Type Inference

CS 4447 / CS 9545 – Stephen M. Watt

***The* University *of* Western Ontario**

Types in C

- Numeric types
  - Exact (char, int) vs Floating point (float, double)
  - Different sizes: short vs normal vs long vs long long
  - Signedness (char, int)
- Pointer types
- Functions, arrays, structs, unions, enums

Issues in Type Analysis for C

- Type compatibility

  int i; short s; …; i = s; ….; s = i;

- Promotion

  extern int printf(char *fmt, …);
  printf(“%d %d %d %d”, c, s, i, l);

- Casts

Type Analysis Algorithm for C

- *Simple enough that it is called "type checking" by some.*
  However we must resolve the types of intermediate expressions.

- Input: A parsed function (CCode) in which all identifiers have had their declarations (including types) resolved.
- Output: A parsed function (CCode) with type annotations (CTYPE) on each expression node. Implied conversions made explicit.

- Method:
  - Single bottom-up pass.
  - Types of constants given by built in language rules.
  - Types of identifiers found from their declarations (symbol table entries).
  - Types of built in operators given by language rules.
  - Types of function arguments determined by promotion rules.
  - Types of function results determined by function type.

Type Analysis in C -- Constants

```
public void visit(CCodeString cc) {
  // TODO deal with const and long/short designations
  cc._optTypeAnnotation = new CTypePointer(new CTypeChar());
}
public void visit(CCodeCharacterConstant cc) {
  // TODO deal with long/short and signedness designations
  cc._optTypeAnnotation = new CTypeChar();
}
public void visit(CCodeIntegerConstant cc) {
  // TODO deal with long/short designations
  cc._optTypeAnnotation = new CTypeInt();
}
public void visit(CCodeFloatingConstant cc) {
  // TODO deal with e/d/l designations
  cc._optTypeAnnotation = new CTypeDouble();
}
```

Type Analysis in C -- Identifiers

```
public void visit(CCodeExprId cc) {
      CSymbol sym = cc._id._optSymAnnotation;
      if (sym == null || sym._optType == null) {
            _senv.error(cc._spos,
                        "The symbol '" + cc._id._s +
                        "' is undeclared.");
            cc._optTypeAnnotation = new CTypeInvalid();
      }
      else
            cc._optTypeAnnotation = sym._optType.realType();
  }
```

Type Analysis in C – Infixed Expressions

```
public void visit(CCodeExprInfix cc) {
  visit(cc._a);
  visit(cc._b);

  CType aType = cc._a._optTypeAnnotation;
  CType bType = cc._b._optTypeAnnotation;
  CType rType = null; // Deduced return type.

  switch (cc._op._type) {
  case CTokenType.COMMA:
       // (a, b)  has type of b.
       cc._optTypeAnnotation = bType;
       break;

  case CTokenType.VBAR_VBAR:
  case CTokenType.AMP_AMP:
       // a && b, a || b  have type int.
       needArithmeticOrPointerType(cc._op, cc._a);
       needArithmeticOrPointerType(cc._op, cc._b);
       cc._optTypeAnnotation = new CTypeInt();
       break;
  ...
```

Infixed Expressions II -- Eg

```
case CTokenType.VBAR:   case CTokenType.XOR:  case CTokenType.AMP:
        // a | b, a ^ b, a & b have unified promoted types of operands.
        needIntegralType(cc._op, cc._a);
        needIntegralType(cc._op, cc._b);
        aType = aType.promote();
        bType = bType.promote();
        rType = unifyTypes(cc, aType, bType);
        // TODO push unified type back down onto operands.
        cc._optTypeAnnotation = rType;
        break;
  case CTokenType.LSH:  case CTokenType.RSH:
        // a << b, a >> b have promoted type of a.
        needIntegralType(cc._op, cc._a);
        needIntegralType(cc._op, cc._b);
        aType = aType.promote();
        aType = aType.promote();
        // TODO push unified types back down onto operands.
        cc._optTypeAnnotation = aType;
        break;
```

Infixed Expressions III -- Eg

```
case CTokenType.PLUS:   case CTokenType.MINUS:
  aType = aType.promote();    bType = bType.promote();
  boolean aIsPointer    = aType instanceof CTypePointer;
  boolean bIsPointer    = bType instanceof CTypePointer;
  boolean aIsArithmetic = aType instanceof CTypeArithmetic;
  boolean bIsArithmetic = bType instanceof CTypeArithmetic;
  if      (aIsArithmetic && bIsArithmetic)
        rType = unifyTypes(cc, aType, bType);
  else if (aIsArithmetic && bIsPointer   )
        { rType = bType;    needIntegralType(cc._op, cc._a); }
  else if (aIsPointer    && bIsArithmetic)
        { rType = aType;    needIntegralType(cc._op, cc._b); }
  else if (aIsPointer    && bIsPointer   ) {
        if (cc._op._type == CTokenType.PLUS)
                  { _senv.error(cc._spos, "Cannot add pointers."); rType = null; }
        else
                  { rType = new CTypeInt(); /* TODO ptrdiff_t */ }
  }
  else { // Error Handling
        needArithmeticOrPointerType(cc._op, cc._a);
        needArithmeticOrPointerType(cc._op, cc._b);
  }
  cc._optTypeAnnotation = rType;
```

More General Type Inference

- With overloaded operators and functions:
  - C++ defined to allow type inference in 1 bottom up pass.
  - Baker's algorithm if return type can be overloaded (Ada).

  - Automatic conversions (e.g. Int -> Float)
  - Default arguments (arity is not known)
  - Automatic instantiation of templates.

  - Priority ranking or error if multiple possibilities.

More General Type Inference II

- Type inference of polymorphic functions, e.g. ML:
  - All occurrences of a fn-bound identifier must have the same type, but
  - Each occurrence of a let-bound identifier may have a different type, provided it is an instance of the principal (most general) type of that id.

ML Example

```
fun  length(x) =
   if null(x) then 0 else length(tl(x)) + 1;
```

length([“Alice”, “Bob”, “Eve”]) + length([2, 3, 5, 7])

ML Example

```
fun  length(x) =
   if null(x) then 0 else length(tl(x)) + 1;
```

- Set

  length: *alpha* -> *beta*

  x: *alpha*

- Infer type *beta* = type of

  if null(x) **then** 0 **else** length(tl(x)) + 1

ML Example

- Infer type *beta* = type of
  if null(x) **then** 0 **else** length(tl(x)) + 1
- **if**: boolean x *gamma x gamma*
  null: list *delta* -> boolean
  unify(list *delta, alpha*)
  0: integer
  +: integer x integer -> integer
  tl: list *epsilon* -> list *epsilon*
  unify(list *delta,* list *epsilon)* => unify(*delta, epsilon*)
  tl(x): list *delta*
  length(tl(x)): *beta*
  unify(*beta,* integer)
  1: integer
  length(tl(x)) + 1: integer
  if (if null(x) then 0 else length(tl(x)) + 1): integer

- length: list *delta* -> integer

Polymorphic Type Inference Algorithm

- Text p. 393. Unary functions only (use products for others)

1. For function defn fun id1(id2) = E
   - Create fresh type variables alpha and beta.
   - Assoc type alpha -> beta with id1 and type alpha with id2.
   - Infer type of E. Let s and t be types of id2 and E afterward.
   - Type of id1 is s -> t. Bind remaining variables with “forall”.
2. For application E1(E2)
   - Infer types of E1 & E2.
   - Let s->s’ and t be the resulting types.
   - Unify s and t. Result is of type s’.
3. For each occurrence of a polymorphic function, replace bound vars (from forall) with fresh variables and remove quantifier. Resulting type is as inferred for this instance
4. For each variable that is encountered for first time, give fresh type variable.