Type Checking- Contd

66.648 Compiler Design Lecture (03/02/98)

Computer Science

Rensselaer Polytechnic

Lecture Outline

- Types and type expressions
- Unification
- Administration

Type Expressions

We are in chapter 6 of the text book. Please read that chapter.

Equivalence of Type Expressions:

The checking rules have the form: if two type expressions are equal then return a certain type else return type-error.

There is an interaction between the notion of equivalence of types and representation of types and we have to take both of them together.

Structural Equivalence

As long as type expressions are built from basic types and constructors, a notion of equivalence between two type expressions is structural equivalence.

Two type expressions are structurally equivalent iff they are identical.

Ex: pointer(Integer) is equivalent to pointer(Integer).

Type Checking

Modifications of the notion of structural equivalence are needed to reflect the actual type checking of the source language. (array bounds may not be part of the types -But in Java, array bound checking is done)

Type Constructor	Encoding
pointer	01
array	10
freturns	11

Encoding-Contd

The basic types are encoded using 4 bits as follows:

Basic Types	Encoding
Boolean	0000
char	0001
Integer	0010
Real	0011

char	000000	0001
pointer(freturns(char))	000111	0001

Type Equivalence

boolean sequival (s, t)

{ if s and t are the same basic type return true;

else if s=array(s1,s2) and t=array(t1,t2) then

return sequival(s1,t1) and sequival(s2,t2)

else if s=s1xs2 and t=t1xt2 then return sequival(s1,t1) and sequival(s2,t2)

else if s= pointer(s1) and t=pointer(t1) then return sequival(s1,t1) else if s=s1->s2 and t=t1->t2 then return sequival(s1,t1) and sequival(s2,t2)

else return false }

Type Equivalence with Encoding

With the encoding testing of structural equivalence becomes simpler.

Languages such as Java use type signatures to state about the return types.

As against structural type equivalence, there is also the notion of name type equivalence.

(This happens in cases where types are given names).

C and Java requires type names to be declared before they are used.

Type Coersions

X + I

x is real and I is integer.

X I into real real +

(The language definition specifies what conversions are necessary.)

Type conversions occur with the overloading of operators.

Coersions: Conversion is implicit when is done by the compiler. Conversion is said to be explicit

Function Overloading

If the programmer must write something to cause the conversion.

An overloaded symbol is one that has different meanings on its context.

Set of possible types for a subexpression:

*: int x int -> int

*: realxreal -> real

*:complex x complex -> complex

Narrowing of Types

A complete expression has a unique type. Given a unique type from the context, we can narrow down the type choices for each subexpression.

function add(I,j) = I+j end.

These will cause a type error. Because, we cannot narrow the type. However, if we state

function add(I:int,j) = I+j end.

Polymorphic Functions

An ordinary procedure allows the statements in its body to be executed with arguments of fixed type.

When a polymorphic procedure is invoked, the statements in its body can be executed with arguments of different types.

Ex: Built in operators for indexing operators, applying functions and manipulating pointers.

Type Variables

Variables representing type expressions allow one to talk about unknown types.

An important application of type variables is checking consistent usage of identifiers in a language that does not require identifiers to be declared before they are used. (In Java, c identifiers have to be declared before they are used.)

Ex: sml, lisp..

Type Inference

Type inference is the problem of determining the type of a language construct from the way it is used.

Ex:

Fun add(I,j) = I+j+2 end.

Fun deref(p) = return *p.

Type Inference

Techniques of type inference and type checking have a lot in common.

The methodology of type inference is similar to the inferences used in Artificial Intelligence.

Type expressions are similar to arithmetic expressions with constants being basic types.

There are notions of substitutions, instances and unification.

By substitution, we mean whenever an expression s appears in t replace s in t with something else.

Substitutions

Type_Expr subst(Type_Expr t)

{ if t is a basic type return t

else if t is a variable return s(t)

else if t is t1 -> t2 return subst(t1)->subst(t2)

}

Ex:

poniter(a) becomes pointer(real) if we substitute a= real.

Instances

Pointer(integer) is an instance of pointer(a)

Pointer(real) is an instance of pointer(a)

Pointer (real) is an instance of a

integer->integer is an instance of a->a

integer is not an instance of real.

Integer-> a is not instance of a->a

Integer->real is not an instance of a->a

<u>Instance impose an ordering</u>

Unification

Two type expressions t1 and t2 unify if there exists some substitutions such that s(t1)=s(t2).

We are interested in the most general unifier (I.e., a substitution with the fewest constraints on the variables in the expressions.)

1. s(t1) = s(t2) and
2. For any other substitution s' such that s'(t1)=s'(t2), the substitution s' is an instance of s.

Comments and Feedback

Project 2 is out. Please start working. PLEASE do not wait for the due date to come.

We are in chapter 6. We will finish the rest of chapter 6 in the next class. Please keep studying this material. It may look difficult.