but the complete list of rules. We can easily calculate this complete list because we have the seperate attributes and the lazy evalution. By doing so, we get the derivation at the root of the tree. We no longer need a topdown traversal to generate this.

```
{
type RuleInf = Maybe ([Int],Int)
}
ATTR STree [ | | nil : RuleInf dst : RuleInf plus1 : RuleInf
                 plus2 : RuleInf plus3 : RuleInf statmt : RuleInf
                 addr : RuleInf regx : RuleInf src : RuleInf
                 con1 : RuleInf con2 : RuleInf tree : Tree ]

SEM STree | ASGN
  --apply rules
  loc.rdst    = Nothing
  loc.rplus1  = Nothing
  loc.rstatmt = ckCost $ minRI $ condCheck @loc.tree
                                 [(applyBRule 1 @left.dst @right.plus1),
                                  (applyBRule 2 @left.dst @right.plus2),
                                  (applyBRule 3 @left.dst @right.plus3)]
  loc.rplus2  = Nothing
  loc.rplus3  = Nothing
  loc.raddr   = Nothing
  loc.rregx   = Nothing
  loc.rsrc    = Nothing
  loc.rcon1   = Nothing
  loc.rcon2   = Nothing
```

There are a couple of functions used to generate the local rule information per storage class. First of all, the list of applicable rules is filtered by condition. The function *condCheck* takes care of this. For every storage class one rule will suffice. With the function *minRI* we select the cheapest rule that satisfies its condition. Since we do not want to use storage classes having infinite costs (meaning they are not derivable) we apply the function *ckCost* to this element. This will set the rule information to *Nothing* for every nonderivable storage class.

```
{
condCheck :: Tree -> [RuleInf] -> [RuleInf]
condCheck tr (ri@(Just ((rnr:rls),cst)):xs) | condition rnr tr = ri : (condCheck tr xs)
                                            | otherwise        = condCheck tr xs
```

```
condCheck _ _ = []

ckCost :: RuleInf -> RuleInf
ckCost Nothing = Nothing
ckCost ri@(Just (rls,cst)) | cst < infCost = ri
                           | otherwise     = Nothing

applyNRule :: Rulenr -> RuleInf
applyNRule rnr = Just ([rnr],rulecost rnr)
applyURule :: Rulenr -> RuleInf -> RuleInf
applyURule _ Nothing = Nothing
applyURule rnr (Just (r1,c1)) = Just ((rnr:(r1)),(rulecost rnr + c1 - 1))
applyBRule :: Rulenr -> RuleInf -> RuleInf -> RuleInf
applyBRule _ Nothing _ = Nothing
applyBRule _ _ Nothing = Nothing
applyBRule rnr (Just (r1,c1)) (Just (r2,c2)) =
                  Just ((rnr:(r1++r2)),(rulecost rnr + c1 + c2 - 1))

minRI :: [RuleInf] -> RuleInf
minRI []  = Nothing
minRI ris = foldl1 fminRI ris

fminRI :: RuleInf -> RuleInf -> RuleInf
fminRI Nothing Nothing  = Nothing
fminRI Nothing (Just a) = Just a
fminRI (Just a) Nothing = Just a
fminRI a1@(Just (_,c1)) a2@(Just (_,c2)) | c1 <= c2  = a1
                                         | otherwise = a2
}
```

Now all we need to do is apply the chainrules. This is quite straightforward:

```
SEM STree | ASGN
  --apply chainrules
  lhs.nil     = Nothing
  lhs.dst     = ckCost $ minRI [@loc.rdst,(applyLab @loc.rregx [7,10] 0),(applyLab @loc.addr [7] 0)]
  lhs.plus1   = ckCost $ minRI [@loc.rplus1]
  lhs.statmt  = ckCost $ minRI [@loc.rstatmt]
  lhs.plus2   = ckCost $ minRI [@loc.rplus2]
  lhs.plus3   = ckCost $ minRI [@loc.rplus3]
  lhs.addr    = ckCost $ minRI [@loc.raddr,(applyLab @loc.rregx [10] 0)]
  lhs.regx    = ckCost $ minRI [@loc.rregx]
  lhs.src     = ckCost $ minRI [@loc.rsrc, (applyLab @loc.rregx [8] 0)]
  lhs.con1    = ckCost $ minRI [@loc.rcon1]
  lhs.con2    = ckCost $ minRI [@loc.rcon2]
{
applyLab :: RuleInf -> [Rulenr] -> Int -> RuleInf
applyLab Nothing _ _ = Nothing
applyLab (Just (rls,cst)) crl cstn = Just (crl++rls,cst+cstn)
}
```

By partially evaluating on a machine description we have derived a far more efficient tree matcher. The only problem is that the tree matcher is rather large. The Haskell HUGS

interpreter can handle the tree matcher for the small machine description. The tree matcher derived from the x86 machine description is too large and it exceeds HUGS program storage space. Allthough the Glaskow Haskell Compiler is able to deal with larger files, compiling also failes. This is for a different reason. The synthesized attributes are combined in one large tuple by the AG system. When using the x86 machine description this tuple becomes a 50+ tuple. Compilation failes because the tuple exceeds the maximum tuple size GHC allows. This is a serious problem. But as we mentioned earlier we will try to remove all unused attributes completely in the next chapter. This will save us a lot of allocation space and will hopefully also solve the problem with the large tuples.

3.5 Evaluating machine descriptions

In the previous sections we explained some techniques which all investigate machine descriptions in some way. Since most analyses are straightforward, we do not intend to discuss them in detail. In this section we will look into some of the issues of the analysis.

3.5.1 Parsing machine descriptions

From the beginning we have been referring to machine descriptions as if they were written in the following form:

```
stmt: ASGNI1(addr,rc)  "mov byte ptr \%0,\%1\n"   1
stmt: ASGNI2(addr,rc)  "mov word ptr \%0,\%1\n"   1
stmt: ASGNI4(addr,rc)  "mov dword ptr \%0,\%1\n"  1
```

But why have we chosen this form? We can just write them in some Haskell data structure, giving us the ability to analyse directly. In fact, there is no good reason for not doing so, besides the fact that we are lazy. We simply do not want to write our own machine descriptions. A lot of work has been done in this field. The machine descriptions described above are the machine descriptions as they are used by Lcc ([5]). They provide us with all the information we needed. LCC is a cross compiler. By supporting this form of machine descriptions, we are immediately able to compile to all machines LCC compiles to. We will give the EBNF for the machine descriptions. Generating a parser from the EBNF is straightforward.

```
root = rules+
rule = nont ruledef rulecode rulecost?

ruledef = nont
        | operator
        | operator nont
        | operator operator
        | operator operator nont
        | operator nont ruledef
        | operator ruledef

rulecode = '"' anychar* '"'

rulecost = digit+
```

```
        | "memop(a)"
        | "move(a)"
        | "hasargs(a)"
        | "imm(a)"
        | "!imm(a)"
        | "range(a," digit+ "," digit+ ")"

nont = lletter+
operator = uletter+ type?
type = uletter size?
size = digit
```

There are some additional possibilities for rulecost. They are left out since they can be transformed to one of the above and giving them will make things needlessly complicated.

After parsing we immediately transform the resulting structure to one which suits our needs. This is the structure we used throughout this thesis. It is written in such a way that the needed analyses (of which some where discussed above) can be done easily. Therefore we will only give this structure and assume any analyses are obvious. When we feel a certain analysis is not straightforward, we will explain it at the time it is first mentioned.

We start with the main datatype MdDes. It holds a complete machine description. It contains information about the different operators, the evaluated chainrules and a list holding all possible storage classes.

```
DATA MdDes
    | MdDes nullOps    : NullOps
            unOps      : UnOps
            binOps     : BinOps
            chainrules : ChainRules
            storage    : Nonterminals
```

Per operator type we have a list with for every operator its name and its definitions.

```
TYPE NullOps = [NullOp]
TYPE UnOps   = [UnOp]
TYPE BinOps  = [BinOp]

DATA NullOp
   | NullOp op   : OpName
            defs : NullDefs
DATA UnOp
   | UnOp op     : OpName
          defs   : UnDefs
DATA BinOp
   | BinOp op    : OpName
           defs  : BinDefs
```

Per operator we have the definition list. It contains all needed rule information.

```
TYPE NullDefs = [NullDef]
TYPE UnDefs   = [UnDef]
TYPE BinDefs  = [BinDef]
```

```
DATA NullDef
   | NullDef rulenr : RuleNr
             deriv  : Nonterminal
             cost   : Int
             cond   : Condition

DATA UnDef
   | UnDef   rulenr : RuleNr
             deriv  : Nonterminal
             left   : Nonterminal
             cost   : Int
             cond   : Condition

DATA BinDef
   | BinDef  rulenr : RuleNr
             deriv  : Nonterminal
             left   : Nonterminal
             right  : Nonterminal
             cost   : Int
             cond   : Condition
```

Furthermore we need the chainrules. We can evaluate them at parse time. This results in evaluated chainrules in the datatype. We get a list with a labelvalue for every storage class. This is the derivable labelvalue, not the reversed chainrule as discussed above. So the labelvalue specifies everything derivable from the storage class for the nonterminal *nt*. Furthermore we need the conditions.

```
TYPE ChainRules = [ChainRule]

DATA ChainRule
   | ChainRule nt    : Nonterminal
               lv    : LabelValue

DATA Condition
   | C_Nothing
   | C_Range    lb : String ub : String
   | C_Memop
   | ...
```

When needed we can easily write attribute grammars for every needed analysis on the above machine description.

Chapter 4

Analyzing Attribute Grammars

4.1 Overview

If we follow the approach of the previous chapter, we may end up with attribute grammars in which many attributes actually correspond to dead code. For a certain nonterminal we have a large set of attributes. The set of attributes we are interested in depends on the parent node for that nonterminal in an input tree. It seems the set of attributes is input dependent. But although it depends on the input, it is possible to precompute which sets of attributes might be needed at some point. Maybe we can remodel the grammar in such a way that we can make use of such a precalculated version of the grammar. In this chapter we will describe a way to do this. Not only the instruction selection problem benefits from such an analysis. We will work on a solution wich works for *any* attribute grammar. After doing so we will review the effect of it on the tree matcher described in the previous chapter.

4.2 A small example

We are looking for a way of disposing unneeded attributes. But what do we mean by this? We will work from an example and expand it until we derive a useful analysis for the tree matcher. Let us take a look at a simple attribute grammar:

```
DATA X | X left : Y right : Y
DATA Y | Y

ATTR X [ | | result : Int ]
ATTR Y [ | | a1 : Int a2 : Int ]

SEM X | X lhs.result = @left.a1 + @right.a2
SEM Y | Y lhs.a1     = 2 * 10
          lhs.a2     = 3 * 4
```

The program for this grammar will always compute both attributes for Y (or when only considering Haskell as a target language, it will always allocate storage space for them). It is easy to see that we do not always need both attributes. At an occurrence of a nonterminal Y in the right hand side of a rule we know in which attributes we are interested. This

leads to a simple observation. If we introduce specialized variants of Y for every set of synthesized attributes we are interested in, we save time or at least storage space. Since the analysis becomes interesting at code generation time, we need not transform the attribute grammar. Just generating different code will suffice. If we look at the above example, the straightforwardly generated resulting Haskell source will look like this:

```
data X = X (Y) (Y)
sem_X (_left) (_right) =
    let ( _left_a1,_left_a2) =
            (_left )
        ( _right_a1,_right_a2) =
            (_right )
    in  (_left_a1 + _right_a2)

data Y = Y
sem_Y  =
    let
    in  (2 * 10,3 * 4)
```

We can make specialized versions of the semantic functions for Y. There are a few cases we are interested in:

- no attributes at all
- in one of the two elements
- both the elements

Since the result of the functions changes, the function for sem_X_X will change too, we get:

```
data X = X (Y) (Y)
sem_X (_left) (_right) =
    let ( _left_a1 ) =
            (_left )
        ( _right_a2) =
            (_right )
    in  (_left_a1 + _right_a2)

data Y = Y
sem_Y__ =
    let
    in ()
sem_Y_a1  =
    let
    in  (2 * 10)
sem_Y_a2  =
    let
    in  (3 * 4)
sem_Y_a1a2 =
    let
    in  (2 * 10,3 * 4)
```

Since the semantic functions change, it is necessary to adapt the part of the program which calls these functions. This part may not be available at the time we are analyzing the attribute grammar. This can for instance be a parser, which has not yet been written. The calls to the semantic functions must be done by hand. When for instance working with a parser one must create a specialized version of the parser. That is possible when it is clear which attributes are needed in which situation (like in the above example) but what if we are working with a large complicated attribute grammar? Changing the semantic function calls by hand is a complicated task. There is another disadvantage. Imagine writing a front end to a compiler using an attribute grammar. The front end calculates two things (attributes): the result of type checking and an intermediate tree. If we only want to know whether our program is typed correct, we only need one attribute. But it seems a bit odd to write three versions of the parser for the front end (one for the first attribute, one for the second and one for both the attributes). We do not want to introduce an overhead like that. What we need is the possibility to use one common parser and specify at top level which attributes we are interested in. The solution is quite simple. We will exploit the catamorphisms. If we generate the catamorphisms for the basic attribute grammar we started with, we get:

```
data X = X_X (Y) (Y)
-- cata
sem_X ((X_X (_left) (_right))) =
    (sem_X_X ((sem_Y (_left))) ((sem_Y (_right))))
sem_X_X (_left) (_right) =
    let ( _left_a1,_left_a2) =
            (_left )
        ( _right_a1,_right_a2) =
            (_right )
    in  (_left_a1 + _right_a2)

data Y = Y_Y
-- cata
sem_Y ((Y_Y )) =
    (sem_Y_Y )
sem_Y_Y  =
    let
    in  (2 * 10,3 * 4)
```

Since we can analyse in which attributes we are interested in for every child of X, the cata-functions can be changed into:

```
sem_X ((X_X (_left) (_right))) =
    (sem_X_X ((sem_Y_a1 (_left))) ((sem_Y_a2 (_right))))
sem_Y_a1 ((Y_Y )) =
    (sem_Y_a1_Y )
sem_Y_a2 ((Y_Y)) =
    (sem_Y_a2_Y )
```

We are analysing the complete grammar. It is not known which element is the root element of the tree. Therefore the analysis and the transformations are applied to all of the nonterminals. The resulting file is:

```
data X = X_X (Y) (Y)
sem_X__ ((X_X (_left) (_right))) =
    (sem_X_X__ (_left) (_right))
```

```
sem_X_result ((X_X (_left) (_right))) =
    (sem_X_X_result ((sem_Y_a1 (_left))) ((sem_Y_a2 (_right))))

sem_X_X__ (_left) (_right) =
    let
    in  ()
sem_X_X_result (_left) (_right) =
    let ( _left_a1 ) =
            (_left )
        ( _right_a2) =
            (_right )
    in  (_left_a1 + _right_a2)

data Y = Y_Y

sem_Y__ ((Y_Y )) = (sem_Y_Y__ )
sem_Y_a1 ((Y_Y )) = (sem_Y_Y_a1 )
sem_Y_a2 ((Y_Y )) = (sem_Y_Y_a2 )
sem_Y_a1a2 ((Y_Y )) = (sem_Y_Y_a1a2 )

sem_Y_Y__ =
    let
    in ()
sem_Y_Y_a1  =
    let
    in  (2 * 10)
sem_Y_Y_a2  =
    let
    in  (3 * 4)
sem_Y_Y_a1a2 =
    let
    in  (2 * 10,3 * 4)
```

This version needs only one parser that constructs the abstract syntax tree. The tree is subequently passed to a semantic function. Which function this is depends on the root attributes we are interested in. We no longer need to specify which attributes are needed at every position if we do not want to. We only need to state which attribute we want delivered at root level. It looks like we have found an interesting new way of writing the semantic functions. There still is a small problem. In the above example we simply create the powerset of all attributes. The analysis must be useful for many purposes. One of them is the compiler from the previous chapter. The main problem there was the huge set of attributes for a single node. The powerset of such a huge set of attributes will be enormous. The resulting file will become so large it will probably be useless. We somehow want to be able to only build those functions which are needed. This will hopefully be a very small subset of the mentioned powerset. Well, can we not simply analyse which attributes are needed for every nonterminal? We are doing this locally to build the catamorphic functions. Can this be taken up to a global level?

When trying to analyse which nonterminals depend on which other nonterminals and which attributes they need, there is one simple problem. There is no starting point. In an attribute

grammar by itself there is no explicit statement stating 'we are interested in this attribute of this nonterminal'. To be able to do the analysis it therefore is necessary for the user to provide us with some extra information. The user must specify a list of nonterminals and attributes we are interested in. This is no problem for the user, because this list corresponds to the semantic function call at the root of the tree. When these starting nonterminals and attributes are given we can simply trace their dependencies recursively until no further functions not already marked as required are found. The file can now safely be generated. If we reconsider the example used above and assume the user stated being interested in the attribute *result* of the nonterminal X this will result in the following, which is no more than a subset of the above example:

```
data X = X_X (Y) (Y)
sem_X_result ((X_X (_left) (_right))) =
    (sem_X_X_result ((sem_Y_a1 (_left))) ((sem_Y_a2 (_right))))

sem_X_X_result (_left) (_right) =
    let ( _left_a1 ) =
            (_left )
        ( _right_a2) =
            (_right )
    in  (_left_a1 + _right_a2)

data Y = Y_Y
sem_Y_a1 ((Y_Y )) = (sem_Y_Y_a1 )
sem_Y_a2 ((Y_Y )) = (sem_Y_Y_a2 )
sem_Y_Y_a1  =
    let
    in  (2 * 10)
sem_Y_Y_a2  =
    let
    in  (3 * 4)
```

4.3 Things to keep in mind

In the previous section we introduced a very simple example. In reality analysis will be done on more complicated attribute grammars and therefore there are some extra things to keep in mind. For starters there is the syntactic sugar and the attribute grammar chainrules. But luckily they are of no concern to us. Since we do not need to do the analysis before it is time to generate code, that has all been taken care of automatically. Currently the abstract syntax tree used for code generation is in a form where all syntactic sugar has been removed and all attribute grammar chainrules have been applied. That gives us a great advantage, because it takes a lot of complexity out of the attribute grammar we are analyzing. But there still is one important thing, the declaration of local functions. Consider a slightly altered version of the example given in the previous section:

```
DATA X | X left : Y right : Y
DATA Y | Y

ATTR X [ | | result : Int ]
ATTR Y [ | | a1 : Int a2 : Int ]
```

```
SEM X | X loc.something = @loc.some2 + @left.a1
          loc.some2     = @right.a2
          loc.result    = @left.a1 + @loc.something
SEM Y | Y lhs.a1     = 2 * 10
          loc.a2     = 3
          lhs.a2     = @loc.a2 + 3
```

The local functions introduce some extra work. It does no longer suffice to examine each rule for the attributes needed. In addition we need to explore all local functions a rule uses. This process must be done because in the given example we need to make sure we have the attribute @*right.a*2 available for the *result* rule. We also need to make sure we only keep those local functions which are actually used. If in the above example *loc.some*2 is not needed, we do not have the attribute @*right.a*2 available. Delivering a function which does contain a reference to that attribute results in an incorrect program. The fact that the attribute is never needed does not matter. If the attribute @*right.a*2 is removed and some local function uses it, the resulting program simply is syntactically wrong. Keeping the above in mind we are now set to model our analysis.

4.4 Details of the analysis

It is time for some details of the actual analysis. We will concentrate on the information gathering part of the analysis. By doing so we hope to create greater understanding of the way the analysis work. If we want to give details about the code generation we first have to explain the entire code generation process of the ag-system. This obviously is beyond the scope of this thesis. The code generation for the analysed attribute grammars is almost exactly the same as normal code generation, provided we gather the right information during analysis. As stated before we will analyse on the abstract syntax tree where all chainrules have been applied and desugaring has been done. The syntax tree is:

```
DATA Grammar      | Grammar typeSyns : {TypeSyns}
                            useMap   : {UseMap}
                            prods    : Productions

TYPE Productions  = [Production]
TYPE Alternatives = [Alternative]
TYPE Children     = [Child]
TYPE Rules        = [Rule]
TYPE LocRules     = [LocRule]

DATA Production   | Production nt   : {String}
                               inh  : {Attributes}
                               syn  : {Attributes}
                               alts : Alternatives

DATA Alternative  | Alternative con      : {String}
                                children : Children
                                rules    : Rules
                                locrules : LocRules
```

```
DATA Child        | Child   name  : {String}
                            tp    : {String}
                            inh   : {Attributes}
                            syn   : {Attributes}
                            rules : Rules

DATA Rule         | Rule    attr  : {Name}
                            rhs   : Expression

DATA LocRule      | LocRule pat   : Pattern
                            rhs   : Expression
                            owrt  : {Bool}
```

As mentioned in the previous section, one of the things needed is the local dependency information. This information can be gathered without any problems. For every nonterminal *LocRule* we need to find out which attributes and which locals it depends on. Then those locals must be traced and any attributes they depend on added. Consider the following example (for some arbitrary nonterminal):

```
loc.a1 = @loc.a2 + @loc.a3
loc.a2 = @left.a1 + @loc.a3
loc.a3 = @left.a2
```

Dependency information is stored in a map from local attribute to a list of attributes per child. To store the local dependencies we use the name *loc* instead of a child name. So for the above example, the complete dependency information is:

```
type AttrMap = Map String (Strings)
type RuleMap = Map String AttrMap

exampleAttrMap :: RuleMap
exampleAttrMap  = list2map [("a1",[("loc",["a1","a2","a3"]),("left",["a1","a2"])]),
                            ("a2",[("loc",["a2","a3"]),("left",["a1","a2"])]),
                            ("a3",[("loc",["a3"]),("left",["a2"])])]
```

To build this complete map, where all dependencies have been traced, we first build the untraced locals list:

```
ATTR LocRules [ | | simplemaplist : {[(String, AttrMap)]} ]
SEM LocRules
    | Cons lhs.simplemaplist = @hd.simplemapelem : @tl.simplemaplist
    | Nil  lhs.simplemaplist = []

ATTR LocRule [ | | simplemapelem : {(String,AttrMap)} ]
SEM LocRule
    | LocRule lhs.simplemapelem = ((@pat.name),list2map (("loc",(@rhs.usedLocals)) :
                                                          (trans @rhs.usedAttrs)))
{
trans :: Ord a => [(a, b)] -> [(a,[b])]
}
```

For the given example, this list will be:

```
[("a1",[("loc",["a2","a3"])]),
 ("a2",[("loc",["a3"]),("left",["a1"])]),
 ("a3",[("left",["a2"])])]
```

The above list can easily be converted to the exhaustively examined list mentioned above. We simply start a worklist algorithm with all the local attributes. The final resulting map is given as a parameter. When exploring a certain local attribute and finding it depends on two other attributes, we look in to the result map whether we know its dependencies. If it is already known we use the information provided. If it has not yet been explored, we start exploring the new attribute. The resulting information can be added to the local attribute we started with. Assume the argument for the worklist algorithm is the ordered list of a1, a2 and a3. We first examine a1 and find we need the dependency information of a2 and a3. We start by examining a2 and again find we need the dependency information for a3. We recurse into examining a3. This gives us an attribute list. The result map has been updated and the dependency of a3 is known. We arrive back at the examination of a2. Since we have all information needed for its dependency we combine the attribute list for a3 with the local one and again update the result map. The map now holds information for both a2 and a3. We arrive back at our starting point, analyzing the dependency of a1. We find the information for a2 and a3 is known and simply add any local attribute dependencies (none in the above example). We have finished exploring a1. The worklist algorithm must now explore the next attribute in the list, being a2. It looks up a2 in the current result map and finds dependency information has already been computed (since it was needed for calculating a1's dependencies). We can consider a2 as analyzed and proceed with the next attribute in the worklist. We again find it already has been explored, and since the worklist is empty, we return the result map.

A quite straightforward algorithm, but there is one risk. It does not necessarily terminate. For the following example it will never terminate:

```
[("a1",[("loc",["a2"])]),
 ("a2",[("loc",["a1"])])]
```

This may seem like a strange example, but the attribute grammar system does not exclude the above example. It of course results in a never ending program, if the attribute a1 or a2 is ever used. But if both attributes are never used, the program will work. Of course we can say the example does not make any sense, but nevertheless we do not want the attribute grammar system to fail because of the above example. This can easily be avoided. When exploring local variables, before recursing into some other local variable we first add the current local variable as being dependent of nothing. This way loops are avoided. The code for the above algorithm is:

```
ATTR LocRules LocRule [ simplemap : RuleMap | | ]
SEM Alternative
   | Alternative locrules.simplemap  = list2map @locrules.simplemaplist
                 locrules.locinfomap = emptyMap
                 loc.locinfomap      = @locrules.locinfomap

ATTR LocRules LocRule [ | locinfomap : RuleMap | ]
```

```
SEM LocRules
   | Cons hd.locinfomap  = @lhs.locinfomap
        tl.locinfomap  = @hd.locinfomap
   | Nil  lhs.locinfomap = @lhs.locinfomap
SEM LocRule
   | LocRule lhs.locinfomap = snd (recurseMap (@pat.name) @lhs.simplemap @lhs.locinfomap)

{
recurseMap :: String -> RuleMap -> RuleMap -> (AttrMap,RuleMap)
recurseMap x simplemap result = let hasinf           = locateMap x result
                                    needrec (Just a) = (a, result)
                                    needrec Nothing  = (frommap dorec, dorec)
                                    frommap dr       = fromMaybe emptyMap (locateMap x dr)
                                    dorec            = doRecurse x simplemap result
                                in needrec hasinf

doRecurse :: String -> RuleMap -> RuleMap -> RuleMap
doRecurse x simplemap result =
     let newmap :: RuleMap
         newmap                    = insert x (list2map [("loc",[x])]) result
         locatt :: AttrMap
         locatt                    = fromMaybe emptyMap (locateMap x simplemap)
         loclst :: [String]
         loclst                    = fromMaybe [] (locateMap "loc" locatt)
         cklocs :: [String] -> (AttrMap, RuleMap)
         cklocs []                 = (locatt, insertComb addattrmap x locatt newmap)
         cklocs xs                 = foldl foldf (locatt, newmap) xs
         foldf :: (AttrMap, RuleMap) -> String -> (AttrMap, RuleMap)
         foldf (at,nm) loc         = foldadd (at,nm) (recurseMap loc simplemap nm)
         foldadd :: (AttrMap, RuleMap) -> (AttrMap, RuleMap) -> (AttrMap, RuleMap)
         foldadd (at1, _) (at2, nm) = (addattrmap at1 at2, nm)
         (na,newm)                 = cklocs loclst
     in insertComb addattrmap x na newm

addattrmap :: AttrMap -> AttrMap -> AttrMap
addattrmap a1 a2 = addmap a1 a2 (\am (x,y) -> insertComb uniadd x y am)

addmap m1 m2 f = foldl f m1 (map2list m2)

uniadd a1 a2 = sort (nub (a1 ++ a2))
}
```

Now we have gathered all the information mentioned at the beginning of this section. The local attribute *locinfomap* at nonterminal *Alternative* contains all the information for the local dependencies. For code generation we are not interested in the local dependencies. We want to know which local functions and childrens attributes each rule depends on. This information can easily be gathered from the local information map.

```
ATTR Rules Rule [ locinfomap : RuleMap | | ]
SEM Alternative
   | Alternative rules.locinfomap = @loc.locinfomap
                 rules.rulemap    = emptyMap
```

```
                loc.rulemap      = @rules.rulemap

ATTR Rules Rule [ | rulemap : RuleMap | ]
SEM Rules
   | Cons hd.rulemap  = @lhs.rulemap
          tl.rulemap  = @hd.rulemap
   | Nil  lhs.rulemap = @lhs.rulemap
SEM Rule
   | Rule lhs.rulemap = let fromMap Nothing  = []
                           fromMap (Just a) = map2list a
                           foldrule lst loc = lst ++ (fromMap (locateMap loc @lhs.locinfomap))
                           locattmap        = list2map (foldl foldrule (trans @rhs.usedAttrs)
                                                                       (@attr : @rhs.usedLocals))
                        in  insert @attr locattmap @lhs.rulemap
```

There is only one thing left to analyze, i.e. which semantic functions do we eventually need. This depends on the list of nonterminals with attributes given by the user when starting the analysis. We can again use the information already gathered. There is one thing to keep in mind, the local rulemap contains information addressed by the names of the children. For the global analysis we are interested in the types of the children. But we can easily lookup the type of a child by its name. So we can simply start another worklist algorithm which traces the dependencies of the given attributes. Any not yet found dependencies are added to the worklist and at the time the worklist is empty we possess the list of all needed nonterminals and the attribute combinations needed for each nonterminal. All these analyses combined give us all the information we need to generate the code for the analyzed version of the attribute grammar.

```
SEM Grammar | Grammar loc.typemap  = list2map @prods.typelist
                      prods.depmap = builddepmap @loc.typemap @loc.startel emptyMap

ATTR Productions [ | | typelist USE {++} {[]} : {[(ProdPair, [ProdPair])]} ]
ATTR Production [ | | typelist : {[(ProdPair, [ProdPair])]} ]
SEM Production | Production alts.typemap = emptyMap
                           lhs.typelist = map2list @alts.typemap

ATTR Alternatives [ | typemap : TypeMap | ]
SEM Alternative [ | typemap : TypeMap | ]
   | Alternative lhs.typemap = let attrlst  = map (\x -> (x, attrmap x @loc.rulemap)) @loc.rulealt
                                   insmap mp (ra,x) = insertComb (++) (@lhs.nt,ra)
                                                                 (toType (map2list x) @children.fields) mp
                                in  foldl insmap @lhs.typemap attrlst

{
type DepMap   = Map String [Strings]
type WorkList = [ProdPair]

type ProdPair = (String,Strings)
type TypeMap = Map ProdPair [ProdPair]

builddepmap :: TypeMap -> WorkList -> DepMap -> DepMap
builddepmap _ [] dp = dp
builddepmap tm ((nt,attrs):wl) dp = let currlst = fromMaybe [] $ locateMap nt dp
                                        chkseen [] = False
```

```
                                        chkseen (x:xs) = (x == attrs) || chkseen xs
                                        newwl = newwlm (locateMap (nt,attrs) tm)
                                        newwlm Nothing  = wl
                                        newwlm (Just a) = wl ++ a
                                        newdp = insertComb (++) nt [attrs] dp
                                    in if chkseen currlst then builddepmap tm wl dp
                                                          else builddepmap tm newwl newdp

toType :: [(String,Strings)] -> [(String,String)] -> [ProdPair]
toType [] _ = []
toType ((x,y):xs) lc | x == "loc" = toType xs lc
                     | otherwise  = (getType x lc, y):(toType xs lc)

getType :: String -> [(String,String)] -> String
getType _ [] = []
getType nm1 ((nm2,tp):xs) | nm1 == nm2 = tp
                          | otherwise  = getType nm1 xs
}
```

4.5 Applying the analysis to the tree matcher

We started the analysis to improve the working of the tree matcher. However the treematcher created by the analysis does not work with the x86 instruction set. For the Simple machine it works fine. We get a tree matcher which needs about 60 percent of the storage space required for the unanalysed version of the same tree matcher.

When we apply the analysis to the tree matcher for the x86 from the previous chapter the resulting Haskell file becomes enormous. The file is about 150 megabytes in size. It contains around six million lines of code. It is obviously not useful for tree matching. The code generation may be improved, making it possible to generate smaller files. But still, the enormous file shows us we have a lot of different combinations of attributes we are interested in. It is worth studying how to improve the analysis in the future. But since the resulting file will still be very large it is probably not the best way to get an efficient tree matcher. Since the last is our goal we will concentrate on different methods of deriving an efficient tree matcher in the next chapter. The analysis is quite interesting, but since the chance of getting an effienct tree matcher with it is quite small (because of the large set of different dependencies) we feel examining it further is beyond the scope of this thesis. We will look into another way of improving the instruction selection in the next chapter hoping to find an efficient and useful approach for the tree matching problem.

One thing must be noted about the analysis. As described in this chapter it is designed to work with any attribute grammar. It needs some work to be complete. At this time for instance we are unable to handle pairs of attributes, for example:

```
ATTR X [ | | a1 : Int a2 : Int ]
SEM X | X loc(a1,a2) = (1,2)
```

Also we have not added all functionality of the original attribute grammar system. Our version is for instance unable to produce correct typing information when working with the analysis. The above described analysis form the basis of a complete analysis. All extra functionality

like the typing information can be added without any extra problems, it will just require a lot of extra work.

The analysis on the smaller machine description shows good results. The x86 machine description has a lot of different dependencies. It is a large subset of the powerset over the attributes. Quite often analysis will give us small subsets. The grammars where the dependencies are like that can benefit from the analysis. Despite the fact that we have not yet found a solution for efficient instruction selection we find the analysis to be a useful addition to the attribute grammar system.

Chapter 5

Automaton based treematching

5.1 Overview

By combining the techniques described in the previous two chapters we theoratically have derived a far more efficient tree matcher than the one we started with. But in practice it is not useful for for instance the x86. Besides that there are still a lot of almost identical calculations done. At every node, a new label value is calculated, purely based on the costs of storage classes at the childrens. We have dropped the normalization for the analysis to work. For this chapter the normalization is reintroduced. We are still working with the tree grammar where operators have been pushed into the grammar, but we drop the seperate attribute per storage class. Figure 5.1 shows a possible way to implement part of the core of the tree matcher. This may not be the most straightforward way to implement things, but it is purely for illustration. This implementation clearly shows that at every node a new labelvalue depends of two things. The rules we applicable (available in advance) and the costs of every derivable storage class at the children. Since normalization is used, these costs are no longer dynamic ([4]). The interesting part of the labelvalue now consists of a limited amount of storage classes with each a cost factor which is also bounded. A technique exploiting this is called automaton based treematching, which was first introduced by E.A. Proebsting in [6]. This technique has one major disadvantage: it does not hold when used with conditional instruction sets. As mentioned before, to implement a tree matcher for a real machine we can not do without conditions. Therefore automaton based treematching might seem useless at first. In the article [4] the author introduces a method to make the automaton based treematching useful for treematching with conditions. We will first explain the ideas of unconditional automaton based tree matching and the techniques to implement it in combination with attribute grammars. In the next chapter we will introduce a technique for handling conditional instruction sets. We have come up with a different approach than the one introduced in ([4]). The ideas described in that article inspired us in developing our approach.

```
{
fromLabVal :: Storage -> LabelValues -> Int
fromLabVal _ [] = infCost
fromLabVal st1 ((Lv st2 _ cst):xs) | st1 == st2 = cst
                                   | otherwise  = fromLabVal st1 xs

lv_norm :: LabelValues -> LabelValues
lv_norm lval = map (lv_min . minimum $ map (\(Lv _ _ c) -> c) lval) lval

lv_min :: Int -> LabelValue -> LabelValue
lv_min min (Lv nt rl cst) = Lv nt rl (cst-min)

fromMaybeList :: [Maybe a] -> [a]
fromMaybeList [] = []
fromMaybeList ((Just x):xs) = x : (fromMaybeList xs)
fromMaybeList (Nothing :xs) = fromMaybeList xs
}

SEM STree | PLUS
    loc.rplus1 = ckCost $ minRI $ condCheck @loc.tree
                            [(applyBRule 17 (fromLabVal Src @left.labval)
                                            (fromLabVal Regx @right.labval))]
    loc.rplus2 = ckCost $ minRI $ condCheck @loc.tree [(applyBRule 13 @left.src @right.con1)]
    loc.rplus3 = ckCost $ minRI $ condCheck @loc.tree [(applyBRule 14 @left.src @right.con2)]
    loc....    = Nothing
    lhs.labval = (lv_norm . fromMaybeList)
                      [(Dst, (ckCost $ minRI [@loc.rdst,(applyLab @loc.rregx [7,10] 0),
                                                        (applyLab @loc.raddr [7] 0)]),
                       (Addr, (ckCost $ minRI [@loc.raddr,(applyLab @loc.rregx [10] 0)])),
                       (Src, (ckCost $ minRI [@loc.rsrc,(applyLab @loc.rregx [8] 0)])]
```

5.2 Unconditional automaton

As mentioned in the previous chapter the number of storage classes and number of costs are limited. Since a labelvalue is nothing but a list with a cost for every storage class (and every storage class occuring no more than once) the number of possible labelvalues must be limited and fixed for a given instruction set. This means it is possible to precalculate all possible labelvalues in advance. Assuming this has been done, we can also precalculate which labelvalue an operator delivers given its childrens labelvalues. We can construct an automaton holding all possible labelvalues and for every operator the resulting labelvalue given its childrens (if any) labelvalues. By constructing such an automaton in advance, the actual tree matching becomes no more than table lookup. None of the calculations from the previous chapters need to be done at compilation time, they have all been done in advance. Needless to say this results in a very fast, but probably very large, tree matcher.

5.2.1 Implementing such an automaton

The actual creation of the automaton is a quite straightforward process which must be easy to understand. The term *state* will be used to represent a labelvalue. The automaton thus consists of a mapping from state to labelvalue and a mapping from an operator and childrens states to a new state. The number of states is fixed for a certain machine. Provided a starting state is available all states it introduces can be calculated. With all those states the same can be done again. When a new state is found, we check if we have already seen it before, or if it is a new state. If it is a new state we can investigate whether the state introduces other new states. What we have just described is just a simple iterative worklist algorithm[1]. The only thing needed is a starting state. In practice we can easily find a collection of 'starting states' since we have a set of nullary operators. As we mentioned new states are found by combining an operator with all found states as children. Since nullary states have no children, initial states can be found by looking at the nullary operators.

We now have all the means necessary to generate an automaton. We get a new state from the first nullary operator. With this state the unary and the binary operators are investigated. For the unary operators each operator is combined with the new state. For the binary operators we must combine each previously found state with this new state, and investigate the situation with the new state as a left child and as a right child for all this earlier states. Ofcourse we must also investigate the situation where an operator has the new state at its left child and at its right child. For the unary and the binary operators we look at every resulting state. If it is a new one, we add it to the worklist. Every found combination is stored ($operator \rightarrow childstates \rightarrow newstate$) as part of our automaton. When every unary and binary operator has been investigated the worklist is examined. If it is not empty we take the first element of the worklist and repeat the above procedure. If it is empty the list of nullary operators is examined. If there is an operator left which introduces a new state it is added to the worklist and the above procedure is repeated. If no new states can be introduced by applying nullary operators and the worklist is empty all states have been found and the automaton is ready.

[1] An iterative worklist algorithm is an algorithm which iterates until a given worklist is empty. If at a certain iteration step we encounter something resulting in extra work to be done, we add it to the worklist.

There are numbers of ways to implement this algorithm. The automaton can for instance directly be generated when parsing machine descriptions. But at the time we are parsing an instruction set (and no files have yet been generated) the operators and storage classes are still strings. Every state found in the above algorithm needs to be compared with all previously found states. It is important to be able to compare states as fast as possible. When a state holds strings representing the storage classes, a couple of string comparisons are needed for every single state comparison. This results in a very slow algorithm. Instead we generate a Haskell file when parsing machine descriptions. This file contains all the means necessary to generate the actual automaton. This has the advantage that automaton generation will be faster, but it also gives us a readable and understandable algorithm for generating the automaton. Since one can easily understand the working of the automaton generation, it will be an easier task to improve this algorithm in the future.

For the algorithm, first the following types are needed:

```
type LabelValue = [LElem]
data LElem = LE Storage [RuleNr] Int deriving Show
```

Then some machine dependent information is needed, which can be easily generated by the machine description parser. We feel the types of these functions will suffice:

```
nullary_ops :: [Operator]
unary_ops :: [Operator]
binary_ops :: [Operator]

app_lab_n :: Operator -> LabelValue
app_lab_u :: Operator -> LabelValue -> LabelValue
app_lab_b :: Operator -> LabelValue -> LabelValue -> LabelValue
```

The functions *app_lab_n*, *app_lab_u* and *app_lab_b* are the operator application functions. In fact, if we recall the attribute grammar from figure 5.1, we could simply import the program this attribute grammar creates and call the resulting semantic functions. For the operator PLUS for example, this is no more than:

```
import OldTreeMatcher

app_lab_b PLUS lv1 lv2 = sem_Tree_PLUS lv1 lv2
```

But this old tree matcher was generated by the same machine description parser as the one generating the automaton builder. So we have chosen to generate the functions directly, but the above illustrates that all used data to generate the automaton builder was already present in the machine description parser.

With the above types and the operator application functions we have all the data needed to write the algorithm. The rest of the algorithm is machine independent. It can be written once and then by combining it with the above we get the complete automaton generator.

First, we need some way of storing all the found states. We can simply use a list but every time we find a state, we want to check whether it has been found before, and if so, which number it has. To make this lookup fast a *Map* from labelvalue to an integer (being the state) is used. Under the hood the Map uses a binary search tree, so lookup will be fast. When we find a new state it gets assigned the number of highest state plus one. We do not want to

investigate what the highest state was every time. We simple combine the last added state number with the state map:

```
type AllState = (Map LabelValue Int, Int)

getState :: AllState -> LabelValue -> (Int, AllState, Bool)
getState allstate []       = (0, allstate, False)
getState (lvmap, count) lv =
   let elm            = locateMap lv lvmap
       reslt Nothing  = (count + 1, (insert lv (count + 1) lvmap, count + 1), True)
       reslt (Just i) = (i, (lvmap, count), False)
   in reslt elm
```

The function *getState* returns the state belonging to its parameters labelvalue, the resulting state collection and a boolean stating whether the state has been added as a new state or it was already known. Since we have a map from labelvalue to int, some way of comparing and ordering labelvalues is needed:

```
eqlv :: LElem -> LElem -> Bool
eqlv (LE nt1 rl1 cst1) (LE nt2 rl2 cst2) = (nt1==nt2) && (rl1==rl2) && (cst1==cst2)
complv :: LElem -> LElem -> Ordering
complv (LE nt1 rls1 cst1) (LE nt2 rls2 cst2)
            | nt1 < nt2 = LT
            | nt1 > nt2 = GT
            | cst1 < cst2 = LT
            | cst1 > cst2 = GT
            | length rls1 < length rls2 = LT
            | length rls1 > length rls2 = GT
            | otherwise = EQ

instance Eq LElem where (==)    = eqlv
instance Ord LElem where compare = complv
```

The worklist algorithm as described above looks like:

```
type State = Int
type StateList = [(Int,LabelValue)]

type TNull   = [(Int, Operator)]
type TUnary  = [(Int, Operator, Int)]
type TBinary = [(Int, Operator, Int, Int)]

worklist :: [Operator] -> [Operator] -> [Operator] -> StateList -> StateList
                                    -> (TNull, TUnary, TBinary, AllState)
                                    -> (TNull, TUnary, TBinary, AllState)

worklist [] _ _ _ [] result = result
worklist (x:xs) unops binops explst [] (tnull, tunary, tbinary, statelst)
        = let newlabval    = (app_lab_n x)
              (sn, sl, sb) = getState statelst newlabval
              newtn        = (sn,x):tnull
              newwl        = if sb && sn > 0 then [(sn, newlabval)] else []
          in worklist xs unops binops explst newwl (newtn, tunary, tbinary, sl)
```

```
worklist xs unops binops explst (w:ws) (tnull, tunary, tbinary, statelst)
= let (newtun, newsl, newwl)     = procUnary w unops tunary statelst ws
              (newtbi, newslbi, newwlbi) = procBinary w binops tbinary newsl newwl explst
          in worklist xs unops binops (w:explst) newwlbi (tnull, newtun, newtbi, newslbi)

procUnary :: (Int, LabelValue) -> [Operator] -> TUnary -> AllState -> StateList
                                                -> (TUnary, AllState, StateList)
procBinary :: (Int, LabelValue) -> [Operator] -> TBinary -> AllState -> StateList
                                          -> StateList -> (TBinary, AllState, StateList)
```

The nullary application is part of the worklist algorithm. The unary application simple takes the state being investigated and checks which states it yields. If any new states are found they are added to the worklist.

```
procUnary _ [] tu cl wl = (tu,cl,wl)
procUnary (inn,lv) (op:unops) tu cl wl =
           let newlabval    = (app_lab_u op lv)
               (sn, sl, sb) = getState cl newlabval
               newtu        = if sn > 0 then (sn, op, inn):tu else tu
               newcl        = sl
               newwl        = if sb then (sn,newlabval):wl else wl
           in  procUnary (inn,lv) unops newtu newcl newwl
```

The binary application is quite similar except that we need to combine the state we are investigating with each previously found state:

```
procBinary _ _ tb cl wl [] = (tb,cl,wl)
procBinary (inn, lv) binops tb cl wl (s:sl) =
              let (newtb, newcl, newwl) = procBinaryOne (inn,lv) binops tb cl wl s
              in procBinary (inn, lv) binops newtb newcl newwl sl
procBinaryOne :: (Int, LabelValue) -> [Operator] -> TBinary -> AllState -> StateList
                                          -> (Int,LabelValue) -> (TBinary, AllState, StateList)
procBinaryOne _ [] tb cl wl _ = (tb,cl,wl)
procBinaryOne (inn, lv) (op:binops) tb cl wl (inn_s,lv_s) =
    let newlabval1      = (app_lab_b op lv lv_s)
        (sn1, sl1, sb1) = getState cl newlabval1
        newtb1          = if sn1 > 0 then (sn1, op, inn, inn_s):tb else tb
        newwl1          = if sb1 then (sn1,newlabval1):wl else wl
    in if (lv == lv_s)
       then
          procBinaryOne (inn,lv) binops newtb1 sl1 newwl1 (inn_s,lv_s)
       else
          let newlabval2      = (app_lab_b op lv_s lv)
              (sn2, sl2, sb2) = getState sl1 newlabval2
              newtb2          = if sn2 > 0 then (sn2, op, inn_s, inn):newtb1 else newtb1
              newwl2          = if sb2 then (sn2, newlabval2):newwl1 else newwl1
          in procBinaryOne (inn,lv) binops newtb2 sl2 newwl2 (inn_s,lv_s)
```

These are al the components needed for automaton generation. The result of the worklist algorithm was of type $(TNull, TUnary, TBinary, AllState)$. We can simply translate this to a set of pattern matched functions. For the Simple machine description a part of the resulting

automaton looks something like this (we have removed some states to make things readible, therefore this is not a functionally correct automaton):

```
app_Nul CONST     = 1
app_Un  MEM   1   = 2
app_Un  MEM   2   = 2
app_Bin PLUS  1 2 = 4
app_Bin ASGN  1 3 = 6
app_Bin PLUS  2 1 = 3
app_Bin ASGN  2 3 = 6

getstate 1 = [LE Con2 [16] 0,LE Con1 [15] 0,LE Dst [7,11] 2,LE Addr [11] 2,
              LE Src [8,6] 2,LE Dst [7,10,6] 2,LE Addr [10,6] 2,LE Regx [6] 2,
              LE Src [8,5] 6,LE Dst [7,10,5] 6,LE Addr [10,5] 6,LE Regx [5] 6]
getstate 2 = [LE Src [9] 0,LE Src [8,4] 3,LE Dst [7,10,4] 3,
              LE Addr [10,4] 3,LE Regx [4] 3]
getstate 3 = [LE Plus1 [17] 2,LE Plus3 [14] 0,LE Plus2 [13] 0]
getstate 4 = [LE Plus1 [17] 0]
getstate 6 = [LE Statmt [3] 0,LE Statmt [2] 4,LE Statmt [1] 2]
```

5.2.2 Treematching with the automaton

With the described automaton, the actual tree matching becomes an easy process (and this is all that needs to be done at compile time, the automaton is generated at compile-compile time). For every node, we simply call the function app_Nul, app_Un or app_Bin. This results in an integer which is the only thing needed to pass to our parents. In this way the entire tree can be labeled in one bottom up traversal. This process goes very fast since at every node only a single lookup is needed. To build the actual deriviation we still need one top down traversal followed by one bottom up traversal. The top down traversal passes the wanted storage classes, by these storage classes we can select the rule to apply at every node. For the top down traversal we need to know which storage classes a rule wants at its children. Therefore we still need the old machine description as generated in the previous chapters. We will no longer do any calculations so we can leave out all chainrule functions but we do need the main list of rules. The complete machine description for the Simple machine thus becomes:

```
type RuleNr = Int
type Deriv  = [RuleNr]
data LElem  = LE Storage [RuleNr] Int
type Rule   = (RuleNr, Storage, Rhs)
data Rhs    = Rhs Operator Storage Storage
type Value  = Maybe Int

data Operator = Noop  | CONST__r_1_1  | CONST  | REG  | MEM
              | ASGN  | PLUS  deriving (Eq, Ord, Show)

data Storage = S_Nil  | Dst  | Plus1  | Statmt  | Plus2
             | Plus3  | Addr  | Regx  | Src  | Con1
             | Con2 deriving (Eq, Ord, Show)

rule :: RuleNr -> Rule
rule 1 = (1,Statmt,(Rhs ASGN Dst Plus1))
```

```
rule 2 = (2,Statmt,(Rhs ASGN Dst Plus2))
rule 3 = (3,Statmt,(Rhs ASGN Dst Plus3))
rule 4 = (4,Regx,(Rhs MEM Addr S_Nil))
rule 5 = (5,Regx,(Rhs CONST__r_1_1 S_Nil S_Nil))
rule 6 = (6,Regx,(Rhs CONST__r_1_1 S_Nil S_Nil))
rule 9 = (9,Src,(Rhs MEM Addr S_Nil))
rule 11 = (11,Addr,(Rhs CONST__r_1_1 S_Nil S_Nil))
rule 12 = (12,Regx,(Rhs REG S_Nil S_Nil))
rule 13 = (13,Plus2,(Rhs PLUS Src Con1))
rule 14 = (14,Plus3,(Rhs PLUS Src Con2))
rule 15 = (15,Con1,(Rhs CONST__r_1_1 S_Nil S_Nil))
rule 16 = (16,Con2,(Rhs CONST__r_1_1 S_Nil S_Nil))
rule 17 = (17,Plus1,(Rhs PLUS Src Regx))
rule _ = (0, S_Nil, (Rhs Noop S_Nil S_Nil))
```

We feel we can now give the tree matcher without any further explanation:

```
ATTR Tree [ | | labval : Int ]
SEM Tree
   | Nullary loc.labval = app_Nul @op
   | Unary   loc.labval = app_Un  @op @left.labval
   | Binary  loc.labval = app_Bin @op @left.labval @right.labval

ATTR Tree [ derivstor : Storage | | ]
SEM Tree
   | Unary  left.derivstor  = rule_lstor (rule_get $ last @loc.deriv)
   | Binary left.derivstor  = rule_lstor (rule_get $ last @loc.deriv)
            right.derivstor = rule_rstor (rule_get $ last @loc.deriv)

ATTR Tree [ | | deriv : Deriv ]
SEM Tree
   | Nullary loc.deriv = deriv_get @lhs.derivstor (getstate @labval)
   | Unary   loc.deriv = deriv_get @lhs.derivstor (getstate @labval)
             lhs.deriv = @deriv ++ @left.deriv
   | Binary  loc.deriv = deriv_get @lhs.derivstor (getstate @labval)
             lhs.deriv = @deriv ++ @left.deriv ++ @right.deriv
```

We have derived a fast tree matcher. In practice the solution is not useful yet. As mentioned we can not handle conditional instruction sets. This is reason enough to make it useless. But there is another disadvantage. In practice the automaton generated as described above becomes very large when using large machine descriptions. When we generate an automaton for an unconditional version of the Intel x86 instruction set, we get an automaton so large it can not be interpreted by Hugs. The Glashow Haskell Compiler also fails to compile such a large automaton. In the next chapter we will try to improve the automaton and the automaton generation process in such a way that it can be used in practice.

Chapter 6

Improving the automaton

6.1 Overview

The automaton must be useful in practice. It currently does not work for instructional condition sets. To make it useful in practice the first thing to do is to change the automaton in such a way that it handles conditions. By doing so we show how one writes an efficient tree matcher using attribute grammars, allthough the files generated are to large to be used with Haskell. We can translate the attribute grammar to another language than Haskell to get a useful tree matcher. After adding the conditions we will introduce some techniques to create a smaller and faster automaton. By doing so we hope we will eventually get an automaton which can be used when translating the attribute grammars to the Haskell language.

6.2 Using conditions

Using an instruction set having conditions for a number of rules, it is no longer possible to precalculate the resulting state at a certain node given its childrens states. Which state this will be depends on which conditions hold for the given node. The latter is something which can only be verified at runtime. We propose a workaround which requires an extra bottom-up traversal of our input tree. In one bottom up traversal the tree is transformed into a specialized input tree where all conditions have already been evaluated. This specialized tree has some extra operators, but no longer needs to mind conditions. For this expanded set of operators the automaton generation and the tree matching can be done in the way described in the previous chapter. The key to this solution is the set of extra operators. At compile-compile time we already know which set of rules can be applied at a certain operator. We know which rules hold which conditions and how they can be verified. These two can be combined, folding the conditions in the operator. The following example will clearify the idea. Assume we have the following instruction set for a certain machine:

```
statmt: ASGN(statmt,statmt) 2
statmt: ASGN(statmt,statmt) memop
statmt: MEM(statmt) 2
```

Here *memop* represents a certain condition. There are two operators, $ASGN$ and MEM. All rules for the operator MEM are unconditional, this operator requires no attention. The

other operator does have a conditional rule. The condition is *memop*. In advance we have no idea whether this condition will hold or fail; this is purely input dependent. We do know that the condition will either hold or fail. We can introduce one specialized version of the operator. This operator gets the name *ASGN_memop*. This operator can be added to the set of operators. The *ASGN_memop* contains all those rules which are unconditional and those rules which require the condition memop to succeed. The original operator *ASGN* still exists, but it only can apply those rules which have no conditions. It is obvious that if we have an input tree using these specialized operators, we can again generate an automaton, because the operator already states whether conditions hold or fail. This specialized tree can easily be generated in one bottom up traversal of a certain given input tree. The attribute grammar handling this conversion can easily be generated from the above machine description. For this example, the transforming attribute grammar will look like this:

```
ATTR Tree [ | | tr : Tree ]
SEM Tree
    | Nil     lhs.tr = Tree_Nil
    | Nullary lhs.tr = Tree_Nullary (applyCond @op Tree_Nil Tree_Nil @value) @value
    | Unary   lhs.tr = Tree_Unary   (applyCond @op @left Tree_Nil @value) @left @value
    | Binary  lhs.tr = Tree_Binary  (applyCond @op @left @right @value) @left @right @value
{
memop :: Tree -> Tree -> Bool

applyCond ASGN left right v | (memop left right) = ASGN_memop
                            | otherwise          = ASGN
applyCond op _ _ _ = op
}
```

The resulting tree can be fed to the tree matcher as described in the previous chapter. The automaton used must ofcourse be generated using the set of operators with the specialized versions added.

The question arises how many extra operators this will introduce. If this introduces too many new operators the resulting automaton can get far too big. The maximum set of possible operator combinations is the powerset. If we have three completely different conditions for a certain operator, this gives us a large set of new operators. But we can do with a subset of the powerset in most cases. It is likely that conditions for a certain operator will at sometimes imply or exclude each other. Let us look at an example, for an operator called OP, we have found the conditions *memop*, $range(0,0)$, $range(1,1)$ and $range(0,31)$. If we simply take the powerset we get the following set of new operators:

```
 1. OP
 2. OP_range(0,31)
 3. OP_range(1,1)
 4. OP_range(1,1)range(0,31)
 5. OP_range(0,0)
 6. OP_range(0,0)range(0,31)
 7. OP_range(0,0)range(1,1)
 8. OP_range(0,0)range(1,1)range(0,31)
 9. OP_memop
10. OP_memoprange(0,31)
11. OP_memoprange(1,1)
12. OP_memoprange(1,1)range(0,31)
13. OP_memoprange(0,0)
```

```
14. OP_memoprange(0,0)range(0,31)
15. OP_memoprange(0,0)range(1,1)
16. OP_memoprange(0,0)range(1,1)range(0,31)
```

This is a large set. But since we know which types of conditions can occur and we know something about them, we might as well use our knowledge. First of all, range might exclude one and another. It is impossible to have a variable in the range (0,0) which also is in the range (1,1). Taking this into account, we can drop the conditions 7, 8, 15 and 16. Furthermore the fact that a certain value satisfies the range (0,0) implies that it satisfies the range (0,31). Condition 4 can for instance be rewritten to $OP\_range(1,1)$. When generating the rule application functions for $OP\_range(1,1)$ we must keep in mind the fact that we have removed those implications. When collecting rules for the operator we do not only take those rules with identical conditions, but also the ones which are implied by the current condition. If we apply the change to condition 4 it will become the same as condition 3. By investigating all possible implecations and afterwards removing any double occurences in the list of conditions, we derive the following set of required conditions:

```
 1. OP
 2. OP_range(0,31)
 3. OP_range(1,1)
 5. OP_range(0,0)
 9. OP_memop
10. OP_memoprange(0,31)
11. OP_memoprange(1,1)
13. OP_memoprange(0,0)
```

We still need seven new operators for this single operator. This can still introduce a large collection of new operators for the entire set of instructions of a certain machine. Nevertheless in practice this is not the case. If we apply the above to the Intel x86 instruction set, we only need ten extra operators. Other instruction sets needed even less. It is possible to design some new machine having so many conditions that the above will introduce a gigantic amount of operators. But for instance all instruction sets the Lcc compiler ([5]) compiles to, the above introduces a maximum of fifteen additional needed operators. This is in the worst case, the average amount of operators is much less.

The above approach depends the assumption that we have no conditons which partially overlap. Either a condition is completely implied by another condition or a condition excludes another. The combination of the conditions `range(0,31)` and `0,-31` is not allowed. The above technique will not work for such a condition. This assumption holds for all of the machine descriptions provided with LCC[5]. If we encounter a machine description which does not satisfy this assumption we can translate it into a machine description for which the assumption does hold. For the above this means we introduce extra rules for the overlapping parts of the conditions. When we have the conditions `range(0,31)` and `0,-31` this means we have to introduce a copy of both of the rules, now having condition `range(0,0)`. When the situation occurs that both conditions would hold, we now have the new condition which implies both of the others.

6.3 Speeding up the lookup in the automaton

In the previous chapter part of our generated automaton looked like this:

```
app_Bin PLUS  1 2 = 4
app_Bin ASGN  1 3 = 6
app_Bin PLUS  2 1 = 3
app_Bin ASGN  2 3 = 6
```

When we have a large set of operators this lookup can get quite slow, since in worst case the numbers of needed comparisons is (#Operators * #States * #States). Therefore we change the functions slightly to make sure this worst case becomes (#Operators + #States + #States). Note that we are talking about the worst case here. If we assume the above automaton is compiled with a good compiler this can be done in constant time. But because we do not want to be completely dependent of the quality of our compiler, we rewrite it a little bit ourselfs. We simply rewrite it into:

```
app_Bin :: Operator -> State -> State -> State
app_Bin PLUS stl str = app_Bin_PLUS stl str
app_Bin ASGN stl str = app_Bin_ASGN stl str

app_Bin_PLUS :: State -> State -> State
app_Bin_PLUS x y = case x of
                      1 -> case y of
                              2 -> 4
                              _ -> 0
                      2 -> case y of
                              1 -> 3
                              _ -> 0
                      _ -> 0
app_Bin_ASGN :: State -> State -> State
app_Bin_ASGN x y = case x of
                      1 -> case y of
                              3 -> 6
                              _ -> 6
                      2 -> case y of
                              3 -> 6
                              _ -> 0
                      _ -> 0
```

This way the lookup at every node is limited at (#Operators + #States + #States).

6.4 Decreasing the size of the automaton

Treematching by the means of an automaton goes very fast. But there is a big problem. The automaton gets very large. In fact, it gets so large that neither Hugs or GHC can handle it, when we generate an automaton for the x86. The question arises whether there is some way to decrease the automaton's size. The most of the size of the automaton is used for representing the binary lookup. Most of the operators are binary. Assuming all of them are binary we have an automaton of size (#Operators * #States * #States). Obviously it is a pity that we need to introduce extra operators to be able to handle conditions. This has increased the size even more. It is however obvious that the number of states is of greater importance for the size of the final automaton. If we can somehow reduce the number of states we will gain a lot. Let us have a look at the way states are currently represented:

```
type State = Int
```

```
type StateList = [(Int,LabelValue)]

type AllState = (Map LabelValue Int, Int)
type LabelValue = [LElem]
data LElem = LE Storage [RuleNr] Int deriving Show
```

A state represents a labelvalue. A labelvalue is no more than a list having for each derivable storage class a set of rules to derive it with and the cost at which it is derived. We need both these rules and costs. We need the set of rules to be able to build the deriviation after treematching has been done. The costs are needed to pass to parents states. The rules and costs are never needed at the same time. The rules need to be known locally. The parent state only cares about the cost for each storage class.

This fact can be exploited by introducing two types of labelvalues. One type holding a set of rules for each storage class and one type holding a cost belonging to each storage class. Since we are only interested in the rules for a storage class locally, new states can be introduced there without any concequences. Only if a new cost state is found it is necessary to investigate if the new state itself introduces any further states. The states needed for the lookup at every node are the cost states. They determine the size of the automaton. Since states having different rules but the same costs for every nonterminal are considered being identical cost states, the number of cost states will likely be smaller than the original number of states. The size of rule states is not that much of an issue, because it is of small importance in the size of the automaton.

The actual cost states are removed from the final automaton. They are not important when tree matching. They have been used to generate the automaton and the integers representing them will be passed throught the tree. But we never care which costs they represented at generation time of the automaton. With the automaton using one state for the cost and the rules we find 257 states for the x86 machine. With our new approach we find only 56 cost states. We still find around 250 rule states, but since they do not contribute that much to the size of the automaton we have gained a lot. Remember, the size of the automaton was (#Operators * #States * #States). With the original approach this would be (66049 * #Operators). Our new approach reduces this to (3136 * #Operators).

The new automaton is still quite large. But it is small enough to be used with Haskell. We can compile the generated automaton, giving us a fast usable tree matcher for for instance the x86. The automaton is still quite large, the compiled tree matcher is around 40 megabytes in size. This is when compiling with GHC. Other compilers may improve this size. But despite its size the tree matcher loads and runs fast. It only needs constant time at every node of an input tree (the time to lookup a value in the automaton).

Chapter 7

Conclusion and future work

7.1 Conclusion

We have shown how one would construct a simple tree matcher using attribute grammars. The most straightforward way of doing this, without any optimizations, gives us a good understanding of the process of tree matching. The attribute clearly expresses what work is done to construct a derivation for a given input program. This fact that the attribute grammar gives us such a clear view has two major advantages. The first is that attribute grammars can be used very well to explain the ideas behind compiler generation. The second and most important advantage is that because it is so obvious how everything works, it becomes an easy task to think of optimizations on the tree matcher.

The attribute analysis in its current form is not useful for compiler generation. But since it is designed to work with any attribute grammar it can be a useful addition to the attribute grammar system. When working with large sets of attributes, we gain a lot by completely leaving out some attributes instead of exploiting Haskell's lazyness. This will also be very useful when we want to translate an attribute grammar to another target language besides Haskell.

The automaton based tree matcher gives us an efficient tree matcher. Since we are able to handle conditional instruction sets it can be used in practice. We have based our work on LCC's machine descriptions and the techniques hold for all of these machine descriptions. This gives us already a couple of target machines we can use our tree matcher with. Looking at the step which took us from the normal attribute grammar tree matcher to the automaton based version, we can conclude we are in fact partially evaluating the attribute grammar.

In [8] Alexey R. Yakushev describes a front-end for a compiler for the C-language using attribute grammars. By describing instruction selection we have handled a large part of the back end. It must be possible to express the remaining parts of the back end using attribute grammars. Therefore we think it is possible to write a complete C-compiler using attribute grammars.

7.2 Future work

7.2.1 The compiler

As mentioned we have not yet derived a complete back end. We do not take function calls into account. Register allocation and the actual code generation are also not discussed. Allthough not much of a challenge it is interesting to see what these parts will look like using attribute grammars. XXX describes a way to handle the register allocation using graph coloring. It is an algorithm which can be used after instruction selection has been done assuming we have infinite registers available. This method fits our instruction selection solution.

All the time we have been working with a tree matcher. A well known optimization while compiling is common subexpression elimination. When common subexpression elimination is used we no longer have a tree shaped intermediate program but the program becomes a directed acyclic graph. Maybe the techniques used for tree matching can be adjusted in a way they can handle those directed acyclic graphs.

7.2.2 The analysis

As mentioned in the chapter describing the analysis, it is far from complete. The analysis is a useful addition to the attribute grammar system, it is interesting to complete the analysis.

Bibliography

[1] Steven W.K. Tjiang Alfred V. Aho, Mahadevan Ganapathi. Code generation using tree matching and dynamic programming. In *ACM Transactions on Programming Languages and Systems*, pages 491–516. ACM, 1989.

[2] Doaitse Swierstra Arthur Baars, Andres Loh. *Implementation of programming languages, lecture notes*. Utrecht University, 2002.

[3] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[4] Eelco Dijkstra. Currently untitled. Currently unfinished thesis about code generation and code generation using DAG's.

[5] Christopher W. Fraser and David R. Hanson. *A retargetable C compiler: design and implementation.* The Benjamin/Cummings Publishing Company, Inc., 1995.

[6] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[7] Rudoplh Landwehr Helmut Engelmann, Friedrich-Wilhelm Schroer. Beg–a generator for efficient back ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and implementation.*, pages 24(7):227–237. ACM, 1989.

[8] Alexey Rodriguez Yakushev. Geen idee. Hoe moet ik het verhaal van Alexey in BIBTEX opnemen?