Roadmap

- Second project has been posted: Type reconstruction for TINY language. Due date: Tuesday, November 29
- Third project will be posted on Wednesday, November 30.
- Thanksgiving recess: no class on Tuesday, November 22.

  Instead **2nd Project meeting**: What is a good time? Monday, Tuesday, or Wednesday next week?
- Homework problem set 4 will be posted by Friday
- Last lecture: Thursday, December 14; overall, classes end Friday, December 15.

Types

type:

A set of values and meaningful operations on them.

Types provide semantic "sanity checks" (consistency checks). Types help identify:

- errors, if an operator is applied to an incompatible operand.
  - dereferencing of a non-pointer
  - adding a function name to something
  - incorrect number of parameters to a procedure
  - ...
- which operation to use for overloaded names and operators, or what type coercion to use (e.g.: $3.0 + 1$)
- identification of polymorphic functions. Such functions can be executed with arguments of several types (e.g.: list of objects of unknown type α)

Type systems

Each language construct (operator, expression, statement, ...) has a type

basic types: integer, real, character, ...

constructed types: arrays, records, sets, pointers, functions

A type system is a collection of *rules* for assigning *type expressions* to operators, expressions, ... in the program. Type systems are language dependent.

A type checker implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system. The type checker "computes" or "reconstructs" type expressions.

Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer (Pascal definition).
  Rule for $+$ (analogue rules for $-$ and $*$):

  $$\frac{E \vdash e_1 : int \quad E \vdash e_2 : int}{E \vdash (e_1 + e_2) : int}$$

  where E is a *type environment* that maps constants and variables to their types.

  In combination with the following axiom template in the type system $\{c : \alpha\} \vdash c : \alpha$
  we can now infer that $(2 + 3)$ is of type integer represented by type expression int:

  $$\frac{E \vdash 2 : int \quad E \vdash 3 : int}{E \vdash (2 + 3) : int}$$

  where $E = \{2 : int, 3 : int\}$.

  In general, type deduction proofs work bottom up.

Example type rules

- The result of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type "foo", then the type of the result is a "pointer to foo". (C and C++ definition)

$$\frac{E \vdash e : \alpha}{E \vdash \&e : \mathit{pointer}(\alpha)}$$

- Two expressions can only be compared if they have the same types. The result is of type boolean.

$$\frac{E \vdash e_1 : \alpha \quad E \vdash e_2 : \alpha}{E \vdash (e_1 = e_2) : \mathit{boolean}}$$

In the examples, *integer* and α are *type expressions*.

Type expressions *inductive definition*

1. A basic type is a type expression. A special basic type, *typeError* will signal an error. A basic type *void* denotes an untyped statement.

2. Since type expressions may be named, a type name is a type expression.

3. Type expressions may contain variables whose values are type expressions (e.g.: useful for languages without type declarations, or polymorphism).

4. A *type constructor* applied to type expressions is a type expression. Examples:

   (a) arrays

   (b) cartesian products

   (c) records

   (d) pointers

   (e) functions

Type checking

The purpose of type checking is to prevent type errors. Type errors can occur during the execution of a program, for instance, when a function is applied with an argument of the wrong type or if a variable with a non–function type is called.

"How much" type checking is done is part of the definition of the programming language.

Type checking can be performed

- at compile–time (only)
- at compile–time and program execution time, or
- at program execution time (only).

Type checking

- A *strongly typed* language guarantees that the compiler will accept only programs that execute without type errors, i.e., the compiler ensures that no type error will *remain undetected.*
  Strongly typed languages can use static and dynamic type checking.
  Standard example: array bounds checking.

  table: array[0. . .255] of char;
  i: integer

  table[i] cannot be guaranteed at compile time to fall in the range of 0 to 255

- In principle, type checking can be done entirely dynamically. However, this requires *type tags* for each value and each function application has to check for matching tags at run–time.

Type checking — discussion

compile–time:

+ points out time errors early
+ no overhead at program execution time (run–time)
– cannot always be done
? parts of the program that may never be executed are still checked (too restrictive)

program execution time:

+ allows more flexibility (e.g.: type may change depending on the use of a variable)
– overhead in terms of space and time
? part of program that are not executed are not checked
  + rapid prototyping
  – harder to debug

Goal: Strongly typed languages with as much type checking as possible at compile–time

A simple type checker

Using a syntax-directed translation scheme (synthesized attributes only), we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

P ::= D ; E
D ::= D ; D | id: T
T ::= char | integer | array [num] of T | $\uparrow$ T
E ::= literal | num | id | E mod E | E[E] | E $\uparrow$

- Basic types *char*, *int*, *typeError*
- assume all arrays start at 1, *e.g.*,
  array [256] of char
  results in the type expression *array*(1...256,*char*)
- $\uparrow$ builds a pointer type, so $\uparrow$ integer
  results in the type expression *pointer(int)*

A simple type checker (cont.)

Partial translation scheme for the type system

D	::=	id: T	$\{$ *addtype*(id.*entry*, *T.type*) $\}$
T	::=	char	$\{$ $T.type \leftarrow char$ $\}$
T	::=	integer	$\{$ $T.type \leftarrow int$ $\}$
T	::=	$\uparrow T_1$	$\{$ $T.type \leftarrow pointer(T_1.type)$ $\}$
T	::=	array [num] of T	$\{$ $T.type \leftarrow$ *array*(*1...*num.*val*, $T_1.type$) $\}$

These rules save the type of identifier id in the symbol table.

A simple type checker (cont.)

Type checking of expressions

E	::=	literal	$\{\ E.type \leftarrow char\ \}$
E	::=	num	$\{\ E.type \leftarrow int\ \}$
E	::=	id	$\{\ E.type \leftarrow lookup(\textsf{id}.entry)\ \}$
E	::=	E_1 mod E_2	$\{\ E.type \leftarrow$ if $E_1.type = int$ and $E_2.type = int$ then int else $typeError\ \}$
E	::=	$E_1[E_2]$	$\{\ E.type \leftarrow$ if $E_2.type = int$ and $E_1.type = array(s,t)$ then t else $typeError\ \}$
E	::=	$E_1 \uparrow$	$\{\ E.type \leftarrow$ if $E_1.type = pointer(t)$ then t else $typeError\ \}$

Is our example language strongly typed?

Type checking statements

Statements do not typically have values, therefore we assign them the type *void*. If an error is detected within the statement, it gets type *typeError*.

S ::=	id ← E	{ $S.type \leftarrow$ if id.$type = E.type$ then $void$ else $typeError$ }
S ::=	if E then S_1	{ $S.type \leftarrow$ if $E.type = boolean$ then S_1.type else $typeError$ }
S ::=	while E do S_1	{ $S.type \leftarrow$ if $E.type = boolean$ then S_1.type else $typeError$ }
S ::=	S_1 ; S_2	{ $S.type \leftarrow$ if $S_1.type = void$ and $S_2.type = void$ then $void$ else $typeError$ }

Type checking functions

We add two new productions to the grammar to represent function declarations and applications

T	::=	T "→" T	declaration
E	::=	E (E)	application

To capture the argument and return type, we use

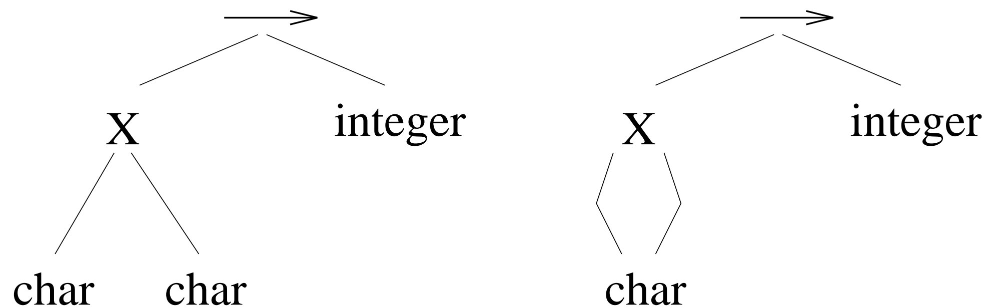
$$T ::= T_1 \text{ `}\rightarrow\text{' } T_2 \;\; \{\; T.type \leftarrow (T_1.type \rightarrow T_2.type) \;\}$$

$$E ::= E_1 \; (\; E_2 \;) \;\; \{\; E.type \leftarrow \textsf{if } E_2.type = s \textsf{ and } E_1.type = s \rightarrow t \textsf{ then } t \textsf{ else } typeError \;\}$$

Representation of type expressions

A convenient way to represent a type expression is to use a graph.

Example:



$$(char \times char) \rightarrow int$$

This makes comparisons of type expressions easier, or helps implementing operations on type expressions such as "unification".

Type names

Type expressions can be named and such names can occur in type expressions (e.g.: typedef in C).

When are two type expressions with type names the same?

structural equivalence:

- either same basic types (e.g., int is equivalent to int), or
- formed by application of the same type constructor to structurally equivalent types, where
- type names are abbreviations for type expressions

In some sense, structural equivalence deletes names in composite constructs and replaces them by the type structures that they represent (unroll the tree or DAG).

name equivalence:

- Two type expressions are equivalent iff they are identical
- each name is assumed to represent a different type.

type link1 = $\uparrow$ A $\neq$ type link2 = $\uparrow$ A

Structural type equivalence algorithm

```
function sequiv (s,t): boolean;
begin
    if s and t are the same basic type then
        return true
    else if s = array (s1, s2) and t = array (t1, t2) then
        return sequiv (s1, t1) and sequiv (s2, t2)
    else if s = s1 × s2 and and t = t1 × t2 then
        return sequiv (s1, t1) and sequiv (s2, t2)
    else if s = pointer (s1) and t = pointer (t1) then
        return sequiv (s1, t1)
    else if s = s1 → s2 and  t = t1 → t2 then
        return sequiv (s1, t1) and sequiv (s2, t2)
    else
        return false
end
```

Problem: Recursive data types

Type equivalence

declaration equivalence

Problems arise in languages that use name equivalence but allow declarations to use a type that is not a name. Solution: Implicit new (fresh) type names are created for such types for each declaration.

type link = $\uparrow$ A
var a_1 : $\uparrow$ A // *implicit type name* `type1`
var a_2, a_3 : $\uparrow$ A // *implicit type name* `type2`
var a_4 : link
var a_5 : link

a_1 ($type1$) has a different type from a_2, a_3 ($type2$).
a_4, a_5 are of the same type (link), but have a different type from a_1 and a_2, a_3.

Note: There are languages that use a hybrid approach.

Example: C uses structural equivalence for all types except structs (records). Structs may contain recursive references. C treats the name of the struct as part of its type, and therefore testing for structural equivalence stops when a struct (record) constructor is reached.

Type variables

Type expressions may contain variables (*type variables*) whose values are type expressions.

Type variables are used for implicitly typed languages or languages with polymorphic types.

Programming languages can be

- explicitly typed — every object is declared with its type (**type checking**)
- implicitly typed — type of object is derived from its use (**type reconstruction**)
- monomorphic — every object has a unique, single type
- polymorphic — allows objects to have more than one type (e.g., *nil*, and & in C)

Type variables — examples

Recall:

$$\frac{E \vdash e_1 : int \quad E \vdash e_2 : int}{E \vdash (e_1 + e_2) : int}$$

where E is a type environment. In other words, "+" has the type expression $(int \times int) \rightarrow int$.

What are the types of the variables a and b in the following program:

```
read(a);
read(b);
...  a + b ...;
```

Here is an idea: Guess the types of variables you don't know about and use *unification* to match guesses with rules.

`read(a)`	{a:α}
`read(b)`	{a:α, b:β}
`a + b`	unify(α, int)
	unify(β, int)
	apply type rule; result int

Type variables — examples

- Polymorphic cons:

$$\frac{E \vdash e_1 : \alpha \quad E \vdash e_2 : list(\alpha)}{E \vdash cons(e_1, e_2) : list(\alpha)}$$

cons has the type expression
$\forall\alpha.(\alpha \times list(\alpha)) \rightarrow list(\alpha)$

- Polymorphic nil:

$$E \vdash nil : list(\alpha)$$

nil has the type expression $\forall\alpha.list(\alpha)$

Questions:

- What is the type of cons(a,nil)?
- What is the type of cons(1,nil)?
- What is the type of cons(a,1)?

Unification

A *unifier* is a substitution of variables by expressions such that, when applied to two expressions $expr_1$ and $expr_2$, both expressions become *syntactically identical*.

The most general unifier (mgu) makes as few and as general bindings as possible.

Example:

times(Z, times(Y, 7)) = times(4, W)

$\Theta_1 =$
$$\{Z \rightarrow 4, Y \rightarrow plus(3,5), W \rightarrow times(plus(3,5),7)\}$$

$\Theta_2 =$
$$\{Z \rightarrow 4, W \rightarrow times(Y,7)\}$$

Question: Which of the two substitutions is more general?

Note: mgu θ is unique modulo renaming of variables

Type systems with polymorphism

How can we infer the type of a polymorphic function such as `length`?

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

- Need type variables to represent unknown types during type reconstruction (type inference).
- Need type environment E that
  - Keeps track of type expressions for program variables and type variables
  - Is initialized to predefined types of constants and functions

Algorithm for typing functions

High–level view of algorithm

1. Add fresh type variables to E for function name, arguments, and result. Add resulting type expression for function to E.
2. Analyze the program (parse tree) bottom up:
   - If constant, look–up type in E.
   - If variable, look–up type in E. If no entry in E, generate entry with fresh type variable.
   - If function application, look–up function in environment. "Apply" application inference rule by unification of type expressions of formal and actual parameters and result.

   $$\frac{E \vdash e_1 : \alpha \rightarrow \beta \quad E \vdash e_2 : \alpha}{E \vdash e_1(e_2) : \beta}$$

   Note: If function is polymorphic ($\forall \alpha. \ldots$) generate fresh type variable for bound variable and remove $\forall$ in resulting type expression.

If at any point unification fails, algorithm returns type error.

Algorithm for typing functions — example

Type inference for function length

Initial type environment:

$$
\begin{aligned}
E = \{&0{:}\text{int},\\
&1{:}\text{int},\\
&\text{if}{:}\forall\alpha.boolean \times \alpha \times \alpha \rightarrow \alpha\\
&\text{null}{:}\forall\alpha.list(\alpha) \rightarrow boolean\\
&\text{tl}{:}\forall\alpha.list(\alpha) \rightarrow list(\alpha)\\
&+{:}int \times int \rightarrow int \;\; \}
\end{aligned}
$$

See ALSU 6.5.4, page 391 — 395

Algorithm for typing functions — example

type expressions	type environment
	l: γ length: $\gamma \rightarrow \delta$
l: γ null: $list(\alpha_1) \rightarrow boolean$ null(l): boolean	$\gamma = list(\alpha_1)$
0: int	
l: $list(\alpha_1)$ tl: $list(\alpha_2) \rightarrow list(\alpha_2)$ tl(l): $list(\alpha_1)$ length: $list(\alpha_1) \rightarrow \delta$ length(tl(l)): δ 1: int +: $int \times int \rightarrow int$ length(tl(l))+1: int	$\alpha_1 = \alpha_2$ $\delta = int$
if: $boolean \times \alpha_3 \times \alpha_3 \rightarrow \alpha_3$ if(...): int	$\alpha_3 = int$
length: $list(\alpha_1) \rightarrow int$	

Since we don't make any assumptions about α_1, we get the polymorphic type for length: $\forall \alpha_1 . list(\alpha_1) \rightarrow int$.

Second Project

A Type Reconstructor for a TINY subset.