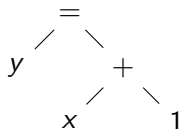


Intermediate Representation Construction in a Nutshell

Christoph Mallon

23. Dezember 2013

Code Generation for Expressions



- ▶ Do *not* evaluate expression
- ▶ Create code, which, when run, evaluates the expression
- ▶ IR construction is code generation, just for a virtual machine
- ▶ *Recursively* create code for expressions
- ▶ Create code for operands, then create code for current node
- ▶ Same order as evaluating, but generating code instead

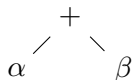
```
virtual Value* Expression::makeRValue();
```

Code Generation for a Constant

1

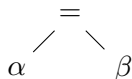
```
Constant::makeRValue() {  
    return createConstantNode(value);  
}
```

Code Generation for +



```
Addition::makeRValue() {  
    l = left->makeRValue();  
    r = right->makeRValue();  
    return createAddNode(l, r);  
}
```

Code Generation for =



- ▶ L-value: *address* of the object denoted by an expression
- ▶ R-value: *value* of an expression
- ▶ L and R stand for left and right hand side (of assignment)
- ▶ Assignment happens as *side effect* of the expression

```
Assignment :: makeRValue() {  
    address = left ->makeLValue();  
    value   = right ->makeRValue();  
    createStoreNode(address, value);  
    return value;  
}
```

Code Generation for * (Indirection)

*
|
 α

- ▶ R-value of $*\alpha$ is the value loaded from the address denoted by the R-value of α
- ▶ L-value of $*\alpha$ is the R-value of α

```
Indirection::makeRValue() {  
    address = operand->makeRValue();  
    return createLoadNode(address);  
}
```

```
Indirection::makeLValue() {  
    return operand->makeRValue();  
}
```

Code Generation for & (Address)

&
|
 α

- ▶ R-value of this expression is the address of the operand
- ▶ Expression is not an L-value

```
Address :: makeRValue() {  
    return operand->makeLValue();  
}
```

```
Address :: makeLValue() {  
    PANIC("invalid L-value");  
}
```

Connection between L-value and R-value

- ▶ R-value is just loading from L-value
- ▶ Unfortunately most expressions are not an L-value

```
virtual Value* Expression::makeRValue() {  
    address = makeLValue();  
    return createLoadNode(address);  
}
```

```
virtual Value* Expression::makeLValue() {  
    PANIC("invalid L-value");  
}
```

Different Code Generation in Different Contexts

```
expr = ... /* L-value */  
... = expr /* R-value */  
if (expr) /* Control flow */
```

- ▶ Code generated depends on context, where the expression appears
- ▶ L-value: *address* of the object denoted by the an expression
- ▶ R-value: *value* of an expression
- ▶ Control Flow: Branch depending on result
- ▶ Different contexts call each other recursively for operands

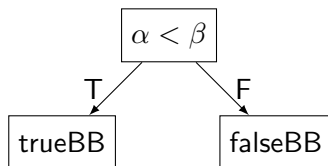
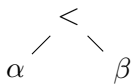
Control-Flow Code Generation for Condition

```
if (C) S1 else S2
```

- ▶ If C evaluates to $\neq 0$ continue at S1
- ▶ Otherwise continue at S2
- ▶ Label/Basic block of S1 and S2 are input for code generation
- ▶ Recall code generation for short circuit evaluation using attribute grammars

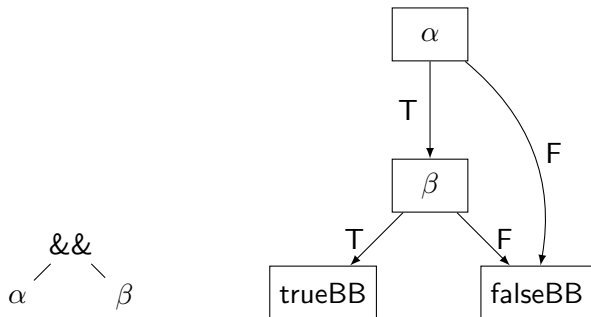
```
virtual void Expression::makeCF(trueBB, falseBB);
```

Control-Flow Code generation for $<$



```
LessThan::makeCF(trueBB, falseBB) {  
    l    = left ->makeRValue();  
    r    = right ->makeRValue();  
    cond = createCmpLessThanNode(l, r);  
    createBranch(trueBB, falseBB, cond);  
}
```

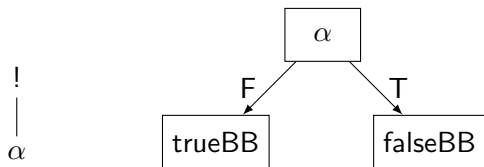
Control-Flow Code generation for &&



- ▶ Lazy evaluation part of semantics
- ▶ Stop evaluation if value of left hand side determines result

```
LogicalAnd :: makeCF(trueBB, falseBB) {  
  extraBB = createBasicBlock();  
  left -> makeCF(extraBB, falseBB);  
  setCurrentBB(extraBB);  
  right -> makeCF(trueBB, falseBB);  
}
```

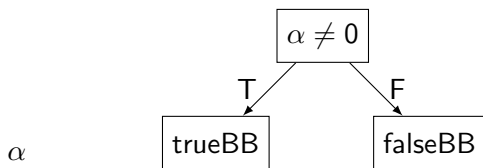
Control-Flow Code Generation for !



- ▶ To negate the condition, just swap the targets

```
LogicalNegation :: makeCF(trueBB, falseBB) {  
    operand -> makeCF(falseBB, trueBB);  
}
```

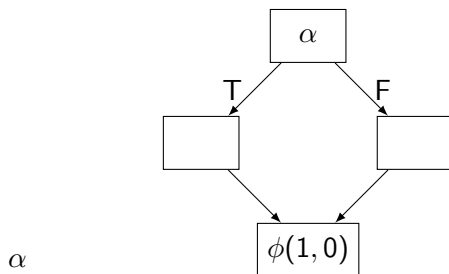
Control-Flow Code Generation for Arbitrary Expression



- ▶ Test R-value $\neq 0$

```
Expression::makeCF(trueBB, falseBB) {  
    PANIC("implement this");  
}
```

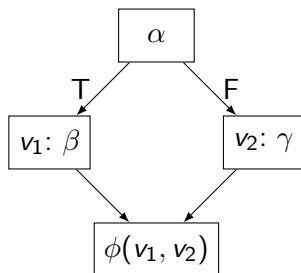
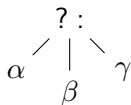
R-value Code Generation for Control Flow Expression



- ▶ Control flow operators produce 1 and 0
- ▶ Select the value depending on whether the true or false basic block was reached

```
ControlFlowExpression::makeRValue() {  
    PANIC("implement this");  
}
```

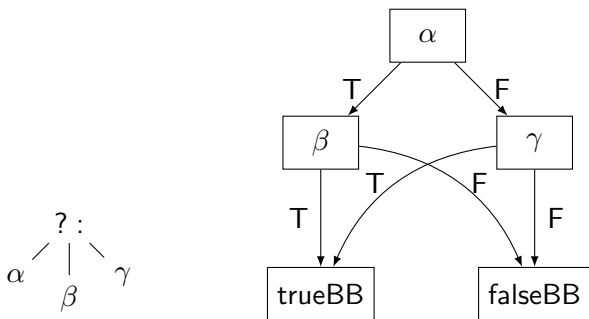
R-value Code Generation for Conditional Expression



- ▶ First evaluate condition α to control flow
- ▶ Then either evaluate consequence β or alternative γ
- ▶ Pick result using a ϕ

```
ConditionalExpression :: makeRValue() {  
    PANIC("implement this");  
}
```

Control-Flow Code Generation for Conditional Expression



- ▶ First evaluate condition α to control flow
- ▶ Then either evaluate consequence β or alternative γ to control flow

```
ConditionalExpression::makeCF() {  
    PANIC("implement this");  
}
```
