Semantic Processing
(Part 2)

All Projects Due: Friday 12-2-05, Noon

Final: Monday, December 5, 2005, 10:15-12:05
Comprehensive
Recursive Type Definitions

type MyRec is record
  f1: integer;
  f2: array of MyRec;
end;

Option #1

Option #2
Our approach is a hybrid...

```plaintext
MyRec

record

array

NamedType

id

"MyRec"

myDef

...  ...

...  ...

...  ...
```

© Harry H. Porter, 2005
Our approach is a hybrid...

```
MyRec
  └─ record
      └─ array
```

```
<table>
<thead>
<tr>
<th>NamedType</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>myDef</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>TypeDecl</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>concreteType</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

© Harry H. Porter, 2005
Testing Type Equivalence

Name Equivalence

- Stop when you get to a defined name
- Are the definitions the same (==)?

Structural Equivalence

- Test whether the type trees have the same shape.
- Graphs may contain cycles!
  - The previous algorithm (“typeEquiv”) will infinite loop.
- Need an algorithm for testing “Graph Isomorphism”

PCAT

Recursion can occur in arrays and records.

```java
type R is record
  info: integer;
  next: R;
end;

type A is array of A;
```

PCAT uses Name Equivalence
Representing Recursive Types in PCAT

type R1 is record
    info: integer;
    next: R1;
end;

<table>
<thead>
<tr>
<th>FieldDecl</th>
<th>id</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldDecl</td>
<td>id</td>
<td>next</td>
</tr>
<tr>
<td>NamedType</td>
<td>id</td>
<td>integer</td>
</tr>
<tr>
<td>NamedType</td>
<td>id</td>
<td>R1</td>
</tr>
</tbody>
</table>

© Harry H. Porter, 2005
type R1 is record
  info: integer;
  next: R1;
end;

Representing Recursive Types in PCAT
Representing Recursive Types in PCAT

type R1 is record
    info: integer;
    next: R1;
end;

© Harry H. Porter, 2005
Representing Recursive Types in PCAT

type R1 is record
    info: integer;
    next: R1;
end;
type R2 is R1;

FieldDecl "info" FieldDecl "next" null

RecordType

NamedType "R2" id

TypeDecl "R1" id
type id

NamedType "integer" id

TypeDecl "R2" id
type id

NamedType "R1" id
Representing Recursive Types in PCAT

```plaintext
Representing Recursive Types in PCAT

type R1 is record
   info: integer;
   next: R1;
end;

type R2 is R1;
```

Diagram:
- TypeDecl "R1" -> TypeDecl "R2" -> RecordType
- FieldDecl "info" -> NamedType "R2" -> myDef
- FieldDecl "next" -> NamedType "null"
- FieldDecl "info" -> NamedType "integer" -> myDef
- FieldDecl "next" -> NamedType "R1" -> myDef
- BasicTypeInteger (no fields)
Representing Recursive Types in PCAT

type R1 is record
  info: integer;
  next: R1;
end;
type R2 is R1;

FieldDecl "info" FieldDecl "next" null

RecordType

TypeDecl "R1" concreteType

TypeDecl "R2" concreteType

NamedType "R2" id myDef

NamedType "integer" id myDef

NamedType "R1" id myDef

BasicTypeInteger (no fields)

FieldDecl "info" id next type

FieldDecl "next" id next type

© Harry H. Porter, 2005
Representing Recursive Types in PCAT

type R1 is record
  info: integer;
  next: R1;
end;
type R2 is R1;

The Symbol Table goes away and we are left with just the type structures!
Type Conversions

```plaintext
var r: real;
i: integer;
... r + i ...
```

During Type-checking...
- Compiler discovers the problem
- Must insert “conversion” code

**Case 1:**
No extra code needed.
```
i = p; // e.g., pointer to integer conversion.
```

**Case 2:**
One (or a few) machine instructions
```
r = i; // e.g., integer to real conversion.
```

**Case 3:**
Will need to call an external routine
```
System.out.print ("i="+i); // int to string
```
Perhaps written in the source language (an “upcall”)

*One compiler may use all 3 techniques.*
Explicit Type Conversions

**Example (Java):**

```java
i = r;
```

Programmer must insert something to say “This is okay”:

```java
i = (int) r;
```

**Language Design Approaches:**

- “C” casting notation
  ```java
  i = (int) r;
  ```
- Function call notation
  ```java
  i = realToInt (r);
  ```
- Keyword
  ```java
  i = realToInt r;
  ```

**Compiler may insert:**

- nothing
- machine instructions
- an upcall

---

**I like this:**

- No additional syntax
- Fits easily with other user-coded data transformations
Implicit Type Conversions ("Coercions")

Example (Java, PCAT):
\[ r = i; \]

Compiler determines when a coercion must be inserted.
Rules can be complex.... Ugh!
Source of subtle errors.

Java Philosophy:
Implicit coercions are okay
when no loss of numerical accuracy.
\[ \text{byte } \rightarrow \text{short } \rightarrow \text{int } \rightarrow \text{long } \rightarrow \text{float } \rightarrow \text{double} \]

My preference:
Minimize implicit coercions
Require explicit conversions

Compiler may insert:
- nothing
- machine instructions
- an upcall
"Overloading" Functions and Operators

What does "+" mean?

- integer addition
  - 16-bit? 32-bit?
- floating-point addition
  - Single precision? Double precision?
- string concatenation
- user-defined meanings
  - e.g., complex-number addition

Compiler must "resolve" the meaning of the symbols

Will determine the operator from types of arguments

\[
i+i \rightarrow \text{integer addition} \\
d+i \rightarrow \text{floating-point addition (and double-to-int coercion)} \\
s+i \rightarrow \text{string concatenation (and int-to-string coercion)}
\]
AST Design Options

Option 1

BinaryOp

PLUS

expr1

expr2

IntegerAdd

expr1

expr2

© Harry H. Porter, 2005
AST Design Options

Option 1

Option 2

Option 3
General Principle:
It is better to ADD new information, than to CHANGE your data structures
Working with Functions

Want to say:

```plaintext
var f: int → real := ... ;
...
x := f(i);
```

Operators Syntax

- `E → E + E`
- `E → E * E`
- `E → E • E`
- `E → ...

Sometimes adjacency is used for function application

- `3N ≡ 3 * N`
- `foo N ≡ foo • N`

Parsing Issues?

- `E → E E`

The programmer can always add parentheses:

- `foo 3 = foo (3) = (foo) 3`

If the language also has tuples...

- `foo (4, 5, 6) = (foo) (4, 5, 6)`
Type Checking for Function Application

Syntax:

\[ E \rightarrow E \cdot E \]

or:

\[ E \rightarrow E E \]

or:

\[ E \rightarrow E (E) \]

Type-Checking Code (e.g., in “checkApply”)... 

\[
t_1 = \text{type of expr1}; \\
t_2 = \text{type of expr2}; \\
\text{if } t_1 \text{ has the form } “t_{\text{DOMAIN}} \rightarrow t_{\text{RANGE}}” \text{ then} \\
\quad \text{if typeEquals}(t_2, t_{\text{DOMAIN}}) \text{ then} \\
\quad \quad \text{resultType} = t_{\text{RANGE}}; \\
\quad \text{else} \\
\quad \quad \text{error}; \\
\quad \text{endIf} \\
\text{else} \\
\quad \text{error} \\
\text{endIf}
\]
**Curried Functions**

*Traditional ADD operator:*

\[
\text{add: int } \times \text{ int } \rightarrow \text{ int} \\
... \text{ add}(3,4) ... \\
\]

*Curried ADD operator:*

\[
\text{add: int } \rightarrow \text{ int } \rightarrow \text{ int} \\
... \text{ add 3 4 ...} \\
\]

Each argument is supplied individually, one at a time.

\[
\text{add 3 4 } \equiv (\text{add 3}) 4 \\
\]

Can also say:

\[
\text{f: int } \rightarrow \text{ int} \\
f = \text{add 3}; \\
... f 4 ... \\
\]

Recall: function application is Right-Associative

\[
\text{int } \rightarrow (\text{int } \rightarrow \text{ int}) \\
\]
Type Checking “apply”

“type” is a synthesized attribute

apply

add

3

type=int-

(type=int→(int→int))
“type” is a synthesized attribute

These types are “matched”
Type Checking “apply”

“type” is a synthesized attribute

This is the Result type
Type Checking “apply”

“type” is a synthesized attribute

```
add  apply
    type=(int→int)
   3  4
  type=int  type=int
```
Type Checking “apply”

“type” is a synthesized attribute

```
apply
  apply
    add
      type=int
    3
      type=int
  type=(int→int)
  4
      type=int
```
A Data Structure Example

Goal: Write a function that finds the length of a list.

```pascal
type MyRec is record
  info: integer;
  next: MyRec;
end;

procedure length (p: MyRec) : integer is
  var len: integer := 0;
  begin
    while (p <> nil) do
      len := len + 1;
      p := p.next;
    end;
    return len;
  end;
```

Traditional Languages: Each parameter must have a single, unique type.
A Data Structure Example

Goal: Write a function that finds the length of a list.

```plaintext
type MyRec is record
  info: integer;
  next: MyRec;
end;

procedure length (p:MyRec) : integer is
  var len: integer := 0;
begin
  while (p <> nil) do
    len := len + 1;
    p := p.next;
  end;
  return len;
end;
```

Traditional Languages: Each parameter must have a single, unique type.

Problem: Must write a new “length” function for every record type!!!
... Even though we didn’t access the fields particular to MyRec
Another Example: The “find” Function

Passed: • A list of T’s
       • A function “test”, which has type T→boolean

Returns: • A list of all elements that passed the “test”
i.e., a list of all elements x, for which test(x) is true

procedure find (inList: array of T;
    test: T→boolean) : array of T is
var result: array of T;
    i, j: integer := 1;
begin
    result := ... new array ...;
    while i < sizeof(inList) do
        if test(inList[i]) then
            result[j] := inList[i];
            j := j + 1;
        endIf;
        i := i + 1;
    endWhile;
    return result;
end;
This function should work for any type T.

**Goal:** Write the function once and re-use.

This problem is typical...
- Data Structure Manipulation

Want to re-use code...
- Hash Table Lookup Algorithms
- Sorting Algorithms
- B-Tree Algorithms
  etc.

...Regardless of the type of data being manipulated.
The “ML” Version of “Length”

Background:

Data Types:
- Int
- Bool
- List(...)

Lists:
- [1,3,5,7,9]
- []
- [[1,2], [5,4,3], [], [6]]

Operations on Lists:

head
- head([5,4,3]) \Rightarrow 5
- head: List(T) \rightarrow T

tail
- tail([5,4,3]) \Rightarrow [4,3]
- tail: List(T) \rightarrow List(T)

null
- null([5,4,3]) \Rightarrow false
- null: List(T) \rightarrow Bool

Type is: List(Int)

Type is: List(List(Int))

Notation: \texttt{x:T} means: “The type of x is T”
The “ML” Version of “Length”

Operations on Integers:
+  
  \[ 5 + 7 = +(5,7) \Rightarrow 12 \]
  
  \[ +: \text{Int} \times \text{Int} \rightarrow \text{Int} \]

Constants:
0: Int
1: Int
2: Int
...

```plaintext
fun length (x) = if null(x)
  then 0
  else length(tail(x)) + 1
```

New symbols introduced here:
- \( x: \text{List}(\alpha) \)
- \( \text{length}: \text{List}(\alpha) \rightarrow \text{Int} \)

No types are specified explicitly! No Declarations!
ML infers the types from the way the symbols are used!!!
Predicate Logic Refresher

Logical Operators (AND, OR, NOT, IMPLIES)
\&, |, ~, →

Predicate Symbols
P, Q, R, ...

Function and Constant Symbols
f, g, h, ... a, b, c, ...

Variables
x, y, z, ...

Quantifiers
∀, ∃

WFF: Well-Formed Formulas
∀x. ¬P(f(x)) & Q(x) → Q(x)

Precedence and Associativity:
(Quantifiers bind most loosely)
∀x. (((¬P(f(x))) & Q(x)) → Q(x))

A grammar of Predicate Logic Expressions? Sure!
Type Expressions

Basic Types
   \( \text{Int, Bool, etc.} \)

constructed Types
   \( \rightarrow, \times, \text{List()}, \text{Array()}, \text{Pointer()}, \text{etc.} \)

Type Expressions
   \( \text{List(Int \times Int)} \rightarrow \text{List(Int \rightarrow Bool)} \)

Type Variables
   \( \alpha, \beta, \gamma, \alpha_1, \alpha_2, \alpha_3, \ldots \)

Universal Quantification: \( \forall \)
   \[ \forall \alpha \ . \ \text{List}(\alpha) \rightarrow \text{List}(\alpha) \]
   (Won’t use existential quantifier, \( \exists \))

Remember: \( \forall \) binds loosely
   \[ \forall \alpha \ . \ (\text{List}(\alpha) \rightarrow \text{List}(\alpha)) \]
   “For any type \( \alpha \), a function that maps lists of \( \alpha \)'s to lists of \( \alpha \)'s.”
Type Expressions

Okay to change variables (as long as you do it consistently)...

\[ \forall \alpha . \ Pointer(\alpha) \rightarrow \text{Boolean} \]

\[ \equiv \forall \beta . \ Pointer(\beta) \rightarrow \text{Boolean} \]

What do we mean by that?

Same as for predicate logic...

• Can’t change \( \alpha \) to a variable name already in use elsewhere
• Must change all occurrences of \( \alpha \) to the same variable

We will use only universal quantification (“for all”, \( \forall \))

Will not use \( \exists \)

Okay to just drop the \( \forall \) quantifiers.

\[ \forall \alpha . \forall \beta . \ (\text{List}(\alpha) \times (\alpha \rightarrow \beta)) \rightarrow \text{List}(\beta) \]

\[ \equiv \ (\text{List}(\alpha) \times (\alpha \rightarrow \beta)) \rightarrow \text{List}(\beta) \]

\[ \equiv \ (\text{List}(\beta) \times (\beta \rightarrow \gamma)) \rightarrow \text{List}(\gamma) \]
Practice

Given:
   x: Int
   y: Int→Boolean

What is the type of (x, y)?
Practice

Given:
    x: Int
    y: Int → Boolean

What is the type of \((x, y)\)?
    \((x, y)\): Int \times (Int → Boolean)
Practice

Given:

\[ x: \text{Int} \]
\[ y: \text{Int} \rightarrow \text{Boolean} \]

**What is the type of \((x, y)\)?**

\( (x, y): \text{Int} \times (\text{Int} \rightarrow \text{Boolean}) \)

Given:

\[ f: \text{List(\(\alpha\)} \rightarrow \text{List(\(\alpha\)} \]
\[ z: \text{List(Int)} \]

**What is the type of \(f(z)\)?**
Practice

*Given:*
- \( x: \text{Int} \)
- \( y: \text{Int} \rightarrow \text{Boolean} \)

**What is the type of \((x, y)\)?**
- \((x, y): \text{Int} \times (\text{Int} \rightarrow \text{Boolean})\)

*Given:*
- \( f: \text{List}(\alpha) \rightarrow \text{List}(\alpha) \)
- \( z: \text{List(\text{Int})} \)

**What is the type of \( f(z) \)?**
- \( f(z): \text{List(\text{Int})} \)
Practice

Given:

\[ x: \text{Int} \]
\[ y: \text{Int} \rightarrow \text{Boolean} \]

What is the type of \((x, y)\)?

\[(x, y): \text{Int} \times (\text{Int} \rightarrow \text{Boolean})\]

Given:

\[ f: \text{List}(\alpha) \rightarrow \text{List}(\alpha) \]
\[ z: \text{List(Int)} \]

What is the type of \(f(z)\)?

\[ f(z): \text{List(Int)} \]

What is going on here?

We “matched” \(\alpha\) to \(\text{Int}\)

We used a “Substitution”

\[ \alpha = \text{Int} \]

What do we mean by “matched”???
Practice

Given:

\[ x: \text{Int} \]
\[ y: \text{Int} \rightarrow \text{Boolean} \]

What is the type of \((x, y)\)?

\[(x, y): \text{Int} \times (\text{Int} \rightarrow \text{Boolean})\]

Given:

\[ f: \text{List}(\alpha) \rightarrow \text{List}(\alpha) \]
\[ z: \text{List}(\text{Int}) \]

What is the type of \(f(z)\)?

\[ f(z): \text{List}(\text{Int}) \]

What is going on here?
We “matched” \(\alpha\) to \(\text{Int}\)

We used a “Substitution”
\[ \alpha = \text{Int} \]

What do we mean by “matched”???

**UNIFICATION!**
Unification

**Given:** Two [type] expressions

**Goal:** Try to make them equal

**Using:** Consistent substitutions for any [type] variables in them

**Result:**
- Success
  - plus the variable substitution that was used
- Failure
A Language With Polymorphic Functions

\[
P \rightarrow D ; E \\
D \rightarrow D ; D \\
\quad \rightarrow \text{id} ; Q \\
Q \rightarrow \forall \ id . \ Q \\
\quad \rightarrow T \\
T \rightarrow T \rightarrow T \\
\quad \rightarrow T \times T \\
\quad \rightarrow \text{List}(T) \\
\quad \rightarrow \text{Int} \\
\quad \rightarrow \text{Bool} \\
\quad \rightarrow \text{id} \\
\quad \rightarrow (T) \\
E \rightarrow \text{id} \\
\quad \rightarrow \text{int} \\
\quad \rightarrow EE \\
\quad \rightarrow (E, E) \\
\quad \rightarrow (E) \\
\]

Quantified Type Expressions
Unquantified Type Expressions
Type Variables
Function Apply
Tuple Construction
Grouping
A Language With Polymorphic Functions

\[
P \rightarrow D ; E \\
D \rightarrow D ; D \\
\quad \rightarrow \text{id} : Q \\
Q \rightarrow \forall \text{id} . Q \\
\quad \rightarrow T \\
T \rightarrow T \rightarrow T \\
\quad \rightarrow T \times T \\
\quad \rightarrow \text{List} (T) \\
\quad \rightarrow \text{Int} \\
\quad \rightarrow \text{Bool} \\
\quad \rightarrow \text{id} \\
\quad \rightarrow (T) \\
E \rightarrow \text{id} \\
\quad \rightarrow \text{int} \\
\quad \rightarrow E E \\
\quad \rightarrow (E, E) \\
\quad \rightarrow (E) \\
\]

Examples of Expressions:

- 123
- (x)
- foo(x)
- find(test, myList)
- add(3, 4)
A Language With Polymorphic Functions

P → D ; E
D → D ; D
    → id : Q
Q → ∀ id . Q
    → T
T → T “→” T
    → T × T
    → List ( T )
    → Int
    → Bool
    → id
    → ( T )
E → id
    → int
    → E E
    → ( E , E )
    → ( E )

Examples of Types:

Int → Bool
Bool × (Int → Bool)
α
α × (α → Bool)
((β → Bool) × List(β)) → List(β)

A Type Variable (id)
A Language With Polymorphic Functions

P → D ; E
D → D ; D
  → id : Q
Q → ∀ id . Q
  → T
T → T “→” T
  → T × T
  → List (T)
  → Int
  → Bool
  → id
  → (T)
E → id
  → int
  → E E
  → (E, E)
  → (E)

Examples of Quatified Types:

Int → Bool
∀ α . (α → Bool)
∀ β . (((β→Bool) × List(β))→List(β))
A Language With Polymorphic Functions

<table>
<thead>
<tr>
<th>P</th>
<th>→ D ; E</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>→ D ; D</td>
</tr>
<tr>
<td></td>
<td>→ id : Q</td>
</tr>
<tr>
<td>Q</td>
<td>→ ∀ id . Q</td>
</tr>
<tr>
<td></td>
<td>→ T</td>
</tr>
<tr>
<td>T</td>
<td>→ T “→” T</td>
</tr>
<tr>
<td></td>
<td>→ T × T</td>
</tr>
<tr>
<td></td>
<td>→ List(T)</td>
</tr>
<tr>
<td></td>
<td>→ Int</td>
</tr>
<tr>
<td></td>
<td>→ Bool</td>
</tr>
<tr>
<td></td>
<td>→ id</td>
</tr>
<tr>
<td></td>
<td>→ (T)</td>
</tr>
<tr>
<td>E</td>
<td>→ id</td>
</tr>
<tr>
<td></td>
<td>→ int</td>
</tr>
<tr>
<td></td>
<td>→ E E</td>
</tr>
<tr>
<td></td>
<td>→ (E, E)</td>
</tr>
<tr>
<td></td>
<td>→ (E)</td>
</tr>
</tbody>
</table>

Examples of Declarations:

- i: Int;
- myList: List(Int);
- test: ∀ α . (α → Bool);
- find: ∀ β . (((β → Bool) × List(β)) → List(β))
A Language With Polymorphic Functions

An Example Program:

\[
\begin{align*}
\text{myList} & : \text{List(Int)}; \\
\text{test} & : \forall \alpha . (\alpha \to \text{Bool}) ; \\
\text{find} & : \forall \beta . ((\beta \to \text{Bool}) \times \text{List} (\beta)) \to \text{List} (\beta) ; \\
\text{find} (\text{test}, \text{myList}) & \\
\end{align*}
\]

GOAL: Type-check this expression given these typings!
Parse Tree (Annotated with Synthesized Types)

Expression:
find (test, myList)
Add known typing info:

\[
\begin{align*}
\text{myList: } & \text{List(Int)}; \\
\text{test: } & \forall \alpha . (\alpha \rightarrow \text{Bool}); \\
\text{find: } & \forall \beta . (((\beta \rightarrow \text{Bool}) \times \text{List}(\beta)) \rightarrow \text{List}(\beta));
\end{align*}
\]
Parse Tree (Annotated with Synthesized Types)

Add known typing info:

- myList: List(Int);
- test: ∀ α . (α → Bool);
- find: ∀ β . (((β → Bool) × List(β)) → List(β));
Parse Tree (Annotated with Synthesized Types)

Tuple Node:
Match $\gamma$ to $(\alpha \rightarrow \text{Bool}) \times \text{List(Int)}$
Semantics - Part 2

Parse Tree (Annotated with Synthesized Types)

Tuple Node:
Match \( \gamma \) to \((\alpha \rightarrow \text{Bool}) \times \text{List(Int)}\)

Conclude:
\( \gamma = (\alpha \rightarrow \text{Bool}) \times \text{List(Int)} \)
Parse Tree (Annotated with Synthesized Types)

Apply Node:
Match
(\beta \rightarrow \text{Bool}) \times \text{List}(\beta)
(\alpha \rightarrow \text{Bool}) \times \text{List}(\text{Int})

Conclude:
\beta = \text{Int}
\alpha = \beta = \text{Int}
Parse Tree (Annotated with Synthesized Types)

**Apply Node:**

**Match**

\[(\beta \rightarrow \text{Bool}) \times \text{List}(\beta)\]

\[(\alpha \rightarrow \text{Bool}) \times \text{List}(\text{Int})\]

**Conclude:**

\[\beta = \text{Int}\]

\[\alpha = \beta = \text{Int}\]
Parse Tree (Annotated with Synthesized Types)

**Apply Node:**
- Match: List(Int)
- \( \delta \)

**Conclude:**
- \( \delta = \text{List(Int)} \)
Semantics - Part 2

Parse Tree (Annotated with Synthesized Types)

Apply Node: Match
List(Int)
δ

Conclude: δ = List(Int)
**Parse Tree (Annotated with Synthesized Types)**

- **apply**
  - type = List(Int)
- **find**
  - type = ((Int → Bool) × List(Int)) → List(Int)
- **tuple**
  - type = (Int → Bool) × List(Int)
- **test**
  - type = Int → Bool
- **myList**
  - type = List(Int)

**Results:**
- $\alpha = \text{Int}$
- $\beta = \text{Int}$
- $\delta = \text{List(Int)}$
- $\gamma = (\text{Int} \rightarrow \text{Bool}) \times \text{List(Int)}$
Unification of Two Expressions

Example:
\[ t_1 = \alpha \times \text{Int} \]
\[ t_2 = \text{List}(\beta) \times \gamma \]

Is there a substitution that makes \( t_1 = t_2 \)?

“\( t_1 \) unifies with \( t_2 \)” if and only if there is a substitution \( S \) such that
\[ S(t_1) = S(t_2) \]

Here is a substitution that makes \( t_1 = t_2 \):
\[ \alpha \leftarrow \text{List}(\beta) \]
\[ \gamma \leftarrow \text{Int} \]

Other notation for substitutions:
\[ \{ \alpha/\text{List}(\beta), \gamma/\text{Int} \} \]
Most General Unifier

There may be several substitutions. Some are more general than others.

Example:
\[ t_1 = \alpha \times \text{Int} \]
\[ t_2 = \text{List}(\beta) \times \gamma \]

Unifying Substitution #1:
\[ \alpha \leftarrow \text{List(\text{List(\text{List(Bool))}})} \]
\[ \beta \leftarrow \text{List(\text{List(Bool))}} \]
\[ \gamma \leftarrow \text{Int} \]

Unifying Substitution #2:
\[ \alpha \leftarrow \text{List(Bool} \times \delta) \]
\[ \beta \leftarrow \text{Bool} \times \delta \]
\[ \gamma \leftarrow \text{Int} \]

Unifying Substitution #3:
\[ \alpha \leftarrow \text{List}(\beta) \]
\[ \gamma \leftarrow \text{Int} \]

This is the “Most General Unifier”
Unifying Two Terms / Types

Unify these two terms:
\[ f(g(a, X), Y) \]
\[ f(Z, Z) \]
Unification makes the terms identical.
Unifying Two Terms / Types

Unify these two terms:
\[ f(g(a,X), Y) \]
\[ f(Z, Z) \]
Unification makes the terms identical.

The substitution:
\[ Y \leftarrow Z \]
\[ Z \leftarrow g(a,X) \]
Unifying Two Terms / Types

Unify these two terms:
\[ f(g(a, X), Y) \]
\[ f(Z, Z) \]

Unification makes the terms identical.

**The substitution:**
\[ Y \leftarrow Z \]
\[ Z \leftarrow g(a, X) \]

**Merge the trees into one!**
Unifying Two Terms / Types

Unify these two terms:
\[ f(g(a, X), Y) \quad f(Z, Z) \quad \rightarrow \quad f(g(a, X), g(a, X)) \]

Unification makes the terms identical.

*The substitution:*
\[ Y \leftarrow Z \]
\[ Z \leftarrow g(a, X) \]

*Merge the trees into one!*

```
   f
  /\  
 /   
/     
Y=Z=g
 
/ \   
/   
/    
/     
/      
av X
```
Unifying Two Terms / Types

Unify these two terms:
\[ f(g(a,X), Y) \]
\[ f(Z, Z) \]
\[ f(g(a,X), g(a,X)) \]

Unification makes the terms identical.

The substitution:
\[ Y \leftarrow Z \]
\[ Z \leftarrow g(a,X) \]

Merge the trees into one!

Same with unifying types!
\[ (\text{Int} \times \text{List}(X)) \times Y \]
\[ Z \times Z \]
Representing Types With Trees

- **Function**
  - domainType
  - rangeType

- **CrossProd**
  - type1
  - type2

- **Integer**
  - (no fields)

- **TypeVar**
  - id “T1” or “α₁”

*Same for other basic and constructed types Real, Bool, List(T), etc.*
Merging Sets

**Approach:** Will work with sets of nodes.
Each set will have a “representative” node.

**Goal:** Merge two sets of nodes into a single set.
When two sets are merged (the “union” operation)... make one representative point to the other!
Merging Sets

**Approach:** Will work with sets of nodes. Each set will have a “representative” node.

**Goal:** Merge two sets of nodes into a single set. When two sets are merged (the “union” operation)... make one representative point to the other!

The new representative

The combined Set

The ‘set’ pointers
Representing Type Expressions

- **Function**
  - domainType
  - rangeType
  - set

- **CrossProd**
  - type1
  - type2
  - set

- **Integer**
  - set

- **TypeVar**
  - id "T1" or "α1"
  - set

The “set” pointers will point toward the representative node.
(Initialized to null.)
Merging Sets

Find(p) → ptr
Given a pointer to a node, return a pointer to the representative of the set containing p.

Just chase the “set” pointers as far as possible.

Union(p,q)
Merge the set containing p with the set containing q.

Do this by making the representative of one of the sets point to the representative of the other set. If one representative is a variable node and the other is not, always use the non-variable node as the representative of the combined, merged sets. In other words, make the variable node point to the other node.
The Unification Algorithm

function Unify \( (s', t'): \text{Node} \) returns bool
\[
\begin{align*}
s &= \text{Find}(s') \\
t &= \text{Find}(t') \\
\text{if } s &= t \text{ then} \\
& \quad \text{return } true \\
\text{elsif } s \text{ and } t \text{ both point to INTEGER nodes} \text{ then} \\
& \quad \text{return } true \\
\text{elsif } s \text{ or } t \text{ points to a VARIABLE node} \text{ then} \\
& \quad \text{Union}(s, t) \\
\text{elsif } s \text{ points to a node } \text{FUNCTION}(s_1, s_2) \text{ and} \\
& \quad t \text{ points to a node } \text{FUNCTION}(t_1, t_2) \text{ then} \\
& \quad \text{Union}(s, t) \\
& \quad \text{return } \text{Unify}(s_1, t_1) \text{ and } \text{Unify}(s_2, t_2) \\
\text{elsif } s \text{ points to a node } \text{CROSSPROD}(s_1, s_2) \text{ and} \\
& \quad t \text{ points to a node } \text{CROSSPROD}(t_1, t_2) \text{ then} \\
& \quad \text{Union}(s, t) \\
& \quad \text{return } \text{Unify}(s_1, t_1) \text{ and } \text{Unify}(s_2, t_2) \\
\text{elsif } \ldots \\
\text{else} \\
& \quad \text{return } false \\
\text{endIf}
\end{align*}
\]

Etc., for other type constructors and basic type nodes
Example: Unify...

\[ t_1 = \alpha \times \text{Integer} \]

\[ t_2 = \text{List}(\beta) \times \gamma \]
Example: Unify...

\[ t_1 = \alpha \times \text{Integer} \]
\[ t_2 = \text{List}(\beta) \times \gamma \]
Example: Unify...

\[ t_1 = \alpha \times \text{Integer} \]

\[ t_2 = \text{List}(\beta) \times \gamma \]
Example: Unify...

\[ t_1 = \alpha \times \text{Integer} \]
\[ t_2 = \text{List}(\beta) \times \gamma \]
**Example: Unify...**

\[ t_1 = \alpha \times \text{Integer} \]
\[ t_2 = \text{List}(\beta) \times \gamma \]

**Recovering the Substitution:**

\[ \alpha \leftarrow \text{List}(\beta) \]
\[ \gamma \leftarrow \text{Integer} \]
Type-Checking with an Attribute Grammar

- **Lookup**(string) → type
  Lookup a name in the symbol table and return its type.
- **Fresh**(type) → type
  Make a copy of the type tree.
  Replace all variables (consistently) with new, never-seen-before variables.
- **MakeIntNode**() → type
  Make a new leaf node to represent the “Int” type.
- **MakeVarNode**() → type
  Create a new variable node and return it.
- **MakeFunctionNode**(type₁, type₂) → type
  Create a new “Function” node and return it.
  Fill in its domain and range types.
- **MakeCrossNode**(type₁, type₂) → type
  Create a new “Cross Product” node and return it.
  Fill in the types of its components.
- **Unify**(type₁, type₂) → bool
  Unify the two type trees and return true if success.
  Modify the type trees to perform the substitutions.
Type-Checking with an Attribute Grammar

E → id  
\[ \text{E.type} = \text{Fresh(Lookup(id.svalue))}; \]

E → int  
\[ \text{E.type} = \text{MakeIntNode}(); \]

\[ \text{E}_0 \rightarrow \text{E}_1 \text{E}_2 \]
\[ p = \text{MakeVarNode}(); \]
\[ f = \text{MakeFunctionNode}(\text{E}_2.\text{type}, p); \]
\[ \text{Unify(E}_1.\text{type, f);} \]
\[ \text{E}_0.\text{type} = p; \]

\[ \text{E}_0 \rightarrow (\text{E}_1, \text{E}_2) \]
\[ \text{E}_0.\text{type} = \text{MakeCrossNode(E}_1.\text{type,} \]
\[ \text{E}_2.\text{type}); \]

\[ \text{E}_0 \rightarrow (\text{E}_1) \]
\[ \text{E}_0.\text{type} = \text{E}_1.\text{type} ; \]
Conclusion

Theoretical Approaches:

- Regular Expressions and Finite Automata
- Context-Free Grammars and Parsing Algorithms
- Attribute Grammars
- Type Theory
  - Function Types
  - Type Expressions
  - Unification Algorithm

Make it possible to parse and check complex, high-level programming languages!

Would not be possible without these theoretical underpinnings!

The Next Step?

Generate Target Code and Execute the Program!