



Chapter 7

Types and type checking

Course “Compiler Construction”

Martin Steffen

Spring 2021



Section

Introduction

Chapter 7 “Types and type checking”

Course “Compiler Construction”

Martin Steffen

Spring 2021

General remarks and overview



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

- Goal here:
 - what are *types*?
 - static vs. dynamic typing
 - how to describe types *syntactically*?
 - how to *represent* and use types in a compiler?
- coverage of various types
 - basic types (often predefined/built-in)
 - type constructors
 - values of a type
 - type operators
 - representation at run-time
 - run-time tests and special problems (array, union, record, pointers)
- specification and implementation of type systems/type checkers
- advanced concepts

Why types?

- crucial, user-visible **abstraction** describing program behavior
- one view: type describes a set of (mostly related) *values*
- static typing: checking/enforcing a type discipline at compile time
- dynamic typing: same at run-time, mixtures possible
- completely untyped languages: very rare to non-existent, types were part of PLs from the start.

Milner's dictum ("type safety")

Well-typed programs cannot go wrong!

- *strong* typing:¹ rigorously prevent "misuse" of data
- types useful for later phases and optimizations
- documentation and partial specification

¹Terminology rather fuzzy, and perhaps changed a bit over time.



Types: in first approximation

Conceptually

- semantic view: set of values *plus* a set of corresponding operations
- syntactic view: notation to *construct* basic elements of the type (its values) *plus* “procedures” operating on them
- compiler implementor’s view: data of the same type have same underlying memory representation

further classification:

- built-in/predefined vs. *user-defined* types
- basic/base/elementary/primitive types vs. compound types
- type constructors: building more complex types from simpler ones
- reference vs. value types



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking



Section

Various types and their representation

Chapter 7 “Types and type checking”

Course “Compiler Construction”

Martin Steffen

Spring 2021

Some typical base types



INF5110
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

base types

<code>int</code>	<code>0, 1, ...</code>	<code>+, -, *, /</code>	integers
<code>real</code>	<code>5.05E4 ...</code>	<code>+, -, *</code>	real numbers
<code>bool</code>	<code>true, false</code>	<code>and or (!) ...</code>	booleans
<code>char</code>	<code>'a'</code>		characters
<code>:</code>			

- often HW support for some of those (including some of the op's)
- mostly: elements of `int` are not exactly mathematical *integers*, same for `real`
- often variations offered: `int32`, `int64`
- often implicit *conversions* and relations between basic types
 - which the type system has to specify/check for legality
 - which the compiler has to implement

Some compound types



INF5110 –
Compiler
Construction

compound types

<code>array[0..9] of real</code>		<code>a[i+1]</code>
<code>list</code>	<code>[], [1;2;3]</code>	<code>concat</code>
<code>string</code>	<code>"text"</code>	<code>concat ...</code>
<code>struct / record</code>		<code>r.x</code>
<code>...</code>		

- mostly reference types
- when built in, special “easy syntax” (same for basic built-in types)
 - `4 + 5` as opposed to `plus(4, 5)`
 - `a[6]` as opposed to `array_access(a, 6) ...`
- parser/lexer aware of built-in types/operators (special precedences, associativity, etc.)
- cf. functionality “built-in/predefined” via libraries

Introduction

Various types and
their
representation

Equality of types

Type checking

Abstract data types

- unit of *data* together with *functions/procedures/operations* ... operating on them
- encapsulation + interface
- often: separation between exported and internal operations
 - for instance `public`, `private` ...
 - or via separate interfaces
- (static) classes in Java: may be used/seen as ADTs, methods are then the “operations”

```
ADT begin
integer i;
real x;
int proc total(int a) {
    return i * x + a // or: ``total = i * x + a''
}
end
```



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Type constructors: building new types

- array type
- record type (also known as struct-types)
- union type
- pair/tuple type
- pointer type
 - explicit as in C
 - implicit distinction between reference and value types, hidden from programmers (e.g. Java)
- *signatures* (specifying methods / procedures / subroutines / functions) as type
- function type constructor, incl. higher-order types (in functional languages)
- (names of) classes and subclasses
- ...



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking



Array type

```
array [<indextype>] of <component type>
```

- elements (arrays) = (finite) functions from index-type to component type
- allowed index-types:
 - non-negative (unsigned) integers?, from ... to ...?
 - other types?: enumerated types, characters
- things to keep in mind:
 - indexing outside the array bounds?
 - are the array bounds (statically) known to the compiler?
 - *dynamic* arrays (extensible at run-time)?

Introduction

Various types and
their
representation

Equality of types

Type checking

One and more-dimensional arrays



INF5110 –
Compiler
Construction

- one-dimensional: efficiently implementable in standard hardware (relative memory addressing, known offset)
- two or more dimensions

```
array [1..4] of array [1..3] of real  
array [1..4, 1..3] of real
```

- one can see it as “array of arrays” (Java), an array is typically a reference type
- conceptually “two-dimensional”- *linear layout* in memory (language dependent)

Introduction

Various types and
their
representation

Equality of types

Type checking

Records (“structs”)



INF5110 –
Compiler
Construction

```
struct {  
  real r;  
  int i;  
}
```

- values: “labelled tuples” ($\text{real} \times \text{int}$)
- constructing elements, e.g.

```
struct point {int x; int y;};  
struct point pt = { 300, 42 };
```

`struct point`

- access (read or update): *dot-notation* `x.i`
- implementation: linear memory layout given by the (types of the) attributes
- attributes accessible by statically fixed *offsets*
- *fast* access
- cf. objects as in Java

Introduction

Various types and
their
representation

Equality of types

Type checking

Tuple/product types

- $T_1 \times T_2$ (or in ascii `T_1 * T_2`)
- elements are *tuples*: for instance: `(1, "text")` is element of `int * string`
- generalization to n -tuples:

value	type
<code>(1, "text", true)</code>	<code>int * string * bool</code>
<code>(1, ("text", true))</code>	<code>int * (string * bool)</code>

- structs can be seen as “labeled tuples”, resp. tuples as “anonymous structs”
- tuple types: common in functional languages,
- in C/Java-like languages: n -ary tuple types often only implicit as *input* types for procedures/methods (part of the “signature”)



Union types (C-style again)



INF5110 –
Compiler
Construction

```
union {  
  real r;  
  int  i  
}
```

- related to *sum types* (outside C)
- (more or less) represents *disjoint union* of values of “participating” types
- access in C (confusingly enough): dot-notation `u.i`

Introduction

Various types and
their
representation

Equality of types

Type checking



Union types in C and type safety

- union types in C: bad example for (safe) type disciplines, as it's simply type-unsafe, basically an *unsafe* hack ...

Union type (in C):

- nothing much more than a directive to allocate enough memory to hold largest member of the union.
- in the example: `real` takes more space than `int`

Explanation

- implementor's (= low level) focus and memory allocation, not "proper usage focus" or assuring strong typing
- ⇒ bad example of modern use of types
- better (type-safe) implementations known since
- ⇒ *variant record* ("tagged"/"discriminated" union) or even inductive data types

Variant records from Pascal



INF5110 –
Compiler
Construction

```
record case isReal: boolean of
  true: (r: real);
  false: (i: integer);
```

- “variant record”
- non-overlapping memory layout²
- programmer responsible to set and check the “discriminator” self
- enforcing type-safety-wise: not really an improvement :-)

Introduction

Various types and
their
representation

Equality of types

Type checking

²Again, that's an implementor-centric view, not a user-centric one.

Inductive types in ML and similar

- *type-safe* and powerful
- allows *pattern matching*

```
IsReal of real | IsInteger of int
```

- allows *recursive* definitions \Rightarrow inductive data types:

```
type int_bintree =  
  Node of int * int_bintree * bintree  
  | Nil
```

- Node, Leaf, IsReal: *constructors* (cf. languages like Java)
- constructors used as discriminators in “union” types

```
type exp =  
  Plus of exp * exp  
  | Minus of exp * exp  
  | Number of int  
  | Var of string
```



Recursive data types in C



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

does not work

```
struct intBST {
  int val;
  int isNull;
  struct intBST left, right;
}
```

“indirect” recursion

```
struct intBST {
  int val;
  struct intBST *left, *right;
};
```

In Java: references implicit

```
class BSTnode {
  int val;
  BSTnode left, right;
}
```

- note: *implementation* in ML: also uses “pointers” (but hidden from the user)
- no nil-pointers in ML (and `NIL` is not a nil-pointer, it’s a constructor)

Pointer types

- *pointer* type: notation in C: `int*`
- “ * ”: can be seen as type constructor

```
int* p;
```

- random other languages: $\hat{\text{integer}}$ in Pascal, `int ref` in ML
- value: *address* of (or reference/pointer to) values of the underlying type
- operations: *dereferencing* and determining the address of an data item (and C allows “ *pointer arithmetic* ”)

```
var a:  $\hat{\text{integer}}$  (* pointer to an integer *)
var b: integer
...
a := &i           (* i an int var *)
                  (* a := new integer ok too *)
b :=  $\hat{a}$  + b
```



Implicit dereferencing

- many languages: more or less hide existence of pointers
- cf. reference vs. value types often: automatic/implicit dereferencing

```
C r;  
C r = new C();
```

- “sloppy” speaking: “ `r` is an object (which is an instance of class `C` /which is of type `C`)”,
- slightly more precise: variable “ `r` contains an object... ”
- precise: “variable `r` will contain a reference to an object”
- `r.field` corresponds to something like “
`(*r).field`, similar in Simula



Programming with pointers

- “popular” source of errors
- test for non-null-ness often required
- explicit pointers: can lead to problems in block-structured language (when handled non-expertly)
- watch out for parameter passing
- aliasing
- null-pointers: “the billion-dollar-mistake”
- take care of concurrency



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Function variables



INF5110 –
Compiler
Construction

```
program Funcvar;
var pv : Procedure (x: integer); (* procedur var *)

  Procedure Q();
  var
    a : integer;
    Procedure P(i : integer);
    begin
      a:= a+i; (* a def'ed outside *)
    end;
  begin
    pv := @P; (* ``return'' P (as side effect) *)
  end; (* "@" dependent on dialect *)
begin (* here: free Pascal *)
  Q();
  pv(1);
end.
```

Introduction

Various types and
their
representation

Equality of types

Type checking

Function variables and nested scopes



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

- tricky part here: nested scope + function definition *escaping* surrounding function/scope.
- here: inner procedure “returned” via assignment to function variable
- **stack discipline** of dynamic memory management?
- related also: functions allowed as return value?
 - Pascal: not directly possible (unless one “returns” them via function-typed reference variables like here)
 - C: possible, but *nested* function definitions not allowed
- combination of nested function definitions and functions as official return values (and arguments): *higher-order functions*
- Note: functions as arguments less problematic than as return values.

Function signatures

- define the “header” (also “signature”) of a function³
- in the discussion: we don't distinguish mostly: functions, procedures, methods, subroutines.
- functional type (independent of the name f): $\text{int} \rightarrow \text{int}$

Modula-2

```
var f: procedure (integer): integer;
```

C

```
int (*f) (int)
```

- *values*: all functions (procedures ...) with the given signature
- problems with block structure and free use of procedure variables.

³Actually, an identifier of the function is mentioned as well.

Escaping

```
1 program Funcvar;  
2 var pv : Procedure (x: integer); (* procedur var *)  
3  
4 Procedure Q();  
5 var  
6     a : integer;  
7     Procedure P(i : integer);  
8     begin  
9         a:= a+i; (* a def'ed outside *)  
10    end;  
11    begin  
12        pv := @P; (* ``return'' P (as side effect) *)  
13    end; (* "@" dependent on dialect *)  
14 begin (* here: free Pascal *)  
15     Q();  
16     pv(1);  
17 end.
```

- at the end of line 15: variable a no longer exists
- possible safe usage: only assign to such variables (here pv) a new value (= function) at the same blocklevel the variable is declared

Classes and subclasses



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Parent class

```
class A {  
  int i;  
  void f() {...}  
}
```

Subclass B

```
class B extends A {  
  int i  
  void f() {...}  
}
```

Subclass C

```
class C extends A {  
  int i  
  void f() {...}  
}
```

- classes resemble records, and subclasses variant types, but additionally
 - visibility: local methods possible (besides fields)
 - subclasses
 - objects mostly created dynamically, *no* references into the stack
 - subtyping and polymorphism (subtype polymorphism): a reference typed by *A* can also point to *B* or *C* objects
- special problems: not really many, nil-pointer still possible

Access to object members: late binding

- notation $rA.i$ or $rA.f()$
- dynamic binding, late-binding, virtual access, dynamic dispatch ...: all mean roughly the same
- central mechanism in many OO language, in connection with inheritance

Virtual access $rA.f()$ (methods)

“deepest” f in the run-time class of the *object*, rA points to

- remember: “most-closely nested” access of variables in nested lexical block
- Java:
 - methods “in” objects are only dynamically bound (but there are class methods too)
 - instance variables not, neither static methods “in” classes.



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Example: fields and methods



INF5110 –
Compiler
Construction

```
public class Shadow {
    public static void main(String [] args){
        C2 c2 = new C2();
        c2.n();
    }
}

class C1 {
    String s = "C1";
    void m () {System.out.print(this.s);}
}

class C2 extends C1 {
    String s = "C2";
    void n () {this.m();}
}
```

Introduction

Various types and
their
representation

Equality of types

Type checking



- *Overloading*
 - common for (at least) standard, built-in operations
 - also possible for user defined functions/methods ...
 - disambiguation via (static) types of arguments
 - “ad-hoc” polymorphism
 - implementation:
 - put types of parameters as “part” of the name
 - look-up gives back a set of alternatives
- type-conversions: can be problematic in connection with overloading
- (generic) polymorphism
`swap(var x, y: anytype)`



Section

Equality of types

Chapter 7 “Types and type checking”

Course “Compiler Construction”

Martin Steffen

Spring 2021

Classes as types



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

- classes = types? Not so fast
- more precise view:
 - design decision in Java and similar languages (but not all/even not all class-based OOLs): that class *names* are used in the *role of (names of) types*.
- other roles of classes (in class-based OOLs)
 - generator of objects (via constructor, again with the same name)⁴
 - containing *code* that implements the instances

```
C x = new C ()
```

⁴Not for Java's *static* classes etc, obviously.

Example with interfaces



INF5110 –
Compiler
Construction

```
interface I1 { int m (int x); }
interface I2 { int m (int x); }
class C1 implements I1 {
    public int m(int y) {return y++; }
}
class C2 implements I2 {
    public int m(int y) {return y++; }
}

public class Noduck1 {
    public static void main(String [] arg) {
        I1 x1 = new C1();           // I2 not possible
        I2 x2 = new C2();
        x1 = x2;                    // ???
    }
}
```

Analogous when using classes in their roles as types

Introduction

Various types and
their
representation

Equality of types

Type checking

When are 2 types “equal”?

- *type equivalence*
- surprisingly *many* different answers possible
- implementor’s focus (deprecated): type `int` and `short` are equal, because they “are” both 2 byte
- type checker must often decide such equivalences
- related to a more fundamental question: what’s a type?

Example: pairs of integers

```
type pair_of_ints = int * int;;  
let x : pair_of_ints = (1,4);;
```

Questions

- Is “the” type of (values of) `x` `pair_of_ints`, or
- the product type `int * int`, or
- both, as they are equal, i.e., `pair_of_int` is an abbreviation of the product type (*type synonym*)?



Structural vs. nominal equality



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

a, b

```
var a, b: record  
  int i;  
  double d  
end
```

c

```
var c: record  
  int i;  
  double d  
end
```

typedef

```
typedef idRecord: record  
  int i;  
  double d  
end
```

```
var d: idRecord;  
var e: idRecord;;
```

what's possible?

```
a := c;  
a := d;  
  
a := b;  
d := e;
```

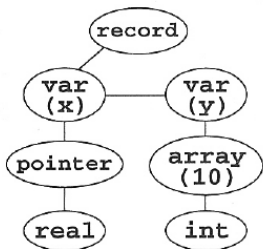


Types in the AST

- types are part of the syntax, as well
- represent: either in a separate symbol table, or part of the AST

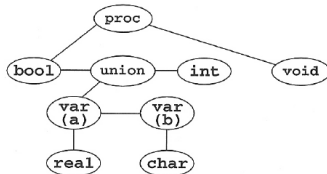
Record type

```
record  
  x: pointer to real;  
  y: array [10] of int  
end
```



Procedure header

```
proc( bool ,  
      union a: real; b: char end ,  
      int ) : void  
end
```



Structured types without names



INF5110 –
Compiler
Construction

var-decls → *var-decls* ; *var-decl* | *var-decl*
var-decl → **id** : *type-exp*
type-exp → *simple-type* | *structured-type*
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array** [*num*] : *type-exp*
| **record** *var-decls* **end**
| **union** *var-decls* **end**
| **pointerto** *type-exp*
| **proc** (*type-exps*) *type-exp*
type-exps → *type-exps* , *type-exp* | *type-exp*

Introduction

Various types and
their
representation

Equality of types

Type checking

Structural equality

```
function typeEqual (t1, t2 : TypeExp) : Boolean;  
var temp : Boolean ;  
    p1, p2 : TypeExp ;  
begin  
    if t1 and t2 are of simple type then return t1 = t2  
    else if t1.kind = array and t2.kind = array then  
        return t1.size = t2.size and typeEqual (t1.child1, t2.child1)  
    else if t1.kind = record and t2.kind = record  
        or t1.kind = union and t2.kind = union then  
begin  
    p1 := t1.child1 ;  
    p2 := t2.child1 ;  
    temp := true ;  
    while temp and p1 ≠ nil and p2 ≠ nil do  
        if p1.name ≠ p2.name then  
            temp := false  
        else if not typeEqual ( p1.child1 , p2.child1 )  
            then temp := false  
        else begin  
            p1 := p1.sibling ;  
            p2 := p2.sibling ;  
        end;  
    return temp and p1 = nil and p2 = nil ;  
end  
    else if t1.kind = pointer and t2.kind = pointer then  
        return typeEqual ( t1.child1 , t2.child1 )  
    else if t1.kind = proc and t2.kind = proc then  
begin  
    p1 := t1.child1 ;  
    p2 := t2.child1 ;  
    temp := true ;  
    while temp and p1 ≠ nil and p2 ≠ nil do  
        if not typeEqual ( p1.child1 , p2.child1 )  
            then temp := false  
        else begin  
            p1 := p1.sibling ;  
            p2 := p2.sibling ;  
        end;  
    return temp and p1 = nil and p2 = nil  
        and typeEqual (t1.child2 , t2.child2)  
end  
    else return false ;  
end ; (* typeEqual *)
```

Test av om to typer er like
(struktur-likhet)
ved rekursiv gjennomgang

Rekursive kall

Om også navnelikhet
er lov, skal dette med



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Types with names

<i>var-decls</i>	→	<i>var-decls ; var-decl</i> <i>var-decl</i>	
<i>var-decl</i>	→	id : <i>simple-type-exp</i>	
<i>type-decls</i>	→	<i>type-decls ; type-decl</i> <i>type-decl</i>	
<i>type-decl</i>	→	id = <i>type-exp</i>	
<i>type-exp</i>	→	<i>simple-type-exp</i> <i>structured-type</i>	
<i>simple-type-exp</i>	→	<i>simple-type</i> id	identifiers
<i>simple-type</i>	→	int bool real char void	
<i>structured-type</i>	→	array [<i>num</i>] : <i>simple-type-exp</i>	
		record <i>var-decls</i> end	
		union <i>var-decls</i> end	
		pointerto <i>simple-type-exp</i>	
		proc (<i>type-exps</i>) <i>simple-type-exp</i>	
<i>type-exps</i>	→	<i>type-exps</i> , <i>simple-type-exp</i>	
		<i>simple-type-exp</i>	

Name equality

- all types have “names”, and two types are equal iff their names are equal
- type equality checking: obviously simpler
- of course: type names may have *scopes*. . . .

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;  
var temp : Boolean ;  
    p1, p2 : TypeExp ;  
begin  
    if t1 and t2 are of simple type then  
        return t1 = t2  
    else if t1 and t2 are type names then  
        return t1 = t2  
    else return false ;  
end;
```



Type aliases

- languages with type aliases (type synonyms): C, Pascal, ML
- often very convenient (`type Coordinate = float * float`)
- light-weight mechanism

type alias; make t_1 known also under name t_2

```
t2 = t1 // t2 is the ``same type``.
```

- also here: different choices wrt. *type equality*



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Type aliases: different choices



INF5110 –
Compiler
Construction

Alias, for simple types

```
t1 = int;  
t2 = int;
```

- often: $t1$ and $t2$ are the “same” type

Alias of structured types

```
t1 = array [10] of int;  
t2 = array [10] of int;  
t3 = t2
```

- mostly $t3 \neq t1 \neq t2$

Introduction

Various types and
their
representation

Equality of types

Type checking



Section

Type checking

Chapter 7 “Types and type checking”

Course “Compiler Construction”

Martin Steffen

Spring 2021

Type checking of expressions (and statements)

- types of subexpressions must “fit” to the expected types the constructs can operate on
- type checking: top-down and *bottom-up* task
- ⇒ *synthesized* attributes, when using AGs
- Here: using an attribute grammar specification of the type checker
 - type checking conceptually done *while parsing* (as actions of the parser)
 - more common: type checker operates on the AST *after* the parser has done its job
- type **system** vs. type **checker**
 - type system: specification of the rules governing the use of types in a language, type discipline
 - type checker: algorithmic formulation of the type system (resp. implementation thereof)



Grammar for statements and expressions



INF5110 –
Compiler
Construction

$program \rightarrow var\text{-}decls ; stmts$
 $var\text{-}decls \rightarrow var\text{-}decls ; var\text{-}decl \mid var\text{-}decl$
 $var\text{-}decl \rightarrow \mathbf{id} : type\text{-}exp$
 $type\text{-}exp \rightarrow \mathbf{int} \mid \mathbf{bool} \mid \mathbf{array} [num] : type\text{-}exp$
 $stmts \rightarrow stmts ; stmt \mid stmt$
 $stmt \rightarrow \mathbf{if} \ exp \ \mathbf{then} \ stmt \mid \mathbf{id} := exp$
 $exp \rightarrow exp + exp \mid exp \ \mathbf{or} \ exp \mid exp [exp]$

Introduction

Various types and
their
representation

Equality of types

Type checking

Type checking as semantic rules



INF5110 –
Compiler
Construction

Grammar Rule	Semantic Rules
$var\text{-}decl \rightarrow id : type\text{-}exp$	$insert(id.name, type\text{-}exp.type)$ ◀
$type\text{-}exp \rightarrow int$	$type\text{-}exp.type := integer$
$type\text{-}exp \rightarrow bool$	$type\text{-}exp.type := boolean$
$type\text{-}exp_1 \rightarrow array$ $[num] \text{ of } type\text{-}exp_2$	$type\text{-}exp_1.type :=$ $makeTypeNode(array, num.size,$ $type\text{-}exp_2.type)$ ◀
$stmt \rightarrow if\ exp\ then\ stmt$	if not $typeEqual(exp.type, boolean)$ then $type\text{-}error(stmt)$
$stmt \rightarrow id := exp$	if not $typeEqual(lookup(id.name),$ $exp.type)$ then $type\text{-}error(stmt)$
$exp_1 \rightarrow exp_2 + exp_3$	if not ($typeEqual(exp_2.type, integer)$ and $typeEqual(exp_3.type, integer)$) then $type\text{-}error(exp_1)$; $exp_1.type := integer$
$exp_1 \rightarrow exp_2 \text{ or } exp_3$	if not ($typeEqual(exp_2.type, boolean)$ and $typeEqual(exp_3.type, boolean)$) then $type\text{-}error(exp_1)$; $exp_1.type := boolean$
$exp_1 \rightarrow exp_2 [exp_3]$	if $isArrayType(exp_2.type)$ and $typeEqual(exp_3.type, integer)$ then $exp_1.type := exp_2.type.child1$ else $type\text{-}error(exp_1)$
$exp \rightarrow num$	$exp.type := integer$
$exp \rightarrow true$	$exp.type := boolean$
$exp \rightarrow false$	$exp.type := boolean$
$exp \rightarrow id$	$exp.type := lookup(id.name)$ ◀

Introduction

Various types and
their
representation

Equality of types

Type checking

More “modern” presentation

- representation as derivation rules
- Γ : notation for symbol table
 - $\Gamma(x)$: look-up
 - $\Gamma, x : T$: insert
- more compact representation
- one reason: “errors” left implicit.



INF5110 –
Compiler
Construction

Introduction

Various types and
their
representation

Equality of types

Type checking

Type checking (expressions)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{TE-ID} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \text{TE-TRUE} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \text{T-FALS}$$

$$\frac{}{\Gamma \vdash n : \mathbf{int}} \text{TE-NUM}$$

$$\Gamma \vdash n : \mathbf{int}$$

$$\Gamma \vdash \mathit{exp}_2 : \mathbf{array_of } T \qquad \Gamma \vdash \mathit{exp}_3 : \mathbf{int}$$

$$\frac{}{\Gamma \vdash \mathit{exp}_2 [\mathit{exp}_3] : T} \text{TE-ARRAY}$$

$$\Gamma \vdash \mathit{exp}_2 [\mathit{exp}_3] : T$$

$$\Gamma \vdash \mathit{exp}_1 : \mathbf{bool} \qquad \Gamma \vdash \mathit{exp}_2 : \mathbf{bool}$$

$$\frac{}{\Gamma \vdash \mathit{exp}_1 \mathbf{or } \mathit{exp}_2 : \mathbf{bool}} \text{TE-OR}$$

$$\Gamma \vdash \mathit{exp}_1 \mathbf{or } \mathit{exp}_2 : \mathbf{bool}$$

$$\Gamma \vdash \mathit{exp}_1 : \mathbf{int} \qquad \Gamma \vdash \mathit{exp}_2 : \mathbf{int}$$

$$\frac{}{\Gamma \vdash \mathit{exp}_1 + \mathit{exp}_2 : \mathbf{int}} \text{TE-PLUS}$$

$$\Gamma \vdash \mathit{exp}_1 + \mathit{exp}_2 : \mathbf{int}$$

Declarations and statements

$\frac{\Gamma, x : \text{int} \vdash \text{rest} : \text{ok}}{\Gamma \vdash x : \mathbf{int}; \text{rest} : \text{ok}}$	$\frac{\Gamma, x : \text{bool} \vdash \text{rest} : \text{ok}}{\Gamma \vdash x : \mathbf{bool}; \text{rest} : \text{ok}}$
$\frac{\Gamma \vdash \text{num} : \text{int} \quad \Gamma(\text{type-exp}) = T \quad \Gamma, x : \mathbf{array} \text{ num of } T \vdash \text{rest} : \text{ok}}{\Gamma \vdash x : \mathbf{array} [\text{num}] : \text{type-exp}; \text{rest} : \text{ok}}$	
$\frac{\Gamma \vdash x : T \quad \Gamma \vdash \text{exp} : T}{\Gamma \vdash x := \text{exp} : \text{ok}}$	$\frac{\Gamma \vdash \text{exp} : \text{bool} \quad \Gamma \vdash \text{stmt} : \text{ok}}{\Gamma \vdash \mathbf{if} \ \text{exp} \ \mathbf{then} \ \text{stmt} : \text{ok}}$
$\frac{\Gamma \vdash \text{stmt}_1 : \text{ok} \quad \Gamma \vdash \text{stmt}_2 : \text{ok}}{\Gamma \vdash \text{stmt}_1 ; \text{stmt}_2 : \text{ok}}$	

References I

Bibliography



**INF5110 –
Compiler
Construction**

Introduction

**Various types and
their
representation**

Equality of types

Type checking