

Type-checking

Types, Storage Layout and Boolean Control Flow

K.R. Chowdhary, Professor of CS

7CS5A Compiler Construction (Intermediate Code Generation) | Lecture 14

14.1 Types and Declarations

Applications of data types fall into two groups:

Type checking

Logical rules reason about program behavior at run time.

- ▶ Operand types must match what the operator expects
- ▶ Java's && needs two Booleans and yields a Boolean

Translation applications

The type of a name drives code generation:

- ▶ Storage needed at run time
- ▶ Address calculation for array references
- ▶ Explicit conversions; choice of arithmetic operator

Relative addresses

Local declarations are examined at compile time, so each name gets a *relative address*: an offset from the start of a data area.

14.1.1 Type Expressions

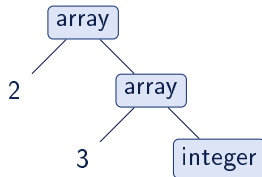
A **type expression** is either a basic type, or a *type constructor* applied to type expressions.

Example 14.1: array types

`int a[2][3]` reads “array of 2 arrays of 3 integers each”:

`array(2, array(3, integer))`

The operator *array* always takes two parameters: a number and a type.



Type expression for `int[2][3]`

- ▶ Basic types: boolean, char, integer, float, void (“absence of a value”)
- ▶ A type name is also a type expression
- ▶ Type expressions may contain variables whose values are type expressions

Type Constructors and the Declaration Grammar

Constructors

- ▶ **array**: applied to a number and a type
- ▶ **record**: field names and their types, implemented via a symbol table of the fields
- ▶ **function** $s \rightarrow t$: from type s to type t
- ▶ **product** $s \times t$: lists or tuples, e.g. function parameters. Left associative, higher precedence than $\rightarrow$

Grammar for declaration lists

$$\begin{aligned} D &\rightarrow T \text{ id}; D \mid \varepsilon \\ T &\rightarrow B C \mid \text{record } \{ D \} \\ B &\rightarrow \text{int} \mid \text{float} \\ C &\rightarrow \varepsilon \mid [\text{num}] C \end{aligned}$$

D : sequence of declarations; T : basic, array or record type; B : basic type; C : zero or more bracketed array dimensions.

14.1.2 Type Equivalence

Many type-checking rules have the form: *“if two type expressions are equal, return a certain type”*. Ambiguity arises when names are given to type expressions and then reused.

Represent type expressions as graphs (the value-number method builds a DAG: interior nodes are constructors, leaves are basic types, names and variables). Two are **structurally equivalent** iff:

1. They are the same basic type, or
2. They are formed by applying the same constructor to structurally equivalent types, or
3. One is a type name that denotes the other.

14.2 Storage Layout for Names

- ▶ Type determines the storage a name needs at run time
- ▶ Type and relative address are stored in the symbol table entry
- ▶ Variable-size data (strings, dynamic arrays) get a fixed-size *pointer* to the data
- ▶ SDT synthesizes *type* and *width*; variables t , w carry them from B down to $C \rightarrow \varepsilon$

Table 14.1: computing types and widths

$T \rightarrow B$	$\{t=B.type; w=B.width;\}$
C	
$B \rightarrow \text{int}$	$\{B.type=\text{integer}; B.width=4;\}$
$B \rightarrow \text{float}$	$\{B.type=\text{float}; B.width=8;\}$
$C \rightarrow \varepsilon$	$\{C.type=t; C.width=w;\}$
$C \rightarrow [\text{num}] C_1$	$\{C.type=\text{array}(\text{num.value}, C_1.type);$ $C.width=\text{num.value} \times C_1.width;\}$

14.3 Type Checking: Strong vs. Weak Typing

The compiler assigns a type expression to every program component and checks that they obey the **type system** of the language.

- ▶ Any check can be done dynamically if the target code carries each value's type
- ▶ A *sound* type system proves statically that type errors cannot occur at run time
- ▶ **Strongly typed**: the compiler guarantees accepted programs run without type errors (e.g. Java)
- ▶ **Weakly typed**: no such guarantee (e.g. C)

Security use

Java compiles to machine-independent bytecodes with detailed type information. Imported code is checked before it may execute, guarding against inadvertent errors and malicious misbehaviour.

Rules for Type Checking: Synthesis and Inference

Type synthesis

Builds the type of an expression from its subexpressions; names must be declared before use.

if f has type $s \rightarrow t$ and x has type s ,
then $f(x)$ has type t

$E_1 + E_2$ is treated as $add(E_1, E_2)$.

Type inference

Determines the type of a construct from how it is used, with type variables $\alpha, \beta, \dots$ for unknowns.

if $f(x)$ is an expression, then for some α, β : f has type $\alpha \rightarrow \beta$ and x has type α

From `null(x)`: x must be a list.

- ▶ Inference is needed for languages like ML that check types but do not require declarations
- ▶ Statements are checked like expressions: `if (E) S;` is a function *if* applied to a boolean and a void, giving void

14.4 Control Flow and Boolean Expressions

Boolean expressions serve two purposes:

1. **Alter control flow:** value is implicit in the position reached, e.g. in `if(E) S`, E is true when S is reached
2. **Compute logical values:** evaluated like arithmetic, using three-address instructions with logical operators

The syntactic context decides which use applies.

Grammar

$$B \rightarrow B \parallel B \mid B \&\& B \mid !B \mid (B) \\ \mid E \text{ rel } E \mid \text{true} \mid \text{false}$$

$\parallel$ and $\&\&$ are left-associative; precedence:
 $\parallel$ lowest, then $\&\&$, then $!$.

- ▶ $B_1 \parallel B_2$: if B_1 is true, skip B_2
- ▶ $B_1 \&\& B_2$: if B_1 is false, skip B_2
- ▶ Side effects in B_2 may then give unexpected results

14.4.2 Short-Circuit Code and Review

In short-circuit code, `&&`, `||`, `!` become jumps: the value is a *position* in the code.

Example 14.2

```
if (x < 100 || x > 200 && x != y) x = 0;

    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1: ...
```

Review questions

- ▶ Meaning of `array(2, array(3, integer))`?
- ▶ Pros and cons of strongly vs. weakly typed languages?
- ▶ Is Java's type checking more rigorous than C's? Why?
- ▶ Type expression for `int[5][6][7]`?
- ▶ Synthesis vs. inference: what is the difference?