Recursive Type Definitions

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

Option #1

Option #2
Our approach is a hybrid...

MyRec

record

array

NamedType

"MyRec"

id

myDef

TypeDecl

"MyRec"

concreteType
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.

```pascal
type R is record
    info: integer;
    next: R;
end;
type A is array of A;
```

PCAT uses Name Equivalence

Representing Recursive Types in PCAT

```pascal
type R1 is record
    info: integer;
    next: R1;
end;
```
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
TypeDecl "R1" id
TypeDecl "R2" id
NamedType "R2" id
myDef
NamedType "integer" id
myDef
NamedType "R1" id
myDef

BasicTypeInteger
(no fields)

... ...

© Harry H. Porter, 2005
The Symbol Table goes away and we are left with just the type structures!
Type Conversions

```
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):**
```
i = r;
```
Programmer must insert something to say “This is okay”:
```
i = (int) r;
```

**Language Design Approaches:**
- “C” casting notation
  ```
i = (int) r;
```
- Function call notation
  ```
i = realToInt (r);
```
- Keyword
  ```
i = realToInt r;
```

**I like this:**
- No additional syntax
- Fits easily with other user-coded data transformations

Compiler may insert:
- nothing
- machine instructions
- an upcall
**Implicit Type Conversions (“Coercions”)**

*Example (Java, PCAT):*

```java
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.

```plaintext
byte → short → int → long → float → double
```

Compiler may insert:
- nothing
- machine instructions
- an upcall

*My preference:*
Minimize implicit coercions
Require explicit conversions

---

**“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` → integer addition
- `d+i` → floating-point addition (and double-to-int coercion)
- `s+i` → string concatenation (and int-to-string coercion)
General Principle: It is better to ADD new information, than to CHANGE your data structures.
Working with Functions

Want to say:
```pascal
case f: int -> real := ... ;
...
x := f(i);
```

Operators Syntax
- 
  - 
- Sometimes adjacency is used for function application
  ```pascal
  3N = 3 * N
  foo N = foo * N
  ```

Parsing Issues?
```pascal
E -> E E
```

The programmer can always add parentheses:
```pascal
foo 3 = foo (3) = (foo) 3
```

If the language also has tuples...
```pascal
foo(4,5,6) = (foo)(4,5,6)
```

Type Checking for Function Application

Syntax:
```pascal
E -> E * E
```

Type-Checking Code (e.g., in “checkApply”)...
```pascal
t1 = type of expr1;
t2 = type of expr2;
if t1 has the form “t_DOMAIN -> t_RANGE” then
  if typeEquals(t2, t_DOMAIN) then
    resultType = t_RANGE;
  else
    error;
  endIf
else
  error;
endIf
```

© Harry H. Porter, 2005
Curried Functions

Traditional ADD operator:
add: int x int → int
... add(3,4) ...

Curried ADD operator:
add: int → int → int
... add 3 4 ...

Each argument is supplied individually, one at a time.
add 3 4 = (add 3) 4

Can also say:
f: int → int
f = add 3;
... f 4 ...

Recall: function application is Right-Associative
int → (int → int)

Type Checking “apply”

“type” is a synthesized attribute
Type Checking “apply”

“type” is a synthesized attribute

These types are “matched”

This is the Result type
Type Checking “apply”

“type” is a synthesized attribute

27

28
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

© Harry H. Porter, 2005
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

```plaintext
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]) ⇒ 5
- head: List(T) → T

tail
- tail([5,4,3]) ⇒ [4,3]
- tail: List(T) → List(T)

null
- null([5,4,3]) ⇒ false
- null: List(T) → Bool

Operations on Integers:

+ 5 + 7 = +(5,7) ⇒ 12
- +: Int×Int → Int

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

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

New symbols introduced here:
- x: List(α)
- length: List(α) → 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
  Int, Bool, etc.

Constructed Types
  →, ×, List(), Array(), Pointer(), etc.

Type Expressions
  List(Int × Int)  →  List(Int → Bool)

Type Variables
  α, β, γ, α₁, α₂, α₃, ...

Universal Quantification: ∀
  ∀α . List(α) → List(α)
  (Won’t use existential quantifier, ∃)

Remember: ∀ binds loosely
  ∀α . (List(α) → List(α))
  “For any type α, a function that maps lists of α’s to lists of α’s.”
Type Expressions

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

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

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.

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

Practice

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

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

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

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

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

---

Practice

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

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 \(f(z)\)?
\(f(z): \text{List}(\text{Int})\)

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

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 → Bool)
- ((β → Bool) × List(β)) → List(β)

A Type Variable (id)

Examples of Quatified Types:

- Int → Bool
- ∀ α . (α → Bool)
- ∀ β . ((β → Bool) × List(β)) → List(β)
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 Declarations:
i: Int;
myList: List(Int);
test: ∀ α . (α → Bool);
find: ∀ β :((β→Bool) × List(β))→List(β))

An Example Program:
myList: List(Int);
test: ∀ α . (α → Bool);
find: ∀ β :((β→Bool) × List(β))→List(β));
find (test, myList)

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

Expression:
find (test, myList)

Add known typing info:

myList: List(Int);
test: ∀ α . (α → Bool);
find: ∀ β . (((β→Bool) × List(β))→List(β));
Add known typing info:

- myList: List(Int);
- test: ∀ α . (α → Bool);
- find: ∀ β . (((β → Bool) × List(β)) → List(β));

Tuple Node:
Match γ to (α → Bool) × List(Int)
**Parse Tree (Annotated with Synthesized Types)**

- **tuple**
  - type: $(\alpha \rightarrow \text{Bool}) \times \text{List(\beta)}$
  - $(\beta \rightarrow \text{Bool}) \times \text{List(\beta)} \rightarrow \text{List(\beta)}$

- **find**
  - type: $((\beta \rightarrow \text{Bool}) \times \text{List(\beta)}) \rightarrow \text{List(\beta)}$

- **myList**
  - type: List(\text{Int})

- **test**
  - type: $\alpha \rightarrow \text{Bool}$

**Tuple Node:**

- Match $\gamma$ to $(\alpha \rightarrow \text{Bool}) \times \text{List(\text{Int})}$
- Conclude:
  - $\gamma = (\alpha \rightarrow \text{Bool}) \times \text{List(\text{Int})}$

**Apply Node:**

- Match:
  - $(\beta \rightarrow \text{Bool}) \times \text{List(\beta)}$
  - $(\alpha \rightarrow \text{Bool}) \times \text{List(\text{Int})}$
- Conclude:
  - $\beta = \text{Int}$
  - $\alpha = \beta = \text{Int}$

© Harry H. Porter, 2005
Parse Tree (Annotated with Synthesized Types)

Apply Node:
Match

\[
\begin{align*}
\beta & \rightarrow \text{Bool} \times \text{List}(\beta) \\
\alpha & \rightarrow \text{Bool} \times \text{List}(\text{Int})
\end{align*}
\]

Conclude:

\[
\begin{align*}
\beta & = \text{Int} \\
\alpha & = \beta = \text{Int}
\end{align*}
\]

© Harry H. Porter, 2005
Parse Tree (Annotated with Synthesized Types)

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

Apply Node:
Match
  List(Int)
  δ
Conclude:
  δ = List(Int)

Results:
  α = Int
  β = Int
  δ = List(Int)
  γ = (Int→Bool) × List(Int)

© Harry H. Porter, 2005
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:
\[ \alpha \leftarrow \text{List}(\text{List}(\text{List}(\text{Bool}))) \]
\[ \beta \leftarrow \text{List}(\text{List}(\text{Bool})) \]
\[ \gamma \leftarrow \text{Int} \]

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

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.

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!**

© Harry H. Porter, 2005
Unifying Two Terms / Types

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

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

*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!
**Representing Type Expressions**

- **Function**
- **CrossProd**
- **Integer**
- **TypeVar**

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

---

**Merging Sets**

- **Find(p)**
  
  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': Node) returns bool
s = Find(s')
t = Find(t')
if s == t then
  return true
elseIf s and t both point to INTEGER nodes then
  return true
elseIf s or t points to a VARIABLE node then
  Union(s,t)
elseIf s points to a node FUNCTION(s₁,s₂) and
t points to a node FUNCTION(t₁,t₂) then
  Union(s,t)
  return Unify(s₁,t₁) and Unify(s₂,t₂)
elseIf s points to a node CROSSPROD(s₁,s₂) and
t points to a node CROSSPROD(t₁,t₂) then
  Union(s,t)
  return Unify(s₁,t₁) and Unify(s₂,t₂)
elseIf ... 
else
  return false
endIf

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...

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

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.

E → id   E.type = Fresh(Lookup(id.svalue));

E → int   E.type = MakeIntNode();

E₀ → E₁ E₂   p = MakeVarNode();
               f = MakeFunctionNode(E₂.type, p);
               Unify(E₁.type, f);
               E₀.type = p;

E₀ → (E₁, E₂)   E₀.type = MakeCrossNode(E₁.type, E₂.type);

E₀ → (E₁)   E₀.type = E₁.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!*