Linear ACSE

Alessandro Barenghi Ettore Speziale Michele Tartara

Politecnico di Milano

Contents

Contents

Work-flow

The LANCE files are a list of `statements`:

- see `Acse.y`

We have just seen a simple statement:

- the `write` statement

It is linear:

- no conditional
- translation depends only on `write` itself

Today we will see something close:

- assignments
- expressions

Interlude

Before going forward:

- how is it possible to generate instructions?

An helper function is associated to every instruction:

- it allows to emit the instruction hiding low level details
- see `axe_gencode.h`

Generating instructions

Instruction	Helper
ADD	gen_add_instruction
ADDI	gen_addi_instruction
READ	gen_read_instruction
BEQ	gen_beq_instruction

Contents

A Complete Statement

Consider the simple assignment `a = 4`:

- we want to copy `4` inside `a`
- we need both `a` (left-hand side) and `4` (right-hand side)

When do we known all the data needed?

- when the parser recognize the `assign_statement` rule

Generalized Assignment I

Think at left-hand sides:

scalar stored in a register

array cell stored somewhere in the memory

Moreover:

- they have different syntax

And right-hand sides:

- just something evaluable to a scalar

Generalized Assignment II

Left-hand sides are too different:

- the rule must be specialized

Right-hand side are equal:

- should be factorized through the `exp` rule

Now, better to switch to code:

- look at the `assign_statement` rule in `Acse.y`
- scalars are stored into registers and manually handled [1]
- arrays are managed exploiting a function from `axe_array.h`

[1] The `if` is explained later.

Contents

The Need to Type

Most of ACSE code deals with expressions:

- assignments
- arrays indexing
- conditionals

The exp has been typed to generalize expressions management:

Expression type [2]

```
typedef struct t_axe_expression {
  int value;
  int expression_type;
} t_axe_expression;
```

[2] See axe_struct.h.

Building Expressions

The expression framework:

- allows to combine expressions together
- generates code to compute expressions
- described in `axe_expressions.h`

They are built recursively:

- two base cases: `IMMEDIATE` and `REGISTER` expressions
- intermediate values kept into `REGISTER` expressions
- `create_expression` allows to build base expressions

Expression Values

The expression value is stored into the `value` field:

immediate the value of the immediate

register the register storing that expression

Un-boxing expressions

```
if($3.expression_type == IMMEDIATE)
  gen_addi_instruction(..., $3.value);
else
  gen_add_instruction(..., $3.value,
                           CG_DIRECT_ALL);
```

Add

Very simple expressions:

Adding two expressions

```
exp:
...
    | exp AND_OP exp {
      $$ = handle_bin_numeric_op(program,
                                 $1,
                                 $3,
                                 ANDB);
    }
...
```

Lesser Than

Relational operators handled with expressions too:

Comparing two expressions

```
exp:
...
    | exp LT exp {
      $$ = handle_binary_comparison(
              program, $1, $3, _LT_);
    }
...
```

Contents

Bibliography

A. Di Biagio and G. Agosta.
Advanced Compiler System for Education.
http://corsi.metid.polimi.it, 2008.

Formal Languages and Compilers Group.
Formal Languages and Compilers – CorsiOnline.
http://corsi.metid.polimi.it, 2010.