Compiler Principle

Prof. Dongming LU

Apr. 15th, 2024

Content

1. INTRODUCTION

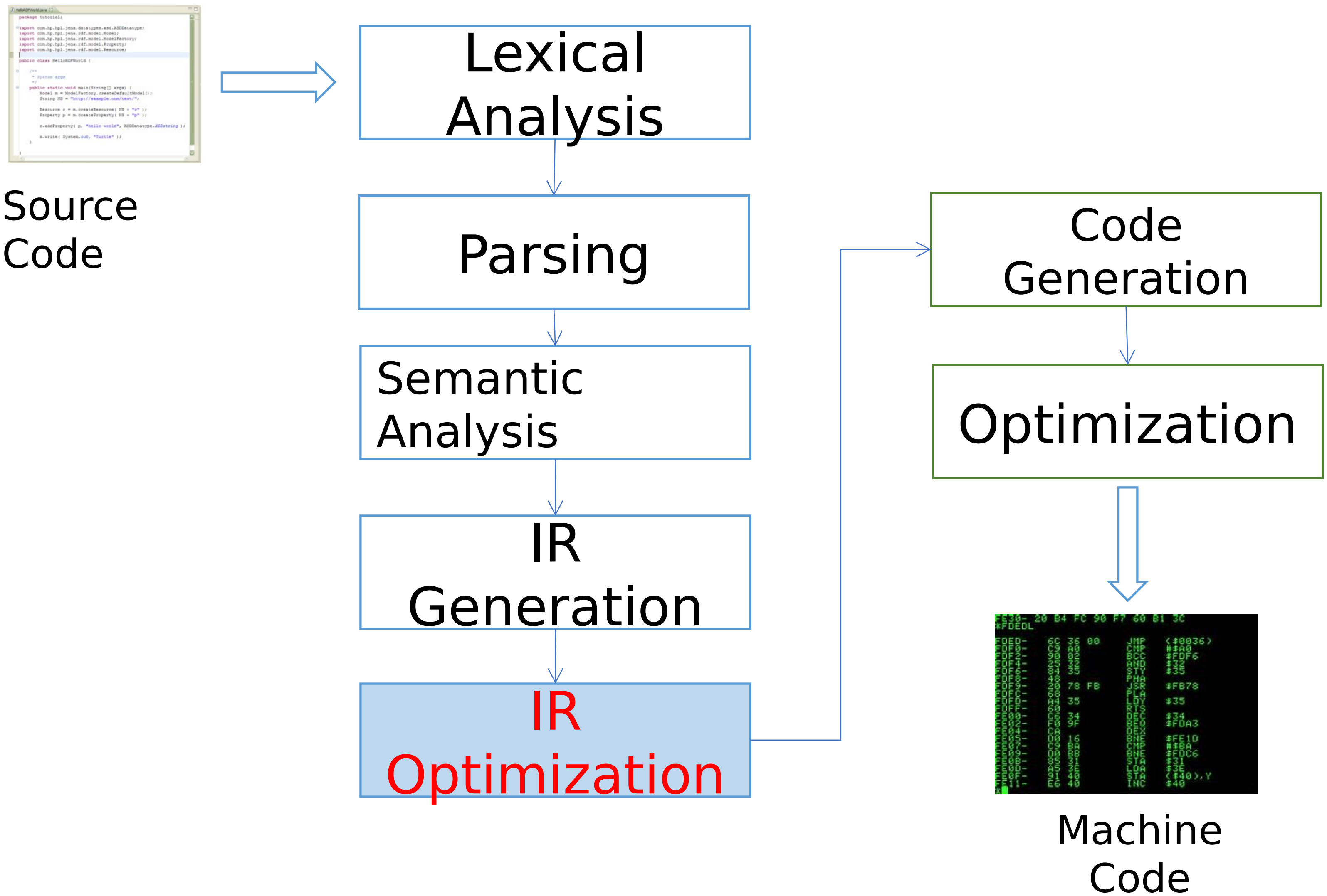
2. LEXICAL ANALYSIS
3. PARSING
4. ABSTRACT SYNTAX
5. SEMANTIC ANALYSIS

6. ACTIVATION RECORD
7. TRANSLATING INTO INTERMEDIATE CODE

8. **BASIC BLOCKS AND TRACES**

8 Basic Blocks and Traces

Where We Are



Source Code
Lexical Analysis
Parsing
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Introduction

- The Mismatches between Trees and machine-language programs
  - ✓ Tree language operators are chosen carefully to match the capabilities of most machines.
  - ✓ Some aspects do not correspond exactly with machine languages;
  - ✓ Some aspects of the Tree language interfere with compile-time optimization analyses.

Some of the Mismatches

- The CJUMP instruction can jump to either of two labels.
  - ✓ Real machines' conditional jump instructions fall through to the next instruction if the condition is false.

- ESEQ nodes within expressions are inconvenient.
  - ✓ Different orders of evaluating subtrees yield different results
- CALL nodes within expressions cause the same problem.

- CALL nodes within the argument-expressions of other CALL nodes will cause problems
  - ✓ When trying to put arguments into a fixed set of formal-parameter registers.

Three stages of the transformation

1. **A tree is rewritten into a list of canonical trees without SEQ or ESEQ nodes**

2. **This list is grouped into a set of basic blocks, which contain no internal jumps or labels**

3. **The basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label.**

The module *canon*

- **The tree-rearrangement functions:**

```
/* canon.h*/
Typedef struct C_stmListList_ *C_stmListList;
struct C_block { C_stmListList stmLists; Temp_label label;};
Struct C_stmListList_ { T_stmList head; C_stmListList tail;};

T_stmList C_linearize(T_stm stm);
Struct C_block C_basicBlocks(T_stmList stmList);
T_stmList C_traceSchedule(struct C_block b);
```

The module *canon*

- Linearize removes the ESEQs and moves the CALLs to top level.
- Basic Blocks groups statements into sequences of straight-line code.
- Trace Schedule orders the blocks so that every CJUMP is followed by its false label.

8.1 Canonical Trees

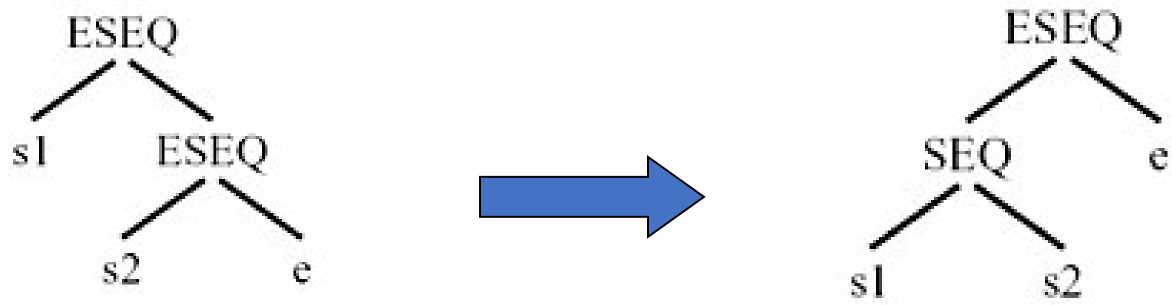
Definition of canonical tree

Canonical trees as having these properties:

1. No SEQ or ESEQ.
2. The parent of each CALL is either EXP(…) or MOVE(TEMP *t*, …).

Transformations on ESEQ

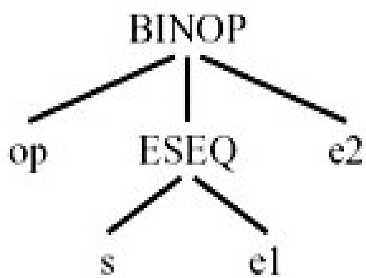
- The idea is to lift the ESEQ nodes higher and higher in the tree, until they can become SEQ nodes.



$\mathrm{ESEQ}(s_1, \mathrm{ESEQ}(s_2, e))$

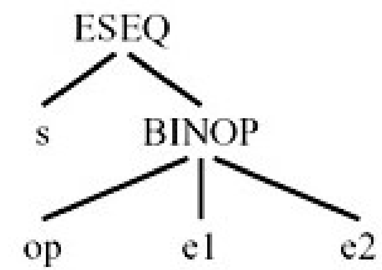
$\mathrm{ESEQ}(\mathrm{SEQ}(s_1, s_2), e)$

Transformations on ESEQ



$$\mathrm{BINOP}(op, \mathrm{ESEQ}(s, e_1), e_2)$$

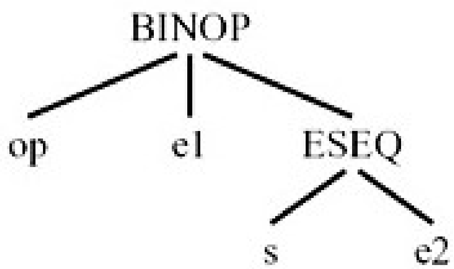
$$\mathrm{MEM}(\mathrm{ESEQ}(s, e_1))$$
$$\mathrm{JUMP}(\mathrm{ESEQ}(s, e_1))$$
$$\mathrm{CJUMP}(op, \mathrm{ESEQ}(s, e_1), e_2, l_1, l_2)$$

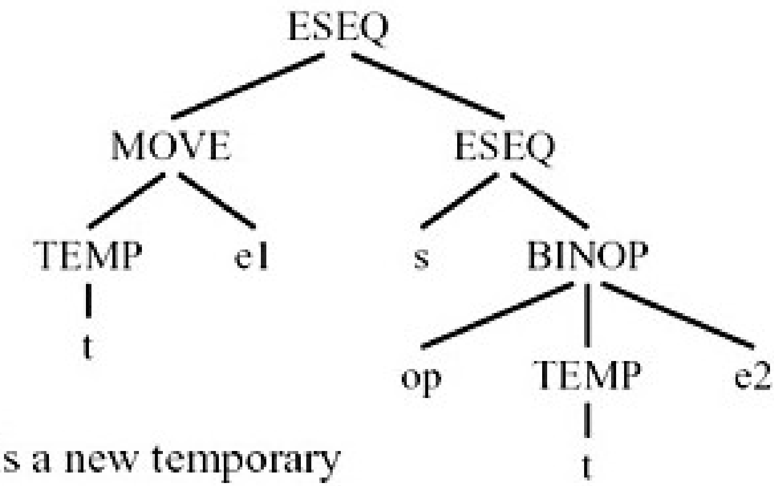


$$\mathrm{ESEQ}(s, \mathrm{BINOP}(op, e_1, e_2))$$

$$\mathrm{ESEQ}(s, \mathrm{MEM}(e_1))$$
$$\mathrm{SEQ}(s, \mathrm{JUMP}(e_1))$$
$$\mathrm{SEQ}(s, \mathrm{CJUMP}(op, e_1, e_2, l_1, l_2))$$

Transformations on ESEQ





t is a new temporary

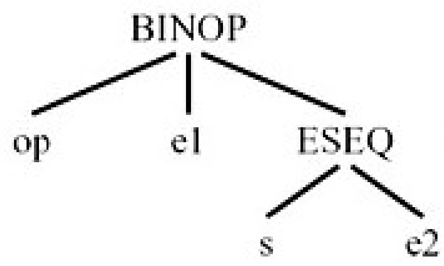
$\mathrm{BINOP}(op, e_1, \mathrm{ESEQ}(s, e_2))$

$\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}\ t, e_1),$
$\quad \mathrm{ESEQ}(s, \mathrm{BINOP}(op, \mathrm{TEMP}\ t, e_2)))$

$\mathrm{CJUMP}(op, e_1, \mathrm{ESEQ}(s, e_2), l_1, l_2)$

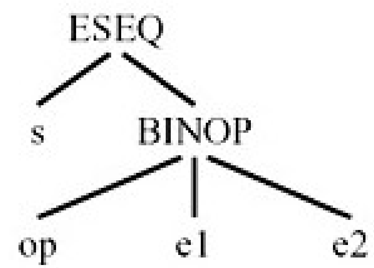
$\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}\ t, e_1),$
$\mathrm{SEQ}(s, \mathrm{CJUMP}(op, \mathrm{TEMP}\ t, e_2, l_1, l_2)))$

Transformations on ESEQ



if s, e_1 commute

$\Rightarrow$



if s, e_1 commute

$$\mathrm{BINOP}(op, e_1, \mathrm{ESEQ}(s, e_2)) \quad = \quad \mathrm{ESEQ}(s, \mathrm{BINOP}(op, e_1, e_2))$$

$$\mathrm{CJUMP}(op, e_1, \mathrm{ESEQ}(s, e_2), l_1, l_2) \quad = \quad \mathrm{SEQ}(s, \mathrm{CJUMP}(op, e_1, e_2, l_1, l_2))$$

Transformations on ESEQ

- The commute function estimates (very naively) whether a statement commutes with an expression:
  - ✓ A constant commutes with any statement .
  - ✓ The empty statement commutes with any expression.
  - ✓ Anything else is assumed not to commute.

General Rewriting Rules

- For each kind of Tree statement or expression, similar rules can be made to pull ESEQs out of the statement or expression.

[*e*1, *e*2, ESEQ(*s*, *e*3)]

s commute with e1, e2

(*s*; [*e*1, *e*2, *e*3])

*e*2 not commute with *s*

(SEQ(MOVE(t1, e1), SEQ(MOVE(t2, e2), s)); [TEMP(t1), TEMP(t2), e3])

*e*2 commutes with *s* but *e*1 does not

(SEQ(MOVE(t1, e1), s); [TEMP(t1), e2, e3])

Algorithm

- Step 1: Make a "subexpression-extraction" method for each kind.
- Step 2: make a “subexpression-insertion” method， given an ESEQ-clean version of each subexpression

```
typedef struct expRefList_ *expRefList
struct expRefList_ {T_exp *head; expRefList tail;};

struct stmExp {T_stm s; T_exp e;}

Static T_stm reorder(expRefList rlist);

Static T_stm do_stm(T_stm stm);
Static struct stmExp do_exp(T_exp exp);
```

Algorithm

- **Reoder: to pull out all the ESEQs out of a list of expression and combine the statement-parts of these ESEQ into one big T_stm**
- **Reoder(*l*) calls upon an auxiliary function do_exp on each expression in list *l***

```
static struct stmExp do_exp(T_exp exp){
Switch(exp->kind){
    case T_BINOP:
        return StmExp(reorder(ExpRefList(&exp->u.BINOP.left,
                      ExpRefList(&exp->u.BINOP.right,NULL))),exp);
    case T_MEM:
        return StmExp(reorder(ExpRefList(&exp->u.MEM, NULL)),exp);
    case T_ESEQ:{
       struct stmExp x=do_exp(exp->u.ESEQ.exp);
        return StmExp(seq(do_stm(exp->u.ESEQ.stm), x.s), x.e) ;
       }
     …
```

Algorithm

```
static T_stm do_stm(T_stm stm){
Switch(stm->kind){
    case T_SEQ:
        return seq(do_stm(stm->u.SEQ.left),
                          do_stm(stm->u.SEQ.right));
    case T_JUMP:
        return seq(reorder(ExpRefList(&stm-
>u.JUMP.exp, NULL)), stm);
     ...
```

- **With the assistance of do_exp and do_stm, the reorder function can pull the statement s*i* out the expression e*i* on its list of references ,going from right to left.**

Move CALLs to Top Level

BINOP(PLUS, CALL(…..), CALL(….))

- The Tree language permits CALL nodes to be used as subexpressions.
- Each function returns its result in the same dedicated return-value register TEMP(RV)
- The second call will overwrite the RV register before the PLUS can be executed.

Move CALLs to Top Level

- The idea is to assign each return value immediately into a fresh temporary register

  CALL(fun, args) $\rightarrow$

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

A Linear List of Statements

- The linearize function repeatedly applies the rule

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

- Just consider this to be a simple list of statements

  $s_1,s_2,s_3,s_4,\ldots\ldots\ldots s_{n-1},s_n$

  none of the s_i contain SEQ or ESEQ nodes

A Linear List of Statements

```
Static T_stmList linear(T_stm stm, T_stmList right)
{
   if (stm->kind == T_SEQ)
      return linear(stm->u.SEQ.left, linear(stm->u.SEQ.right,right));
  else return T_StmList(stm, right);
}

T_stmList C_linearize(T_stm stm){
     return linear(do_stm(stm), NULL);
```

The end of Chapter 8(1)