CSC 425 - Principles of Compiler Design I

Type Checking

Outline

- General properties of type systems
- Types in programming languages
- Notation for type rules
  - logical rules of inference
- Common type rules

Static Checking

- Static checking refers to the compile-time checking of programs in order to ensure that the semantic conditions of the language are being followed
- Examples of static checks include:
  - Type checks
  - Control flow checks
  - Uniqueness checks
  - Name related checks

Static Checking

- Control flow checks: statement that cause the flow of control to leave a construct must have some place where control can be transferred; for example, `break` statements in C
- Uniqueness checks: a language may dictate that in some contexts, an entity can be defined exactly once; for example, identifier declarations, labels, values in case expressions
- Name related checks: sometimes the same name must appear two or more times; for example, in Ada a loop or block can have a name that must then appear both at the beginning and at the end

Types and Type Checking

- A type is a set of values together with a set of operations that can be performed on them
- The purpose of type checking is to verify that operations performed on a value are in fact permissible
- The type of an identifier is typically available from declarations, but we may have to keep track of the type of intermediate expressions

Type Expressions and Type Constructors

- A language usually provides a set of base types that it supports together with ways to construct other types using type constructors
- Through type expressions we are able to represent types that are defined in a program

Type Expressions

- A base type is a type expression
- A type name is a type expression
- A type constructor applied to type expressions is a type expression, for example:
  - arrays: if T is a type expression and I is a range of integers, then $\mathit{array}(I, T)$ is a type expression
  - records: if $T_1, \ldots, T_n$ are type expressions and $f_1, \ldots, f_n$ are field names, then $\mathit{record}((f_1, T_1), \ldots, (f_n, T_n))$ is a type expression
  - pointers: if T is a type expression, then $\mathit{pointer}(T)$ is a type expression
  - functions: if $T_1, \ldots, T_n$ and T are type expressions, then so is $(T_1, \ldots, T_n) \rightarrow T$

Notions of Type Equivalence

- Name equivalence: in many languages, for example, Pascal, types can be given names. Name equivalence views each distinct name as a distinct type. Two type expressions are name equivalent if and only if they are identical
- Structural equivalence: two expressions are structurally equivalent if and only if they have the same structure, that is, if they are formed by applying the same constructor to structurally equivalent type expressions

Example of Type Equivalence

- In the Pascal fragment:

```
type nextptr = ^node;
     prevptr = ^node;
var p : nextptr;
    q : prevptr;
```

p is not name equivalent to q, but p and q are structurally equivalent

Static Type Systems and their Expressiveness

- A static type system enables a compiler to detect many common programming errors
- The cost is that some correct programs are disallowed
  - some argue for dynamic type checking instead
  - others argue for more expressive static type checking
  - but, a more expressive type system is also more complex

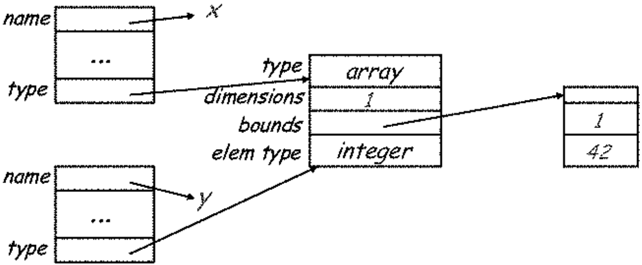
Compile-time Representation of Types

- Need to represent type expressions in a way that is both easy to construct and easy to check
- Approach: Type Graphs
  - Basic types can have predefined “internal values”, for example, small integer values
  - Named types can be represented using a pointer into a hash table
  - Composite type expressions: the node for $f(T_1, \ldots, T_n)$ contains a value representing the type constructor f, and pointers to the nodes for the expressions $T_1, \ldots, T_n$

Compile-time Representation of Types

- Example:

```
var x, y : array[1..42] of integer;
```



Compile-time Representation of Types

- Approach: Type Encodings
  - Basic types use a predefined encoding of the low-order bits

Basic Type	Encoding
boolean	0000
char	0001
integer	0002

  - The encoding of a type expression *op*(T) is obtained by concatenating the bits encoding *op* to the left of the encoding of T

Type Expression	Encoding
char	00 00 00 0001
array(char)	00 00 01 0001
ptr(array(char))	00 10 01 0001
ptr(ptr(array(char)))	10 10 01 0001

Compile-time Representation of Types

- Type encodings are simple and efficient
- On the other hand, named types and type constructors that take more than one type expression are arguments are hard to represent as encodings. Also, recursive types cannot be represented directly.
- Recursive types (for example, lists and trees) are not a problem for type graphs; the graph simply contains a cycle

Types in an Example Programming Language

- Let us assume that types are:
  - base types: integers and floats
  - arrays of a base type
  - booleans (used in conditional expressions)
- The user declares types for all identifiers
- The compiler infers a type for every expression

Type Checking and Type Inference

- Type checking is the process of verifying fully typed programs
- Type inference is the process of filling in missing type information
- The two are different, but are often used interchangeably

Rules of Inference

- We have seen two examples of formal notation for specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)
- The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rues have the form: *If Hypothesis is true, then Conclusion is true*
- Type checking computes via reasoning: *If E_1 and E_2 have certain types, then E_3 has a certain type*
- Rules of inference are a compact notation for "If-Then" statements

From English to an Inference Rule

- The notation is easy to read (with practice)
- Start with a simplified system and gradually add features
- Building blocks:
  - Symbol $\wedge$ is "and"
  - Symbol $\Rightarrow$ is "if-then"
  - $x : T$ is "x" has type "T"
- Example:
  - If e_1 has type *int* and e_2 has type *int*, then $e_1 + e_2$ has type *int*
  - $(e_1 \text{ has type } int \wedge e_2 \text{ has type } int) \Rightarrow e_1 + e_2 \text{ has type } int$
  - $(e_1 : int \wedge e_2 : int) \Rightarrow e_1 + e_2 : int$
- The statement $(e_1 : int \wedge e_2 : int) \Rightarrow e_1 + e_2 : int$ is a special case of $H_1 \wedge \ldots \wedge H_n \Rightarrow C$; this is an inference rule

Notation for Inference Rules

- By tradition, inference rules are written

$$\frac{\vdash \mathit{Hypothesis}_1 \ldots \vdash \mathit{Hypothesis}_n}{\vdash \mathit{Conclusion}}$$

- Type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "it is provable that ..."

Example Rules

- Example

$$\frac{i \text{ is an integer}}{\vdash i : int}[\text{Int}]$$

$$\frac{\vdash e_1 : int \qquad \vdash e_2 : int}{\vdash e_1 + e_2 : int}[\text{Add}]$$

- Thes rules give templates describing how to type integers and $+$ expressions
- By filling in the templates, we can produce complete typings for expressions

Example: $1 + 2$

$$\dfrac{\dfrac{1 \text{ is an integer}}{\vdash 1 : \mathit{int}}\,[\text{Int}] \qquad \dfrac{2 \text{ is an integer}}{\vdash 2 : \mathit{int}}\,[\text{Int}]}{\vdash 1 + 2 : \mathit{int}}\,[\text{Add}]$$

Soundness

- A type system is sound if, whenever $\vdash e : T$, then e evaluates to a value of type T
- We only want sound rules, but some sound rules are better than others:

$$\frac{i \text{ is an integer}}{\vdash i : \mathit{number}}$$

Type Checking Proofs

- Type checking proves facts $e : T$
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each kind of AST node
- In the type rule used for a node e
  - Hypotheses are the proofs of types of e's subexpressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

Rules for Constants

$$\frac{i \text{ is an integer}}{\vdash i : \mathit{number}}\text{[Int]}$$

$$\frac{}{\vdash \mathit{true} : \mathit{bool}}\text{[Bool]}$$

$$\frac{}{\vdash \mathit{false} : \mathit{bool}}\text{[Bool]}$$

$$\frac{f \text{ is a floating point number}}{\vdash \mathit{false} : \mathit{bool}}\text{[Float]}$$

Some Other Rules

$$\frac{\vdash e_1 : int \qquad \vdash e_2 : int}{\vdash e_1 + e_2 : int}[\text{Add}]$$

$$\frac{\vdash e : bool}{\vdash note : bool}[\text{Not}]$$

$$\frac{\vdash e_1 : bool \qquad \vdash e_2 : T}{\vdash while\ e_1\ do\ e_2 : T}[\text{While}]$$

A Problem

- What is the type of a variable reference?

$$\frac{x \text{ is an identifier}}{\vdash x :?}[\text{Var}]$$

- The local, structural rule does not carry enough information to give x a type

A Solution

- Put more information in the rules
- A type environment give types for free variables
  - A type environment is a function from identifiers to types
  - A variable is free in an expression if it is not defined within the expression

Type Environments

- Let E be a function from identifiers to types
- The sentence

  $$E \vdash e : T$$

  is read: under the assumption that variables have the types given by E, it is provable that the expression e has type T

Type Environments and Rules

- The type environment is added to the earlier rules, for example

$$\frac{i \text{ is an integer}}{E \vdash i : \mathit{int}}[\text{Int}]$$

$$\frac{E \vdash e_1 : \mathit{int} \qquad E \vdash e_2 : \mathit{int}}{E \vdash e_1 + e_2 : \mathit{int}}[\text{Add}]$$

- And we can now write a rule for variables:

$$\frac{E(x) = T}{E \vdash x : T}[\text{Var}]$$

Type Checking Expressions

Production	Semantic Rules
$E \rightarrow id$	$\{if(declared(id.name))$ $then\ E.type := lookup(id.name).type$ $else\ E.type := error()\}$
$E \rightarrow int$	$\{E.type := integer\}$
$E \rightarrow E_1 + E_2$	$\{if(E_1.type == integer \wedge E_2.type == integer)$ $then\ E.type := integer$ $else\ E.type := error()$

Type Checking Expressions

- May have automatic type coercion
- Example:

$E_1.type$	$E_2.type$	E.type
integer	*integer*	*integer*
integer	*float*	*float*
float	*integer*	*float*
float	*float*	*float*

Type Checking of Statements: Assignment

- Semantic Rules:

$$S \rightarrow \mathit{Lval} := \mathit{Rval} \quad \{\mathit{check\_types}(\mathit{Lval}.\mathit{type}, \mathit{Rval}.\mathit{type})\}$$

- Note that in general *Lval* can be a variable or it may be a more complicated expression, for example, a dereferenced pointer, an array element, or a record field
- Type checking involves ensuring that:
  - *Lval* is a type that can be assigned to, for example, it is not a function or a procedure
  - The types of *Lval* and *Rval* are "compatible", that is, the language rules provide for coercion of the type of *Rval* to the type of *Lval*

Type Checking of Statements: Assignment

- Semantic Rules:

$$Loop \rightarrow while\ E\ do\ S \quad \{check\_types(E.type, bool)\}$$

$$Cond \rightarrow if\ \ E\ then\ S_1\ elseS_2 \quad \{check\_types(E.type, bool)\}$$