Advanced Models and Programs:

MicroC, pointer programming, type system, compilation

Peter Sestoft
Monday 2010-02-15

Agenda

- Concepts: Pointers, lvalue, rvalue
- MicroC parsing challenges
- MicroC abstract syntax
- MicroC compilation schemes
The essence of C: Pointers

- Main innovations of C (1972) over Algol 60:
  - Structs, as in COBOL and Pascal
  - Pointers, pointer arithmetics, pointer types, array indexing as pointer indexing
  - Using {} for blocks, as in C++, Java, C#
- Java and safe C# lack pointers and are completely different from C/C++
- Except for the core language, which is Algol with {} instead of begin end.

Ptr basics

- A pointer \( p \) refers to a storage location
- The dereference expression \( *p \)
  - may mean the content of the location as in \( *p+4 \)
  - may mean the storage location itself, as in \( *p = x+4 \)
- The pointer that points to \( x \) is \&\( x \)
- Pointer arithmetics: \( *(p+1) \) is the storage location after \( *p \)
- If \( p \) equals \&\( a[0] \) then \( *(p+i) \) equals \( p[i] \) equals \( a[i] \), so arrays are pointers
Using pointers for return values

- Example test12.c, computing square(x):

```c
void main() {
    int res;
    square(5, &res);
    write res;
    square(res, &res);
    square(res, &res);
    write res;
}

void square(int x, int *p) {
    *p = x*x;
}
```

MicroC abstract syntax
Concepts

• Expression statements
• Storage model
• Activation records (frames)
  – Base pointer bp
  – Return address
  – Array layout (B rather than C)
• Lvalue and Rvalue

Expression statements in C, C++, Java and C#

• The “assignment statement”
  \[ x = 2+y; \]
  is really an expression
  \[ x = 2+y \]
  followed by a semicolon
  \[
  \]
  • The semicolon means: ignore value

```csharp
public class ExprStatement : Statement {
    private readonly Expression e;
    public override void Compile(CEnv env, Generator gen) {
        e.Compile(env, gen);
        gen.Emit(new INCSP(-1));
    }
}
```

Value: none
Effect: change x

Value: 2+y
Effect: change x

Eval. expr. then pop its value
Storage model

- The store is an indexable stack
  - Bottom: global variables at fixed addresses
  - Plus, a stack of activation records

<table>
<thead>
<tr>
<th>globals</th>
<th>main</th>
<th>fac(3)</th>
<th>fac(2)</th>
<th>fac(1)</th>
<th>fac(0)</th>
</tr>
</thead>
</table>

- An activation record is an executing function
  - return address
  - old base pointer
  - local variables
  - temporaries

Array layout

- An array int arr[5] consists of
  - its 5 int elements
  - a pointer to arr[0]

- This is the array representation of B, the predecessor to C
Lvalue and rvalue of an expression

- Rvalue is “normal” value, right-hand side of assignment: 17, true
- Lvalue is “location”, left-hand side of assignment: x, a[2]
- In \( e_1 = e_2 \), expression \( e_1 \) must have lvalue
- Where else must an expression have lvalue in C#? In C?

<table>
<thead>
<tr>
<th></th>
<th>lvalue</th>
<th>rvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>a[2]</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>*p</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>x+2</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;x</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

MicroC parsing

- Parsing variable declarations
- Parsing lvalue and rvalue expressions
- Parsing right-associative operators
C variable type declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n</td>
<td>n is an integer</td>
</tr>
<tr>
<td>int *p</td>
<td>p is a pointer to integer</td>
</tr>
<tr>
<td>int ia[3]</td>
<td>ia is array of 3 integers</td>
</tr>
<tr>
<td>int *ipa[4]</td>
<td>ipa is array of 4 pointers to integers</td>
</tr>
<tr>
<td>int (*iap)[3]</td>
<td>iap is pointer to array of 3 integers</td>
</tr>
<tr>
<td>int *(*ipap)[4]</td>
<td>ipap is pointer to array of 4 pointers to ints</td>
</tr>
</tbody>
</table>

The Unix program cdecl helps understand C types:

cdecl> explain int *(*ipap)[4]
declare ipap as pointer to array 4 of pointer to int
cdecl> declare n as array 7 of pointer to pointer to int
int **n[7]

Parsing C variable declarations

- Cumbersome because declarations are mixfix, eg:
  int *x[4]
- Parser trick: Use C# delegate type T2V that transforms Type to VarDecl:
  public delegate VarDecl T2V(Type t);
- Return a T2V and apply it to the type:

```csharp
VarDec<out VarDecl varDecl>     (. Type ty; T2V t2v; .)
= Typ<out ty>                   (. varDecl = t2v(ty); .)
  VarDesc<out t2v>              (. String name; .)
= (Ident<out name>)            (. t2v = delegate(Type ty) { return new VarDecl(name, ty); });
| "*" VarDesc<out t2v>         (. t2v = delegate(Type ty) { return t2v(new PointerType(ty)); });
...```
Parsing Lvalue and Rvalue expressions

- LR parsing can distinguish expressions having lvalues from those that do not.
- LL parsing cannot, and must parse assignment as `Expr "=" Expr`.
- But `x+2=78;` should be illegal.
- Solution: Make grammar too general `Expr<out lhs> "=" Expr<out rhs>` and then check that `lhs` has `lvalue`:
  ```
  e = new Assignment(lhs.MakeLvalue(), rhs);
  ```

Only accesses (x, a[2], *p) have lvalue

```csharp
public abstract class Expression {
    public virtual LvalueExpression MakeLvalue() {
        throw new Exception("Illegal lvalue expression");
    }
}

public class LvalueExpression : Expression {
    private readonly Access access;
    public override LvalueExpression MakeLvalue() {
        return this;
    }
}

public abstract class Access : Expression {
    public override LvalueExpression MakeLvalue() {
        return new LvalueExpression(this);
    }
}

public class VariableAccess : Access {
    public readonly String name;
}
```
Parsing assignment \( x = y = e \), which is right associative

- Expr. \( x = y = z = v \) should parse as \( x = (y = (z = v)) \)
- Parse as Expr AssignExpr where AssignExpr is [ ‘=’ Expr ]

```java
Expr<out Expression e> (. Expression rhs; .) = LogOrExp<out e>
[ ’=’ Expr<out rhs>
  (. e = new Assignment(e.MakeLvalue(),
    rhs);
  .)
] .
```

Throws if no lvalue

MicroC checking

- Check that variables and functions are declared when used
- Check that types match
  - in arithmetic and logical expressions
  - in pointer and array indexing
  - in pointer dereferencing and address-of
  - in function parameter passing
  - in assignment expressions
Ten-minute exercise

• Assume \texttt{p} is pointer to \texttt{int}
• What kinds of pointer arithmetics are allowed by MicroC.cs?
  – Addition of integer: \texttt{p+2}
  – Multiplication by integer: \texttt{p*2}
  – Subtraction of integer: \texttt{p-2}

• What should be changed to make pointer difference \texttt{p-q} have type \texttt{int}?

Compiling arithmetic expressions and assignment

• \texttt{<e1>} means: the result of compiling \texttt{e1}

  • Compile 17 as rvalue:
    \texttt{CSTI 17}

  • Compile \texttt{e1 + e2} as rvalue:
    \texttt{<e1> as rvalue}
    \texttt{<e2> as rvalue}
    \texttt{ADD}

  • Compile \texttt{e1 = e2} as rvalue:
    \texttt{<e1> as lvalue}
    \texttt{<e2> as rvalue}
    \texttt{STI}
Compiling lvalues and rvalues

- Compile $e$ as rvalue: $<e>$ as lvalue
  LDI

- Compile $&e$ as rvalue: $<e>$ as lvalue

- Compile $x$ as lvalue:
  GETBP
  CSTI $<xoffset>$
  ADD

- Compile $e_1[e_2]$ as lvalue:
  $<e_1>$ as rvalue
  $<e_2>$ as rvalue
  ADD

- Compile $*e$ as lvalue: $<e>$ as rvalue

Compiling blocks and declarations

- To compile a block $\{ s_1 \ s_2 \ \ldots \ s_n \}$
  - Enter new scope
  - $<s_1> \ <s_2> \ \ldots \ <s_n>$
  - Leave scope

- To compile declaration $\text{int } x$:
  - Generate code to push 0; this takes a stack place

- To compile declaration $\text{int } a[5]$:
  - Generate code to allocate 5 stack locations, that is, increment SP by 5
  - Generate code to compute address of the first of those locations
## Statement compilation schemes

- Compile
  \[
  \text{if (e) s1 else s2:}
  \]
  \[
  \begin{align*}
  &<e> \text{ as rvalue} \\
  &\text{IFZERO L1} \\
  &<s1> \\
  &\text{GOTO L2} \\
  &L1: <s2> \\
  &L2:
  \end{align*}
  \]

- Compile
  \[
  \text{while (e) s:}
  \]
  \[
  \begin{align*}
  &L1: <e> \text{ as rvalue} \\
  &\text{IFZERO L2} \\
  &<s> \\
  &\text{GOTO L1} \\
  &L2:
  \end{align*}
  \]

## Ten-minute exercise

- What code should be generated for a do-while block:
  \[
  \text{do s while (e) ;}
  \]

- What code should be generated for a for statement:
  \[
  \text{for (e1; e2; e3) s}
  \]
The Micro-C stack machine

- Compiling and running Micro-C

```c
Coco -namespace MicroC MicroC.ATG
csc MicroC.cs Scanner.cs Parser.cs
csc Machine.cs
unzip test-c.zip
MicroC test5.c check
MicroC test5.c compile
Machine a.out
Machine /trace a.out
```

Build compiler
Build stack machine
Compile example
Run it with trace

The stack of frames

- Example test5.c, computing factorial:

```c
void main() {
    int res;
    fac(3, &res);
    write res;
}

void fac(int n, int *p) {
    if (n == 0)
        *p = 1;
    else {
        int tmp;
        fac(n-1, &tmp);
        *p = n * tmp;
    }
}
```

- n is input parameter
- p is output parameter, a pointer to where to put the result
- tmp holds the result of the recursive call to fac
- &tmp gets the pointer to tmp
The code for test5.c

```
0 CALL 0 L1
3 STOP
4 L1:
  33 LDI
  34 CSTI 0
  36 EQ
  37 IFZERO L3
  39 GETBP
  40 CSTI 1
  42 ADD
  43 LDI
  44 CSTI 1
  46 STI
  47 INCSP -1
  49 GOTO L4
  51 L3:
  51 CSTI 1
  53 GETBP
  54 CSTI 0
  55 INCSP -1
  56 ADD
  57 LDI
  58 CSTI 0
  59 INCSP -1
  60 SUB
  61 CSTI 2
  62 CSTI 2
  63 ADD
  64 ADD
  65 CALL 2 L2
  68 INCSP -1
  70 GETBP
  71 CSTI 1
  73 ADD
  74 LI
  75 GETBP
  76 CSTI 0
  78 ADD
  79 LI
  80 GETBP
  81 CSTI 2
  83 ADD
  84 LI
  85 MUL
  86 STI
  87 INCSP -1
  88 GETBP
  89 INCSP -1
  91 L4:
  93 RET 1
```

The stack of frames

- Example test5.c: computing fac(0)
- Stack frame for fac(0):
  - What stack frame?
    - [3 -999 0]

```
    ret-addr  old bp  n  p
      15 2 0 2
```
Highlights from computing fac(3)

Exercises for week 4

- Write Micro-C programs
  - to sum elements of an array
  - to make histogram of the elements of array

- Change Micro-C parser and compiler to
  - Allow (==) and (!=) to compare pointers and ints
  - Add a C/C++/Java/C#-style for-loop
  - Add prefix increment (++) operator
  - Add conditional operator (?:)

- Investigate the (Java or .NET) bytecode
  - for selection sort
  - for a generic class and generic method