Basic Blocks and Traces for Intermediate Representation

Hunyadi Ioan Daniel
Department of Informatics
University "Lucian Blaga" of Sibiu
5-7 Dr. Ioan Ratiu Street, Sibiu
ROMANIA

Abstract: The semantic analyses phase of a compiler must translate abstract syntax into abstract machine code. It can do this after type-checking, or at the same time. An intermediate representation is a kind of abstract machine language that can express the target-machine operations without committing to too much machine-specific details. But it is also independent of the details of the source language. The front-end of the compiler does lexical analysis, parsing, semantic analyses, and translation to intermediate representation. The back-end does optimization of the intermediate representation and translation to machine language.

Key-Words: compiler, lexical analysis, abstract syntax, intermediate representation, abstract machine language

1 Introduction

The intermediate representation tree language is defined by the package `Tree`, containing abstract classes `Stm` and `Exp` and their subclasses.

A good intermediate representation has several qualities:

- It must be convenient for the semantic analyses phase to produce.
- It must be convenient to translate into real machine language, for all the desired target machines.
- Each construct must have a clear and simple meaning, so that optimizing transformations that rewrite the intermediate representation can easily be specified and implemented.

Individual pieces of abstract syntax can be complicated things, such as array subscripts, procedure calls, and so on. And individual "real machine" instructions can also have a complicated effect. Unfortunately, it is not always the case that complex pieces of the abstract syntax correspond exactly to the complex instructions that a machine can execute.

The intermediate representation should have individual components that describe only extremely simple things: a single fetch, store, add, move, or jump. Then any "chunky" piece of abstract syntax can be translated into just the right set of abstract machine instructions.

2 Problem Formulation

The trees generated by the semantic analyses phase must be translated into assembly or machine language. The operators of the `Tree` language are chosen carefully to match the compatibilities of most machines. However, there are certain aspects of the tree language that do not correspond exactly with machine languages, and some aspects of the `Tree` language interfere with compile-time optimization analyses.

For example, it's useful to be able to evaluate the subexpressions of an expression in any order. But the subexpressions of `Tree.exp` can contain side effects – `ESEQ` and `CALL` nodes that contain assignment statements and perform input/output. If tree expressions did not contain `ESEQ` and `CALL` nodes, then the order of evaluation would not matter.

```
package Tree;

abstract class Exp
CONST(int value)
NAME(Label label)
TEMP(Temp.Temp temp)
BINOP(int binop, Exp left,Exp right)
MEM(Exp exp)
CALL(Exp func, ExpList args)
ESEQ(Stm stm, Exp exp)

abstract class Stm
MOVE(Exp dst, Exp src)
EXT(Exp exp)
JUMP(Exp exp, Temp.LabelList targets)
```

```
CJUMP(int rel, Exp left,Exp right,
      Label iftrue, Label iffalse)
SEQ(Stm left, Stm right)
LABEL(Label label)
```

Here is a description of the meaning of each tree operator. First, the expression (Exp), which stand for the computation of some value (possibly with side effects):

`CONST(i)` – The integer constant *i*.
`NAME(n)` – the symbolic constant *n* (corresponding to an assembly language label)
`TEMP(t)` – Temporary *t*. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.
`BINOP(o,e1,e2)` – The application of binary operator *o* to operands *e1*, *e2*. Subexpression *e1* is evaluated before *e2*. The integer arithmetic operator are PLUS , MINUS, MUL, DIV; the integer bitwise logical operators are AND, OR, XOR; the integer logical shift operators are LSHIFT, RSHIFT; the integer arithmetic right-shift is ARSHIFT. The MiniJava language has only one logical operator, but the intermediate language is meant to be independent of any source language; also, the logical operators might be used in implementing other features of MiniJava.
`MEM(e)` – The content of *wordSize* bytes of memory starting at address *e* (where *wordSize* is defined in the Frame module). Note that when MEM is used as the left child of a MOVE, it means "store", but anywhere else it means "fetch".
`CALL(f,l)` – A procedure call: the application of function *f* to argument list *l*. The subexpression *f* is evaluated before the arguments which are evaluated left to right.
`ESEQ(s,e)` – The statement *s* is evaluated for side effects, then *e* is evaluated for a result.

Some of the mismatches between `Trees` and machine-language programs are:

- The `CJUMP` instruction can jump to either of two labels, but real machines conditional jump instructions fall through to the next instruction if the condition is false.
- `ESEQ` nodes within expressions are inconvenient, because they make different orders of evaluating subtrees yield different results.
- `CALL` nodes within expressions cause the some problem.
- `CALL` nodes within argument-expressions of other `CALL` nodes will cause problems when trying to put arguments into a fixed set of formal-parameter registers.

3 Problem Solution

The transformation is done in three stages: First, a tree is rewritten into a list of *canonical trees* without `SEQ` or `ESEQ` nodes; then the list is grouped into a set of *basic blocks*, which contain no internal jumps or labels; then the basic blocks are ordered into a set of *traces* in each every `CJUMP` is immediately followed by its false label.

Thus the module `Canon` has these tree-rearrangement functions:

```
package Canon;
public class Canon{
  static public Tree.StmList
            linearize(tree.Stm s);
}
public class BasicBlocks{
  pubic StmListList blocks;
  public temp.label done;
  public BasicBlocks
             (Tree.StmList stms);
}
StmListList(Tree.StmList
       head,StmListList tail);
public class TraceSchedule{
  public traceSchedule(BasicBlocks b);
  public Tree.StmList stms;
}
```

`Liniarize` removes the `ESEQs` and moves the `CALL` to top level. Then `BasicBlocks` groups statements into sequences of straight-line code. Finally, `TraceSchedule` orders the blocks so that every `CJUMP` is followed by its `false` label.

3.1. Transformations on ESEQ

How can the `ESEQ` nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become `SEQ` nodes.

Figure 1 gives some useful identities on trees.

Identity (1) is obvious. So is identity (2): Statement *s* is to be evaluated; then e_1, e_2 and then the sum of the expressions is returned. Is *s* has side effects that affect e_1 or e_2, then either the left-hand side or the right-hand side of the first equation will execute those side effects before the expressions are evaluated.

Identity (3) is more complicated, because of the need not to interchange the evaluations of *s* an e_1. For example, if *s* is MOVE(MEM(x),y) and e_1 is BINOP(PLUS,MEM(x),z), then the program will compute a different result if *s* is evaluate before $_{e1}$ instead of after. Our goal is simply to pull *s* out of the BINOP expression. To do so, we assign e_1 into a new temporary *t*, and put *t* inside the BINOP.

It may happen that s causes no side effects that can alter the result produced by e_1. This will happen if the temporaries and memory locations assigned by *s* are not referenced by e_1.

We cannot always tell if two expressions commute. For example, whether MOVE(MEM(x),y) commute with MEM(z) depends on whether x=z, which we cannot always determine at compile time.

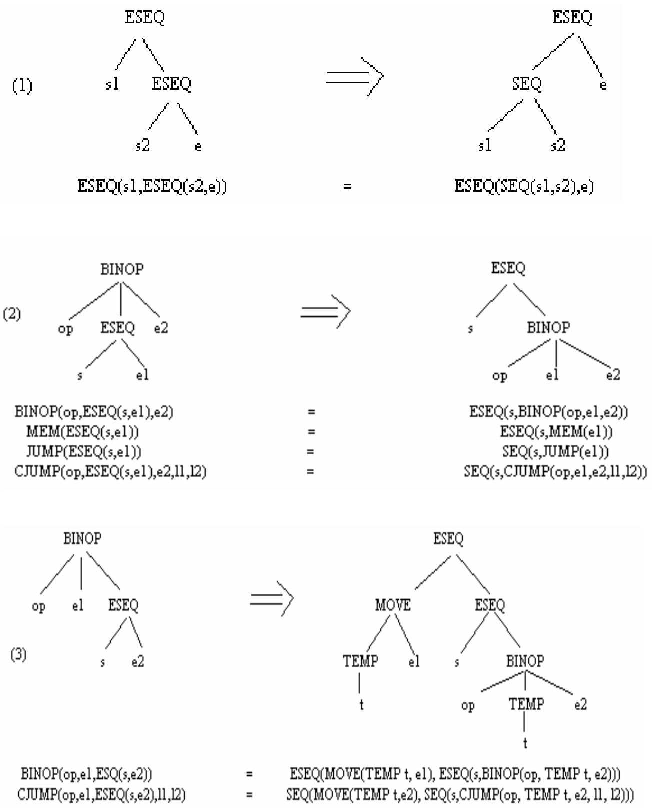

Figure 1.

The `comute` function estimates (very naively) whether a statement commutes with an expression:

```
static  boolean  commute(Tree.Stm  a,
Tree.Exp b){
   return isNop(a)
     || b instanceof Tree.NAME
     || b instanceof Tree.CONST;
}

static boolean isNop(Tree.Stm a){
  Return a instanceof Tree.EXP
   &&((Tree.EXP)a).exp instanceof
       Tree.CONST;
}
```

A constant commutes with any statement, and the empty statement commutes with any expression. Anything else is assumed not to commute.

3.2. General Rewriting rules

In general, for each kind of `Tree` statement or expression we can identify the subexpressions. Then we can make rewriting rules, similar to the ones in Figure 1, to pull the ESEQs out of the statement or expression.

For example, in [e_1,e_2,*ESEQ(s,*e_3*)*], the statement *s* must be pulled left-ward past e_2 and e_1. If they commute, we have (*s*;[e_1,e_2,e_3]). But suppose e_2 does not commute with *s*. Then we must have

(*SEQ*(*MOVE*(t_1,e_1),*SEQ*(*MOVE*(t_2,e_2)));
[*TEMP*(t_1),*Temp*(t_2),e_3])

Or if e_2 commutes with *s* but e_1 does not, we have

(*SEQ*(*MOVE*(t_1,e_1),*s*);[*TEMP*(t_1), e_2, e_3])

The `recorder` function takes a list of expressions and returns a pairs of (statement, expression-list). The statement contains all the things that must be executed before the expression-list. As shown in these examples, this includes all the statement-parts of the ESEQs, as well as any expressions to their left with which they did note commute. When there are no EXEQs at all we will use EXP(CONST 0), which does nothing, as the statement.

Algorithm. Step one is to make a "subexpression-extraction" method for each kind. Step two is to make a "subexpression-insertion" method: given an ESEQ-clean version of each subexpression, this builds a new version of the expression or statement.

These will be methods of the `Tree.Exp` and `Tree.Stm` classes:

```
package Tree;
abstract public class Exp{
   abstract public ExpList kids();
   abstract public Exp build
                  (ExpList kids);
}
abstract public class Stm{
   abstract public ExpList kids();
   abstract public Stm build
                  (ExpList kids);}
```

Each subclass `Exp` or `Stm` must implement the methods. For example:

```
package Tree;
public class BINOP extends Exp{
  public int binop;
  public Exp left, right;
  public BINOP(int b, Exp l, exp r)
  {
     binop=b;…
  }
  public final static int PLUS=0,
    MINUS=1,MUL=2,DIV=3,AND=4,OR=5,
    LSHIFT=6,RSHIFT=7,ARSHIFT=8,XOR=9;
  public ExpList kids();
  public Exp build(ExpList kids);
}
```

Other subclasses have similar (or even simpler) `kids` and `build` methods. Using these `build` methods we can write functions

```
static Tree.Stm do_stm(Tree.Stm s)
static tree.ESEQ do_exp(Tree.Exp e)
```

that pull all the ESEQs out of a statement or expression, respectively. That is, `do_stm` uses `s.kids()` to get the immediate subexpressions of *s*, which will be an expression-list *l*. It then pulls all the ESEQs out of *l* recursively, yielding a clump of side-effecting statements s_1 and a cleaned–up list *l'*. Then `SEQ(`s_1`,s.build(l'))` constructs a new statement, like the original *s* but with no ESEQs. These functions rely on auxiliary functions `reorder_stm` and `reorder_exp` for help.

The left-hand operand of the MOVE statement is not considered a subexpressions, because is the destination of the statement – its value is not used by the statement. However, if the *destination* is a memory location, then the *address* acts like a source. Thus we have,

```
public class MOVE extends Stm{
  public Exp dst,src;
  public MOVE(Exp d, Exp s)
    {dst=d; src=s;}
  public ExpList kids();
  public Stm build (ExpList kids);
}
```

Now, given a list of "kids", we pull the ESEQs out, from right to left.

3.3. Moving calls to top level

The `Tree` language permits CALL nodes to be used as expressions. However, the actual implementation of CALL will be that each function return its result in the same dedicated return-value register `TEMP(RV)`. Thus, if we have

```
BINOP(PLUS,CALL(…),CALL(…))
```

the second call will overwrite the RV register before the PLUS can be executed.

We can solve this problem with a rewriting rule. The idea is to assign each return value immediately into a fresh temporary register, that is

CALL(fun,args) →
ESEQ(MOVE(TEMP t,CALL(fun,args)),TEMP t)

Now the ESEQ-eliminator will percolate the MOVE up outside of its containing BINOP expressions. This technique will generate a few extra MOVE instructions, which the register allocator can clean up.

The rewriting rule is implementing as follows: `reorder` replaces any occurrence of `CALL(f, args)` by

ESEQ(MOVE(TEMP t_{new}*,CALL(f,args)), TEMP* t_{new}*)*

and calls itself again on the `ESEQ`. But `do_stm` recognizes the pattern

MOVE(TEMP t_{new}*, CALL(f,args))*

and does not call `reorder` on the CALL node in that case, but treats the *f* and *args* as the children of the MOVE node. Thus, `reorder` never "sees" any CALL that is already the immediate child of MOVE. Occurrences of the pattern *EXP(CALL(f,args))* are treated similarly.

3.4. A linear list of Statement

Once an entire function body s_0 is processed with `do_stm`, the result is a tree $s_0^{'}$ where all the SEQ nodes are near the top (never underneath any other kind of node). The `liniarize` function repeatedly applies the rule

```
SEQ(SEQ(a,b),c)=SEQ(a,SEQ(b,c))
```

The result is that $s_0^{'}$ is linearized into an expression of the form

$$SEQ(s_1,SEQ(s_2,…,SEQ(s_{n-1},s_n)...))$$

Here the SEQ nodes provide no structuring information at all, and we can just consider this to be a simple list of statements,

$$s_1, s_2, …, s_{n-1}, s_n$$

where none of the s_i contain SEQ or ESEQ nodes.

These rewrite rules are implemented by `linearize`, with an auxiliary function `linear`:

```
static Tree.StmList linear
         (Tree.SEQ s,Tree.StmList l){
return linear(s.left,linear(s.right,l));
}
static Tree.StmList linear
         (Tree.Stm s,Tree.StmList l){
  if(s istanceof Tree.SEQ)
    return linear((Tree.SEQ)s,l);
  else return new Tree.StmList(s,l);
}
static public Tree.StmList lineariaze
         (Tree.Stm s){
    return linear(do_stm(s),null);
}
```

3.5. Taming conditional branches

Another aspect of the `Tree` language that has no direct equivalent in most machine instruction sets is the two-way branch of the `CJUMP` instruction. The `Tree` language `CJUMP` is designed with two target labels for convenience in translating into trees and analyzing trees. On a real machine, the conditional jump either transfers control (on a true conditions) or "falls through" to the next instruction.

To make the trees easy to translate into machine instructions, we will rearrange them so that every `CJUMP(cond,`l_t`,`l_f`)` is immediately followed by `LABEL(`l_f`)`, its "false branch". Each such `CJUMP` can be directly implemented on a real machine as a conditional branch to label l_t.

We will make this transformation in two stages: first, we take the list of canonical trees and form them into *basic blocks*; then we order the basic blocks into a *trace*.

In determining where the jumps go in a program, we are analyzing the program's *control flow*. Control flow is the sequencing of instructions in a program, ignoring the data values in registers and memory, and ignoring the arithmetic calculations. Of course, not knowing the data values means we cannot know whether the conditional jumps will go to their true or false labels.

In analyzing the control flow of a program, any instruction that is not a jump has an entirely uninteresting behavior. We can lump together any sequence of nonbranch instructions into a basic block and analyze the control flow between basic blocks.

A basic block is a sequence of statements that is always entered at the beginning and exited at the end, that is:

- The first statement is a `LABEL`.
- The last statement is a `JUMP` or `CJUMP`.
- There are now other `LABELs, JUMPs`, or `CJUMPs.`

The algorithm for dividing a long sequence of statements into basic blocks is quite simple. The sequence is scanned from beginning to end. Whenever a `LABEL` is found, a new block is started (and the previous block is ended). Whenever a `JUMP` or `CJUMP` is found, a block is ended (and the next block is started).If this leaves any block not ending with a `JUMP` or `CJUMP`, then a `JUMP` to the next block's label is appended to the block. If any block has been left without a `LABEL` at the beginning, a new label is invented and stuck there.

We will apply this algorithm to each function body in turn. The procedure "epilogue" will not be part of this body, but is intended to follow the last statement. When the flow of program execution reaches the end of the last block, the epilogue should follow. But is inconvenient to have a "special" block that must come last and that has no `JUMP` at the end. Thus, we will invent a new label `done` – intended to mean the beginning of the epilogue – and put a `JUMP(NAME done)` at the end of this block.

3.6. Traces

Now the basic blocks can be arranged in any order, and the result of executing the program will be the same – every block ends with a jump to the appropriate place. We can take advantage of this to choose an ordering of the blocks satisfying the condition that each CJUMP is followed by its false label.

At the same time, we can also arrange that many of the unconditional JUMPS are immediately followed by their target label. This will allow the deletion of these jumps, which will make the compiled program run a bit faster.

A *trace* is a sequence of statement that could be consecutively executed during the execution of the program. It can include conditional branches. A program has many different, overlapping traces. For our purposes in arranging `CJUMPs` and false-labels, we want to make a set of traces that exactly covers the program: each block must be in exactly one trace. To minimize the number of `JUMPs` from one trace to another, we would like to have as few traces as possible in our covering set.

A very simple algorithm will suffice to find a covering set of traces. The idea is to start with some block – the beginning of a trace – and follow a possible execution path – the rest of the trace.

4 Conclusion

An efficient compiler will keep the statements grouped into basic blocks, because many kinds of analysis and optimization algorithms run faster on basic blocks than on individual statements. For the MiniJava compiler we seek simplicity in the implementation of later phases. So we will flatten the ordered list of traces back into one long list of statements.

For some application of traces, it is important that any frequently executed sequence of instructions (such as a body of a loop) should occupy its own trace. This helps not only to minimize the number of unconditional jumps, but also may help with other kinds of optimizations, such as register allocation and instruction scheduling.

Acknowledgements: The research was supported by the Grant Agency of CNMP (grant No. 73-CEEX-II-03 from 31/07/2006).

References:

[1] Daniel Hunyadi, Translating Programming Languages into executable cod, *Wseas Transactions on information science and Applications*, WSEAS Press, 2007, ISSN: 1790-0832, pp. 145-152

[2] Cattell R.G., Automatic derivation of code generators from machine descriptions, *ACM Trans. on Programming Languages and Systems 2(2),* 1980, pp.173-190

[3] Chambers C., Leavens G.T., Typechecking and modules for multimethods, *ACM Trans. on Programming Languages and Systems 17(6),* 1995, pp.805-843

[4] Chen W., Turau B., Efficient dynamic look-up strategy for multi-methods, *European Conference an Object Oriented Programming (ECOOP '94),* 1994

[5] Burkl M.G., Fisher G.A., A practical method for LR and LL syntactic error diagnosis and recovery, *ACM Trans. on Programming Languages and Systems 9(2),* 1987, pp.164-167

[6] Flanagan C., Sabry A., Duba B.F., Felleisen M., The essence of compiling with continuation, *Proceedings of the ACM SINGLAN '93Conference on Programming Language Design and Implementation.* ACM Press, New York, 1993, 237-247

[7] Pelegri-Llopart E., Graham S.L., Optimal Code generation for expression tree: An application of BURS theory, *15th ACM Symp. on Principles of Programming Languages*, 1988, ACM Press, New York , 294-308.