Intermediate Representation Construction in a Nutshell

Christoph Mallon and Johannes Doerfert

18. Dezember 2015
**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
Code Generation for a Constant

```cpp
Constant::makeRValue() {
    return createConstantNode(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 an expression
- R-value: `value` of an expression
- Control Flow: Branch depending on result of an expression
- Different contexts call each other recursively for operands
Code Generation for +

\[ \begin{array}{c}
\alpha \\
+ \\
\beta 
\end{array} \]

- Generate code for operands
- Then generate code for +

```c
Addition::makeRValue() {
    l = left->makeRValue();
    r = right->makeRValue();
    return createAddNode(l, r);
}
```
Code Generation for $\Rightarrow$

$\Rightarrow$

\[
\begin{array}{c}
\alpha \\
\downarrow \\
\Rightarrow \\
\downarrow \\
\beta
\end{array}
\]

- **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 $\ast$ (Indirection)

$\ast$

$\alpha$

- R-value of $\ast\alpha$ is the value loaded from the address denoted by the R-value of $\alpha$
- Address of the object denoted by $\ast\alpha$ is the value of $\alpha$: L-value of $\ast\alpha$ is the R-value of $\alpha$

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

Indirection::makeLValue() {
    return operand->makeRValue();
}
```
Code Generation for & (Address)

\[ \& \]

\[ \alpha \]

- Value of \&\(\alpha\) is the address of the object denoted by \(\alpha\):
  - R-value of \&\(\alpha\) is the L-value of \(\alpha\)
- \&\(\alpha\) does not denote an object: \&\(\alpha\) is not an L-value

---

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

\texttt{Address::makeLValue()} {
  \texttt{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, i.e. do not denote an object

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

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

```cpp
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 is part of semantics: \( \beta \) might have side effects
Stop evaluation if value of left hand side determines result

```c
LogicalAnd :: makeCF(trueBB, falseBB) {
    extraBB = createBasicBlock();
    left -> makeCF(extraBB, falseBB);
    setCurrentBB(extraBB);
    right -> makeCF(trueBB, falseBB);
}
```
Control-Flow Code Generation for $\alpha$

To negate the condition, just swap the targets

```plaintext
LogicalNegation :: makeCF(trueBB, falseBB) {
    operand -> makeCF(falseBB, trueBB);
}
```
Control-Flow Code Generation for Arbitrary Expression

Test R-value ≠ 0

```
virtual 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

```cpp
ControlFlowExpression::makeRValue() {
  PANIC("implement this");
}
```
R-value Code Generation for Conditional Expression

- First evaluate condition $\alpha$ to control flow
- Then either evaluate consequence $\beta$ or alternative $\gamma$
- Pick result using a $\phi$

```
ConditionalExpression :: makeRValue() {
    PANIC(”implement this”);
}
```
Control-Flow Code Generation for Conditional Expression

- First evaluate condition $\alpha$ to control flow
- Then either evaluate consequence $\beta$ or alternative $\gamma$ to control flow

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