

---

# CS 421 – Spring 2007

## Lecture Notes Set 12:

### User Defined Recursive Types

Elsa L. Gunter<sup>1</sup>

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 07 - typederiv-1 (slides start - 40)

Made available: February 16, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Types Systems (slide 2)

Previously, we discussed creating types yourself. Infinitely recursive types, trees, etc. In this lecture, we will discuss how types can be used (data abstraction) and how they are interpreted (type checking)

### 2.1 Terminology (slides 3-4)

When we refer to a type  $t$  we are defining a set of possible data values that  $t$  can hold. So, for example, in C, a short is an value in the set  $-2^{15}, \dots, 2^{15} - 1$ . Booleans have a value in the set true, false. The values in this set are said to have type  $t$ . So type systems are the rules in a language that assign types to expressions.

We use these types to describe properties. For example, types can be used to define if data is read-write or read-only. We can use a type to decided if some operation has the authority to access the data. We can also check to see if data came from the right place or that see if some operation might raise an exception. Like in Java, you can throw an input/output exception, as the signature of a method and that can be thought of as part of the type for that method. Most languages have the types describe the layout of the data and how they can be accessed.

### 2.2 Sound Type Systems (slide 5)

Types can guarentee us some amount of safety within a program. What this safety is, is a sound type system. It is where if an expression has some type  $t$  and then the expression evaluates to some value  $v$ , then we know that  $v$  is in the set of values defined by the type  $t$ . OCaml has a sound type system, but languages like C and C++ generally do not.

### 2.3 Strongly Typed Languages (slides 6-7)

If we are writing in a language, where when you apply an operator to some arguments, and there can be no run-time type errors, the language is considered strongly typed. For example,  $1 + 2.3$  in OCaml will complain at compile time. It will tell you that you are using a float where there should be an integer.

However, this is also a vague term and depends on the definition of type error. That is, C++ claims to be strongly typed, but with Unions allowing types to be used in other places, type coercion causing undesirable effects, etc, some call the claim to be false.

OCaml is strongly typed, but still makes sure it does array bound checking, runtime type case analysis, etc.

---

<sup>1</sup>© 2007, Share and Enjoy

## 3 Type Checking

### 3.1 Static vs. Dynamic Types (slide 8)

A static type is a type that is assigned to an expression at compile time, where a dynamic type is a type assigned to a storage location at run time.

So, statically typed languages (also known as typed languages) are those where every expression is statically type assigned at compile time. Dynamically typed languages (also known as untyped languages) where types of an expression are determined at run time.

### 3.2 What and when (slides 9-10)

If we have an operation, applied to some arguments, type checking makes sure that we apply the operation to the right number of arguments of the right types. Right type means that the argument has the same type as what was specified earlier or it means that the type used was a predefined implicit coercion. This is used to resolve overloaded operations.

So, type checking may be done at two different times. It can be done statically at compile time, or dynamically at run time. Dynamically typed languages (like LISP and Prolog) do only dynamic type checking. Where statically typed languages (like C, C++, and Java) can do most of its type checking statically.

### 3.3 Closer look at Dynamic Type Checking (slides 11-12)

Here, there is no type checking at compile time. During run time, before an operation is performed, the type of the operation and the arguments are checked. The types of the variables and operations are all left unspecified until runtime. There is an advantage to this, and that is the same variable can be used as different types. So for example, you started out using a variable as an integer, but later on, you wanted to redeclare it as a string. Dynamically typed languages will allow this.

The disadvantages are that, the data object that you are using must hold the type information needed, since your environment doesn't have it. Also, since everything is done at runtime, some errors may not be detected until the erroneous part of the code is executed, which may not until years after the code was written.

### 3.4 Closer look at Static Type Checking (slides 13-14)

Static type checking is performed after parsing, but before running the program. So we parse the program and get its syntax tree, and then we check to see if it's type safe. Then if we find that it is type safe then we can throw out all the type information because we already know that it is type safe and then start running the program.

So an advantage is that once static type checking finds that a program is type safe, the information does not need to be kept, because we know the program makes sense. So we get rid of the need to store that information like dynamic type checking needs. Static type checking also catches many program errors at early points. However, a problem arises because we can't check types that depend on dynamically computed values, like array bound checking. That has to be done at runtime.

This all depends on the idea that for every variable, every expression, every operator, their types have to be known at compile time. This can either be done by type inference or by declaring variables. OCaml mainly does type inference.

### 3.5 Type Declarations and Inference (slide 16-17)

A type declaration is the explicit assignment of types to variables or functions. But if you have type inference, type declarations are not needed. Type inference is a program analysis that infers the types for you. OCaml uses this, but types like Records cause problems for this.

So, when you use the addition operator, the +, OCaml sees it and says this operator works on two integers, so that is what we are going to look for. Then OCaml also knows that after performing that operation, an integer should be returned.

## 4 Mathematics of Type Checking

### 4.1 Type Judgements (slides 18-19)

A type judgement looks like:

$$\Gamma \mid\text{- exp: } \tau$$

$\Gamma$  is the typing environment, which supplies the types of variables and functions. It is a list in the form  $[x: \sigma, \dots]$ . So, it has a list of variable names and associated types.

The 'exp' is the program expression.

$\tau$  is a type that is being assigned to the exp.

$\mid\text{-}$  is called 'turnstile', 'entails' or 'satisfies'.

So you can interpret the line above as 'the environment gamma satisfies that the expression has type tau'. Or you can also read it as saying, 'the expression is guaranteed to have type tau in the type environment gamma'.

Here are some simple examples of valid type judgements.

$$[] \mid\text{- true or false : bool}$$

This first example is saying that the empty environment satisfies that the expression true or false has type bool.

$$[x : \text{int}] \mid\text{- } x + 3 : \text{int}$$

This is saying that in the environment where  $x$  is an integer, we are going to take  $x$  and add it to three, and that will return an integer.

$$[p : \text{int} \rightarrow \text{string}] \mid\text{- } p(5) : \text{string}$$

Here, in the environment, we have that  $p$  is a function that takes an integer to a string, and that satisfies that we have an expression where  $p$  is applied to the integer 5 and that has the type string.

### 4.2 Typing Rules (slides 20-21)

We have a set of typing rules to show that the expression have a certain type. The typing rules have the following format.

$$\frac{\Gamma_1 \mid\text{- } exp_1 : \tau_1 \dots \Gamma_n \mid\text{- } exp_n : \tau_n}{\Gamma \mid\text{- } exp : \tau}$$

So below the line, there will be one judgement. This is the conclusion, what we want to prove. Above the line are our assumptions. And this is saying that our conclusion is true if our assumptions hold.

The main idea is that the expression in the judgement below the line is proven by the expressions above the line. If there are no assumptions made for a judgement, then that rule is called an axiom and it is accepted to be true. Lastly, if  $\Gamma$  doesn't matter, it may be omitted.

Note that  $\Gamma$ , exp, and  $\tau$  are parameterized, so they may contain things like meta-variables.

### 4.3 Axioms (slides 22-23)

These are rules that are true with any typing environment. There are no assumptions, simply, they are considered true.

$$\overline{\mid\text{- true : bool}} \quad \overline{\mid\text{- false : bool}}$$

Note that you, for axioms, the overline may be omitted. Or, it may be there, but only a dot above the line.

Now we will look at variable axioms. If we see the notation  $\Gamma(x)$  we are looking for the value of the variable  $x$  in the environment  $\Gamma$ . This goes back to previous discussions of environments, where you look for the first notation of the variable in the environment.  $x$  So  $\Gamma(x) = \sigma$  if  $x : \sigma \in \Gamma$ . That is,  $x$  has the type  $\sigma$  if there exists an expression in  $\Gamma$  saying that  $x$  has type  $\sigma$ .

So that would look like:

$$\overline{\Gamma \mid\text{- } x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

### 4.4 Simple Rules (slides 24-25)

So we will start out with the Arithmetic Rules.

We know we have these primitive operators for arithmetic. ( $\oplus \in +, -, *, \dots$ )

So, you can have something that looks like:

$$\frac{\Gamma \mid - e_1 : \text{int} \quad \Gamma \mid - e_2 : \text{int}}{\Gamma \mid - e_1 \oplus e_2 : \text{int}}$$

What this is saying is that we have two variables  $e_1$  and  $e_2$  and we want to show that these are integers. And then above the line, that is what we try to show. Then, if they are integers, then we know the arithmetic operator used is ok.

Then there relational operators. ( $\sim \in >, <, =, <=, >=$ )

$$\frac{\Gamma \mid - e_1 : \text{int} \quad \Gamma \mid - e_2 : \text{int}}{\Gamma \mid - e_1 \sim e_2 : \text{int}}$$

Here, we again are trying to see if the types under the line make sense. So, we check to see if  $e_1$  and  $e_2$  are integers again.

Now there are operation between booleans. There is the logical *and* and the logical *or*.

$$\frac{\Gamma \mid - e_1 : \text{bool} \quad \Gamma \mid - e_2 : \text{bool}}{\Gamma \mid - e_1 \ \&\& \ e_2 : \text{bool}}$$

$$\frac{\Gamma \mid - e_1 : \text{bool} \quad \Gamma \mid - e_2 : \text{bool}}{\Gamma \mid - e_1 \ \|\| \ e_2 : \text{bool}}$$

#### 4.5 Simple Example (slides 26-47)

We are going to look at a simple example. We will start with defining  $\Gamma$  to be the set  $[x : \text{int}; y : \text{bool}]$ . We want to show that  $\Gamma \mid - y \|\| (x + 3 > 6) : \text{bool}$ . Now we will start will building the proof tree from the bottom up, to show that the conclusion is valid. We will use the typing rules we have just learned to do this. This is what we start with.

$$\frac{?}{\Gamma \mid - y \|\| (x + 3 > 6) : \text{bool}}$$

So this expression is being operated on by the boolean *or*. So that means that both sides of the boolean *or* has to have boolean types. So that is what goes above the line, that both expressions are of type bool. So we get:

$$\frac{\Gamma \mid - y : \text{bool} \quad \Gamma \mid - (x + 3 > 6) : \text{bool}}{\Gamma \mid - y \|\| (x + 3 > 6) : \text{bool}}$$

Now we have to show that these expressions are of type boolean. So we arbitrarily choose an expression to look at. Let's look at the left expression first. The variable rule is the rule that shows that  $y$  is a boolean. To check this, we look at  $\Gamma$ . And when we look there, we see that  $y$  is a boolean. So we do have that this assumption holds.

Looking at the right expression, we have  $x + 3 > 6$ . We see the relational operator. So that leads to to integers being on both sides. So we get:

$$\frac{\Gamma \mid - x + 3 : \text{int} \quad \Gamma \mid - 6 : \text{int}}{\Gamma \mid - y : \text{bool} \quad \Gamma \mid - (x + 3 > 6) : \text{bool}}$$

So we can easily see that 6 is an integer, so that side is done by the Rule for Constants. The other side has an arithmetic operator as we have the rule for arithmetic operations. Now we have to split up the other expression and we get:

$$\frac{\Gamma \mid - x : \text{int} \quad \Gamma \mid - 3 : \text{int}}{\Gamma \mid - x + 3 : \text{int}} \quad \Gamma \mid - 6 : \text{int}$$

$$\frac{\Gamma \mid - y : \text{bool} \quad \Gamma \mid - (x + 3 > 6) : \text{bool}}{\Gamma \mid - y \|\| (x + 3 > 6) : \text{bool}}$$

Now, this is easy, we do the rule for variables by checking  $\Gamma$ . In it, we have that  $x$  is an integer, so that is done. Then we check the 3, and that is the rule for constants, so we see that 3 is an integer. So that assumption is correct. And that's it we have shown all the assumptions hold, making out conclusion at the bottom of the proof tree, correct.