Compilerbau mit Phoenix

Analyse und Synthese
Symbol Tables

• information about source code constructs
• symbol table entries:
  – identifier (as a character string, lexeme)
  – type
  – "meta-type"
  – storage location (incl. storage class, ...)
  – visibility/access
  – ...

© 2008 Martin v. Löwis
Scopes

• symbol table per scope
• name hiding
• name resolution: search names in scopes
  – name lookup
  – typically "most-closely nested"
  – static vs. dynamic scopes
  – overloading
  – lookup by metatype
Intermediate Code: Three-Address Code

- instructions of the form $x = y \ op \ z$
- $x$, $y$, $z$ either source code constructs, or compiler-generated temporaries
- $\text{op}$: binary operator
  - also supports arrays:
    - $x = y[z]$
    - $x[y] = z$
- additionally instructions for control flow
  - ifFalse $x$ goto $L$
  - ifTrue $x$ goto $L$
  - goto $L$
- additional instruction for assignment: $x = y$
Three-Address Code (2)

• "addresses"
  – name
  – constant
  – temporary

• operands:
  – binary
  – unary (no z-axes)
  – indexed copy
  – jumps
  – procedure calls: "param" instruction, y = call p,n
  – address and pointer operations: x = &y, x = *y, *x = y
Three-Address Code (3)

- quadruple form: \( \text{op, arg1, arg2, result} \)
- triple form: \( \text{op, arg1, arg2} \)
  - each instruction has a code address
  - result referred-to by code address
    - minus, \( c \)
    - \( \ast, b, (0) \)
    - minus, \( c \)
    - \( \ast, b, (2) \)
    - \( +, (1), (3) \)
    - \( =, a, (4) \)
Static Single-Assignment Form (SSA)

• all assignments to distinct variables
  – often: versioned variables (name, number)
• issue: assignment in alternative code paths
  – if(cond) x=1; else x=-1;
    y = x*a;
• solution: symbolic function
  – if(cond) x₁=1; else x₂=-1;
    y = \Phi(x₁,x₂) * a
• advantages: simpler optimization algorithm
  – dead stores: variable is never read
  – constant propagation: variable's value is a constant; variable then itself becomes a constant
Types and Declarations

- type checking
  - aspect of well-formedness
- types in translations
  - storage layout
  - field access
Type Expressions

- basic types
- type combinators/constructors
- type expression: terms describing types
  - basic types
  - type names
  - array type: T[n] or T[]
  - record type: list of (fieldname, field type)
  - function types: T1 → T2
  - cartesian product (e.g. for parameter types): T1 × T2
  - union types: list of alternatives (fieldname, field type)
    - optionally: discriminated unions
  - type variables (e.g. for parametrized types)
Type Equivalence

• types T1, T2 are equivalent if:
  – they are the same basic type
  – if T1 is a type alias/named type, they are equivalent if the type that T1 refers to is equivalent to T2 (likewise if T2 is an alias)
  – if they are constructed types: if the same constructor is used, and recursively the type components are the same

• type compatibility may go beyond equivalence:
  – basic types can get converted
  – OO: assignments may be polymorphic
  – C++: user-defined type conversions
Storage Layout

- assign width and alignment to each type
- basic types: width, alignment defined by target machine
- record types: traverse fields
  - assign offset to each field
  - offset(previous_field) + width(previous_field) + padding(alignment(field))
  - alignment of record = max(alignment(field))
  - width = offset(last_field) + size(last_field) + alignment(padding(maxalignment))
- array types: field width * n
Translation of Expressions

- AST: unary, binary operators, funktion calls, field and array references, assignments
  - possibly pointer dereferences, ternary operator, casts
- typing: assign a type to every expression
  - typically bottom-up
- generating stack machine code: post-order traversal
- generating three-address code: introduction of temporary variables
  - store generated variable in AST node
Type Checking

- type system
- static vs. dynamic type checking
  - *sound type system*: no need for dynamic type checks
  - *strongly typed*: programs with type errors are rejected by compiler
- type synthesis: determine type of expression from types of operands, definition of operator
- type inference: determine type of expression from context
  - e.g. for Python's `len(x)`, infer that `x` must be a list/sequence
  - use type variables to denote types of expressions not yet inferred
Type Conversion

• foo((float)i);
• two effects
  – typeof (T)i is T, not typeof(i)
  – a temporary variable is introduced holding the value of (T)i
    • can be eliminated if typeof(i) has the same internal representation as T
• implicit type conversion
  – *widening vs narrowing* (typically: widening implicit, narrowing explicit)
  – in binary operation: T1 op T2: widen to max(T1, T2)
  – in assignments/parameter passing: widen to target type
Overload Resolution

- Example: C++

1. Compute set of *candidate functions*
   - for function calls: visible functions of global or static scope
   - for operators: also include operators defined in class of left operand
   - Koenig lookup: also include functions in associated namespaces
   - Also consider templates
   - ...

2. Restrict to set of *viable functions*
   - for each candidate and each parameter, try to find *implicit conversion sequence*
   - standard conversion sequence (exact match, promotion, conversion), user-defined conversion sequence, ellipsis conversion sequence

3. Compute *best viable function*
   - by comparing conversion sequences

- Error if either no viable function, or multiple best viable functions
Control Flow

• boolean expressions
  – alter control flow
  – compute logical values

• short-circuit expressions: &&, ||
  – translate to if statements

• possible representation of control flow: associate "next" statements with each instruction
  – two different next statements for boolean expressions
Control Flow (2)

- **P → S**
  - S.next = newlabel()
  - P.code = S.code || label(S.next)

- **S → assign**
  - S.code = assign.code

- **S → if (B) S₁**
  - B.true = newlabel()
  - B.false = S₁.next = S.next
  - S.code = B.code || label(B.true) || S₁.code

- **S → S₁ S₂**
  - S₁.next = newlabel()
  - S₂.next = S.next
  - S.code = S₁.code || label(S₁.next) || S₂.code
Control Flow (3)

- boolesche Ausdrücke
- z.B. \( B \rightarrow !B_1 \)
  - \( B_1.\text{true} = B.\text{false} \)
  - \( B_1.\text{false} = B.\text{true} \)
  - \( B.\text{code} = B_1.\text{code} \)
- \( B \rightarrow E_1 \text{ rel } E_2 \)
  - \( B.\text{code} = E_1.\text{code} || E_2.\text{code} || \)
    \( \text{gen('if' } E_1.\text{addr } \text{rel}.\text{op } E_2.\text{addr 'goto' } B.\text{true}) || \)
    \( \text{gen('goto' } B.\text{false}) \)
Switch Statements

• small number of cases: sequence of gotos
• many cases: hashtable, mapping case to jump label
• special case: all indices lie in a small range
  – create label array with indices 0..max-min, indexed by case-min
  – "case" instructions in intermediate code
    • three-address: case <var> <value> <label>