Compiler Theory: Type Checking

Marc Moreno Maza
University of Western Ontario
CS447b – Winter 2002

December 1, 2004

1 Type Systems

<u>SEMANTIC CHECKING.</u> A compiler must perform many semantic checks on a source program.

- Many of these are straightforward and are usually done in conjunction with syntax-analysis.
- Checks done during compilation are called *static checks* to distinguish from the *dynamic checks* done during the execution of the target program.
- Static checks include:

  Type checks. The compiler checks that names and values are used in accordance with type rules of the language.

  Type conversions. Detection of implicit type conversions.

  Dereferencing checks. The compiler checks that dereferencing is applied only to a pointer.

  Indexing checks. The compiler checks that indexing is applied only to an array.

  Function call checks. The compiler checks that a function (or procedure) is applied to the correct number and type of arguments.

Uniqueness checks. In many cases an identifier must be declared exactly once.

Flow-of-control checks. The compiler checks that if a statement causes the flow of control to leave a construct, then there is a place to transfer this flow. For instance when using **break** in C.

In this chapter we will focus on type checking and type conversions. Most of the other static checks are easy to implement.

<u>TYPE INFORMATION</u> may be needed for the code generation. For instance, for the translation of *overloaded symbols* like $+$. Indeed, overloading may be accompanied by coercion of types, where a compiler supplies an operator to convert an operand into the type expected by the context.

<u>OVERLOADED SYMBOLS AND POLYMORPHIC FUNCTIONS.</u> An *overloaded symbol* is a symbol that represent different operations in different contexts. This is different from a *polymorphic function* which is a function whose body can be executed with arguments of several types.

<u>TYPES HAVE STRUCTURE.</u> Generally types are either basic or structured.

- Basic types are the atomic types with no internal structure. For instance, in C: `char`, `int`, `double`, `float`, etc.
- Structured types are the types of constructs like arrays, sets, (typed) pointers, structs, classes, and functions.

<u>TYPE EQUIVALENCE.</u> Because of overloaded symbols and structured types, we will be concerned with the non trivial problem of type equivalence.

- That is, given two language constructs, decide whether they have the same type.
- We will use the notion of a *type expression.*

<u>TYPE EXPRESSION.</u> The idea is to associate each language construct with an expression describing its type and so called *type expression.* Type expressions are defined inductively from basic types and constants using *type constructors.* Here are the type constructors that we shall consider in the remaining of this chapter. They are close to type constructors of languages like C or PASCAL.

Basic types. `char`, `int`, `double`, `float` are type expressions.

Type names. It is convenient to consider that names of types are type expressions.

Arrays. If T is a type expression and n is an `int` then

$$\texttt{array}(n, T)$$

is a type expression denoting the type of an array with elements in T and indicies in the range $0 \cdots n - 1$.

Products. If T_1 and T_2 are type expressions then

$$T_1 \times T_2$$

is a type expression denoting the type of an element of the Cartesian product of T_1 and T_2. We assume that products of more than two types are defined in a similar way and that this product-operation is left-associative. Hence

$$(T_1 \times T_2) \times T_3 \text{ and } T_1 \times T_2 \times T_3$$

are equivalent.

Records. If $name_1$ and $name_2$ are two type names, if T_1 and T_2 are type expressions then

$$\texttt{record}(name_1\text{: } T_1, \, name_2\text{: } T_2)$$

is a type expression denoting the type of an element of the Cartesian product of T_1 and T_2. The only difference between a record and a product is that the fields of a record have names.

Pointers. If T is a type expression, then

$$\texttt{pointer}(T)$$

is a type expression denoting the type *pointer to an object of type T*.

Function types. If T_1 and T_2 are type expressions then

$$T_1 \rightarrow T_2$$

is a type expression denoting the type of the functions associating an element from T_1 with an element from T_2.

GRAPHICAL REPRESENTATIONS FOR TYPE EXPRESSIONS. Let Σ_{val} be the alphabet consisting of

- the basic types,
- the possible values of an array index, say $\mathbb{N}$,
- the possible names for a type.

Let Σ_{op} be the alphabet consisting of the keywords `array`, $\times$, `record`, `pointer` and $\rightarrow$. Then it is not hard to see that the set of type expressions can be viewed as a language of expressions in the sense of the section about syntax trees in the chapter dedicated to syntax-directed definitions. Indeed we could write

$$(\texttt{array}\ n\,T),\ (\texttt{record}\ name_1\ T_1\ name_2\ T_2) \text{ and } (\texttt{pointer}\ T)$$

instead of

$$\texttt{array}(n, T),\ \texttt{record}(name_1\text{: } T_1,\ name_2\text{: } T_2) \text{ and } \texttt{pointer}(T).$$

That way we would satisfy the definition of a language of (arithmetic) expressions. It follows that we can use syntax trees and DAGs to represent type expressions graphically.

TYPE SYSTEMS. A *type system* is a set of rules assigning type expressions to different parts of the program.

- Type systems can (usually) be implemented in a syntax-directed way.
- The implementation of a type system is called a type checker.

A LANGUAGE IS STRONGLY TYPED if the compiler can guarantee that the accepted programs will **execute without type errors.** Lisp languages are usually weakly typed:

```
CannaLisp listener 3.5 beta 2
-> (defun f (l) (car l))
f
-> (f '(+ 1 2))
+
-> (f (+ 1 2))
Bad arg to car 3
```

2 Specification of a Simple Type Checker

We consider the language generated by the following grammar.

$$
\begin{array}{l}
P \longmapsto D;\ E \\
D \longmapsto D;\ D \\
D \longmapsto \mathbf{id} : T \\
T \longmapsto \mathbf{character} \\
T \longmapsto \mathbf{integer} \\
T \longmapsto \mathbf{array}[\mathbf{num}]\mathbf{of}T \\
T \longmapsto \uparrow T \\
T \longmapsto T \rightarrow T \\
E \longmapsto \mathbf{literal} \\
E \longmapsto \mathbf{num} \\
E \longmapsto \mathbf{id} \\
E \longmapsto E \,\mathbf{mod}\, E \\
E \longmapsto E[E] \\
E \longmapsto \uparrow E \\
E \longmapsto E(E)
\end{array}
$$

- A sentence of this language is a *P*rogram.
- A *P*rogram consists of a sequence of *D*eclarations followed by an *E*xpression.
- **character** and **integer** are the basic types whereas **literal** and **num** stand for elements of these types.
- **id** is the token for identifiers.
- $\mathbf{array}[\mathbf{num}]\mathbf{of}T$ is an array type construct whereas $E[E]$ refers to an element of an array.

- $\uparrow T$ is a pointer type construct whereas $\uparrow E$ is a pointer dereference.
- $T \rightarrow T$ is a function type construct whereas $E(E)$ is a function call.
- $E \,\mathbf{mod}\, E$ represents a remainder computation.

We consider the following attributes.

Grammar symbol	Synthesized attribute	Inherited attribute
E	$E.type$ (type expression)	
T	$T.type$ (type expression)	
id	$\mathbf{id}.entry \in \mathbb{Z}$	
num	$\mathbf{num}.val \in \mathbb{Z}$	
literal	$\mathbf{literal}.val$ character	

First we give a translation scheme that saves the type of an identifier.

$D \longmapsto \mathbf{id} : T$	$\{ \ \mathsf{addtype}(\mathbf{id}.entry, \ T.type) \ \}$
$T \longmapsto \mathbf{character}$	$\{ \ T.type := \mathbf{character} \ \}$
$T \longmapsto \mathbf{integer}$	$\{ \ T.type := \mathbf{integer} \ \}$
$T \longmapsto \mathbf{array}[\mathbf{num}]\mathbf{of} T_1$	$\{ \ T.type := \mathsf{array}(0 \cdots \mathbf{num}.val - 1, T_1.type) \ \}$
$T \longmapsto \uparrow T_1$	$\{ \ T.type := \mathsf{pointer}(T_1) \ \}$
$T \longmapsto T_1 \rightarrow T_2$	$\{ \ T.type := (T_1.type \rightarrow T_2.type)$

Note that because of the rule

$$D \quad \longmapsto \quad \mathbf{id} : T \ \{\mathsf{addtype}(\mathbf{id}.entry, T.type)\} \tag{1}$$

the types of all identifiers are saved in the symbol table before the expression generated by E is checked. Now we give a translation scheme for type checking rules for expressions. We assume that lookup retrieves the type information about an entry of the symbol table.

$E \longmapsto$ **literal**	$\{\ E.type :=$	**character** $\}$
$E \longmapsto$ **num**	$\{\ E.type :=$	**integer** $\}$
$E \longmapsto$ **id**	$\{\ E.type :=$	$\mathsf{lookup}(\mathbf{id}.entry)\ \}$
$E \longmapsto E_1\ \mathbf{mod}\ E_2$	$\{\ E.type :=$	if $E_1.type =$ **integer** and $E_2.type =$ **integer** then **integer** else **type_error** $\}$
$E \longmapsto E_1[E_2]$	$\{\ E.type :=$	if $E_1.type =$ **integer** and $E_2.type = \mathsf{array}(n, T)$ then T else **type_error** $\}$
$E \longmapsto \uparrow E_1$	$\{\ E.type :=$	if $E_1.type = \mathsf{pointer}(T)$ then T else **type_error** $\}$
$E \longmapsto E_1(E_2)$	$\{\ E.type :=$	if $E_2.type = S$ and $E_1.type = S \rightarrow T$ then T else **type_error** $\}$

Now we extend our language by stating that

- A *P*rogram consists of a sequence of *D*eclarations followed by a sequence of *S*tatements.
- A statement is either an assignment, an alternative or a **while** loop.
- **boolean** is a basic type.
- An expression could be $E_1 \leq E_2$ which evaluates to **boolean** provided that E_1 and E_2 have the same basic type, otherwise evaluates to **type_error**.

Therefore we obtain the following new grammar.

$P \longmapsto D;\ S$
$D \longmapsto D;\ D$
$D \longmapsto \mathbf{id} : T$
$T \longmapsto \mathbf{boolean}$
$T \longmapsto \mathbf{character}$
$T \longmapsto \mathbf{integer}$
$T \longmapsto \mathbf{array}[\mathbf{num}]\mathbf{of}T$
$T \longmapsto \uparrow T$
$E \longmapsto \mathbf{literal}$
$E \longmapsto \mathbf{num}$
$E \longmapsto \mathbf{id}$
$E \longmapsto E\, \mathbf{mod}\, E$
$E \longmapsto E\ \leq\ E$
$E \longmapsto E[E]$
$E \longmapsto \uparrow E$
$S \longmapsto S_1;\ S_2$
$S \longmapsto \mathbf{id} := E$
$S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ S$
$S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ S$

Extending the translation scheme for statements leads to:

$S \longmapsto S_1;\ S_2$	$\{\ S.type :=$	if $S_1.type =$ void and $S_2.type =$ void then void else **type_error** $\}$
$S \longmapsto \mathbf{id} := E$	$\{\ S.type :=$	if $E.type = \mathbf{id}.type$ then void else **type_error** $\}$
$S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ S_1$	$\{\ S.type :=$	if $E.type =$ **boolean** then $S_1.type$ else **type_error** $\}$
$S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ S_1$	$\{\ S.type :=$	if $E.type =$ **boolean** then $S_1.type$ else **type_error** $\}$

3 Type Equivalence

<u>Type checking rules</u> usually have the form

> **if** two type expressions are equivalent
>
> **then** return a given type
>
> **else** return **type_error**

<u>Key ideas.</u> The central issue is then that we have to define when two given type expressions are equivalent.

- The main difficulty arises from the fact that most modern languages allow the naming of user-defined types.
- For instance, in C and C++ this is achieved by the `typedef` statement.
- When checking equivalence of named types, we have two possibilities.

  Name equivalence. Treat named types as basic types. Therefore two type expressions are *name equivalent* if and only if they are identical, that is if they can be represented by the same syntax tree, with the same labels.

  Structural equivalence. Replace the named types by their definitions and recursively check the substituted trees.

<u>Structural Equivalence.</u> If type expressions are built from basic types and constructors (without type names, that is in our example, when using products instead of records), structural equivalence of types can be decided easily.

- For instance, to check whether the constructed types `array(n1,T1)` and `array(n2,T2)` are equivalent
  - we can check that the integer values `n1` and `n2` are equal and recursively check that `T1` and `T2` are equivalent,
  - or we can be less restrictive and check only that `T1` and `T2` are equivalent.

- Compilers use representations for type expressions (trees or dags) that allow type equivalence to be tested quickly.

RECURSIVE TYPES. In PASCAL a *linked list* is usually defined as follows.

```
type link = ^ cell;
     cell = record
                info: type;
                next: link;
             end;
```

The corresponding type graph has a cycle. So to decide structural equivalence of two types represented by graphs PASCAL compilers put a *mark* on each visited node (in order not to visit a node twice). In C, a *linked list* is usually defined as follows.

```
struct cell {
    int info;
    struct cell *next;
};
```

To avoid cyclic graphs, C compilers

- require type names to be declared before they are used, except for pointers to records.
- use structural equivalence except for records for which they use name equivalence.

4 Type Conversions

Type Conversions can be

- explicit (casts) if the programmer must write something to cause these conversions,
- implicit (coercions) if done automatically by the compiler.

From the point of view of type checking,

- explicit conversions are just like function applications,

- implicit conversions requires the type checker
  - to detect that the two types are different and
  - to insert the appropriate cast.

Coercions

- occur for instance with overloaded operators (say in C++) or in assignment statements and
- for efficiency reasons should be done at compile time.

Example 1 Let us extend the set of expression constructs of the language from the previous section

- We add a type **real** for floating point numbers with some fixed precision.
- Elements of this new type must match the pattern **num.num**.
- We also add binary infix arithmetic operators and use **op** in what follows to denote any of them.
- These new operators are defined over the types **integer** and **real**.

$$\begin{array}{|l|}\hline T \longmapsto \textbf{real} \\ E \longmapsto \textbf{num}.\textbf{num}. \\ E \longmapsto E\, \textbf{op}\, E \\ \hline \end{array}$$

The translation scheme for these new rules are:

$T \longmapsto \textbf{real}$	$\{\ T.type :=$ **real** $\}$
$E \longmapsto \textbf{num}.\textbf{num}$	$\{\ E.type :=$ **real** $\}$
$E \longmapsto E_1\, \textbf{op}\, E_2$	$\{\ E.type :=$ if $E_1.type =$ **integer** and $E_2.type =$ **integer** then **integer** else if $E_1.type =$ **real** and $E_2.type =$ **real** then **real** else if $E_1.type =$ **integer** and $E_2.type =$ **real** then **real** else if $E_1.type =$ **real** and $E_2.type =$ **integer** then **real** else **type_error** $\}$

5 Overloading of Functions and Operators

An overloaded operator may have different meanings depending upon its context.

- Normally overloading is resolved by the types of the arguments,
- but sometimes this is not possible and an expression can have a set of possible types.

Example 2 In the previous section we were resolving overloading of binary arithmetic operators by looking at the the types of the arguments. Indeed we had two possibles types, say $\mathbb{Z}$ and $\mathbb{R}$, with a natural coercion due to the inclusion

$$\mathbb{Z} \subseteq \mathbb{R} \tag{2}$$

But what could we do if we had the three types $\mathbb{Z}$, $\mathbb{Z}/p\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z}$ for two different integers m and p?

- There is no natural coercion between $\mathbb{Z}/p\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z}$.
- So the type of an expression like $1 + 2$ and consequently the signature of $+$ may also depend on what is done with $1 + 2$.

Set of possible types for a subexpression. The **first step** in resolving the overloading of operators and functions occuring in an expression E' is to determine the possible types for E'.

- For simplicity, we restrict here to unary functions.
- We assign to each subexpression E of E' a synthesized attribute $E.types$ which is the set of possible types for E.
- These attributes can be computed by the following translation scheme.

$E' \longmapsto E$	$\{\ E'.types :=\quad E.types\ \}$
$E \longmapsto \mathbf{id}$	$\{\ E.types :=\quad \mathsf{lookup}(\mathbf{id}.entry)\ \}$
$E \longmapsto E_1[E_2]$	$\{\ E.types :=\quad \{t \mid (\exists\ s \in E_2.types) \mid (s \rightarrow t) \in E_1.types\}\ \}$

Narrowing the set of possible types. The **second step** in resolving the overloading of operators and functions in the expression E' consists of

- determining if a unique type can be assigned to each subexpression E of E' and
- generating the code for evaluating each subexpression E of E'.

This is done by

- assigning an inherited attribute $E.unique$ to each subexpression E of E' such that
  - either E can be assigned a unique type and $E.unique$ is this type,
  - or E cannot be assigned a unique type and $E.unique$ is **type_error**.
- assigning a synthesized attribute $E.code$ which is the target code for evaluating E and
- executing the following translation scheme (where the attributes $E.types$ have already been computed).

$E' \longmapsto$		$\{ E.unique :=$	if $E'.types = \{t\}$
			then t
			else **type_error** $\}$
	E		
		$\{ E'.code :=$	$E.code$ $\}$
$E \longmapsto$	**id**	$\{ E.code :=$	`generate`(**id**.$lexeme$ ':' $E.unique$) $\}$
$E \longmapsto$			$\{ t := E.unique \}$
			$\{ S := \{s \in E_2.types \mid (s \to t) \in E_1.types\} \}$
		$\{ E_2.unique :=$	if $S = \{s\}$
			then s
			else **type_error** $\}$
		$\{ E_1.unique :=$	if $S = \{s\}$
			then $s \to t$
			else **type_error** $\}$
	$E_1[E_2]$		
			$\{ code :=$ `generate`(`apply`':' $E.unique$) $\}$
		$\{ E.code :=$	$E_1.code \Vert E_2.code \Vert code$