Compiler Design

Type Checking

Winter 2010
Static Checking

- **Static (Semantic) Checks**
  - Type checks: operator applied to incompatible operands?
  - Flow of control checks: break (outside while?)
  - Uniqueness checks: labels in case statements
  - Name related checks: same name?
Type Checking

- **Problem:** Verify that a type of a construct matches that expected by its context.

- **Examples:**
  - `mod` requires integer operands (PASCAL)
  - `*` (dereferencing) - applied to a pointer
  - `a[i]` - indexing applied to an array
  - `f(a1, a2, ..., an)` - function applied to correct arguments.

- Information gathered by a type checker:
  - Needed during code generation.
Type Systems

- A collection of **rules** for assigning **type expressions** to the various parts of a program.
- **Based on**: Syntactic constructs, notion of a type.
- **Example**: If both operators of “+”, “-”, “*” are of type integer then so is the result.
- **Type Checker**: An implementation of a type system.
  - Syntax Directed.
- **Sound Type System**: eliminates the need for checking type errors during run time.
Type Expressions

- Implicit Assumptions:
  - Each program has a type
  - Types have a structure

<table>
<thead>
<tr>
<th>Basic Types</th>
<th>Type Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Arrays</td>
</tr>
<tr>
<td>Real</td>
<td>(strings)</td>
</tr>
<tr>
<td>Enumerations</td>
<td>Records</td>
</tr>
<tr>
<td>Void</td>
<td>Sets</td>
</tr>
<tr>
<td>Variables</td>
<td>Pointers</td>
</tr>
<tr>
<td>Character</td>
<td>Functions</td>
</tr>
<tr>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>Sub-ranges</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Names</td>
<td></td>
</tr>
</tbody>
</table>
Representation of Type Expressions

$\rightarrow x$

$\rightarrow x$

(char x char)$ \rightarrow$ pointer (integer)

(const cell {
    int info;
    struct cell * next;
});
Type Expressions Grammar

Type → int | float | char | ...
| void
| error
| name
| variable
| array(size, Type)
| record((name, Type)*)
| pointer(Type)
| tuple((Type)*)
| fcn(Type, Type) (Type → Type)

Basic Types

Structured Types
A Simple Typed Language

Program → Declaration; Statement

Declaration → Declaration; Declaration
  | id: Type

Statement → Statement; Statement
  | id := Expression
  | if Expression then Statement
  | while Expression do Statement

Expression → literal | num | id
  | Expression mod Expression
  | E[E] | E ↑ | E (E)
Type Checking Expressions

\[
\begin{align*}
E &\rightarrow \text{int\_const} \quad \{ E.\text{type} = \text{int} \} \\
E &\rightarrow \text{float\_const} \quad \{ E.\text{type} = \text{float} \} \\
E &\rightarrow \text{id} \quad \{ E.\text{type} = \text{sym\_lookup(id.\text{entry}, \text{type})} \} \\
E &\rightarrow E_1 + E_2 \quad \{ E.\text{type} = \text{if } E_1.\text{type} \notin \{\text{int, float}\} \mid E_2.\text{type} \notin \{\text{int, float}\} \} \\
&\quad \quad \text{then error} \\
&\quad \quad \text{else if } E_1.\text{type} == E_2.\text{type} == \text{int} \\
&\quad \quad \quad \text{then int} \\
&\quad \quad \quad \text{else float} \\
\end{align*}
\]
Type Checking Expressions

\[ E \rightarrow E_1 \, [E_2] \quad \{ E.\text{type} = \text{if} \ E_1.\text{type} = \text{array}(S, \, T) \, \land \, E_2.\text{type} = \text{int} \, \text{then} \, T \, \text{else} \, \text{error} \} \]

\[ E \rightarrow *E_1 \quad \{ E.\text{type} = \text{if} \ E_1.\text{type} = \text{pointer}(T) \, \text{then} \, T \, \text{else} \, \text{error} \} \]

\[ E \rightarrow &E_1 \quad \{ E.\text{type} = \text{pointer}(E_1.\text{type}) \} \]

\[ E \rightarrow E_1(E_2) \quad \{ E.\text{type} = \text{if} \ (E_1.\text{type} = \text{fcn}(S, \, T) \, \land \, E_2.\text{type} = S, \, \text{then} \, T \, \text{else} \, \text{error} \} \]

\[ E \rightarrow (E_1, \, E_2) \quad \{ E.\text{type} = \text{tuple}(E_1.\text{type}, \, E_2.\text{type}) \} \]
Type Checking Statements

\[ S \rightarrow \text{id := E} \quad \{S.\text{type := if id.type = E.type then void else error}\} \]

\[ S \rightarrow \text{if E then } S_1 \quad \{S.\text{type := if E.type = boolean then } S_1.\text{type else error}\} \]

\[ S \rightarrow \text{while E do } S_1 \quad \{S.\text{type := if E.type = boolean then } S_1.\text{type}\} \]

\[ S \rightarrow S_1; S_2 \quad \{S.\text{type := if } S_1.\text{type = void \&\& S}_2.\text{type = void then void else error}\} \]
Equivalence of Type Expressions

Problem: When in $E_1\.type = E_2\.type$?

- We need a precise definition for type equivalence
- Interaction between type equivalence and type representation

Example:

<table>
<thead>
<tr>
<th>Type Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type vector = array [1..10] of real</code></td>
<td></td>
</tr>
<tr>
<td><code>type weight = array [1..10] of real</code></td>
<td></td>
</tr>
</tbody>
</table>

var x, y: vector; z: weight

Name Equivalence: When they have the same name.
- x, y have the same type; z has a different type.

Structural Equivalence: When they have the same structure.
- x, y, z have the same type.
Structural Equivalence

- **Definition: by Induction**
  - Same basic type \((\text{basis})\)
  - Same constructor applied to SE Type \((\text{induction step})\)
  - Same DAG Representation

- **In Practice: modifications are needed**
  - Do not include array bounds - when they are passed as parameters
  - Other applied representations \((\text{More compact})\)

- **Can be applied to: Tree/ DAG**
  - Does not check for cycles
  - Later improve it.
Algorithm Testing

Structural Equivalence

\[
\text{function sequiv}(s, t): \text{boolean}
\]
\{
\text{ if } (s \land t \text{ are of the same basic type) return true;}
\text{ if } (s = \text{array}(s_1, s_2) \land t = \text{array}(t_1, t_2))
\]
\text{ return sequiv}(s_1, t_1) \land \text{sequiv}(s_2, t_2);
\text{ if } (s = \text{tuple}(s_1, s_2) \land t = \text{tuple}(t_1, t_2))
\]
\text{ return sequiv}(s_1, t_1) \land \text{sequiv}(s_2, t_2);
\text{ if } (s = \text{fcn}(s_1, s_2) \land t = \text{fcn}(t_1, t_2))
\]
\text{ return sequiv}(s_1, t_1) \land \text{sequiv}(s_2, t_2);
\text{ if } (s = \text{pointer}(s_1) \land t = \text{pointer}(t_1))
\]
\text{ return sequiv}(s_1, t_1);
\}

Winter 2010 based on CSE 504, Stony Brook University
Recursive Types

Where: Linked Lists, Trees, etc.
How: records containing pointers to similar records
Example: type link = ↑ cell;
          cell = record info: int; next = link end

Representation:

DAG with Names

Substituting names out (cycles)
Recursive Types in C

- **C Policy**: avoid cycles in type graphs by:
  - Using structural equivalence for all types
  - Except for records → name equivalence

- **Example**:
  - `struct cell {int info; struct cell * next;}`

- **Name use**: name `cell` becomes part of the type of the record.
  - Use the acyclic representation
  - Names declared before use - except for pointers to records.

- **Cycles** - potential due to pointers in records
- **Testing** for structural equivalence stops when a record constructor is reached ~ same named record type?
Overloading Functions & Operators

- **Overloaded Symbol**: one that has different meanings depending on its context
- **Example**: Addition operator +
- **Resolving (operator identification)**: overloading is resolved when a unique meaning is determined.
- **Context**: it is not always possible to resolve overloading by looking only the arguments of a function
  - Set of possible types
  - Context (inherited attribute) necessary
Overloading Example

function "*" (i, j: integer) return complex;
function "*" (x, y: complex) return complex;

* Has the following types:
  \[ \text{fcn(tuple(integer, integer), integer)} \]
  \[ \text{fcn(tuple(integer, integer), complex)} \]
  \[ \text{fcn(tuple(complex, complex), complex)} \]

int i, j;
k = i * j;
Narrowing Types

\[ E' \rightarrow E \quad \{E'.\text{types} = E.\text{types}\} \]

\[ E.\text{unique} = \text{if } E'.\text{types} = \{t\} \text{ then } t \text{ else error}\] 

\[ E \rightarrow \text{id} \quad \{E.\text{types} = \text{lookup}(\text{id}.\text{entry})\} \]

\[ E \rightarrow E_1(E_2) \quad \{E.\text{types} = \{s' \mid \exists s \in E_2.\text{types} \text{ and } s \rightarrow s' \in E_1.\text{types}\}\} \]

\[ t = E.\text{unique} \]

\[ S = \{s \mid s \in E_2.\text{types} \text{ and } S \rightarrow t \in E_1.\text{types}\} \]

\[ E_2.\text{unique} = \text{if } S = \{s\} \text{ then } S \text{ else error} \]

\[ E_1.\text{unique} = \text{if } S = \{s\} \text{ then } S \rightarrow t \text{ else error} \]
Polymorphic Functions

- **Defn**: a piece of code (functions, operators) that can be executed with arguments of different types.

- **Examples**: Built in Operator indexing arrays, pointer manipulation

- **Why use them**: facilitate manipulation of data structures regardless of types.

- **Example ML**:
  ```ml
  fun length(lptr) = if null (lptr) then 0
                        else length(+l(lptr)) + 1
  ```
A Language for Polymorphic Functions

\[
P \rightarrow D ; E
\]

\[
D \rightarrow D ; D | id : Q
\]

\[
Q \rightarrow \forall \alpha. Q | T
\]

\[
T \rightarrow fcn (T, T) | tuple (T, T)
\]

\[
| unary (T) | (T)
\]

\[
| basic
\]

\[
| \alpha
\]

\[
E \rightarrow E (E) | E, E | id
\]
Type Variables

- Why: variables representing type expressions allow us to talk about unknown types.
  - Use Greek alphabets $\alpha$, $\beta$, $\gamma$ ...
- Application: check consistent usage of identifiers in a language that does not require identifiers to be declared before usage.
  - A type variable represents the type of an undeclared identifier.
- Type Inference Problem: Determine the type of a language constant from the way it is used.
  - We have to deal with expressions containing variables.
Examples of Type Inference

Type link ↑ cell;
Procedure mlist (lptr: link; procedure p);
{ while lptr <> null
  { p(lptr); lptr := lptr↑ .next} }

Hence: p: link → void

Function deref (p)
{ return p ↑; } 

P: β, β = pointer(α)

Hence deref: ∀ α. pointer(α) → α
Program in Polymorphic Language

deref: $\forall \alpha. \text{pointer}(\alpha) \rightarrow \alpha$
q: pointer (pointer (integer))
deref (deref( (q)))

Notation:
→ fcn
x tuple

Subscripts $i$ and $o$ distinguish between the inner and outer occurrences of deref, respectively.
Type Checking Polymorphic Functions

- Distinct occurrences of a p.f. in the same expression need not have arguments of the same type.
  - deref (deref(q))
  - Replace α with fresh variable and remove ∀(α_i, α_o)

- The notion of type equivalence changes in the presence of variables.
  - Use unification: check if s and t can be made structurally equivalent by replacing type vars by the type expression.

- We need a mechanism for recording the effect of unifying two expressions.
  - A type variable may occur in several type expressions.
Substitutions and Unification

- Substitution $S$: a mapping from type variables to type expressions.

Function `apply` ($t$: type Expr, $S$: Substitution): type Expr

```plaintext
{ if ($t$ is a basic type) return $t$;
  if ($t$ is a variable) return $S(t)$;    -- check if $t \notin S$
  if ($t$ is $t_1 \rightarrow t_2$) return (apply ($t_1$) \rightarrow apply ($t_2$)); }
```

- Instance: $S(t)$ is an instance of $t$ written $S(t) \prec t$.
  - Examples: pointer (integer) $\prec$ pointer ($\alpha$), int $\rightarrow$ real $\neq a \rightarrow a$

- Unify: $t_1 \approx t_2$ if $\exists S. S(t_1) = S(t_2)$

- Most General Unifier $S$: A substitution $S$:
  - $S(t_1) = S(t_2)$
  - $\forall S'. S'(t_1) = S'(t_2) \Rightarrow \forall t. S'(t) \prec S(t)$.
Polymorphic Type checking
Translation Scheme

\[ E \rightarrow E_1 (E_2) \quad \{ \text{p := mkleaf(newtypevar);} \]
\[ \quad \text{unify (E}_1\.\text{type, mknode(‘→’, E}_2\.\text{type, p);} \]
\[ \quad \text{E.type = p} \} \]

\[ E \rightarrow E_1, E_2 \quad \{ \text{E.type := mknode(‘x’, E}_1\.\text{type, E}_2\.\text{type);} \} \]

\[ E \rightarrow \text{id} \quad \{ \text{E.type := fresh (id.type)} \} \]

fresh (\( t \)): replaces bound variables in \( t \) by fresh variables. Returns pointer to a node representing result type.

fresh(\( \forall a.\text{pointer}(a) \rightarrow a \)) = \text{pointer}(a_1) \rightarrow a_1.

unify (m, n): unifies expressions represented by m and n.
- Side-effect: keep track of substitution
- Fail-to-unify: abort type checking.
PType Checking Example

Given: derefo (derefi (q))
q = pointer (pointer (int))

Bottom Up: fresh (\(\forall a. \text{Pointer}(a) \rightarrow a\))

\[\begin{array}{c}
\text{deref} \\
\rightarrow: 3 \\
\text{pointer}: 2 \\
\text{ao}: 1 \\
\rightarrow: 3 \\
\text{pointer}: 2 \\
\text{ao}: 1 \\
\end{array}
\]

\[\begin{array}{c}
\text{derefi} \\
\rightarrow: 6 \\
\text{pointer}: 5 \\
\text{ai}: 4 \\
\rightarrow: 6 \\
\text{pointer}: 5 \\
\text{ai}: 4 \\
\end{array}\]

\[\begin{array}{c}
\text{q} \\
\rightarrow: 9 \\
\text{pointer}: 9 \\
\text{integer}: 7 \\
\rightarrow: 6 \\
\text{pointer}: 8 \\
\text{integer}: 7 \\
\end{array}\]

\[\begin{array}{c}
\rightarrow: 3 \\
\text{pointer}: 2 \\
\text{ao}: 1 \\
\end{array}\]

\[\begin{array}{c}
\rightarrow: 6 \\
\text{pointer}: 5 \\
\text{ai}: 4 \\
\end{array}\]

\[\begin{array}{c}
\rightarrow: 6 \\
\text{pointer}: 5 \\
\beta: 8 \\
\end{array}\]

\[\begin{array}{c}
\rightarrow: 8 \\
\text{pointer}: 8 \\
\text{integer}: 7 \\
\end{array}\]