Course Script

INF 5110: Compiler construction

INF5110, spring 2024

Martin Steffen

Contents

7 Types and type checking **1**
7.1 Introduction . . . 1
7.2 Various types (and their representation) . . . 4
7.2.1 Some typical basic types . . . 4
7.2.2 Some compound types . . . 6
7.2.3 Abstract data types . . . 7
7.2.4 Type constructors: building new types . . . 10
7.2.5 Arrays . . . 10
7.2.6 Record (“structs”) . . . 12
7.2.7 Tuple or product types . . . 13
7.2.8 Typing alternatives: Union types, sum types, and inductive data types . . . 15
7.2.9 Recursive and inductive types . . . 18
7.2.10 Pointer and reference types . . . 22
7.2.11 Classes and types . . . 25
7.2.12 Polymorphism . . . 28
7.3 Equality of types . . . 30
7.3.1 Type aliases or synonyms . . . 37
7.4 Type checking . . . 38
7.4.1 General remarks about a type checker . . . 38
7.4.2 Attribute grammar specification . . . 41
7.4.3 Type system given by derivation rules . . . 43

Chapter 7
Types and type checking

What is it about?

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

7.1 Introduction 1
7.2 Various types (and their representation) 4
7.3 Equality of types 30
7.4 Type checking 38

7.1 Introduction

This chapter deals with "types". As 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).

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 properties or attributes 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 aspects, not normally connected with types, like for instance: bounds on memory usage, guarantees of termination, assertions about secure information flow (like no information leakage), and many more. The chapter here focuses on *bread-and-butter types*, like the ones for instance supported by the compila language from the oblig.

In some of last years (2021 and 2020, and 2023), there had been groups doing the oblig in Haskell. Haskell's type system is rather expressive even in its core version. Language extensions allow to do serious steps in the direction of what is called *type-level programming* and programming with *dependent types*. This leads to systems where type inference and other questions become undecidable and the type system starts resembling a specification of the program behavior (expessing non-trivial invariants, etc.). Indeed, a type system fully embracing dependent types is a form of combining computation for programming and logic (for specification) in a common framework.

As a final random example for expressive, not quite run-of-the-mill type systems: a language like *Rust* is known for its non-standard form of memory management based on the notion of *ownership* to pieces of data. Ownership tells who, i.e., which piece of code at a given point, has the right to access the data when and how, and that's important to know 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 introduce 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 *broader* and *deeper* (and more practically relevant) 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 the one from integers to strings) actually are done in the compiled code. Of course, the user of the programming language does not want to know those details, the coder 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, how to **represent** or **encode** what the programmer calls "an integer" or "a string". So, it's fair to say the most fundamental role of types 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 represention al details.[2]

[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.

The lecture will have some look at both aspects of type systems, the "interface" and "abstraction" perspective and the "internal representation" perspective. The representational aspect of types is more felt in languages like C, which is closer to the operating system and to memory in hardware than many other language. Besides that, we will also 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.

Abstracting away from internal, restricting which data can be meaningfully combined with which other, enforcing that restrictions by type checking, and at the same time internally do the proper manipulations at the underlying representations, all that can contribute to more safe and robust programs. All this is captured in a well-known slogan:

Milner's dictum ("type safety")

Well-typed programs cannot go wrong!

What does it mean when saying a language has a "strong" type system, is strongly typed, or is type safe? The latter, type safety, is more clearly defined. It's connected to Milner's dictum. It is meant to be a statement about *static typing* and static type system. Types, as mentioned, can be understood as *abstractions*. As far as static type systems are concerned, these abstractions are also *predictions* of what will happen at run-time. The predictions are not exact, it's approximative. For example, typically the type system cannot determine, which concrete integer will be given back as a result of a function, but it can determine that the result will be an integer, it's just unclear at compile time which it will be. That hangs together with fundamental limitations of what can be algorithmically determined (Halting problem, Rice's theorem). But beside those fundamental limitations, there is another obvious reason. Let's stick with the example of determining what value a function will return. A function will have an input and typically the behavior of the function, in particular the resulting value of the function will *depend* on the input (otherwise, what would be the point of having a function with an input that does not influence the outcome). Not knowing which concrete input to expect statically, implies not knowing what outcome to expect. It's beyond the ambitition and capabilities of standard type systems to be no more specific than saying that, for instance, if the input is an integer, then the output is an integer again. So, in this case, one is dealing with a function from integers to integers. And that's information enough for the compiler, to prepare for enough memory, since no matter what integer it will be called with, they all are represented uniformely.

When claiming that "one cannot be more specific", then that's not actually true: with standard type systems, one cannot be more specific. Of course, if one assumes that the function receives an arbitrary integer as argument, then, to stick with the example, one cannot know what particular integer is returned in most cases. If one had a function like `add (x: int) = x+1`, one knows that if the input is an odd integer, the result is

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.*

even, and vice versa. Unless perhaps a `MAXINT` overflow exception is raised, which might be also something that the type system can specify. The Java type system for instance allows to specify what potential exceptions could occur.

But let's leave the exception-discussion aside, and focus on the even and odd situation. Normally, type systems support integer as types, but not the type of even integers or the odd ones. Or mechanisms for the programmer to define such things like the type of all even numbers when wished. In the extreme a type system could allow to capture on the type level, what specific outcome to expect for specific input. That would lead to what is known as **dependent types** and is beyond this lecture (and the vast majority of current type system for general purpose languages).

In contrast to (standard) types: many other abstractions in static analysis (like the control-flow graph or data-flow analysis and others) are not directly visible in the source code. We called types an important abstraction *for programmers*. Data flow information and other such representations used in semantic analysis are like-wise abstractions and equally important. Though one could say those are mostly not abstractions *for the programmer*, those are abstractions inside the compiler. Many types, in contrast are visible to the programmer.

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 no *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 disaster anyway. Note that "dynamically typed" or "weakly typed" is not the same as "untyped".

7.2 Various types (and their representation)

This section shows a parade of different types, which can be found across many languages (variations apply). Most should be familiar in one form or the other.

7.2.1 Some typical basic types

Let's not define exactly what is or is not a basic type; there is not much insight to gain from that. Let us just discuss aspects of types which we reasonably call basic or elementary, and show common examples.

One aspect is that (elements of) basic types have no sub-parts in the sense that they are composed of other (elements of) more primitive types. For instance the type for pairs of integers, perhaps written `int` × `int`, is not basic, it's composite or compound. That is often hand in hand with the fact that the values belonging to the type in question are structured or not. For instance, a pair $(1, 2)$ consists of the integer 1 in the first position and 2 in the second, and the language will offer possibilities to access those two elements. So, for compound values (which are not elements of basic types), there are ways to *deconstruct*

them. Deconstructing a composite data item means accessing its constituents or sub-parts. The opposite of deconstructing values is, of course, constructing them. In the pair example, there is special syntax (_, _) to construct a pair. It's characteristic for non-basic to have possibilities to construct values and to decompose them again. Integers is an example of a basic type. Though one can make the argument, that internally an integer has parts (like being represented by a sequences of bits), the bits are not seen as being "parts" of the integer, and are not typically accessible at the programming-language level.

There are **corner cases,** depending on particular languages. For instance, `string` may feel like a quite basic type, but actually, for instance C considers strings as compound. C 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 almost as if they were primitive.

Basic types are predefined by the language resp. the compiler. Often, a fair selection of those is provided by the language, like the ones in Table 7.1 (and partly mapped to representations with HW support on a platform). Often they are lexically represented by using reserved keywords, i.e., it's typically not allowed to redefine a type like `bool` to represent something else, even if one believes one can come up with a better implementation of booleans than the one provided (which is highly unlikely anyway...).

Note, being built-in is not the same as basic or elementary. For instance, `List` may be a built-in keyword in the language used in connection with the types of lists. List *values* like `[1;2;3]` are certainly composite (just the empty list `[]` cannot be called "composite" in a meaningful way). By what about `List` as type? `List` as keyword may be predefined, as said, but it's best **not** called as *type* in a strict sense, when using the words carefully.

How comes, does `List` not represents lists as members? Well, there is (often) no type containing lists *in general*, there are only lists of integers, of type `List of int`, lists of booleans, of type `List of bool`, list of lists of pairs of string of type `List of (List of (string * string))`. But is `List of Object` not a type containing lists in general (for instance in Java or similar languages)? That's true, but `List of Object` is not the same as `List` in isolation, and the fact that `List of Object` is a type for (basically) all kind of lists has to do with a further property of type system, sub-type polymorphism (see later) and the fact that `Object` may be the super-type of (almost) all types. The point here is: `List` per se is not a type, basic or otherwise, neither is $\star$ (describing pairs, written also $\times$ in non-ascii), those are examples of **type constructors**. A bit more later.

We hope it's clear enough what basic types conceptually are; let's comment on some of the basic types from Table 7.1. All languages will offer various numeric types, like `int` and `real` or `float`. Those can often rely on some form of HW support. It should be clear, that elements from types like `int` or `real` are not exactly mathematical *integers*

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

Table 7.1: Basic types

or reals from $\mathbb{R}$. The computer-versions all suffer from limited precision.[3] Languages also offer variations of fixed precision, like `int32` and `int64`.

When dealing with different numbers of different precision, they are conceptually all machine-representations of numbers. On the bit-level representations of numbers, numerical operations on, say, `int32` and `int64`, work analogously, but not identically in the sense that one can use exactly the same steps, or at least one has to be careful. On the implementation level, their representations may often be "consistent" to some extent. For instance, an `int32` number padded with leading 32 bits of 0's may be an `int64` representation of the "same" number. Still, it's not the same representation, one uses 64 bits and one 32.

On the level of abstraction of programming source code, one would like to do the normal numerical operations like addition, substraction, etc. *consistently.* Often, the design of the language will use the same (special) syntax for operations on different numerical types. Like "+" in infix notation is a good choice for adding two numbers. On the level of representation, the +-operator is implemented differently on `int32` and on `int64`. The fact that some syntactic construct, like the operator +, is implemented differently when operating on different types is known as **overloading**. In this particular case of *operator* overloading. Often, operators like + cannot only be used on two `int32` numbers or alternatively on two `int64` numbers (or else on two strings, in which case it's interpreted as string concatenation, perhaps). The language may also support mixed-type arguments, like using + on one `int32` and one `int64` argument. That would involve some behind-the-scenes **conversion,** like turning the 32-bit integer representation into a 64-bit one (like padding it with leading 0's in this case, which would in plausible representations of those two numerical data). Of course for the compiler writer, there's no such thing as behind-the-scenes, the compiler is responsible to use or generate appropriate conversion code, and it will consult type information to chose the appropriate conversions. Both overloading and conversions are two forms of type *polymorphism.* A bit more later.

7.2.2 Some compound types

Table 7.2 contains a few common compound types, available in most languages. Especially, when built-in into the language core, as opposed to be available as library functionality,

[3]There is also something called infinite or arbitrary precision arithmetic, but let's not go there.

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

Table 7.2: Compound types

the most common ones are often supported by special syntax. For instance, accessing the 7th slot of an one dimensional array `a` is typically written as `a[6]` (with `a[0]` being the first slot), as opposed to `array_access(a,6)`. Some such data structures may come in a built-in version and in a library version. For instance, there may be a built-in fixed-size standard array data type and a dynamically-sized version with further bells and whistles from the library.

Compound types (or compound data structures), resp. their types, like the ones from Table 7.2, are often **reference types**. What that means is that a variable of the corresponding type, for instance, a variable `a` of type `array[0,..,9] of real` does not "contain" the array, the variable contains a reference to the place where the array is stored. That fact is often suppressed in the syntax. I.e., the type of the mentioned variable is not `ref (array[0,..,9])` which would make that fact explicit in the type. Still, the compiler writer must keep that in mind, and likewise the user of the language. The fact that some data structures are implicitly handled via references makes a difference if the data is shared and modified. That may happen when using the data as argument in function calls. We will discuss the related issue of parameter passing in a later chapter.

7.2.3 Abstract data types

Let's discuss the notion of **abstract data types** on a conceptual level at least. Again,we don't attempt to define what an abstract data type precisely is and what not. One can find many different standpoints or opinions on that, partly contradicting. One finds that classes are (a form of) abstract data types, and one finds that classes are categorically different from ADTs. One finds that ADT are nothing else than modules (if that explains something), one finds that ADTs are multisorted algebras (if that explains something) etc. One encounters such terminological fuzziness not just with the concept of ADTs, it also can be found for other programming language concepts. That's partly due to the fact that sometimes the realization in one or two languages are confused with a general *concept*. For instance, a particular language may support a keyword `module` or `class` for the concept of modules or classes, and the manual of that language describes in detail what that is and what not, and how to use it when programming in that language etc. But that is sometimes just one particular angle or interpretation of a more general concept, leaving out aspects resp., throwing in additional ones in a particular language. And then, programmer used to that language, may contribute to discussions with statements like "Do I know what a module is? Good lord, I have programmed ADA for 30s years, a language

famous for modules, and trust me, I know I what I am talking about and your definition is plain wrong". On the other hand, trying to find out exactly (across all languages) what a concept like ADTs is, is also futile. Therefore we don't attempt that. So much as philosophical *disclaimer* before discussing ADTs (and actually a disclaimer for a number of others concepts covered in the lecture).

Let's mention a few aspects often cited in connection with abstract data types. Let's also not bother to distinguish in detail between the type aspect of an ADT, resp. the implementation thereof. So the word, abstract data *type* is often used not just for the type level information (like the signature) but also for it's implementation, like "this stack is an example for an ADT".

Anyway, what are then ingredients or characteristics of ADTs? One is the typing aspect, and one is **abstraction**, hence the name. Another one is that they **bundling up** data structures together with code operating on the data.

In the light of what we have said about what types are ("central user-level abstractions"), in some sense, all types have aspects of "abstract data types". They provide abstractions on the underlying representation. Additionally, *all* meaningful data comes with operations and functions to work with them, Data without operators *on* it useless. It's not for nothing that lecture IN2010 (formerly INF2020) is not just called *Datastrukturer* but *Algoritmer og datastrukturer*: discussing data structures without algorithms that do something interesting on them makes no sense. Also the basic, primitive types like `int32` or `int64` we discussed earlier come with operations on them (like +). And those operations hide type the internals of the representation, they provide an *abstraction*. In that sense, one might call already that an "abstract data type", a very primitive though.

One will seldom hear that a built-in type like `int32` is an ADT, though. One reason for that is because other important aspects are missing. One is that numbers and their operations are not "bundled up" as one structure with one explicitly given signature or interface.[4] When saying a language supports ADTs one means, the language supports a mechanism for the programmer to **introduce new, custom-made** ADTs. So, `int32` is not user-defined, but built in, and it's not compound, but primitive, so that does not qualify...

To allow to introduce ADTs as a "bundle" consisting of data + operations on it, languages then support corresponding syntax (of various sorts). Like (in some ad-hoc syntax):

```
ADT X
begin
   ....
end
```

Languages may use `module` as keyword. Indeed, classes also bundle up data (the instance variables) with "operations" (the methods), which is why some people claim, classes are object-oriented ADTs (or at least that static classes are modules). In particular, one can make a case to see *static* classes as in Java as a form of ADTs. Some will protest, though,

[4]At least not the basic kinds of numbers, with basic, built-in operations. A language like Java, on top of that, also supports classes like `Integer` which indeed supports quite a number of operations in its interface.

finding differences between classes and ADTs more important. Typically, what is lacking in ADTs is the notion of *instantiation*, and same for static classes.

Proving (user-defined) abstraction to the user of an ADT means also, when using an ADT, one has not to bother about the internals, actually, one has *no access* to the internal representation of the ADT. That's also known as **encapsulation**: an ADT not just bundles up data and operations, but it **encapsulates** and protects the internal representation from access other than via the offered **interface**.

We stated that classes bundle up data and procedures, with instance variables playing the role of the internal data. The problem is, Java classes don't enforce encapsulation. One can partly achieve that in declaring instance variables as `private`. That provides some encapsulation (but not 100%, private fields of instances from other instances of the same class, which breaks encapsulation). Encapsulation (or hiding of internals or abstraction, whatever it is called) can in Java also achieved by using Java interfaces as types, not the class names themselves as types ("programming against interfaces" is the slogan). Indeed, the concept of **interface** demarcating the separation between inside and outside of the abstract data type is central. Also Java knows supports interfaces (with the keyword `interface`); some other languages could use the name *signature* for interfaces of modules or ADTs. *Modula*, a language promoting the idea of modules as a form of ADTs differentiates between `DEFINITION MODULE`'s and `IMPLEMENTATION MODULE`'s, the former the interface or signature for the latter (the implementation).

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

If one still wants to know, what REALLY is the difference between ADTs, modules, and classes, a reasonable standpoint (if one wants to differentiate) in my eyes can be found at `http://www.cs.man.ac.uk/~pjj/cs2111/ho/node18.html`. It roughly says, modules have a (mutable) state, ADTs have not (which basically says: ADTs are functional).

With ADTs being "stateless" (in that view), the behavior of the externally offered functionality can be captured *declaratively*, so the interface is functional (rather than an imperative interface as would be characteristic for modules). One may find it a clear distinction (or not). At any rate, a functional interface would allow to specify the outside behavior by *equations*. The latter point is the reason why some say, ADTs are an implementation of algebraic data types (algebras are some equational formalism).

Finally classes: they typically have a state (more precisely, instances of a class have a state), but there are much more complicated mechanisms on top: inheritance, overriding, late binding and what not.

7.2.4 Type constructors: building new types

A language with only basic types (see Section 7.2.1), would have only a **finite** number of types, built into the language. So, languages offer mechanisms to **introduce new types**, compound types like the ones mentioned in Section 7.2.2. In that way, the language supports an unbounded number of types. Building new types from old ones is done by so-called **type constructors**.[5] We have mentioned a few before, like the one for lists and for pairs or tuples. Those are all examples of compound types. Also ADTs, if supported syntactically in a language, are composed.

There is another mechanism to introduce "new" types, not connected to the question of whether the type is composed or not. That is to give **(new) names to a type.** For instance, **calling** a tuple `real` $\times$ `real` under the name `complex`. Introducing a new name for some type may not be seen as introducing a new type. But it may be that the type system insists that pairs of type `real` $\times$ `real` cannot be use when values of `complex` are needed. If that is the case, `complex` is in a way a new type and different from `real` $\times$ `real`. Questions like that, "is `complex` the same as `real` $\times$ `real` or not?" will be discussed in Section 7.3; different languages can take different choices about which types to treat as equal and which not. The current section here is not much concerned with naming of types, it's about the constructions themselves.

Central type constructors are built-in to a language and are written in "special" syntax. An example is the constructor $\times$ in infix notation for tuple types as in `int` $\times$ `int`. We will see some examples for that.

Generally a type consists of members or elements of that type, the data values of that type. So, it's not enough to be able to define new types and perhaps declare variables to be of that type. One needs a way to *construct values* of the introduced type. For pairs, we have seen the syntax already, writing for instance $(1, 2)$ to construct a pair, here of type `int` $\times$ `int`. Constructing values is one thing, there needs also a way of *deconstructing* them, i.e., way to access the individual parts, in our example the first element 1 and the second one 2. All type constructions in the following will have these three ingredients: 1) forming new types via type constructors, 2) constructing and 3) deconstructing values of the introduced type.

In the following we will have a look at a few of composed types in programming languages. The Compila language of this year's oblig 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 subtyping? etc.).

7.2.5 Arrays

Array types may be notationally represented as in Listing 7.1 or similar. Note that in the code snippet, the array type is unnamed or **anonymous**. Many languages would allow to declare array types only together with giving it a name, but this section here focuses on the types themselves without much emphasis of how to give names to them.

[5] Types / classes that take other types as arguments are also known as **generics** or parametrized types.

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

Listing 7.1: Some conventional syntax for array types

Conceptually, arrays, i.e., values of array types, are finite (and "mutable") functions from the index-type to the component type. Often, there are restrictions on what can be used as index type. First of all, the index domain has to be finite (though one can see extensible arrays as having a potentially unbounded index domain). Perhaps a finite range of non-negative (unsigned) integers. Typical is a bounded range from 0 to $n-1$. Also possible may be syntax like `from ... to ...`. Other types make also be allowed, like *enumerated* types, characters, etc.

In their basic variant, arrays are **fixed size** data structures. Still, the language may be more or less restrictive there. Restrictive (and easiest to realize) would be that the size of the array is statically known at compile time. More liberal would be to allow that the dimension of the array, while still fixed, is known only at run-time, like that the size depends on the content of a variable.

Either way, for most arrays one faces the problem that, at run-time the array may be accessed *outside* its bounds. That danger does not exist for *all* arrays. For arrays allowing indexes ranging over *all* characters or by some from some fixed enumeration of elements (i.e., values from an enumeration type), a halfway decent type system can statically assure that no out-of-bounds errors occur.

By half-way decent I mean the following, in the context of enumeration types: Assume an enumerate type like the following (using ad-hoc syntax and giving the enumeration a useful name):

```
type Weekday = enum {Monday, Tuesday, Wednsday, Thursday, Friday, Saturday, Sunday}
```

Such an enumeration will, in all likelihood, be represented in the compiler by (short, unsigned) integers, ranging from 0 to 6, and probably in the order as listed in source code. Integers can be handled efficiently, in particular when used as indexes for array access. A safe type system would make sure that the array can be indexed only by the official entries in the enumeration. An unsafe one would "allow" the user to exploit the knowledge that `Monday` corresponds to 0 or whatever the scheme is. The programmer would have no advantages of that exploit, at least not in terms of execution speed, since, as said, the compiler will translate the enumeration anyway to numbers. The only thing one could gain is that one could do in an unsafe type system is things like

```
var x : Weekday := Monday;
....
x:=x+1;
```

where one "calculates" with the weekdays (mis-)using integer operations. Of course, if careless, one may end up doing `Sunday + 1`, and there is the out-of-bound error. A type system that would tolerate such calculations would be an example of an unsafe type system, and it is an example where the type system, here the enumeration, does not provide proper *abstraction*. It looks like weekdays and a finite enumerations, but still one can do integer-related things on it, exploiting knowledge of the underlying representation.

For conventional, integer-index arrays, however, there is not much the compiler can do to really and reliably prevent out-of-bounds situations (being an undecidable problem in general). One thing to make sure is that at least at run-time, array-bound checks are done (the compiler must generate code for that). If the check fails, it has the positive effect of raising an error at run-time, if the check fails, and crashing the program (if the error is not handled). That's a good thing to do, compared to the alternative of *not* raising the error. The alternative allows read and write access to memory parts that are not meant to be accessed, at least not in *this* way. And that would be *much* worse (for security reasons and otherwise).

Integer-indexed arrays are typically a very efficient data structure, as they mirror the layout of standard random access memory and customary hardware.[6] Indeed, contiguous random-access memory can be seen as one big array of "cells" or "words" and standard hardware *supports* fast access to to those cells by indirect addressing modes (like making use of an off-set from a base address, even offset multiplied by a factor which represents the size of the entries). In the later chapters about code generation, we will look a bit into different addressing modes of machine instructions.

There are also **multi-dimensional** arrays, not just one-dimensional. One can see it as "array of arrays" (for instance in Java). Often there is specific syntax also for that, not just defining an array of array, like

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

Listing 7.2: Multi-dimensional arrays

As mentioned, one dimensional, i.e., linear, arrays can be mapped straightforwardly onto standard memory. Also two-dimensional arrays or higher-dimensional ones need to be mapped to a *linear* layout in memory, the way that's done may vary and is language dependent (row-by-row or column-by-column etc.) A last word: Array types are typically *reference* types, as many compound types.

7.2.6 Record ("structs")

Structs or records are another well-known data type. For clarity, one should distinguish between **record types** and **records**, the latter being the values of record types. For even more clarity, one should separate also between the record type and the *name* of the record type (same as with array types). Often that precision is loosened a bit and one just says "this is a record" or "this is a struct" for the name, for the record type and for the record value all the same. Structs is a different name for records, coming from the C-keyword for records. Java does not support structs, but of course classes and objects can be used as if one had structs. If one ignores inheritance and methods, the analogy is close.

A struct type could be declared as follows:

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

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

Listing 7.3: Record type ("struct")

The values of a record type, i.e., the records are "labeled tuples". In this example, elements roughly corresponding to `real`$\times$ `int`. When using the labels explicitly, the *order* of occurrence, whether one mentions the real before the integer component is irrelevant on source code level. That's of course different for tuples, the position there matters.

Besides forming new types, one needs a way to construct elements of that type, and also to deconstruct those elements; we mentioned that in the introductory remarks.

Unlike the record type from Listing 7.3 which was anonymous, the one in Listing 7.4 is given a name (many languages don't support anonymous records). The special syntax to define a record in the example is `{300, 42}` and the value is stored in variable `pt`. Of course, as the value is constructed *without* mentioning the fields, resp. mentioning them only in when introducing the type `point`, the order of the two values matter.

Languages may (additionally) allow do define the record value writing `{x=300, y=42}` or `{y=42, x=300}`. To deconstruct a value one uses the well-known dot-notation, like `pt.x`. Again the analogy to objects (and field access) is very close. The dot-notation can be used for read and write access to the record value.

```
struct point {int x; int y;};  // defining a record type (and
                               // give it a name.
struct point pt = { 300, 42 }; // constructing a record value
int z = pt.x;;                 // field access
```

Listing 7.4: Constructing and deconstructing records

As far as the **implementation** of such records is concerned, they are arranged in a *linear* memory layout with slots whose size is given by the (types of the) attributes or fields. The fields are accessible by statically fixed *offsets* which allows *fast* access.

7.2.7 Tuple or product types

Tuples are **pairs** of values, written for instance `(1,2)`. One can pair up values of different types, for instance `(1, "text")`. Typical syntax for tuple types, also called product types is `int * string` (in ascii, for the second sample tuple) or $T_1 \times T_2$ in math notation, with T_1 and T_2 are arbitrary types. Languages may also support be n-tuples, like $(1, 2, 4)$, a "triple", or $(1, "a", \texttt{true}, "b", 7)$, a "quintuple".

Notationally, the syntax to form an n-tuple value here resembles syntax to form a list (or is identical to list forming syntax in some languages). Even if notationally similar, there are differences between n-tuples and lists, Mainly in what one can "do" with the corresponding elements, the ways of "deconstructing" them. Often, in statically typed languages, lists have to be of uniformly-typed elements, pairs not.

Table 7.3 shows the different types of a triple and of a tuple containing another tuple. On some level, both values are "the same". Indeed, both values may actually be *implemented identical*, as a pair of a value followed by a second pair. Still, at the user level, the two values may (or may not) be treated differently. To access the "last element" `true`, the triple may have a syntax to access the 3rd element directly, for the second value one may need to access the second element of the second element.

The table shows that the two values also carry different types. However, the language may treat the two values as different notations for the same "triple" value. If so, and in line with that, it makes no sense to distinguish the two types as different. In other words, the type system would treat the tuple type constructor $\times$ as *associative*, and the two types as the "same", or equivalent. That would be an example of an issue we discuss a bit later, namely when are two types *equivalent*; different approaches for various types exist and other aspects than associativity-or-not factor in as well, in particular **names** of types. The tuples here is a small illustration as preview for the fact that a type system is not just about values and their types, it's also about **relations between types** (when are two types equal? When is one type a subtype of another?).

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

Table 7.3: tuples of tuples and triples

Tuples and tuple types are common in functional languages, less so in other languages. For example, Java (like C etc.) do not support them. Of course, one can simulate them. If one feels the need to return a pair of values, one can return an object containing the pair of values stored in two instance variables (it's more like a record).

One could remark that in some way languages like Java support tuples in their special role as arguments to methods or functions. It's often no presented like that, i.e., it's not said "n-tuples are limited in their use as method arguments", it's rather often said that methods or functions take *"lists of arguments"* as input (lists in an informal manner, not lists as instance of the Java collection type(-constructor) `List`). Java notation is illustrated in Listing 7.5 on a simple example.

```
public int add (int a, int b) { return a + b;}
```

Listing 7.5: Simple method in Java

A method definition like that mixes up declaration aspects, specifying input and output types, with definitorial aspects, providing the code for the method body. Isolating the type, one could write (in this example)

$$add : \texttt{int} \times \texttt{int} \to \texttt{int} \tag{7.1}$$

where the type specifies methods (or functions) which take a 2-tuple or pair of integers (or a list of integers of length 2) and returns another integer.

Aside: Interface for methods in Java Above, in connection with the method definition in Listing 7.5, we mentioned that this code snipped mixes declarational and definitorial aspects, whereas equation (7.1) focuses on the type in isolation. In Java (and other such languages), there is the notion of **interface**, not just conceptually, but as language construct with the keyword `interface` (in Java). So let's revisit the `add` method in a larger context shown in Listing 7.6.

```
interface Iadd {
    int add (int x, int y);
}

class Add implements Iadd {
    public int add (int a, int b) { return a + b;}
};
```

Listing 7.6: Simple method and corresponding interface in Java

The information concerning `add` now corresponds to the one from equation (7.1), with *one extra piece of information*, namely the names of the variables.

As the example also shows, however, the names `x` and `y` as given in the interface *play no role*. The method can be defined using other variables instead. The variables in the interface are not completely random. They cannot be left out, and it's not allowed to use the same variable twice. But the type system does not check if the corresponding method follows the choice of names. As far as type checking is concerned, the signature in the interface is *treated* as if it were `int add(int,int)` (and information given back from the type checker in case of a mismatch between the interface and the class also does not mention which variable names are used, because it is of no concern to the type system).

Why does Java insist on mentioning the names in the interface if it obviously plays no role? I don't know what the designers factually had in mind (or maybe they just followed earlier languages). One motivation is perhaps *documentation*. Interfaces show important information about public methods, how to use them in a type-correct way. The input and output types already give some information based on which one can guess what the method is intended for. In the same way, that a well-chosen method name can give useful information indicating what a method is for, the same holds for well-chosen names for the arguments.

7.2.8 Typing alternatives: Union types, sum types, and inductive data types

Next we discuss the related concepts of **union** and **sum** types. They both realize conceptually situations where a value belongs to one of different alternative types. Like a value which is an integer **or** a boolean.

It should be stressed that it's supposed to be a real alternative, it's *either* an integer *or else* a boolean. In that sense, the word "union" is a bit ill-chosen. If one sees types as sets of their values, it's not really the union of two sets (which would allow overlap). The concepts correspond more precisely to a *disjoint unions*. Later we discuss inductive data types, which add a dimension independent from offering a form of disjoint union types; the extra dimension is recursively defined types. That recursion dimension is ultimately

independent from the sum-type construction, but both work very well together and give expressive mechanisms to build types for unbounded data structures.

Union types (C-style)

Listing 7.7 show the notion of union types in C. The *members* of the union type (as they are called) are discriminated, in this example, by `r` vs. `i`. We could leave it at that, like: that's the way in C, "alternatives" are captured. However, we discuss a bit how (values of) union types are represented and in which way the union types as in C have serious weaknesses. We discuss that in the context of **type safety**, and the weakness is, that union types may be useful, but are definitely not type safe. We will later see, how to do better.

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

Listing 7.7: Union type in C

Union types are C's way to represent the mentioned *concept* for types, that of "alternatives" or disjoint union, in the example from Listing 7.7, the members of that type are either reals or integers. What makes the situation in C not ideal is that union types there are not explained and realized *conceptually*, but in an implementation-centric way. One can find definitions of union types like this:

> A union is a special data type available in C that allows to store different data types in the same-sized memory location.

The weakness of union types comes from that fact, that this is all they do: they *allow* the programmer to use the memory in a particular way. How it's done is clear, if one builds a union type from integers and reals, a value is stored at a place whose size corresponds to reals, since the representation of reals requires *more* space than that of integers.

That makes sure that, when storing a value, no matter whether it's a real or an int, there's enough space to store it. That's welcome, of course, and it avoids overwriting inadvertently neighboring data, thereby corrupting the program. The trouble may start when reading back the stored value. The access in C is written for unions the same way as for records or structs with a dot notation (like `u.i`, when `u` is a (variable containing a reference to) value of the above union type.

The problem is: there is no mechanism, when reading a value of union type to figure out which it is (here integer or real). Neither the static type system has that information, nor is it possible for the programmer to insert a check at run-time to determine, which it is. I.e., the notation `u.i` is does actually not mean "give me the integer stored there", it is more wishful thinking: "give me the value stored there, I think it's an integer".

That being so, it should be clear that the treatment of union types is definitely not **type safe**. It's nothing much more than a directive to allocate enough memory to hold largest

member of the union. As can be seen also on the quoted "definition" of union types in C, the type is treated clearly with an implementor's (= low level) focus and wrt. memory allocation needs, not with a "proper usage focus" or assuring strong typing. Thus, it might be seen as a bad example of modern use of types and better (type-safe) ways of realizing the notion of "alternatives" are known since. Next, we discuss a (small) improvement, namely *variant records*, also called tagged unions or discriminated unions, before we have a short look at inductive data types.

Variant records from Pascal

The union type from before is basically nothing else than a piece of space big enough to store each possible alternative, but contains no information about which it actually is. To improve that, one can simply store additional information about which alternative is meant. The corresponding data type is known as **variant record type**, or also **tagged union type** or **discriminated union type**. Listing 7.8 shows an example in Pascal, again describing the alternative between reals and integers, as before.

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

Listing 7.8: Variant record (Pascal)

The memory layout, i.e., the representation of value in memory, is *different* than that for C union types. The layout now is *non-overlapping*.[7] The disadvantage is that the implementation uses space for all potential alternatives (plus information about the "tag") even if only exactly one alternative is the actual one. The representation resembles therefore closely *record* or *struct*, namely a record where only one field is meant "for real", the others are "empty" in the sense of containing bit patterns without any meaning, so better not touch them...

Now, is that wasteful memory usage worth it? Well, the programmer is responsible to set and check the "discriminator". The type system does not give assistance there. So, the improvement is the following: instead of remembering what kind of variant is meant, an integer or a real for instance, the data structure carries that information. That's actually something of quite some use. The *code* can *use* that information to make case distinctions, to **discriminate** the between the case of integers vs. reals in the example. That's of course very useful. Without that, it's hard to work meaningfully with elements of union type. If given a value that is, say a string or a bool or an object of some sort without being told what it is, what can one do with that piece of data? In particular, one cannot make a **case distinction** based on what actually it is.

The possibility of making **case distinctions** on alternative data is essential for types intended to represent alternative data. The ability to make case distinctions is **the very essence** of something like union types (when done properly).

[7] Again, that's an implementor-centric view, not a user-centric one.

The C union types not only lack type safety guarantees, they also don't offer that case distinction feature. The latter one is what is added to variant record.

But how do variant records improve on the **type safety** front? Alas, type-safety-wise, they are not really an improvement. The problem is, that the user can profit from the extra information, the "tag". However, to make proper use of that is the responsibility of the programmer, the type system does not check it or enforce it. The careless programmer can thus confuse things up, read a bit-pattern that represents a value of some specific type as if it were of a different one by confusing up the alternatives and that can mess up everything. This has been discussed as "Pascal's type hole" (at the time when Pascal was kind of hot in some corners) with examples that show how to trick Pascal to do pointer arithmetic, (mis-)using variant records. Of course, one can do the same with C, though no one ever mentioned a type hole in C's union type. The reason is that no one claimed C being type safe in the first place, union types or otherwise. Pascal on the other had came with a very restrictive (and inflexible) type system. The loss of flexibility might be justified by increased safety. At any rate, especially disciples of C would not tire to point out the gaping type hole in Pascal: "their type system is like a straitjacket, it's almost impossible to do real programming, and what for? Safety? My ass, look at this example, with their union types, I can trick their oh-so-strong type system into doing pointer arithmetic just like in C. Let's call it Pascal's type hole.".

Remark 7.2.1 (A word on terminology)**.** The types in the previous discussion about type safety contrasted the plain *union types* as in C and their improvement in the form of *variant records types*. The latter ones got their name probably because they are represented in memory very similar than records (as mentioned). I would, however, not consider the name to be too well-chosen. Considering types as a central user-level abstraction on data, the fact how (commonly) a particular type is laid out in memory is of prime interest for the programmer that uses elements of a type and should actually not be relevant: the type is supposed to provide an *abstraction* from the layout.

And on the user level, records and variant records (or members of tagged union type) are **very** different. Records are like n-tuples or members of product types (more precisely labeled product types). And actually, as concept, union types are the **opposite** of product types. That's why they are also called **sum** types (with "sum" and "product" denoting duals). There are good, mathematical reasons that make sum types (for alternatives) and product types (for tuples) the exact opposites or *duals* of each other, but let's leave those out from the discussion. At any rate, calling two very different concepts (records and variant records) by quite similar names is unfortunate in my eyes. Likewise it's not ideal, that the "destructor syntax" in both cases is often similar. Both records and elements of a discriminated union are accessed via the same *dot-notation*. □

7.2.9 Recursive and inductive types

Next we discuss *inductive types* or inductive data types. One way of seeing them is basically: (disjoint) union types done right plus the possibility of "recursion" (on the type level). Recursion is a concept orthogonal to that of describing alternatives, so we could do a discussion focusing solely on *sum types*. But their combination is so common and useful, that we use examples making use of recursion as well.

Inductive data types are very common in (statically-typed) functional languages, but appear in other languages as well. We will use ML or ocaml in the code examples, but many functional languages use quite similar syntax (many are influenced by ML, anyway).

Listing 7.16 shows a corresponding notation for the integer-or-real example. The vertical bar `|` denotes the alternative.[8] The syntactic ingredients `isReal` and `isInteger` are called the constructors for elements of that type.

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

Listing 7.9: Alternative (inductive data type without recursion)

In the code snippet of Listing 7.16, the type is **anonymous**, i.e., no name is given to type. That limits the usefulness of the example, but of course one can easily give it a name, like writing `type intorreal = IsReal of real | IsInteger of int`.

In ML-like languages, this form of sum-types is **type safe**. Elements from the types are constructed by, well, using the **constructors** of the sum type, that's why constructors are called like that .... For instance one can obtain a real number resp. an integer number as elements of they sum type by `IsReal 4.5` resp. `IsInteger 5`.

As stressed, one needs not only a way of constructing elements of a compound data type, one needs also ways of *destructing* (or deconstructing) them, i.e., pull composed data apart. Like accessing the components of a record, slots in an array etc. *Elements* of a sum-type, which is a compound or composed type, are not really "composed". It makes no sense to talk about the real part of the value `isInteger 5`, it's of course only an integer. Deconstructing values is not so much understood as access parts (which is not a useful picture at least here), but as being the *opposite* of constructing or building values. The opposite of creating a value is *using* it. As mentioned, the **very essence** of how to make use of a value `intorreal` is to do a **case distinction**, here covering the two cases.

That can elegantly be done by so-called **pattern matching**, in combination with a case construct (in the code here with the keyword `match`):

```
type intorfloat =  IsFloat of float | IsInteger of int;;

let discriminate (n: intorfloat): unit =
  match n with
    IsFloat   f -> print_float f
  | IsInteger i -> print_int i
;;
```

Listing 7.10: Alternative and pattern matching

Pattern matching in this style is type-safe. Often the type system provides checks whether the match is *exhaustive*, i.e., that no alternative is forgotten, and whether no alternative is covered more than once. In a duplicate match situation, typically the first match is the relevant one, the second one is "dead code", which is presumably unintended. The type system may still accept the code, but will at least issue warnings about unmatched cases or unused cases. That is very helpful.

[8] As for regular expressions and for context-free grammars.

Listing 7.11 shows how one can combine the idea of sum-types with recursive definition. It encodes directly the idea that a tree is *either* a leaf *or else* a node that carries an integer and two trees, the sub-trees.

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

Listing 7.11: Inductive data type (binary tree)

The example could be improved (the language ML or ocaml and many others would allow that): A more useful binary tree would not fix that the values stored are fixed integer, but that type would be treated as *parameter*. But discussing also that would take us too far from the issue at hand.

In the compiler lecture, we have seen *many* examples of concepts that can be represented by inductive data types, and we will see more. Listing 7.12 shows how one can represent expressions of some form, basically describing the type for syntax trees (for expressions).

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

Listing 7.12: Expressions as inductive data type

Recursive data types in C Of course, one can define tree data structures also in languages like C, and have them reflected (to some extent) in the type system. This paragraph is no so much concerned with sum-types, but more with the "recursion" aspect of inductive data types. Listing 7.13 shows an attempt to recursively define trees analogous to way using inductive data types from before.

```
struct int_bintree  {
  int val;
  struct int_bintree left , right;
}
```

Listing 7.13: Recursive record type for binary trees (does **not** work in C)

Conceptually there is nothing wrong defining a recursive record type like that, only that C does not allow it. Shortly, we will see how it's done properly. The code from Listing 7.13, of course, also covers only one of the two alternative cases of (the type for) binary trees, the one for proper nodes, which is represented as a record or struct with three members. That is in contrast to the situation in Listing 7.11, which lists both alternatives (leaf or else inner node).

Leaves are "represented" by null-pointers, resp. structs where both left and right subtrees are null-pointer can be considered as leaves (in this case a leave carrying a value). Record types are, as mentioned, reference types, and references may be "undefined", i.e., null. So the case distinction between being a proper node or a leaf, something that is done in ML or similar by pattern matching, involves here checking for null-ness.

Back to the question how to achieve something like recursive (record) types, since the notation from Listing 7.13 is not allowed in C. One can do it by using pointers or references, resp. explicit **reference types** as shown in Listing 7.14. Why this is allowed and the other is not has more to do with design decisions of how (early) C-compilers worked and is not so interesting for us.

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

Listing 7.14: Recursive types for binary trees in C (indirect)

Let's have also a look at Java (Listing 7.15). Java does not force the user to mention references in the definition (Java does not have an *explicit* notation for reference types anyway).

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

Listing 7.15: Binary trees in Java

Note that also in languages like ML, ocaml, Haskell etc., the implementation of trees and other such structures use "pointers", but they are **hidden** from the user. Note further, there are no null-pointers in ML, and the `NIL` we used for leaves is not a null-pointer but a constructor of a sum data type. Probably a better name in that definition would have been `Leaf`, but we wanted to draw a parallel to the situation in C and Java. Which is not really a parallel, since `NIL` as said, is not a way of introducing null-pointers in ML (there is no such thing as null-pointers in ML).

"Pattern matching" in Java You may know that there is no such thing as pattern matching in Java. Still, we have seen (in other parts of the lecture) how to implement inductive data structures like the one for expressions, similar to the ones from Listing 7.12. We discussed earlier one "recipe" how to implement ASTs (using abstract super-classes and multiple concrete sub-classes). The list of sub-classes correspond to the list of *alternatives* of a sum-type. In the expression example of Listing 7.12, there would be one abstract super-class (say `Exp`), and 4 concrete sub-classes, say `Plus`, `Minus`, `Number`, and `Var`). Only there's no pattern matching over those.

The job achieved by pattern matching is done differently in Java. If you followed the recipe in your oblig, you will have done it, for instance for the type checker and for the code generator and the pretty printer. Remember the purpose of pattern matching. It's to *discriminate* between the different cases of expressions, name whether the expression is constructed via `Plus`, `Minus`, `Number`, and `Var`. A printing procedure would handle these 4 cases differently. In Java, one has to implement the print procedure differently for the 4 classes, same for type checking etc. But the effect is comparable. Depending on which expression object one invokes the method, a different reaction is done.

7.2.10 Pointer and reference types

Many data structures make use of pointers one way or the other. In many languages, complex data types are often reference types, as mentioned. Languages, however, may more or less hide the use of pointers to some degree or more or less completely. Still, one should be aware that, for instance, the array type is a reference type, because not knowing that and changing a shared array may lead to trouble. But the fact that one is dealing with references is neither visible in the type nor when accessing the data structure itself.

We are here talking about **pointers** and **references**. On some level, it's the same concept. A distinction done by many is that, when having pointers, one can "calculate" with them, doing pointer arithmetic, like obtaining a pointer or address and then accessing "the next slot afterwards". Reference are tamed pointers and mostly implicit. One cannot calculate with them, cannot determine the address of a thing, and dereferencing is done implicitly. Java, in that terminology, uses references, but C uses pointers.

Pointer types as in C

C is explicit about its use of pointers including that there is a special type for it, resp. a type constructor. For instance, the type of a pointer to a integer value is `int*`, where " * " in postfix notation can be seen as type constructor.

```
int* p;
```

Listing 7.16: Variable of pointer type

Not only C, which allows pointer arithmetic, knows such types. The corresponding type is written, for instance `^integer` in Pascal and `int ref` in ML. The value of such types is an *address* of (or reference or pointer to) a value of the underlying type. Operations on such references are *dereferencing*, i.e., "following" the reference to access the underlying value. There is in C also an operation that determining the address of an data item, written `&x` ("address of x"). Remember: C allows *pointer arithmetic*. Listing 7.17 show some operations involving pointer (in Pascal).

```
var a:  ^integer  (* pointer to an integer    *)
var b:   integer
...
a := &i           (* i an int var             *)
                  (* a :=  new integer ok too *)
b:= ^a + b
```

Listing 7.17: Operations involving pointer in Pascal

Implicit dereferencing As mentioned, many languages more or less hide the existence of pointers. Still, they may distinguish between reference vs. value types, it's only not visible in the types, and with such a design choice, the language will often do automatic, implicit dereferencing. Class types in Java is an example of reference types.

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

Listing 7.18: Objects

In the code snippet of Listing 7.18, in a sloppy manner of speaking, one could say " `r` is an object" (which is an instance of class `C` /which is of type `C`). Slightly more precise is to say "variable `r` contains an object... ", and even more precise "variable `r` contains a reference to an object". In Java and other languages, `r.field` involves an implicit dereferencing and corresponds to something like " `(*r).field`" when done explicitly.

Programming with pointers Pointers or references are a "popular" source of **errors.** To avoid null-pointer exceptions and to program defensively, one typically has to insert tests for non-null-ness. Explicit pointers can lead to problems in block-structured language (when handled non-expertly). We will mention that (again) in the chapter about run-time environments. In that later chapter, we also will discuss parameter passing, including call-by-reference, which is a mechanism to hand over parameters from caller to callee that involves references.

Another aspect to watch out for is **aliasing**. That's when two variables contain a reference to a shared piece of data. This is troublesome if also mutation enters the picture. In an alias situation, changing the piece of data via one variable changes "also" the value for the other variables. That may be intended. We will see that later for the mentioned call-by-reference mechanism. If the alias-situation is unknown, the change may be unintended and erroneous.

Null pointers are generally attributed (actually including self-attributed) to *Tony Hoare*, famous for many landmark contributions. He himself refers to the introduction of null pointers or null references (1965 for ALGOL-W) as his billion dollar mistake.[9] One can also consult Hoare's Turing Award lecture from 1980, where he talks about similar topics. Also the text of the lecture is available on the net. In the lecture, he interestingly mentions as the *first* and foremost design principle for the design of ALGOL resp. the corresponding compiler: **security**. So it's not that the intention was to say "to hell with security, speed rules". From the text, though, it seems that he speaks about "security" of the compiler itself, in that it should never crash (= "... no core dumps should ever be necessary").

Function variables: references to functions

The following shows problems in situations when one can reference more "powerful" things than "dead data". So far, the data was all *passive* but, of course, also function or procedures need to be stored somewhere, ultimately it's also just a block of bits. Often, in traditional layouts, one thinks of functions code residing in one portion of the memory, and data in a another.[10] Either way, there is no principal reason why variables could not refer

[9]See also the link here (the video seems no longer to work, but there is some notes or rudimentary transcript).

[10]Though in a shared address space in the traditional von Neumann architecture. In the so-called Harvard-architecture, the separation would be stricter.

to functions, as well. That goes in the direction of *higher-order functions* where the distinction between data and code is completely blurred.

The example here is not based on higher-order programming, but uses just Pascal. What one can do in Pascal (as opposed to C) is *nested* function declarations and "returning" variables "containing" functions (referring to them). The problem, illustrated here ("escaping"), is something that one also has to deal with for higher-order functions. In a way, the lesson from this example is: Pascal has this facility, but somehow did not deal with it properly. Dealing properly with it would have required **closures** (see later), but Pascal did not do that.

```
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.
```

Listing 7.19: Function variable (in Pascal)

The tricky part in the example from Listing 7.19 is the nested, lexical scope and the fact that the nested function definition **escapes** the surrounding function resp. scope. The escape is done with the help to the assignment to the function variable. The function variable, containing a reference to the procedure `P`, is used to "return" the corresponding function (resp. a pointer to it).

That is problematic in that the procedure `P` comes with its own scope and local variables. By returning the procedure that scope **outlives** the surrounding scope. Pascal (like C and Java and other languages) uses a *stack* to manage the memory needs of procedures at run-time (as part of the so-called run-time environment). So, this is example *cannot* be handled with a stack-based run-time environment? So what does Pascal do then? Does the semantic analysis checks it (via an escape analysis) and issues a warning? Will the memory for the escaping scope be stored elsewhere, not on the stack, maybe the heap? Nothing like that, the program unceremoneously **crashes**. At least in the Pascal version I used (free Pascal), there may be other versions with compilers that offer earlier warnings. But crashing is at least better than silently accessing parts of the memory that should not be accessed.

Functional languages, which typically support higher-order functions, allow function abstractions as return value. They face the same challenge, and the solution is: since the stack discipline does not work, the memory needs to realize the local scope are stored on the heap (in what is called a *closure*). This will be picked up when talking about run-time environments.

In C, one can return functions in the same way as in the Pascal example since C supports pointers to functions. C does not support closures, but it *does not* suffer from the "escape problem" as discussed for Pascal. The reason is, that C does not allow *nested* function definitions. It's the combination (combined with the lack of closures) that crashes the Pascal program.

What about (classic) Lisp. It supports higher-order functions, i.e., it allows to return functions (taking a function as argument is less problematic), it allows nested definitions. And, originally, it did not work with closures. But it still did *not* suffer from escaping scopes in the way described. Simply because classic Lisp uses *dynamic scopes* not *static* ones.

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

Functions of course carry types. The corresponding type constructor is $\rightarrow$ (or -> in ascii). So `int`$\rightarrow$`int` is the type representing functions from `int` to `int`, i.e., taking an integer as argument and return one as well. Not all languages use explicitly the $\rightarrow$ constructor, and different notations exist. One is shown in Listing 7.20.

Sometimes the term *signature* is used to when taking about type information in connection with a function. The signature may include the name of the function, as in the example from Listing 7.20 and 7.21.

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

Listing 7.20: Function signature (Modula 2)

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

Listing 7.21: Function signature (C)

As mentioned before, functions are *arguments* are less problematic than returning them (for instance via function variabless), and the reason is that the stack-discipline in that case is still doable.

7.2.11 Classes and types

Let's also talk about *object-oriented* languages and what role types play there. We don't dig deep here, we stick to vanilla, class-based, single-inheritance languages, say Java.[11] Basically saying a few words about types for objects in such a language.

Objects are instances of classes and are typed by "classes". Classes are connected in a hierarchy via **inheritance.** In a single-inheritance setting, the inheritance hierarchy is a tree. In Java, the root class of that tree is called `Object`. One speaks also of super-classes and sub-classes, with `Object` being the super-class of every other class. Classes resemble

[11] There exist languages that qualify for being object-oriented, but don't have classes.

record types to some extent, and we have also discussed (at the part about inductive types) that sub-classes can be used in ways the resemble variant types (sum types, variant records).

There are many bells and whistles that factor in when looking at typing, even in a simple language like Java (and Java at least started out as a quite simple language). Complicating details are questions concerning visibility, overloading etc. We don't go into those details (of Java or other languages), we are interested in a few general aspects in the context of type systems.

In particular, we focus on the central aspect of **class inheritance** and **sub-classing**. Note that I have not called it **subtyping** (yet), but sub-classing. In general, for the sake of clarity, it's better to distinguish different roles of class names. Listing 7.22 defines three classes `A`, `B`, and `C`, the latter two, unrelated among each other, are direct **sub-classes** of `A`. So far, so well-known. `A`, `B` and `C` are also **(names of) types**. Also in their roles as type names, the three identifiers are related, and the relationship is called **subtyping**. Isn't that then not the same? Well, Java is carefully designed (to assure that the type system is type-safe)[12] and designed in such a way that inheritance between classes implies that the corresponding types (or the class names in their role as types, if you prefer) are in subtype relation. As a consequence of this fact (in Java), one finds categorical statements like *inheritance is subtyping* here and there, and I have talked to people that who insisted strongly that types and classes are the same thing and likewise inheritance and subtyping is the same thing, denying that obvious thing as just delusional...

Of course, when programming with the classes of Listing 7.22, for instance doing `B x = new B()`, it's definitely fine to say `x` is of type `B` and the object created and stored in `x` is of type `B` as well. Still, using words carefully when the situation requires it can't hurt, and when writing a type checker or compiler is very definitely a situation, that requires attention to details. One see that the the roles of `A`, `B`, and `C` as classes and as types cannot be 100% the same by considering that `x` is an instance of class `B` and is of type `B` and of type A, with `A` being the super-type of `B`. Typically, one does not consider `x` to be an instance of `A`, though inheritance between the two classes leads to the situation that things defined in `A` are relevant for `x` (either via late-bound methods or via instance variables that are taken from the super-class).

The fact that `x` is both of type `B` and of type `A` (and also of type `Object` by default), i.e., that there can be code pieces that have more than one type, is known as *polymorphism*. The form here is called, **subtype polymorphism** (or subtyping for short). Java (and other languages) support often different forms of polymorphism, at least for some aspects of the language.

Actually, class names like `B` play 3 different roles not just 2, in languages like Java, we will pick up on that a bit later in Section 7.3 when discussing equality between types.

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

class B extends A {
  int i
```

[12]There are unsafe corners, but they are well-defined and let's ignore them.

```
  void f() {...}
}

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

Listing 7.22: Class inheritance resp. subclassing

The three classes from above illustrate sub-classing (and in many object-oriented languages, connected to that, subtyping). Note that the classes are *also* names of types. What is also is illustrated is *overriding* as far as `f` is concerned. Inheritance is actually not illustrated, insofar that `f` as only method involved is overridden, not inherited both in `B` and `C`. The methods `f` and the instance variables `i` are treated differently as far as binding is concerned. That will be discussed next. In the slides we use `rA` to refer to a variable of *static* type/class `A`.

Access to object members

Instance variables and methods of a object are accessed (in many languages) via the dot-notion, similar to field access in records. A central aspect when calling a *method* is **late binding** also called dynamic binding, virtual access, dynamic dispatch, all mean roughly the same. We discuss a few issues around that also later, in the context of run-time environments

When invoking `rA.f()`, what is meant is the "deepest" `f` in the run-time class of the *object*, `rA` points to. It's not determined by the static type of `rA`. Only for `static` methods (in Java terminology), the static type determines which code is executed

```
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();}
}
```

Listing 7.23: Fields vs. methods ("shadowing")

The code illustrates the difference in the treatment of fields and methods, as far as binding is concerned. While the mechanism for methods (which are late or dynamically bound) is called overriding, the similar (but of course not same) situation for fields (which are

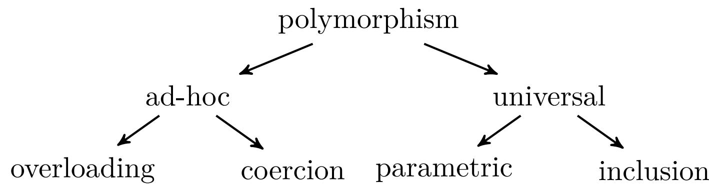


Figure 7.1: Classification of polymorphism

statically bound) is called *shadowing*. One may also see it like that: fields are treated as if they were *static* methods.

7.2.12 Polymorphism

Let's round off this section by shortly introducing the concept of **polymorphism.** It's a general property of type systems. Some type systems are monomorphic and some are polymorphic. As a matter of fact, truly monomorphic type systems are rare in programming languages. Basically all realistic type system offer some form of polymorphism or more than one form. In short, polymorphism is the opposite of monomorphism.

> A type system is **monomorphic**, if all syntactic entities have at most one type. I.e., it's either ill-typed, or, when well-typed it has exactly one type. A type system is **polymorphic** if that's not the case.

Of course a type system can contain both aspects. At some corners of the language, constructs are handled monomorphically, whereas in others, there is more flexibility, due to polymorphism. According to a classical categorization (by Cardelli and Wegner [1]) one can broadly distinguish between ad-hoc polymorphism and non-ad-hoc polymorphism (in the paper called universal polymorphism). See Figure 7.1. What is called inclusion polymorphism is also called *subtype polymorphism* or *subtyping* for short.

Ad hoc polymorphism

In contrast to universal polymorphism, the ad-hoc form of polymorphism represents resp. treats different situations differently. Overloading is a well-known form, we have mentioned it elsewhere and it is also called "abuse-of-notation". Languages can use overloading for different (classes of) language constructs. Very typical is overloading of (built-in) *operators* (**operator overloading**). For instance, the binary infix operator + is such a nice and familiar operator (and there are not so many symbols in ascii) that it would be a shame to use it for one single use. Thus it's overloaded to operate on pairs of integers, pairs of floats, and pairs of strings. The different pieces of code that implement the different operations are *unrelated*, though of course one can make the argument the addition on representations of integers and on operations on floats have some conceptual connection. But there is no "shared" code.

Coercion (or **conversion**) is slightly different. It refers to situations, where implicitly an internal representation is converted to a different one before operated on. The + operation

is a good illustration again. Many languages allow using + on mixed arguments, like an addition of an integer and a float. In this situation, the integer will be converted to a float, and the operation for pair of floats is applied (and a float is returned).

So, on the surface, overloading and coercion look rather similar, like that + works on all combinations of integers and floats and in that sense overloaded in multiple ways. There are, however, only two implementations of + (leaving strings out of the picture now), the one on integers and the one on floats. Depending on the situation, one of the two is "chosen" in that the compiler uses the corresponding instructions. That's the overloading part. Overloading is resolved with the help of the type system in that, before code generation, the overloaded + at user level is disambiguated by mentioning syntactically one of two routines `add_float` or `add_in`, for instance in an processed AST. Most languages use overloading (and coercions) at least for some built-in routines. They may also allow the programmer to make use of overloading. Typical, and as known from Java, is method overloading and, a special form of that, constructor overloading. Other languages offer advanced user-defined capacities for overloading in the form of *type classes* as known from Haskell (not part of the pensum).

Overloading is a convenient concept but should be employed in moderation, i.e., there can be too much of a good thing. One should keep in mind that it's not just how far one can push overloading of concepts and the support of the type system for it. Typically, overloading is not the hardest problem in a static type system: the different interpretations are disambiguated early on (like + being replaced by `add_float` or `add_int`), based on the type checker and *before* any code generation starts. Being "too overloaded", making choices on fine-tuned and intricate situational criteria, may obscure to the user what actually is going on. Especially, if overloading is implemented in combination with subtype of polymorphism or taking into account more or less complex rules when types are equal. In that case, the type system has too much "ad-hoc-ness", and that's not positive.

Universal polymorphism

Let's cover the last two forms of polymorphism, without going into details. One is **generic polymorphism**. It's characteristic for many functional languages. It's about functions (or procedures or methods) that work identical for all situations. For instance, swapping to integers in a pair works identical to swapping two booleans (at least identical as far as the swapping is concerned). So instead of having functions `swap_int: (int * int) -> (int * int)` and `swap_bool: (bool * bool) -> (bool * bool)` and infinitely many others for all things that could potentially be swapped, a generically polymorphic function would be of type `swap: ('a * 'b) -> ('b * 'a)` with identifiers like `'a` and `'b` representing type variables (in some languages).

The other form of universal polymorphism is **subtype polymorphism** or subtyping. It's supported by many object-oriented languages (as in Java). There, the key is that elements of a subtype can be used without problems at places where elements of a super-type are expected so that elements of a subtype are *also* at the same time an element of each supertype. That requirement is known as **subsumption**.

Subtyping is a relationship between types, like $T_1 \leq T_2$, with often some minimal requirements like being reflexive and transitive. Sub-typing can become complex, but we leave

it at that mostly. We will have a look at another relationship on types in Section 7.3, namely *equality*.

7.3 Equality of types

This section discusses issues in connection with the question: when are two types equal. Different languages give different answers to that questions (sometimes also differently for various types). We also discuss **naming** for types in this section, since the names of types is an important criterion one can use for equality of types.

It should go without saying that given a program fragment of a type T_1 which is equal or equivalent to type T_2, then the program fragment is also of type T_2. One could be tempted to say that this is an instance of **polymorphism**, after all, the program fragment has more than one type, namely T_1 and T_2. Normally, though, one would not count that as polymorphism. After all, T_1 and T_2 are the "same" type, in the sense of type equality or type equivalence (if one prefers that word). Figure 7.1 from before did likewise not include that phenomenon as a form of polymorphism.

So, **when are 2 types equal?** There are surprisingly **many different answers** possible. At any rate, it's the type system's resp. the type checker's task to implement that check equivalence of types. Those checks may be very simple. Or then also not, it all depends on design choices one makes. One **(non-)answer** to the question is that two types are equal if they are **represented** equally. In such a view, for instance, type `int` and `short` are equal, since the *are* both (two different names to refer to) 2 bytes on some platform. That, however, is at odds with the modern role of types as **abstraction**. So let's look at other approaches.

Listing 7.24 works with pairs of integers, a compound type. Not only that, it gives it a specific *name*, namely `pairs_of_ints`, and declares a variable to be of that type.

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

Listing 7.24: Pairs of integers

Now, is "the" type of (values of) x `pair_of_ints`, or the product type `int * int` , or both, as they are considered equal? In the latter case `pair_of_int` is an abbreviation of the product type. One speaks of `pairs_of_int` to be a **type synonym** for `int * int`. For the particular language (ocaml), the piece of code is correct: the pair `(1,4)` is of type `int * int` and of type `pair_of_ints`.

The example involves two aspects, the fact that the type is *compound* not basic, and the fact that it's given a name. The fundamental decision a type system has to make concerning type equivalence is, which one counts (or counts more): *the name or the structure of the types.* This is the crucial difference between

structural vs. nominal equivalence of types.

Let's have a look at Listing 7.25, which introduces a few record types and some variables for those.

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

var c: record
   int i;
   double d
 end

typedef idRecord: record
   int i;
   double d
 end

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

Listing 7.25: A few structs

There are altogether 5 variables, `a` to `e`, all of them containing records of the same shape or structure, i.e., with members of the same name and of the same type (in this case, **identical** types `int` and `double`, not just equivalent types).

Now, the question is , which ones of the various (=? different) record types are treated equivalent? Or, to say it differently, which one of the following assignment are accepted by a type checker, on the ground that the types of the involved variables are equivalent. Of course, the values of the "different" record types and all *represented* the same way. For that implementation-centric view, all assignments should be unproblematic.

```
a := c;
a := d;

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

Every language, that allows to define multiple such synonyms simultaneously, will treat `a` and `b` to be of the same type, being declared at the same time with the same (anonymous) record type. Whether the types for `a` and `c` are treated equal is a different issue. They are declared separately, with two record types (again anonymous). Both record types are structurally equal, but a type system based on **nominal** (name-based) principles would treat them as different.

The `typedef` definition introduces the name `idRecord` for the record type, and the last two variables are introduced in two separate declarations. In this case, the only plausible behavior of a type system is to treat `d` and `e` as of the same type, namely of the type called `idRecord`. Otherwise, what would be the use of having the same name. A matter of choice might be whether `a` and `d` are of the same type.

The two most plausible and consistent interpretations would be the following. In a **nominal** treatment, `a` and `b` would carry the same type as well as `d` and `e`, but all others would be different. In a **structural** type system, all types would be equivalent and all assignments allowed.

Classes as types (in Java)

Let's have a look at Java, and the way classes are treated, at least in their role as (names for) types. Let's ignore the issue of anonymous classes, classes without a name. Therefore each class has a name which, at the same time, is also the name of a type. Namely the name of a type of all instances of that class and of instances of all its sub-classes.

It's a decision of the designers of Java and similar languages, that class names have this dual role, referring to the class and at at the same to the corresponding type. At the same time, there is a third role, namely referring to the constructor method(s) of the class. For instance, in the statement `C x = new C()`, the class name `C` on the left-hand side is used as type and on the right-hand side as constructor (and somewhere else in the program there will be the class definition, which fixes the code of the constructor plus the code for other methods and fields). To use the same name for these three different roles is not a law of nature, it's a design decision, and other object-oriented languages may make different design choices.

Java also supports **interfaces** as separate concept, which (also) play the role of types. Indeed, some people recommend as good programming practice, to *only* use interfaces as types (and not the class names). That's sometimes called *code-against-interfaces* or similar. Whether one follows the code-against-interface style of programming or not, Java's type system is **nominal** as far as classes and interfaces are concerned. That's illustrated in Listing 7.26.

```
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;                 // ???
    }
}
```

Listing 7.26: No duck typing in Java, an example with interfaces

The example works analogous when using classes in their roles as types instead of interfaces. Why does the example refers to "duck typing"? Well, Java used nominal principles, not structural. In some corners, people find the word "duck-typing" to be more clear (or funnier) than structural typing ("if it walks like a duck, swims like a duck, quacks like a duck, then it must be a duck no matter how you call the animal"). The duck typing terminology seems popular in scripting languages (and some, not all, connect duck typing exclusively to dynamic type systems). Since there is no complete agreement what duck typing actually is (except that it sure sounds funny), the traditional distinction between nominal and structural typing seems preferable.

We discussed the name-vs.-structure question in the context of when two types are equal. It applies also to subtyping, which is a more complex relation than equivalence. Also there, languages can support nominal subtyping (Java for example) or structural subtyping, but we leave it out from the discussion here.

Types in the AST

Shortly, we sketch in code how recursive routines could look like that check the equality of two types, one for structural equivalence and one for nominal equivalence. The routines work recursively over pairs of ASTs for the two types ot be compared. Before we do that we have a few words on ASTs for types.

But actually, there as no many breathtaking insights to be gained here. We have seen a few types and some typical syntax for them. ASTs, as we know, is a tree-like representation for syntax, and compared to concrete syntax tree, often pruned and cleaned up a bit. Furthermore, there is no such thing as *the* AST for a given concrete syntax, there is quite some amount of freedom how to design a AST and how to realize in the the language in which the compiler is written. That's generally the case, and that's still the case for syntax that represents the types. Of course, basic, non-compound types do not correspond to trees, they typically just leaves in an AST, and covered by keywords on the language (like `int` and `bool`). Only non.trivial compound types, which may be composed to complex types like `List of (Int * (Array of struct {....} from 1..10))` correspond to trees.

So the following (sketches of) AST are just some impressionistic illustrations of how such trees could look like. Listing 7.27 show some record type and Listing 7.28 some syntax for procedure headers. Figures 7.2a and 7.2b illustrate possible trees.

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

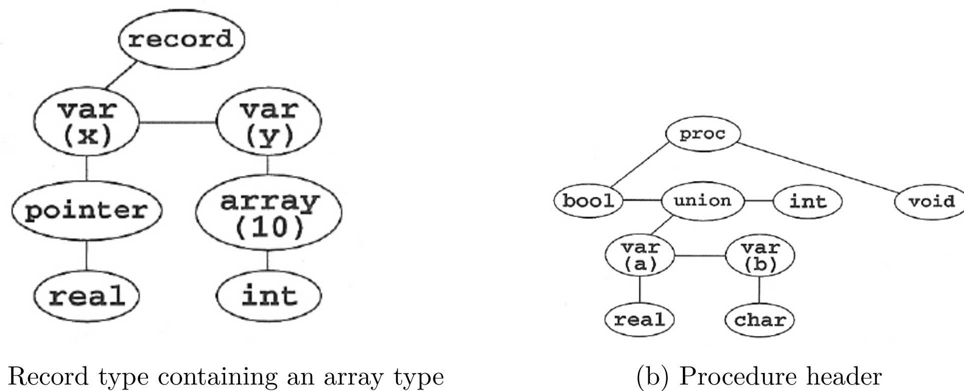
Listing 7.27: Sample record type containing an array type

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

Listing 7.28: Sample procedure header

Structural equality

The pictorial sketches of the AST maybe too sketchy when talking about a equality checking as part of a type checker. Table 7.4 shows a grammar for (abstract) syntax for types which is intended for checking structural equality. Afterwards, we will deal with syntax used for nominal equality (Table 7.5). The two versions of the syntax are pretty similar. In the first version, structured types are "anonymous", in the second version not. Of course, one can have a type system using structural equality which *also* allows to give names to types (or not). The tuple types in Listing 7.24 showed an example: the product type as such is anonymous, but it was given the name `pair_of_ints`, too.



(a) Record type containing an array type (b) Procedure header

Figure 7.2: Abstract syntax trees

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*

Table 7.4: Type syntax intended for structural equality

In the presentation here, the two versions of syntax are either anonymous, or force the programmer that uses records or similar to give it a name.

```
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.child, t2.child)
  else if     t1.kind = record and t2.kind = record
          or  t1.kind = union and t2.kind = union
       then begin
              p1 := t1.child;
              p2 := t2.child;
              temp := true;
              while temp and p1 ≠ nil and p2 ≠ nil
              do
                 if   p1.name ≠ p2.name
                 then temp := false
                 else
                 begin
                    p1 := p1.sibling;
                    p2 := p2.sibling;
                 end;
               return temp and p1 = nil and p2 = nil;
```

```
  else if t1.kind = pointer and t2.kind = pointer
  then     return typeEqual(t1.child , t2.child)
  else if t1.kind = proc and t2.kind = proc
  then     begin
               p1 := t1.child;
               p2 := t2.child;
               temp := true;
               while temp and p1 ≠ nil and p2 ≠ nil
               do
                  if   not typeEqual(p1.child , p2.child)
                  then temp := false
                  else
                    begin
                      p1 := p1.sibling;
                      p2 := p2.sibling;
                    end;
                  return temp and p1 = nil and p2 = nil
                              and typeEqual(t1.child,t2.child=
                  end
  else if t1 and t2 are type names  (* if also names are checked *)
  then     return typeEqual(getTypeExp(t1), getTypeExp(t2)
  else     return false
end; (* typeEqual)
```

Listing 7.29: Checking for structural equality

We see how the recursive procedure descends the two trees, as long as they are of the same "shape". If there is some deviation, the traversal of the two trees stops and reported that the trees are not equal. The pseudo code resembles a bit how it could be done in C. In particular, care has to be taken of the nil-pointer. Not only that there should be no nil-pointer exceptions. Also the case where one tree is fininished (a nil-case) but the other is not has to be counted as that the two types are not equal.

Listing 7.30 shows some simple data structure to represent abstract syntax trees for types. It's an example of an inductive data type, supporting 3 basic or simple types (booleans, integers, and floats), which represent leaves in a abstract syntax tree, and some compound types. The representation may be a bit simplistic, in that one represents the "signature" of a record or a union by simple lists of pairs, but then again, it would work, and the representation is used for illustration only. As far as records and unions are concerned: it's *assumed* that the lists contain the field names (respresented as strings) are contained in the same order. We know that the language may allow that the order of writing down the fields may not matter. If that's so, one should make sure that in the abstract syntax tree, the fields are listed in a "standardized" form, maybe by ordering them. Or using not lists in the AST, but some look-up structure, where the order does not matter.

```
type texp = (* abstract syntax for types *)
    TBool
  | TInt
  | TFloat
  | TArray of int * texp
  | TRecord of string * (string * texp) list
  | TUnion  of string * (string * texp) list
  | TPointer of texp
  | TFunc    of texp * texp
```

Listing 7.30: Checking for structural equality

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

Table 7.5: Type syntax intended for nominal equality

Listing 7.31 then show a procedure checking for type equality assuming **structural** equality for types.

```
let rec t_structequal ((t1, t2): texp * texp) =
  match (t1, t2) with
    (TBool, TBool) | (TInt,TInt) | (TFloat,TFloat) -> true
  | (TArray(size1, t1'), TArray (size2, t2')) ->
     ( size1 = size2 && t_structequal (t1', t2'))
  | TRecord(name_1,flist1), TRecord(name_2,flist2)
    | TUnion(name_1,flist1), TUnion(name_2,flist2) ->
     (name_1 = name_2 && t_structequal_fields (flist1 ,flist2))
  | (TPointer t1', TPointer t2')  ->
     t_structequal(t1',t2')
  | (TFunc(s1',t1'), TFunc(s2',t2'))  ->
     t_structequal (s1', s2') && t_structequal (t1',t2')
  | _ -> false
and t_structequal_fields (l1,l2) = (* assume field names in same order *)
  match (l1,l2) with
    ([],[]) -> true
  | ((fn1, tl1)::rest1, (fn2, tl2)::rest2) ->
     (fn1 = fn2)  && t_structequal (tl1, tl2) && t_structequal_fields(rest1,rest2)
  | _ -> false ;;
```

Listing 7.31: Checking for structural equality

Nominal equality

Let's do the same for nominal equality and for the variation of the syntax from Table 7.5. It should be obvious that checking for nominal equality is **simpler** than to check for structural equality, actually pretty much so, it's quite trivial (see Listing 7.32).

```
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
```

Listing 7.32: Checking for nominal equality

Of course, in practice, complications may enter. For instance the names of types may occur in scopes, and that has to be taken into account, but those are orthogonal issues.

7.3.1 Type aliases or synonyms

We have mentioned the concept already earlier, to give (alternative) names to a type. In the example from Listing 7.24, the type for pair of integers was named `pair_of_ints`, so that is a type synonym for `int * int`. If one has a mechanism to give names to types, one can also do multiple synonyms for the same type. In that case, the different names would be called `type aliases`. So basically it refers to the same mechanism. Of course and as hinted at, to be known under different names does not count typically as a form of polymorphism. Many languages offer type synonyms, including C, Pascal, ML, ... For a programmer, it's a convenient mechanism to work with abbreviations (like type `Coordinate = float * float`), and it's a rather light-weight mechanism. It can be use to

In Listing 7.33, type `t1` is made known also under the name `t2`.

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

Listing 7.33: Two type names

All that seems straightforward, but what type aliasing implies for type equality for different classes of types may differ. In that sense, it can be more confusing than it looks at first sight.

Let's compare the situations inListing 7.35 and in Listing **??**. The first example introduces two synonyms of the basic type of integers.

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

Listing 7.34: Type alias for simple types

In this situation, `t1` and `t2` are often treated to be the *"same"* type. That may be different when deadling with compound types.

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

Listing 7.35: Type alias for structured types

In the second example, it's often that `t3` $\neq$ `t1` $\neq$ `t2` (but `t2` and `t3` are the same). The upshot is: even within one language, it may be that different rules apply when it comes to different kinds of types. Perhaps for synonyms of basic types (like integers), the equality "carries over" but for more complex one (like arrays in the illustration), it may not.

7.4 Type checking

Finally, we have to discuss how to realize a type checker. A bit of it we have seen when talking about type equality in Section 7.3.

The task of static type checking resp. the static type checker is to determine whether at given program is well-typed, i.e., adheres the type discipline for a given language, or ill-typed. In the latter case, the compilation process stops and hopefully the type checker generates a meaninful error message.

Actually, the type checker does not only give this binary decision, well-typed vs. ill-typed, it checks for well-typedness of a program including all substructures (expressions, statements, procedures ...) and give back the *type* (when well-typed). That means type *checking* is not really the problem of checking whether a program has an (expected) type, it's to *determine* the type if any.

The type checker operates on the AST and it should not come as a surprise that it's a recursive procedure with the AST as input (and additionally the symbol table that may to be consulted and updated during the run of the type checker).

Part of the type checker, as subroutine, is typically the check for type equivalence. For example, if a procedure is called with an argument, the type checker determines the type of the argument, it determines which type the procedure expects, and compares them, checking whether they are equivalent. If one had a more flexible type system that allowed subtype polymorphism, instead of checking for equality, a subroutine for subtype checking would be used (but we don't really cover that).

Type checking, as said involves traversing the AST and that typically involves *top-down* and *bottom-up* parts; in the terminology of attribute grammars, it involves both inherited and synthesized parts.

7.4.1 General remarks about a type checker

Before we look concretely at a simple type system for a fragment of a simple language involving expressions and statements.

Type system vs. type checker

The (static) type checker is the part of the compiler that decides which syntactically correct program is well-typed and which not (and when doing so determines the types of the syntactic constructs). That the type checker determines well-typedness is almost a

tautology. It's equally not too insightful to say: a program is well-typed if survives the type checker.

There is of course the issue of type safety which we discussed ("well-typed programs cannot go wrong"). So it's a requirement (in a type-safe language) that the type system prevents certain errors. That's a correctness criterion that the type checker should satisfy.

It is, in my eyes, useful to distinguish between between a type **system** and a type **checker**. Why is that? The language's type discipline, the regiment that says what is allowed, type-wise and what not, needs not only to be implemented, it needs also to be communicated to the programmer.

The above viewpoint that a program is well-typed if it survives the type checker is not only tautological, it's also not very helpful.

One can of course describe the type discipline in English text and perhaps using illustrative examples; we do that for *compila* in the oblig. Additionally one can design a bunch of specific small programs, that cover different aspects including corner cases and ill-typed programs. Also those examples can be informative, resp. can be used to test a given type checker. Also that we do in the second oblig.

What's then a type checker vs. a type system. A useful distinction is the following: The type system is the **specification** of the rules or regiment governing the use of types in a language, a specification of type discipline, and also the specification what the type checker has to realize. The type checker has to be **algorithmic**, i.e., correspond to an algorithm, traversing the AST in one way or the other and determining types.

Note that I did not say the type checker is the actual *implementation*. Ultimately the corresponding part of an compiler implementation is of course a type checker which hopefully realizes the type system as specification. It's the difference between a (description) of an algorithm and it's programmatic realization or implementation in a programming language.

Doesn't that mean, in a way, one has 2 specificiations of the implementation, the type system and the algorithmic version, the type checker? Yes, indeed. The question is, however, why does one need or wants sometimes two specifications so to say? In our lecture, we don't actually see much of a need for that. The type system we will look at later is quite simplistic. The same holds for the type system of the oblig. That means, specifying a type system (like with a set of rules or with an attribute grammar) gives enough information and guidance to straightforwardly implement it. In the oblig, we even describe the type discipline only in English.

Modern programming languages, however, can have very complex and intricate type disciplines. Often, it's simpler to describe a type discipline *without* first focusing on algorithmic aspects. That makes a formal description simpler, and when investigating novel and complex aspects of a newly invented type investigating, a English text may not cut it any longer. Especially not, if one needs to investigate whether the newfangled discipline is type safe or has other desired or undesired properties.

In simple situations, the type system directly corresponds to an algorithmic type checking specification. In more advanced systems, that's seldomly the case. There is work to be done to massage the specification into a algorithm, and sometimes it's not even possible.

With advanced feature like complex forms of polymorphism and type constructor, it's easy to specify a (meaningul and even type safe) type discipline, only that, with the type system as specification, the problem is *undecidable*.

For instance, C^{++} has an undecidable type system. One may see that as problem, or maybe not insofar in practice it works. There are no naturally occuring programs whose correct type checking simulates the solution of the halting problem for Turing machines. . . .

In Sections 7.4.2 and 7.4.2 we provide two ways how one can specify a type system, one with attribute grammars, one with derivation rules. Both specifications describe the same simple type discipline covering expression and statements. The corresponding grammar is given in Table 7.6. Since the type discipline is so simple (in particular there is no polymorphism), there rules of the type system more or less directly correspond to an algorithic solution.

Polymorphism

Mentioning polymorphism, we could make some very high-level remarks there, without going in any form of details. As indicated, polymorphism or other advanced features like type inference can complicate type systems and corresponding algorithmic problems considerably. In connection with polymorphism, I like point out only one thing, more like food for thought and not providing concrete solutions for concrete forms of polymorphism.

The thing is the following. We mentioned that the task of a type checker is to check well-typedness, of course, but, when well-typed, also give back the type of a construct. Now, in a polymorphic setting, a construct can have *multiple types*, that's by definition of being polymorphic. Now that leads to the question: what should a type checker do then? For type system, the specification, it's not a big issue. One can specify a type system loosely in a way, that allows to derive multiple types, depending on how one does the derivation steps. The different possible types are thus represented in the type system by the fact that the system incorporates some *non-determinism*.

NB: for attribute grammars, non-determinism is not forseen. In their standard form as covered by the lecture, solving an dependency graph means finding *the* unique solution. Attribute grammars must be formulated in such a way that dependency graphs a acyclic for instance, to make sure that this unique solution exists. In other words, attribute grammars are ill-equipped to specify with polymorphism. In our simple illustration in Section 7.4.2, it's not an issue (and it's connected to the fact that there is not much difference here between the type system and the type checking algorithm. That's also the reason, why AG may not be the formalism of choice when specifying type system (and type checking algorithm). Therefore we look afterwards in Section 7.4.3 to a formulation based on derivation rules. Of course, in a way, it's just a different representation of the same thing. However, the derivation rule based representation is more flexible in more complex situations. This is the reason, why it's a more suitable format and preferred often when studying type systems.

That's good to hear, but we have actually not addressed the question from the beginning: in a polymorphic discipline, what should the type checker return, if the type system non-detrministically allows many types?

Without going into details: the type *checker* need to give back deterministically *one* type, not many, it needs to be deterministic. A non-option is to give back literally *all* possible types. It's a non-option because there can be infinitely many (or at least very many) and it's not practical anyway. Another non-option is to just give back an *arbitrary* type, for instance the first one found that is among the allowed one. That does not really work, since the "first type found" may turn out the not be compatible with the rest of the program, and then one may be forced to **backtrack**, and try if another type can be found the work better. While possible, it's unpractical as well, leading to a combinatorial search of all combinations of types here and there, until one may find a combination that works for the whole program.

But what options remain then? The trick typically is that the type system is designed in a favorable way so that one can derive **"the best type"** at each given point. For instance for subtype polymorphism, it would correspond to the **most specific** (or **minimal**) type. Intuitively that makes sense. For instance, one one type checks the instance of a class `C` (in Java), that instance can be typed by `Object`, and one could continue with the rest of the program with that type. That's obviously stupid to do, one will in all probability encounter a situation where one does something *specific* for objects from `C`. There is not much one can do with instances from `Object`, perhaps cloning, printing, and comparing for identity, but that's it. Using `C` as the most specific type is obviously the right thing to do.

At any rate, if a polymorphic type system is designed in this way, namely that it can operate always with a "best" type (minimal in the setting of subtyping), then there is a good chance that one can turn the specification into a more or less efficient algorithm. If no such "best" type exists, the system is probably ill-designed. It might still be possible to solve type checking algorithmically (probably resorting to combinatorial search and backtracking), which would make type checking of high computational complexity. That would mostly be unacceptable, maybe not even mainly because of the complexity, but because it's difficult to explain what the type systems does for the user: it gives back some type, but there could also be others unrelated, neither better nor worse. It would be like given back some random type. For the subtyping the message is pretty clear: the type system gives back the best, the minimal type of an expression, and there is *one* best type, and all other types of that expression are supertypes of that. That's how it's supposed to be and the key for a type checking algorithm.

7.4.2 Attribute grammar specification

Let's start with a representation of type checking with attribute grammars. The syntax for which we want to do type checking is shown in Table 7.6

When drawing the parallel that type checking is a bottom-up (synthesized) task, that is only *half* of the picture. The presentation focuses in a large part on type checking of expressions (and statements). When it comes to *declarations* (i.e., declaring a type for a variable, for instance), that part corresponds to inherited attributes. Remember that one standard way of implementing the association of variables ("symbols") with (here) types (which can be seen as an "attribute") are symbol tables.

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

Table 7.6: Grammar for statements and expressions

The attribute grammar from Table 7.7 are pretty straightforward. Actually, the attribute grammar deviates from the purist view we sometimes used earlier. In particular, the rules make use for some exception mechanism (`type-error(exp)` or similar). That would not officially be possible in an attribute grammar, and in earlier examples, we used attributes to mimick exceptional behavior. That cluttered the semantic rules with additional checks, namely whether or not an error had occured. This is not needed if one assumes exceptions and the specification become more readable. We also see that the type equality procedure `typeEqual` from earlier is used in a couple of place as subroutine.

Coming back to the issue of exceptions. Exceptions are raised when a type error is detected. Most productions, in particular those for compound expressions, contain a positive case, when the types "fit" and a negative one, when that's not the case and a type error is raised.

The derivation-rule-based presentation in Section 7.4.3 afterwards does an even more "economic" representation. It focuses solely on the **positive cases**, i.e. which conditions need to be met to be well-typed. The negative cases, which here raise an error (and in the mentioned purist attribute grammars resulted in that an error-attribute was set to `true`) are simply not covered by rules. The type system stated: a programm is well-typed, if there **exists** a derivation in the system of type rules that derive its type. Left *implicit* is, that no if such type is derivable, the program is ill-typed.

It's not that it it is impossible, one could clearly either do rules with an `if-then-else` construct or basically duplicate the rules. Assume the rule that requires for of a sum expression $e_1 + e_2$, that both e_1 and e_2 are of integer type (and then concluding that sum is of interger type as well). This could be accompagnied by a second rule for the negative case, stating that, should e_1 or e_2 or both be not of type integer, then that's a type error.

As said, that's easy to do. And in an actually implementation, one need to cover the negative cases as well.[13] Still, in most specifications of type systems one focuses on the positive cases, simply not mentioning the negative ones. This allows to focus on the core of the type system (knowing of course, that type errors need proper handling in an implementation as well).

[13] Having cases in a case construct uncovered would derail the type checking, leading to errors, but that would be an undignified way of signalling a program ill-typed ...

We will say a few more words comparing the attribute grammar presentation with the rule-based representation once we have seen it in Section 7.4.3.

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*)

Table 7.7: Type system as attribute grammar

7.4.3 Type system given by derivation rules

The following formalizes basically the same type system one more time. It uses a style of representation, which borrows from “logics”, capturing the type system as a set of *derivation rules*. It’s a form of presentation often employed specifying type systems, including those of a more complex nature. It’s not a coincidence that such presentations resemble logical derivations. There are deep connections between (mostly intuitionstic or constructive) logics and type systems, but that goes beyond this lecture.

We know that type checking makes used of a symbol table. That was also visible in the attribute grammar and the semantic rules of Table 7.7. The symbol table is not actually mentioned in the attribute grammar, but the procedures `lookup` and `enter` operate on a symbol table data structure left unmentioned otherwise.

Also here, we need a **symbol table**. Conventionally, one uses a greek letter for that, commonly Γ. In presentations like the one here, Γ is mostly not called symbol table but context or environment. That’s probably also the reason for choosing Γ (being the greek letter for “C” as in “context”). A symbol table (or context, etc.) is a data structure to store associations of (here) types with variables or identifiers, and we need to be able to

add a new binding and to look up the type associated with a variable (and also we need to refer to the empty symbol table; at the beginning the table is empty). As for notation, we use $\Gamma(x)$ for the type associated with x. That corresponds to the `lookup` function. In this notation, Γ is simply seen as a finite function: applying it to x gives back the type, if defined. If not defined, an error would be raised, but as discussed earlier, we focus on the positive cases... The notation $\Gamma, x : T$ represents the context that works like Γ but extended by a new binding, namely associating type T to variable x.

Now to the derivation rules, split into two parts in Table 7.8 and Table 7.9.

The rules are to be read as follows: There are premises (above the horizonal line) and one conclusion (below the horizonal line). The derivable "assertions" are of the following form:

$$\Gamma \vdash p : T \tag{7.2}$$

Those as are also called judgements sometimes, in particular typing judgments. A judgement of the one from Equation (7.2) is to be read as follows:

> given the context Γ then progam p is of type T.

The context Γ, as said correspond to the symbol table. One find also the terminology that Γ is called *assumption* or *hypothesis*. That terminology also makes sense: A (derivable) judgment like

$$x : \mathtt{int}, y : \mathtt{int} \vdash \; x + y : \mathtt{int}$$

expresses "*assuming* that both x and y are integers, then the expression $x+y$ is an integer as well". Also the terminology "context" for Γ is not too bad. It reminds us that type checking is a *context-sensitive* analysis. The syntax of programs, in our case of expressions, statements etc from Table 7.6 is **context-free**, i.e., without context. Type checking (as most semantic analyses) fall in the broad category of being context-sensitive (and no longer context-free). And Γ is just the context, resp. the particular form of context needed to do the particular problem of type checking.

What is written as p in equation (7.2) is generically meant as "program piece", concretely the type rules with work with expressions, statements etc.

So the rules specify how one can derive such judgements from other judgements. That may directly be translated into a algorithm, or may not be used as algorithm directly, depending on the way the rules are formulated. In our simple case, the rules directly correspond to an algorithm.

Typically, when the language and the type system is complex, one may *specify* well-typedness in such a manner, without the rules immediatedly translatable to a type checker, or maybe not at all, insofar one may have specified an *undecidable* typing relation. A major complicating factor maybe polymorphism, as mentioned, but we don't have that here.

A derivation system simply says, p is of a type T if, with the given rules, one can derive the corresponding judgment, i.e., if there *exists* a derivation. It does not per se require, that the deriviation is unique, or that p may not have other types (in which case the type

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

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

$$\frac{\Gamma \vdash exp_2 : \ \texttt{array_of}\ T \qquad \Gamma \vdash exp_3 : \texttt{int}}{\Gamma \vdash exp_2\,[\,exp_3\,] : T}\ \text{TE-ARRAY}$$

$$\frac{\Gamma \vdash exp_1 : \texttt{bool} \qquad \Gamma \vdash exp_2 : \texttt{bool}}{\Gamma \vdash exp_1\ \textbf{or}\ exp_2 : \texttt{bool}}\ \text{TE-OR}$$

$$\frac{\Gamma \vdash exp_1 : \texttt{int} \qquad \Gamma \vdash exp_2 : \texttt{int}}{\Gamma \vdash exp_1 + exp_2 : \texttt{int}}\ \text{TE-PLUS}$$

Table 7.8: Type system (expressions)

system is polymorphic). But that's fine. In such more complex situations, the rules would not directly yield an algorithm, it may be seen as a specification of a type discipline.

Note: the way we presented the attribute grammars, we can't allow ourselves such a relaxed attitude, being happy if there is one solution among different possible ones. Attribute grammars require one definite solution, no non-determinism or cycles or undefined situations allowed. That (among other reasons) makes it often less straightforward to use for specifying a type system. One aspect where it's also visible is: in the attribute grammar, we explictly had to specify (in a not too elegant way) error-situations. The rules here don't do that. For instance, in the treatment of the conditionals, it's required that the expression is a boolean. If it should be the case that it's not a boolean, there is no rule that covers that situation, which means, the well-typedness judgment for a program containing such a situation is not derivable. Which means, it's not well-typed and contains thereby a type error. A concrete type checker would have to produce a meaningful type error message in that alternative scenario, but that's supressed in the rule-based presentation. The core of the type system is focusing on the positive cases, leaving the type errors implicit and leaving it up to the implementor to figure out how to deal with uncovered situations. Similar relaxedness applies to rules that would include non-determinism: the implementor has to figure out how to deal with it, i.e., how to turn the specification in an algorithm. The concrete type system here is so simple (monomorphic) that the rules are basically an algorithm already.

One property of a static type discipline is: to be well-typed *all* parts of the program code needs to be well-type. Even if there is *dead code*, i.e., code that will never be executed, it *still* need to be type-correct, and the type system checks everything.

That principle is clearly visible in the rule-based formulation. In the attribute grammar of Table 7.7, for instance in the case of `if` *exp* `then` *stmt*, the corresponding semantic rule requires that *exp* is of boolean type (same as rule TS-IF). However, nothing seems to be required for *stmt*. That gives the impression (in the attribute grammar) that a conditional is well-typed if the expression is of boolean type and that's all. But that's not

$$\frac{\Gamma, x : \texttt{int} \vdash \mathit{rest} : \texttt{ok}}{\Gamma \vdash x : \textbf{int};\, \mathit{rest} : \texttt{ok}}\ \text{TD-INT} \qquad \frac{\Gamma, x : \texttt{bool} \vdash \mathit{rest} : \texttt{ok}}{\Gamma \vdash x : \textbf{bool};\, \mathit{rest} : \texttt{ok}}\ \text{TD-BOOL}$$

$$\frac{\Gamma \vdash \mathit{num} : \texttt{int} \qquad \Gamma(\mathit{type\text{-}exp}) = T \qquad \Gamma, x : \texttt{array}\ \mathit{num}\ \texttt{of}\ T \vdash \mathit{rest} : \texttt{ok}}{\Gamma \vdash x : \textbf{array}\,[\,\mathit{num}\,] : \mathit{type\text{-}exp}\,;\, \mathit{rest} : \texttt{ok}}\ \text{TD-ARRAY}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash \mathit{exp} : T}{\Gamma \vdash x := \mathit{exp} : \texttt{ok}}\ \text{TS-ASSIGN} \qquad \frac{\Gamma \vdash \mathit{exp} : \texttt{bool} \qquad \Gamma \vdash \mathit{stmt} : \texttt{ok}}{\Gamma \vdash \textbf{if}\ \ \mathit{exp}\ \ \textbf{then}\ \ \mathit{stmt} : \texttt{ok}}\ \text{TS-IF}$$

$$\frac{\Gamma \vdash \mathit{stmt}_1 : \texttt{ok} \qquad \Gamma \vdash \mathit{stmt}_2 : \texttt{ok}}{\Gamma \vdash \mathit{stmt}_1\,;\, \mathit{stmt}_2 : \texttt{ok}}\ \text{TS-SEQ}$$

Table 7.9: Type system (statements and declarations)

the case. Depending on what case of a statement it is, the corresponding check is specified in connection with other productions.

One can probably reformulate the attribute grammar. For instance, in the attribute grammar, *stmt* does not have a type, i.e., there is no attribute for that. To be more in line with the treatment using rules, one should have added a type (perhaps `void`) to convey information that a *stmt* nodes is well-typed. Instead of `void` one could have used `ok` indicating well-typedness, as it's done in the rules of Table 7.9, same thing. That could have been used to make more clear that *stmt* nodes also need to be type checked, something that is not so transparent in the given attribute grammar. For the derivation rules, it's obvious that all parts of a syntax tree are traversed and checked.

A final remark. I mentioned that the type system or type checker relies on equality checking as subroutine. The routine is explicitly invoked in the semantic rules of the attribute grammar. But what about here? Also that is a bit suppressed. One could have been more explicit. For instance, instead of the shown rule TE-PLUS, one might formulate it equivalently as follows.

$$\frac{\Gamma \vdash \mathit{exp}_1 :\ T_1 \qquad \Gamma \vdash \mathit{exp}_2 :\ T_2 \qquad T_1 =^{?} \texttt{int} \qquad T_2 =^{?} \texttt{int}}{\Gamma \vdash \mathit{exp}_1 + \mathit{exp}_2 : \texttt{int}}\ \text{TE-PLUS}'$$

For clarity I used $=^?$ to denote the type equality check, where TE-PLUS simply wrote "=" (why bother to write $=^?$, the equal sign is clear enough, even if equality may be not so trivial, as discussed).

Anyway, the more verbose version from TE-PLUS′ is perhaps also more clearly in the spirit of an algorithm. Like

> Assume the type checker wants to determine the type (if any) of $\mathit{exp}_1 + \mathit{exp}_2$ in a given context Γ: let's write $\Gamma \vdash \mathit{exp}_1 + \mathit{exp}_2 : ?$ for the problem of having the

type checker determine the type of the sum (actually if the sum is well-typed at all, it must be `int` in our simple type system). To figure it out, the type checker is recursively invoked on the subexpression exp_1, still with context Γ, let's write $\Gamma \vdash exp_1 : ?$ for it. That gives back, when successful, some type T_1. That's the premise $\Gamma \vdash exp_1 : T_1$. Do the same for the other subexpression exp_2, corresponding to the call $\Gamma \vdash exp_2 : ?$ resp. the premise $\Gamma \vdash exp_2 : T_2$. Then check that both T_1 and T_2 are (equivalent to) `int`, and if so, report back `int` for the sum-expression. That corresponds to the conclusion $\Gamma \vdash exp_1 + exp_2 : \texttt{int}$.

Anyway, especially in relatively simple type systems, most are happy enough with a formulation like TE-Plus, suppressing all unnecessary side issues (type equality, error situations) and focus on the core.

Bibliography

Bibliography

[1] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522.

[2] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index

abstraction, 2
ad hoc polymorphism, 28
aliasing, 23
array, 12
array type, 10

basic type, 4

CAM, 12
class
  and types, 25
closure, 24
coercion, 28
compound type, 7
constructor, 19
content-addressable memory, 12
conversion, 28

data-flow analysis, 4
dependent type, 1, 4
dereference, 22
discriminated union type, 17
dynamic typing, 3

function
  higher-order, 25
function type, 10
function variable, 23

generic polymorphism, 29

higher-order function, 25

inclusion polymorphism, 29
inductive data type, 19
`int32`, 6
`int64`, 6

late binding, 27

Milner's dictum, 3
ML, 19
monomorphism, 28

null pointer, 23

overloading, 6
ownership, 1

parameter passing, 7
Pascal, 17
pattern matching, 19, 21
pointer
  dereference, 22
pointer arithmetic, 22
pointer type, 10, 22
polymorphism, 28, 30, 40
  ad hoc, 28
  generic, 29
  inclusion, 29
  subtype, 29
  universal, 29
product type, 13

record
  variant, 17
record type, 10, 12
reference type, 7
run-time type, 3
Rust, 1

signature, 10
static typing, 1, 3
subsumption, 29
subtype polymorphism, 29
subtyping, 10, 38
sum type, 15, 17
symbol table, 43

tagged union type, 17
tuple, 14
tuple type, 10, 13
type, 1, 3
  array, 10
  dependent, 1, 4
  derivation rule, 44
  inductive, 18
  sum, 15
  union, 15
type checking, 1
type class, 29
type constructor, 5, 10
type safety, 3, 16
type synonym, 30

Index

type theory, 1

union type, 10, 15–17
universal polymorphism, 29

variant record, 17