Course Script

INF 5110: Compiler construction

INF5110, spring 2018

Martin Steffen

Contents

Chapter 7 Types and type checking

Learning Targets of this Chapter

1. the concept of types
2. specific common types
3. type safety
4. type checking
5. polymorphism, subtyping and other complications

Contents

What is it about?

7.1 Introduction

This chapter deals with "types". Since the material is presented as part of the static analysis (or semantic analysis) phase of the compiler, we are dealing mostly with *static* aspects of types (i.e., static typing etc).

The notion of "type" is **very** broad and has many different aspects. The study of "types" is a research field in itself ("type theory"). In some way, types and type checking is the very essence of semantic analysis, insofar that types can be very "expressive" and can be used to represent vastly many different aspects of the behavior of a program. By "more expressive" I mean types that express much more complex one than the ones standard programmers are familiar with: booleans, integers, structured types, etc. When increasing the "expressivity", types might not only capture more complex situations (like types for higher-order functions), but also unusual aspects, not normally connected with types, like for instance: bound on memory usage, guarantees of termination, assertions about secure information flow (like no information leakage), and many more.

As a final random example: a language like *Rust* is known for its non-standard form of memory management based on the notion of *ownership* to a piece of data. Ownership tells who has the right to access the data when and how, and that's important to know as as simultaneous write access leads to trouble. Regulating ownership can and has been formulated by corresponding "ownership type systems" where the type expresses properties concerning ownership.

That should give a feeling that, with the notion of types such general, the situation is a bit as with "attributes" and attribute grammars: "everything" may be an attribute since an attribute is nothing else than a "property". The same holds for types. With a loose interpretation like that, types may represent basically all kinds of concepts: like, when interested in property "A", let's intoduce the notion of "A"-types (with "A" standing for memory consumption, ownership, and what not). But still: studying type systems and their expressivity and application to programming languages seems a much *broader* and *deeper* (and more practical) field than the study of attribute grammars. By more practical, I mean: while attribute grammars certainly have useful applications, stretching them to new "non-standard" applications may be possible, but it's, well, stretching it.[1] Type systems, on the other hand, span more easily form very simple and practical usages to very expressive and foundational logical system.

In this lecture, we keep it more grounded and mostly deal with concrete, standard (i.e., not very esoteric) types. Simple or "complicated" types, there are at least two aspects of a type. One is, what a user or programmer sees or is exposed to. The second one is the inside view of the compiler writer. The user may be informed that it's allowed to write `x + y` where `x` and `y` are both integers (carrying the type `int`), or both strings, in which case + represents string addition. Or perhaps the language even allows that one variable contains a string and the other an integer, in which case the + is still string concatenation, where the integer valued operand has to be converted to its string representation. The compiler writer needs then to find representations in memory for those data types (ultimately in binary form) that actually *realize* the operations described above on an abstract level. That means choosing an appropriate encoding, choosing the right amount of memory (long ints need more space than short ints, etc, perhaps even depending on the platform), and making sure that needed conversions (like from integers to string) actually are done in the compiled code (most likely arranged statically). Of course, the programmer does not want to know those details, he typically could not care less, for instance, whether the machine architecture is "little-endian" or "big-endian" (see `https://en.wikipedia.org/wiki/Endianness`). But the compiler writer will have to care when writing the compiler itself to *represent* or *encode* what the programmer calls "an integer" or "a string". So, apart from the more esoteric and advanced roles types play in programming languages, perhaps the most fundamental role is that of **abstraction**: to shield the programmer from the dirty details of the actual representation.

Types are a central abstraction for programmers.

Abstraction in the sense of hiding underlying representational details.[2]

The lecture will have some look at both aspects of type systems. One is the representational aspect. That one is more felt in languages like C, which is closer to the operating system and to memory in hardware than languages that came later. Besides that, we will also more look at type system as *specification* of what is allowed at the programmer's level ("is it allowed to do a + on an a value of `integer` type and of `string` type?"), i.e., how to specify a type system in a programming language independent from the question how to choose proper lower-level encodings that the abstraction specified in the type system.

[1] That's at least my slightly biased opinion.

[2] Beside that practical representational aspect, types are also an abstraction in the sense that they can be viewed as the "set" of all the values of that given type. Like `int` represents the set of all integers. Both views are consistent as all members of the "set" `int` are consistently represented in memory and consistently treated by functions operating on them. That "consistency" allows us as programmers to think of them as integers, and forget about details of their representation, and it's the task of the compiler writer, to reconcile those two views: *the low-level encoding must maintain the high-level abstraction.*

General remarks and overview

- 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 and 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, types were part of PLs from the start.

Milner's dictum ("type safety")

Well-typed programs cannot go wrong!

- *strong* typing:[3] rigourously prevent "misuse" of data
- types useful for later phases and optimizations
- documentation and partial specification

In contrast to (standard) types: many other abstractions in SA (like the control-flow graph or data flow analysis and others) are not directly visible in the source code. However, in the light of the introductory remarks that "types" can capture a very broad spektrum of semantic properties of a language if one just makes the notion of type general enough ("ownership", "memory consumption"), it should come as no surprise that one can capture *data flow* in appropriately complex type systems, as well…

Besides that: there are not really any *truly* untyped languages around, there is always some discipline (beyond syntax) on what a programmer is allowed to do and what not. Probably the anarchistic recipe of "anything (syntactically correct) goes" tends to lead to desaster or is too complex to implement in a rational and understandable manner. Note that "dynamically typed" or "weakly typed" is not the same as "untyped".

[3]Terminology rather fuzzy, and perhaps changed a bit over time. Also what "rigorous" means.

Types: in first approximation

Conceptually

- semantic view: A 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 compex types from simpler ones
- reference vs. value types

7.2 Various types and their representation

Some typical base types

	base types		
`int`	`0, 1, ...`	`+, −, *, /`	integers
`real`	`5.05E4 ...`	`+, −, *`	real numbers
`bool`	`true, false`	`and or (|) ...`	booleans
`char`	`'a'`		characters
⋮			

- 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

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

- 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

Being a "conceptual" view means, it's about the "interface", it's an abstract view of how one can make *use* of members of a type. It not about implementation details, like "integers are 2 bit-bit-words in such-and-such representation". See also the notion of abstract data type on the next slide.

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
```

Type constructors: building new types

- array type
- record type (also known as struct-types)
- union type
- pair/tuple type
- pointer type
  - explict as in C
  - implict 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
- ...

Basically all languages support to build more complex types from the basic one and ways to use and check them. Sometimes it's not even very visible, for instance, one may already see strings as compound. For instance in C, which takes a very implementation-centric view on types, explains strings as one-dimensional array of characters terminated by a null character '\0'

Of course, there is special syntax to build values of type string, writing `"abc"` as opposed to `string-cons('a, string_cons('b, ...))` or similar... This smooth support of working with strings may make them feel as if being primitive.

In the following we will have a look at a few of composed types in programming languages. The compila language of the oblig this year supports records but also "names" of records. We will also discuss the issue of "types as such" vs. "names of types" later (for instance in connection with the question how to "compare types: when are they equal or compatible, what about subtping? etc.).

Arrays

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)?

Integer-indexed arrays are typically a very efficent data structure, as they mirror the layout of standard random access memory and customary hardware.[4] Indeed, contiguous random-access memory can be seen as one big array of "cells" or "words" and standard hardward *supports* fast access to to those cells by indirect addressing modes (like making use of an off-set from a base address, even multiplied by a factor).

One and more-dimensional arrays

- one-dimensional: effienctly 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)

[4]There exist unconventional hardware memory architectures which are *not* accessed via addresses, like content-addressable memory. Those don't resemble "arrays". They are a specialist niche, but have applications.

Records (“structs”)

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

- values: “labelled tuples” (`real`× `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`
- implemenation: linear memory layout given by the (types of the) attributes
- attributes accessible by statically fixed *offsets*
- *fast* access
- cf. objects as in Java

Structs in C

The definition, declaration etc. of struct types and structs in C is slightly confusing. For one thing, in

```
struct Foo { // Foo is called a ``tag''
  real r;
  int  i
```

The `foo` is a *tag*, which is almost like a type, but not quite, at least as C concerned (i.e. the definition of C distinguishes even it is not so clear why). Technically, for instance, the name space for tags is different from that for types. Ignoring details, one can make use of the tag almost as if it were a type, for instance,

```
struct foo b
```

declares the structure `b` to adhere to the struct type tagged by `foo`. Since `foo` is not a proper type, what is illegal is a declaration such as `foo b`. In general the question whether one should use `typedef` in commbination with struct tags (or *only* `typedef`, leaving out the tag), seems a matter of debate. In general, the separation between tags and types (resp. type names) is a messy, ill-considered design.

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
`(1, "text", true)`	`int * string * bool`
`(1, ("text", true))`	`int * (string * bool)`

- 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")

The two "triples" and their types touches upon an issue discussed later, namely when are two types equal (and related to that, whether or not the corresponding values (here the "triples") are equal.

Union types (C-style again)

```
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`

Union types in C and type safety

- union types is 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 above example: `real` takes more space than `int`
- role of type here is more: implementor's (= low level) focus and memory allocation need, 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[5]

Variant records from Pascal

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

- “variant record”
- non-overlapping memory layout[6]
- programmer responsible to set and check the “discriminator” self
- enforcing type-safety-wise: not really an improvement :-(

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

Pointer types

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

```
int* p;
```

- random other languages: `^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:  ^integer  (* pointer to an integer    *)
var b:   integer
...
a := &i           (* i an int var             *)
                  (* a :=  new integer ok too *)
b:= ^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();
```

[5]Basically: it’s union types done right plus possibility of “recursion”.
[6]Again, it’s a implementor-centric, not user-centric view

- "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

Function variables

```
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.
```

Function variables and nested scopes

- tricky part here: nested scope + function definition *escaping* surrounding function/scope.
- here: inner procedure "returned" via assignment to function variable[7]
- think about *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.

[7]For the sake of the lecture: Let's not distinguish conceptually between functions and procedures. But in Pascal, a procedure does not return a value, functions do.

Function signatures

- define the “header” (also “signature”) of a function[8]
- in the discussion: we don’t distinguish mostly: functions, procedures, methods, subroutines.
- functional type (independent of the name f): `int→int`

Modula-2

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

C

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

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

Escaping: function var’s outside the block structure

```
 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 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
- note: function *parameters* less problematic (stack-discipline still doable)

[8] Actually, an identfier of the function is mentioned as well.

Classes and subclasses

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 (independent from the *static* class type of `rA`.

- 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.

Example: fields and methods

```
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();}
}
```

Inductive types in ML and similar

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

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

- allows *recursive* definitions ⇒ 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

does not work

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

“indirect” recursion

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

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-point, it’s a constructor)

7.3 Equality of types

Classes as types

- 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)[9]
  - containing *code* that implements the instances

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

[9]Not for Java’s *static* classes etc, obviously.

Example with interfaces

```
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 effects when using classes in their roles as types

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 "equivalence"
- 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
- is "the" type of (values of) x the product type `int * int` ,
- or both, because they are equal, i.e., `pair_of_int` simply an abbreviation of the product type (*type synonym*)?

Structural vs. nominal equality

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?

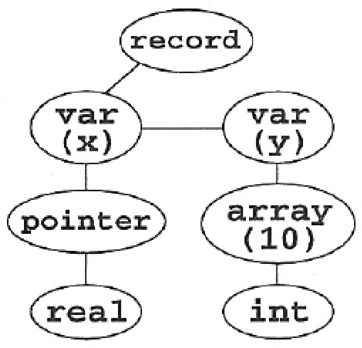
```

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

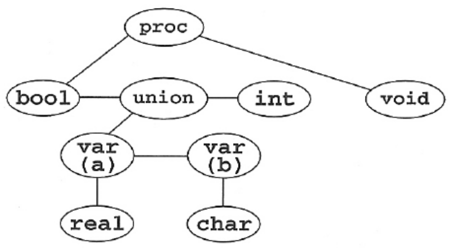
Record type

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



Procedure header

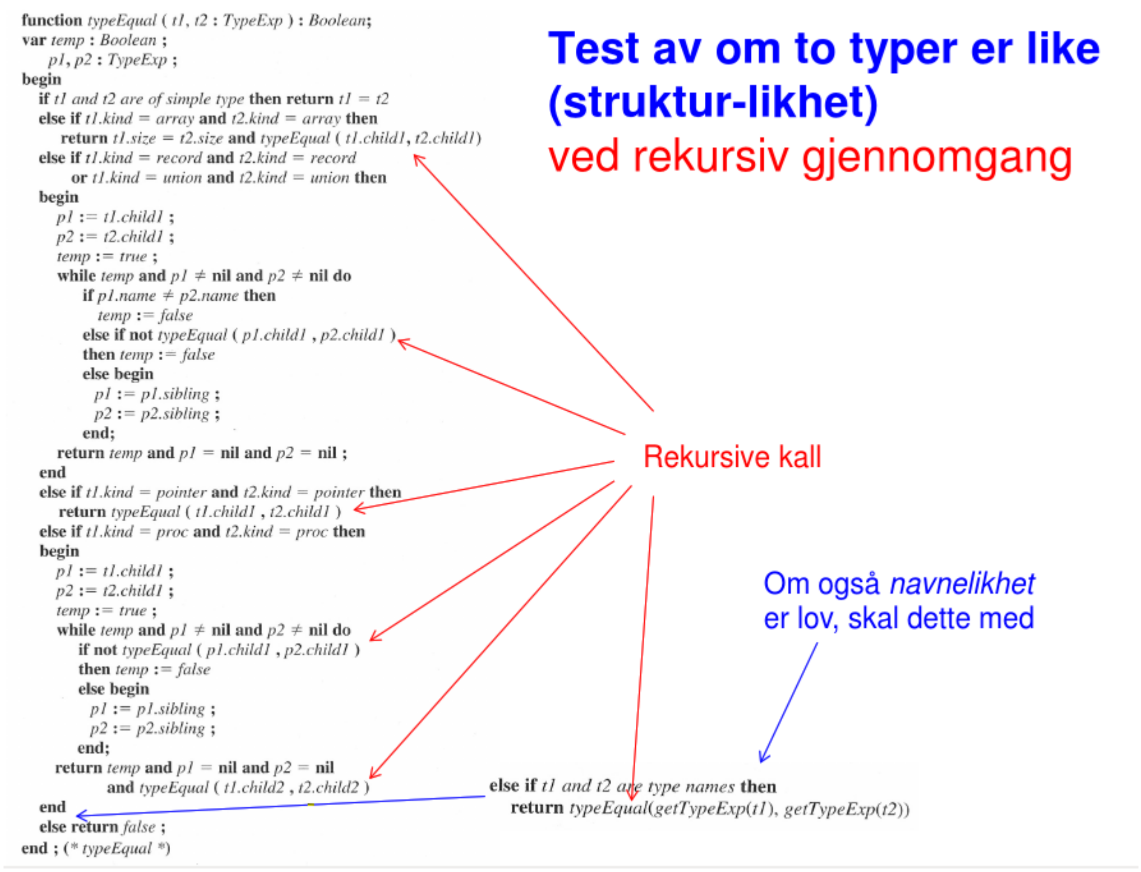
```
proc(bool,
      union a: real; b:char end,
      int):void
 end
```



Structured types without names

$$\begin{array}{rcl}
\textit{var-decls} & \to & \textit{var-decls}\,\textbf{;}\,\textit{var-decl} \mid \textit{var-decl} \\
\textit{var-decl} & \to & \textbf{id}\,\textbf{:}\,\textit{type-exp} \\
\textit{type-exp} & \to & \textit{simple-type} \mid \textit{structured-type} \\
\textit{simple-type} & \to & \textbf{int} \mid \textbf{bool} \mid \textbf{real} \mid \textbf{char} \mid \textbf{void} \\
\textit{structured-type} & \to & \textbf{array}\,[\,\textit{num}\,]:\textit{type-exp} \\
 & \mid & \textbf{record}\ \textit{var-decls}\ \textbf{end} \\
 & \mid & \textbf{union}\ \textit{var-decls}\ \textbf{end} \\
 & \mid & \textbf{pointerto}\ \textit{type-exp} \\
 & \mid & \textbf{proc}\,\textbf{(}\,\textit{type-exps}\,\textbf{)}\,\textit{type-exp} \\
\textit{type-exps} & \to & \textit{type-exps}\,\textbf{,}\,\textit{type-exp} \mid \textit{type-exp}
\end{array}$$

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 *)
```

```
else if t1 and t2 are type names then
   return typeEqual(getTypeExp(t1), getTypeExp(t2))
```

Types with names

$$
\begin{array}{rcll}
\mathit{var\text{-}decls} & \rightarrow & \mathit{var\text{-}decls}\,;\mathit{var\text{-}decl} \mid \mathit{var\text{-}decl} & \\
\mathit{var\text{-}decl} & \rightarrow & \mathbf{id}:\mathit{simple\text{-}type\text{-}exp} & \\
\mathit{type\text{-}decls} & \rightarrow & \mathit{type\text{-}decls}\,;\mathit{type\text{-}decl} \mid \mathit{type\text{-}decl} & \\
\mathit{type\text{-}decl} & \rightarrow & \mathbf{id} = \mathit{type\text{-}exp} & \\
\mathit{type\text{-}exp} & \rightarrow & \mathit{simple\text{-}type\text{-}exp} \mid \mathit{structured\text{-}type} & \\
\mathit{simple\text{-}type\text{-}exp} & \rightarrow & \mathit{simple\text{-}type} \mid \mathbf{id} & \text{identifiers} \\
\mathit{simple\text{-}type} & \rightarrow & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{real} \mid \mathbf{char} \mid \mathbf{void} & \\
\mathit{structured\text{-}type} & \rightarrow & \mathbf{array}\,[\,\mathit{num}\,]:\mathit{simple\text{-}type\text{-}exp} & \\
 & \mid & \mathbf{record}\ \mathit{var\text{-}decls}\ \mathbf{end} & \\
 & \mid & \mathbf{union}\ \mathit{var\text{-}decls}\ \mathbf{end} & \\
 & \mid & \mathbf{pointerto}\ \mathit{simple\text{-}type\text{-}exp} & \\
 & \mid & \mathbf{proc}\,(\,\mathit{type\text{-}exps}\,)\,\mathit{simple\text{-}type\text{-}exp} & \\
\mathit{type\text{-}exps} & \rightarrow & \mathit{type\text{-}exps}\,,\mathit{simple\text{-}type\text{-}exp} & \\
 & \mid & \mathit{simple\text{-}type\text{-}exp} &
\end{array}
$$

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 `t1` known also under name `t2`

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

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

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`

7.4 Type checking

Type checking of expressions (and statements)

- types of subexpressions must "fit" to the expected types the contructs can operate on[10]
- type checking: a *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)
  - also common: type checker operates on the AST *after* the parser has done its job[11]
- 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)

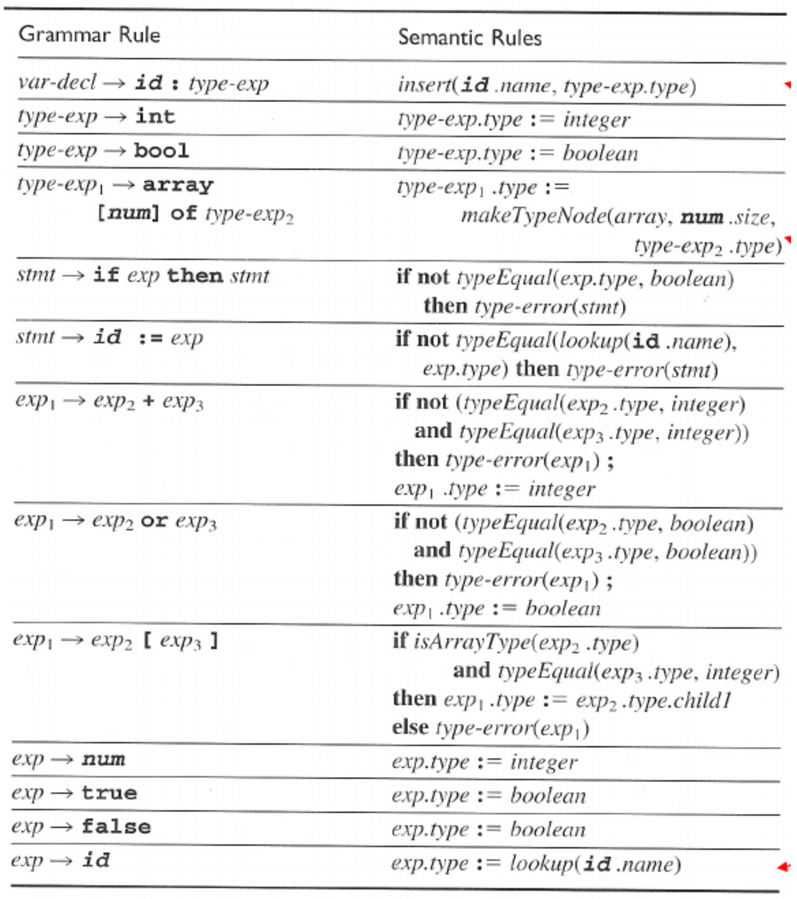
Grammar for statements and expressions

$$
\begin{array}{rcl}
\mathit{program} & \to & \mathit{var\text{-}decls}\,;\mathit{stmts} \\
\mathit{var\text{-}decls} & \to & \mathit{var\text{-}decls}\,;\mathit{var\text{-}decl} \mid \mathit{var\text{-}decl} \\
\mathit{var\text{-}decl} & \to & \mathbf{id}:\mathit{type\text{-}exp} \\
\mathit{type\text{-}exp} & \to & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{array}\,[\,\mathit{num}\,]:\mathit{type\text{-}exp} \\
\mathit{stmts} & \to & \mathit{stmts}\,;\mathit{stmt} \mid \mathit{stmt} \\
\mathit{stmt} & \to & \mathbf{if}\ \mathit{exp}\ \mathbf{then}\ \mathit{stmt} \mid \mathbf{id} := \mathit{exp} \\
\mathit{exp} & \to & \mathit{exp} + \mathit{exp} \mid \mathit{exp}\ \mathbf{or}\ \mathit{exp} \mid \mathit{exp}\,[\,\mathit{exp}\,]
\end{array}
$$

[10] In case (operator) overloading: that may complicate the picture slightly. Operators are selected depending on the type of the subexpressions.

[11] one can, however, use grammars as specification of that *abstract* syntax tree as well, i.e., as a "second" grammar besides the grammar for concrete parsing.

Type checking as semantic rules



Grammar Rule	Semantic Rules
var-decl → **id :** *type-exp*	*insert*(**id** .*name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
type-exp$_1$ → **array** **[num] of** *type-exp*$_2$	*type-exp*$_1$.*type* := *makeTypeNode*(*array*, **num** .*size*, *type-exp*$_2$.*type*)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp.type*, *boolean*) **then** *type-error*(*stmt*)
stmt → **id** **:=** *exp*	**if not** *typeEqual*(*lookup*(**id** .*name*), *exp.type*) **then** *type-error*(*stmt*)
exp$_1$ → *exp*$_2$ **+** *exp*$_3$	**if not** (*typeEqual*(*exp*$_2$.*type*, *integer*) **and** *typeEqual*(*exp*$_3$.*type*, *integer*)) **then** *type-error*(*exp*$_1$) ; *exp*$_1$.*type* := *integer*
exp$_1$ → *exp*$_2$ **or** *exp*$_3$	**if not** (*typeEqual*(*exp*$_2$.*type*, *boolean*) **and** *typeEqual*(*exp*$_3$.*type*, *boolean*)) **then** *type-error*(*exp*$_1$) ; *exp*$_1$.*type* := *boolean*
exp$_1$ → *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-error*(*exp*$_1$)
exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** .*name*)

Type checking (expressions)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{TE-Id} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \texttt{bool}}\ \text{TE-True} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \texttt{bool}}\ \text{T-False}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \text{TE-Num} \qquad \frac{\Gamma \vdash exp_2 : \texttt{array\_of}T \qquad \Gamma \vdash exp_3 : \texttt{int}}{\Gamma \vdash exp_2\,[\,exp_3\,] : T}\ \text{TE-Array}$$

$$\frac{\Gamma \vdash exp_1 : \texttt{bool} \qquad \Gamma \vdash exp_3 : \texttt{bool}}{\Gamma \vdash exp_2\ \mathbf{or}\ exp_3 : \texttt{bool}}\ \text{TE-Or}$$

$$\frac{\Gamma \vdash exp_1 : \texttt{int} \qquad \Gamma \vdash exp_3 : \texttt{int}}{\Gamma \vdash exp_3 + exp_3 : \texttt{int}}\ \text{TE-Plus}$$

Diverse notions

- *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) polymporphism
  `swap(var x,y:  anytype)`

Index