Translation for Intermediate Code

Hunyadi Ioan Daniel, Popa M. Emil, Fabian Detlef Ralf, Mocan Ionela
Department of Informatics
University "Lucian Blaga" of Sibiu
5-7 Dr. Ioan Ratiu Street, Sibiu
ROMANIA

Abstract: - Any large software system is much easier to understand and implement if the designer takes care with the fundamental abstractions and interfaces. Each phase is implemented as one or more software modules. The semantic analyses of a compiler must translate abstract syntax into abstract machine code. It can do this after type-checking, or at the same time. Though it is possible to translate directly to real machine code, this hinders portability and modularity. Suppose we want compilers for N different source languages, targeted to M different machine. In principle this is N-M compilers, a large implementation task.

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

1 Introduction

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 (IR) 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.

A portable compiler translates the source language into IR ant then translates the IR into machine language, as illustrated in Fig.1. Now only N front ends and M back ends are required. Such an implementation task is more reasonable.

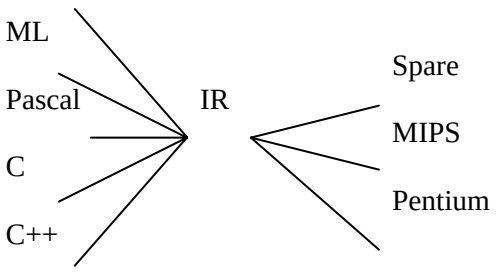


Fig.1. Compilers for four languages and three target machines with an IR.

Even when only one front-end and one back-end are being built, a good IR cam modularize the task, so that the front end is not complicated with machine-specific details, and the back-end is not bothered with information specific to one source language. Many different kinds of IR are used in compilers. For this compiler we have chosen simple expression syntax.

2 Problem Formulation

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.

3 Problem Solution

Therefore, 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.

```
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)

other classes:
ExpList(Exp head, ExpList tail)
StmList(Stm head, StmList tail)

other constants:
final static int BINOP.PLUS,
 BINOP.MINUS, BINOP.MUL, BINOP.DIV,
 BINOP.AND, BINOP.OR,
 BINOP.LSHIFT,BINOP.RSHIP,
 BINOP.ARSHIFT, BINOP.XOR;
final static int CJUMP.EQ,
   CJUMP.NE, CJUMP.LT, CJUMP.GT,
   CJUMP.LE, CJUMP.GE, CJUMP.ULT,
   CJUMP.ULE, CJUMP.UGT, CJUMP.UGE;
```

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.

The statements (`stm`) of the tree language perform side effects and control flow:

MOVE(TEMP t,e) – Evaluate *e* and move it into temporary *t*.

MOVE(MEM(e1),e2) – Evaluate *e1*, yielding address *a*. The evaluate *e2*, and store the result into *wordSize* bytes of memory starting at *a*.

EXP(e) – Evaluate *e* and discard the results.

JUMP(e,labs) – Transfer control (jump) to address *e*. The destination *e* may be a literal label, as in NAME (*lab*), or it may be an address calculated by any other kind of expression. For example, a C-language `switch(i)` statement may be implemented by doing arithmetic on *i*. The list of labels *labs* specifies all the possible location that the expression *e* can evaluate to; this is necessary for dataflow analysis later. The common case of jumping to a known label *l* is written as `JUMP(NAME l, new LabelList(l,NULL))`, but the `JUMP` class has an extra constructor so that this can be abbreviated as `JUMP(l)`.

CJUMP(o,e1,e2,t,f) – Evaluate *e1, e2* in that order, yielding values *a, b*. Then compare *a, b* using the relational operator *o*. If the result is `true`, jump to *t*; otherwise jump to *f*. The relational operators are EQ and NE for integer equality and nonequality (signed or unsigned); signed integer inequalities LT, GT, LE, GE; and unsigned integer inequalities ULT, ULE, UGT, UGE.

SEQ(s1,s2) – The statement *s1* followed by *s2*.

LABEL(n) – Define the constant value of name *n* to be the current machine code address. This is like a label definitions in assembly language. The value NAME(n) may be the target of jumps, calls, etc.

It is almost possible to give a formal semantic to the `Tree` language. However, there is no provision in this language for procedure and function definitions – we can specify only the body of each function. The procedure entry and exit sequences will be added later as special "glue" that is different for each target machine.

Translation of abstract syntax expressions into intermediate tree is reasonably straightforward; but there are many cases to handle. We will cover the translation of various language construct, including many from MiniJava.

The MiniJava grammar has clearly distinguished statements and expression. In languages such as C, the distinction is blurred. For example, an assignment in C can be used as an expression. When translating such languages, we will have to ask the following question. What should the representation of an abstract syntax expression be in `Tree` language? At first it seems obvious that it should be `Tree.Exp`. This is true only for certain kind of expressions, the ones that compute a value. Expressions that return no value are more naturally represented by `Tree.Stm`. And expressions with boolean values, such as `a>b`, might best be represented as an conditional jump – a combination of `Tree.Stm` and a pair of destinations represented by `Temp.Labels`.

It is better instead to ask, "how might the expression be used?" Then we can make the right kind of methods for an object-oriented interface to expressions. Both for MiniJava and other languages, we end up with `Translate.Exp`, not the same class as `Tree.Exp`, having three methods:

```
package Translate;
public abstract class Exp{
      abstract Tree.Exp unEx();
      abstract Tree.Stm unNx();
      abstract Tree.Stm unCx
       (Temp.Label t, Temp.Label f);
  }
```

`Ex` – stands for an "expression", represented as a `Tree.Exp`.

`Nx` – stands for "no result", represented as a `Tree` statement.

`Cx` – stands for "conditional", represented as a function from label-pair to statement.

For example, the MiniJava statement

```
if (a<b && c<d){
   //true block
}
else {
   //false block
}
```

might translate to a Translate.Exp whose unCx method is roughly like

```
Tree.Stm unCx(Label t, Label f){
  Label z=new Label();
  return new SEQ(new
    CJUMP(CJUMP.LT,a,b,z,f), new
    SEQ(new LABEL(z),new
    CJUMP(CJUMP.LT,c,d,t,f)));
  }
```

The abstract class `Translate.Exp` can be instantiated by several subclasses: `Ex` for an ordinary expression that yields a single value, `Nx` for an expression that yields no value, and `Cx` for a "conditional" expression that jumps to either *t* or *f*:

```
class Ex extends Exp{
  Tree.Exp exp ;
  Ex(Tree.Exp e) {exp=e;}
  Tree.Exp unEx() {return exp;}
  Tree.Stm unNx() {...?...}
  Tree.Stm unCx(
        Label t, Label f) {...?...}
}
class Nx extends Exp {
  Tree.Stm stm;
  Nx(Tree.Stm s) {stm=s;}
  Tree.Exp unEx() {...?...}
  Tree.Stm unNx() {return stm;}
  Tree.Stm unCx(
        Label t, Label f) {...?...}
}
abstract class Cx extends Exp{
Tree.Exp unEx(){
```

```
   Temp r=new Temp();
   Label t=new Label();
   Label f=new Label();
return new Tree.ESEQ(new Tree.SEQ
(new Tree.Move(new Tree.Temp(r),new
Tree.Const(1)),new Tree.SEQ(unCx(t,
f),new Tree.SQE(new Tree.LABEL(f)
,new Tree.SEQ(new Tree.Move(new
Tree.TEMP(r), new Tree.CONST(0)),
new          Tree.LABEL(t))))),new
Tree.TEMP(r));
}
abstract Tree.Stm unCx(Label t,
   Label f);
Tree.Stm unNx(){...}
}
```

Each kind of `Translate.Exp` class must have similar conversion methods. The `unCx` method is still abstract. But the `unEx` and `unNx` methods can still be implemented in terms of the `unCx` method.

3.1 Structured *l*-values

An *l*-value is the result of an expression that can occur on the *left* of an assignment statement. An *r*-value is one that can only appear on the right of an assignment. That is, an *l*-value denotes a *location* that can be assigned to, and an *r*-value does not. Of course, an *l*-value can occur on the right of an assignment statement. In this case the contents of the location are implicitly taken.

All the variables and *l*-values in MiniJava are scalar, since it has only one component. Even a MiniJava array or object variable is really a pointer.

In C or Pascal there are structured *l*-value – structs in C, records in Pascal - that are not scalar. In a C compiler, the `access` type would require information about the size of the variables. Then, the `MEM` operator of the `TREE` intermediate language would need to be extends with a notation of size:

```
package Tree;
abstract class Exp
 MEM(Exp exp, int size)
```

The translation of a local variable into an IR tree would look like

```
   MEM(+(TEMP fp, CONST kn),S)
```

where the *S* indicates the size of the object to be fetched or stored (depending on whether this tree appears on the left or right of a MOVE).

Leaving out the size on MEM nodes makes the MiniJava compiler easier to implement, but limits the generality of its intermediate representation.

3.2 Subscripting and field selection

To select field *f* of a record *l*-value *a* (to calculate *a.f*), simply add the constant field offset of *f* to the address *a*.

An array variable *a* is an *l*-value; so is an array subscript expression *a[i]*, even if *i* is not an *l*-value. To calculate the *l*-value a[i] from *a*, we do arithmetic on the address of *a*. Thus, in a Pascal compiler, the translation of an *l*-value (particularly a structured *l*-value) should not be something like in Fig. 2, but should instead be the `Tree` expression representing the base address of the array (Fig. 3).

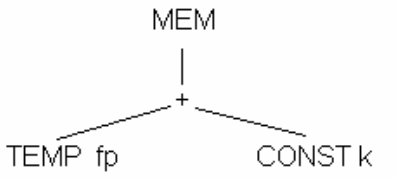


Fig. 2.

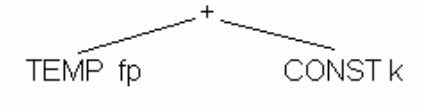
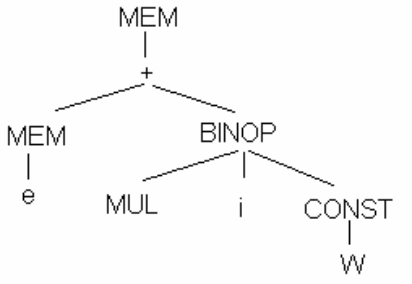


Fig. 3.

In the MiniJava Language, there are no structured, or "large" *l*-value. This is because all object and array values are really pointers to object and array structures. The "base address" of the array is really the contents of a pointer variable, so `MEM` is required to fetch this base address.

Thus, if *a* is a memory-resident array variable represented as `MEM(e)`, then the contents of address *e* will be a one-word pointer value *p*. The contents of addresses *p*, *p+W*, *p+2W*,…, will be the elements of the array. Thus, *a[i]* is just *l*-values and MEM nodes, like in Fig. 4.




```
MEM{+(MEM(e),
    BINOP(MUL,i,CONST(W)))
```

Fig. 4.

Technically, an *l*-value should be represented as an address (without the top MEM node in the diagram above). Converting an *l*-value to an *r*-value (when it is used an expression) means *fetching* from that address. Assigning to an *l*-value means *storing* to that address. We are attaching the MEM node to the *l*-value before knowing whether it is to be fetched or stored. This works only because in the `Tree` intermediate representation, MEM means both store (when used as the left child of a MOVE) and fetch (when used elsewhere).

3.3 Function definitions

A function is translated into a segment of assembly language with a *prologue*, a *body*, and an *epilogue*. The body of a function is an expression, and the *body* of the translation is simply the translation of that expression.

The *prologue*, which precedes the body in the assembly-language version of the function, contains

1. pseudo-instructions, as needed in the particular assembly language, to announce the beginning of function.
2. a label definition for the function name.
3. an instruction to adjust the stack pointer (to allocate a new frame).
4. instructions to save "escaping" arguments into the frame, and to move nonescaping arguments into fresh temporary registers.
5. store instructions to save any callee-save registers – including the return address register – used within the functions.

Then comes

6. the function *body*.

The *epilogue* comes after the body and contains

7. an instruction to move the return value (result of the function) to the register reserved for that purpose.
8. load instructions to restore the callee-save registers.
9. an instruction to reset the stack pointer (to deallocate the frame).
10. a return instruction (JUMP to the return address)
11. pseudo-instructions, as needed, to announce the end of function.

Some of these items (1,3,9 and 11) depend on exact knowledge of the frame size, which will not be known until after the register allocator determines how many local variables need to be kept in the frame because they don't fit in registers. So these instructions should be generated very late, in a `FRAME` functions.

To implement 7, the `Translate` phase should generate a move instruction

```
MOVE(RV,body)
```

that puts the result of evaluating the body in the return value (`RV`) location specified by the machine-specific `frame` structure:

```
package Frame;
public abstract class Frame{
      .
      .
      .
      abstract public Temp RV();
}
```

4 Conclusion

To simplify the implementation of the translator, we may do without the `Ex, Nx, Cx` constructors. The entire translation can be done with ordinary value expression. This means that there is only one `Exp` class. This class contains one field of type `Tree.Exp` and only an `unEx()` method. Instead of `Nx(s)`, use `Ex(EXEQ(s.CONST 0))`. For conditionals, instead of a Cx, use an expression that just evaluates to 1 or 0.

The intermediate representation trees produced from this kind of naïve translation will be bulkier and slower than a "fancy" translation. But they *will* work correctly, and in principle a fancy back-end optimizer might be able to clean up the clumsiness. In any case, a clumsy but correct translator is better than a fancy one that doesn't work.

References:

[1] 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*

[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]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

[6] 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