

Course Script

Static analysis and all that

IN5440 / autumn 2024

Martin Steffen

Contents

2 Type (and effect) systems **1**
2.1 Introduction 1
2.1.1 Types and type systems 1
2.2 Type checking 2
2.2.1 A λ-calculus 2
2.2.2 Adding simple types 4
2.3 Type inference 12
2.3.1 Type inference problem 13
2.3.2 Substitution, unification, and unification constraints 16
2.3.3 Unification 23

Chapter 2
Type (and effect) systems

What is it about?

Learning Targets of this Chapter

type systems
effects
functional languages
type inference and unification

Contents

2.1 Introduction 1
2.1.1 Types and type systems 1
2.2 Type checking 2
2.2.1 A λ-calculus 2
2.2.2 Adding simple types . 4
2.3 Type inference 12
2.3.1 Type inference problem 13
2.3.2 Substitution, unification, and unification constraints 16
2.3.3 Unification 23

2.1 Introduction

In this part, we cover various *type systems*. In the book [2], the focus is on the "non-standard type systems" parts (covering flow information and effects). The slides reorder the material a bit, in particular we start covering standard type systems and type inference first (in case to have that covered, should the time run out).

In the overall introduction, we used the while-language to illustrate type and effect systems or annotated type systems in a simple setting, where the typing part was trivial. In this section, we deal mainly with functional languages, i.e., λ-calculi; we touched upon also that in the introduction already.

2.1.1 Types and type systems

As said in the introduction, the notion of "type" is rather broad. Types play different roles in programming languages. We focus here on *static* type system. There are connections between those roles.

Describing the memory needs for data

2.2 Type checking

2.2.1 A λ-calculus

As in the introduction, when talking about type systems, we focus on functional languages. Of course, for non-functional languages, type systems are equally important. Also the dividing line between functional languages and non-functional languages, in practice, is not so clear. Most languages, that are know as "functional" have non-functional, imperative aspects. And in languages, not considered as functional, one may design programs "functionally" when that fits.

The while-language was presented as some *core* imperative language (with or without procedures). Executing a program resulted in *state-change* (via assignment) and did non involve returning *value*, thus being completely non-functional. In addition to assigmment, there were simple control constructs like sequential composition, conditionals, and loops. Those aspects indeed are core features of a "programming language" and make the language Turing complete.

Now we are shifting to a functional "core language", known as the *lambda*-calculus. Though the syntax, shown in Table 2.1 is not "the" λ-calculus, like one of its canonical pure versions. We say a bit about the selection of constructs soon, but indeed it's a Turing complete functional core calculus, analogous to the way that the while-calculus is a Turing-complete imperative core calculus.

Unlabelled syntax

$$\begin{array}{rcll} e & ::= & c \mid x \mid \texttt{fn}\, x \Rightarrow e \mid \texttt{fun}\, f\ x \Rightarrow e \mid e\ e & \text{terms} \\ & \mid & \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{let}\, x = e\, \texttt{in}\, e \mid e\, \texttt{op}\, e & \end{array} \tag{2.1}$$

Table 2.1: Syntax of some λ-calculus (unlabelled)

The syntax is a variation of an (untyped) λ-calculus, the prototypical "functional language". Instead of "λ" as symbol for functional abstraction, the language here uses `fn` and `fun`. The language distinguishes syntactically between the "standard" abstraction `fn` and abstraction for recursive functions. Standard representations of the untyped λ-calculus would not bother to make that distinction: the **untyped** λ-calculus is Turing complete and therefore expressive enough to *encode* recursion. The most well-known encoding makes use of the so-called **Y-combinator** (see for instance here); those encodings don't play a role for us here, our language directly supports recursion.

Likewise, the untyped λ-calculus could encode conditionals etc. However, the lecture uses the more elaborate syntax for two reasons. We are interested in illustrating program analysis, where control flow constructs like conditionals are most likely to be part of the abstract syntax. Even if conditional would be encodable in a more basic syntax, that will obscure the analysis (and make it less "realistic"). Likewise, recursion will pose specific problems, the treatment of which would be obscured when relying on an encoding.

Finally, we are *not* working with an untyped calculus: we make use of **typed** versions of the language, and for the typed λ-calculus, things change: without adding recursion explicitly, the calculus *is not Turing complete* (which makes it hard to argue that it's a simple form of a standard functional programming language).

Labelled syntax

What does *not* belong to the standard calculus are the **labels**. Those will be needed (as before) for the intended analysis *later* (not for the basic stuff like standard type checking and standart type inference). See Table 2.2.

$$\begin{array}{rcll} e & ::= & c \mid x \mid \texttt{fn}_\pi x \Rightarrow e \mid \texttt{fun}_\pi f\ x \Rightarrow e \mid e\ e & \text{terms} \\ & \mid & \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{let}\, x = e\, \texttt{in}\, e \mid e\, \texttt{op}\, e & \end{array} \tag{2.2}$$

Table 2.2: Syntax of some λ-calculus (labelled)

It's worthwile to remember the CFA from the introduction, where the functional calculus was labelled, as well. Unlike here, the labelling in the introduction was *for all constructs* whereas here, we label only the two forms of abstractions. Abstractions are *anonymous* functions, i.e., functions without a name. Of course, one can give functions a "name"; in the syntax here, that's done with the `let`-construct. When writing something like $\texttt{let}\, f = (\texttt{fn}\, x \Rightarrow e)$, then $\texttt{fn}\, x \Rightarrow e$ is the function itself, whereas f is a *variable* and one can the variable f see it as a *name* for the function.

$$\begin{array}{rcll} \pi & \in & \textbf{Pnt} & \text{program points, labels} \\ e & \in & \textbf{Expr} & \text{expressions} \\ c & \in & \textbf{Const} & \text{constants} \\ \texttt{op} & \in & \textbf{Op} & \text{operators} \\ f, x & \in & \textbf{Var} & \text{variables} \end{array} \tag{2.3}$$

It's not always that one thinks or should think of variables as "names" for entities, like when doing $x := 5$ is a name for 5, or, even weirder, when doing $x := y$, thinking "x is a name for y", i.e. they are synonymous. But we are in a functional setting. That makes variables like f declarative (or single-assignment, and not just *static* single assignment, but really single-assignment). So f is never changed, it just used to represent the anonymous function (which then is no longer anonymous, as it has at least one name). In that way, it's justified to think of f as the name of the function. That a variable is a a way to refer to an (immutable) value whereever it's mentioned and can be replaced by that value, that is known as *referential transparency*. This replacement "whereever you see f you may replace it by $\texttt{fn}\, x \Rightarrow e$, as it's the same" refers to **substitution**, i.e., substution is the technical term for "replacement", and we will encounter that as well: the operational semantics of the language is based on *substitution*. For imperative, updatable variables, thinking of variables as "names" is inappropriate, and one cannot have a purely substitution-based semantics.

Back to the abstract syntax: here, as introduced, the anonymous functions are "labelled" (by a subscript). This is how the analysis keeps track of the function definition, in that sense it's the "name" under which the function is known by the *analyses* (not by the user). In the introduction, we had already a look at some analyses and at some variant of the functional calculus. There, *all* constructs where labelled, whereas here, it's only the abstractions. This difference is motivated by the fact that here we will be interested in a (slightly) different analysis.

Now some simple **examples**:

Example 2.2.1 (Application). The first one is very simply: it applies an anonoymous function to another one, both representing the identity function.

$$(\texttt{fn}_X\, x \Rightarrow x)\ (\texttt{fn}_Y\, y \Rightarrow y) \tag{2.4}$$

Example 2.2.2 (Recursion).

$$\begin{array}{ll} \texttt{let}\, g & = (\texttt{fun}_F\, f\ x \Rightarrow f(\texttt{fn}_Y\, y \Rightarrow y)) \\ \texttt{in} & g\ (\texttt{fn}_Z\, x \Rightarrow x) \end{array} \tag{2.5}$$

It's worthwile to think about the latter example and especially the role of `fun` (the "recursive" function abstraction). To understand the function, one has to look at $(\texttt{fun}_F\, f\ x \Rightarrow f(\texttt{fn}_Y\, y \Rightarrow y))$, of course. The fact that this function is called g is irrelevant, but the name is introduced for later usage (in the second line). It's probably also clear: f is *not* the name of the function, the (recursive) function is anynymous (but given an "outside" name g by the let-construct. Or rather one should say: f is the "inside" name of the function. Recursion means, a function calls itself, and to do some properly, it needs a name, f in this case. The name f acts like a *bound* variable inside the body of the function, completely analogous to x for the function's formal parameter in that example. So, both f and x are *bound* variables scoped within the body of the function.

Remark 2.2.3. *The latter program* loops. *The example is repeated from [2, p.143] (that's from the chapter about constraint-based analysis, which we did not cover). There, the function is the same, but the labelling is different (all constructs are labelled). To interpret what it is doing one needs to recall the syntax and especially what* `fun` *does. The f is the function name, not the argument. Of course, the g is the outside name, but the f is needed for the recursion. So the function ignores its argument (the identity function) and in the body calls itself (on another id-function which is ignored again ...).* □

2.2.2 Adding simple types

We now present a typed version of our λ-calculus. It pretty much looks like what is know as **simply typed λ-calculus**, but there is some twist, in that recursion is expicitly added (wait for some more explanation later). There are many versions of the simply-typed λ-calculus or typed λ-calculi in general. One point of variation is the syntax of the λ-calculus. Here, for instance we are working with 2 different abstractions, the standard one, and one for recursion. Also we assume some syntax (actually not even shown) two

write booleans an integers: the types show that booleans and integers are part of the language, so those and corresponding operators ($+$, $\wedge$ ... are assumed to be supported. The fact that one considers some "real-live" data and types makes it into an **"applied"** λ-calculus, as opposed to a pure one, that might be studied for theoretical purposes ("is it Turing complete?"). Different applied λ-calculi may include (much) more data structures, but still remaining a λ-calculus at heart. Scheme is an example, where there is not much on top of the funtional core, if one ignores the imperative part, and it's not an applied **typed** λ-calculus (typed in the sense of statically typed).

But first to the types. They are shown in equation (2.7).

$$\begin{array}{lcll} T & \in & \textbf{Type} & \text{types} \\ \Gamma & \in & \textbf{TEnv} & \text{type environment} \end{array} \tag{2.6}$$

$$T ::= \mathsf{int} \mid \mathsf{bool} \mid T \to T \tag{2.7}$$

As mentioned, we assume two base types bool and int and standard constants and operators ($\mathsf{true}, 5, +, \leq, \ldots$). Each built-in constant c has one base type T_c (for instance 5 is of type int). We don't bother much about that part of the type system (as it's trivial).

For the formulation of the type system, we need to introduce the concept of type environment. They are basically finite mappings to variables to types. In a practical implementation, they could be implemented by a hash-table or an appropiate dictionary data structure.

$$\Gamma ::= [] \mid \Gamma, x{:}T \tag{2.8}$$

Curry-style typing

One characteristic of the presentation is called **Curry**-style typing. There is an alternative (both equally ancient) which is known as **Church**-style typing (and there are mixed forms). "Curry-style" means that abstractions `fn` $x.e$ *don't* mention a type for the formal parameter (same for let). The alternative with a syntax like `fn` $x{:}T.e$ is called "Church style". Both variants exist in real programming languages. Also, leaving it up to a user whether mentioning a type or leaving it out is preferred on a case by case basis is possible. Some programmers may prefer to add types for being more explicit and documenting the type of expected arguments or prefer to leave it out, for compactness or convenience (and adding them or leaving them out may be decided case by case).

In Curry-style typing, it's a problem to figure out what the type of an argument actually is (if any). The process of figuring that out is commonly known as **type inference**; some people prefer the word **type reconstruction** for it (and also type synthesis is used here and there). Some other interpretations of the terminology of "type inference" also exists, but they are not so common. In case that explictly giving the types is *optional* for the user, the problem is called **partial** type inference (or partial type reconstruction).

The *base* types won't play a prominent role in the development, the calculus simply picks some common ones, where Booleans are needed for the conditionals. There is one restriction for the *constants* here, namely they are typed by one of the *base types*. That means there are no functional constants. More precisely: the operators can be seen as built-in functional constants, but their arguments are base types, which means, the operators are not *higher-order*. That may very well be different in real languages, and is done here just to simplify the representation here.

As for *type environments* Γ from equation (2.8). They play the role of "contexts" (also in the technical sense of dealing with context-sensitive analyses). It is assumed that they work as *finite mappings*, basically as a representation of the **symbol table**. Three fundamental things one can do with such a "table" is: creating an empty one, adding a new binding ("enter"), and looking up an entry ("lookup"). The first two operations are part of the definition from above, for the lookup of x in Γ we write $\Gamma(x)$. We may also write $\Gamma[x \mapsto T]$, and use $dom(\Gamma)$. Γ in general acts like a stack.

Judgments and derivation system The type system will be formulated as **derivation system**. That's the standard representation for type systems. It will consist of a number of rules that allow to derive well-typedness for a given expression. More precisely, the derivation system operations with so-called type **judgements**:

$$\Gamma \vdash_{\mathsf{UL}} e : T \tag{2.9}$$

The intended meaning is: given the type information from the context Γ, expression e is of type T. Now to the derivation rules. They are shown in Table 2.3.

$$\Gamma \vdash c : \tau_c \quad \textsc{Con} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e_1 : T^1_{\mathsf{op}} \qquad \Gamma \vdash e_2 : T^2_{\mathsf{op}}}{\Gamma \vdash e_1\ \mathtt{op}\ e_2 : T_{\mathsf{op}}}\ \textsc{Op}$$

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash\ \mathtt{fn}_\pi x \Rightarrow e : T_1 \to T_2}\ \textsc{Fn} \qquad \frac{\Gamma, x{:}T_1, f{:}T_1 \to T_2 \vdash e : T_2}{\Gamma \vdash\ \mathtt{fun}_\pi f\ x \Rightarrow e : T_1 \to T_2}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{bool} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash\ \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T}\ \textsc{If}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash e_2 : T_2}{\Gamma \vdash\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : T_2}\ \textsc{Let}$$

Table 2.3: Curry-style

The rules are meant the same way as other such rules we have seen in the lecture (for instance when formulating an SOS semantics). They have premises and a conclusion; Rules without premises are also called axioms.

Type system vs. type checking algorithm The rules should be straightforward to understand. There is one rule, however that deserve a closer look. It has also to do with what perspective we take on the rules. One is the pure **logical** one. A rule sais: if the premises are true, then so is the conclusion, or rather: if the premises are derivable, then so is the conclusion: the conclusion's derivation (tree) consists of the derivation (trees) of the premises. Unlike standard presentations of trees, the current root of the tree (the judgement that's being derived) is "depicted" on the bottom, of course. If that's the mental interpretation of the rules, (which is a valid one), everything should be clear and we could stop the discussion here. It correspond to seeing the rules as a **type system**.

But one could shift the perspective, and try to see it as a *type checking* algorithm. One would say, the rule are interpreted in a **goal-directed** manner. In this way one see at a judgment $\Gamma \vdash e : T$ in the way described, but think of it as consisting of *input* and *output* of (ideally) an algorithm. In this interpretation, the rules are seen as describing a **recursive procedure**, with the context Γ and the expression as input, and the output a type of the expression (which implies the expression is well-typed):

$$\underbrace{\Gamma \vdash e}_{\uparrow \text{input}} : \underbrace{T}_{\downarrow \text{output}}$$

The different rules cover different cases of abstract syntax. Seen as a recursive procedure, the body of the type checking procedre would consist of a big case-construct, matching against the different syntactic forms. Most of the cases lead to **recursive** calls. Cases that correspond to axioms from the derivation system are **base cases** in the recursion.

Most cases work straightforwardly. In case of conditionals (cf. rule FUN), for instance , checking involves three recursive calls, one checking well-typedness of e_0 resp. that it's well-typed and of boolean type, and doing the same for e_1 and e_2, and making sure that they have the *same* type (T in the rule), which then is given back as type of the whole construct.

One word on **ill-typedness.** The rules describe when an expression is well-typed, resp. the type checker covers only the cases with a positive outcome. For conditionals, it states that e_0 has to be a boolean. But it does **not** describe what happens if it's not a boolean (or if the types of e_1 and e_2 are not the same). That's in the general spirit of derivation rules: the specify the positive outcomes, here, something is well-typed if you are able to use the rules to derive it, if there *exists* a derivation. It's understood that if there is no derivation, the expression is *ill-typed*. If one would implement a recusive type checker ignoring the negative outcome, it would probably mean, the type checker would raise an exception ("uncovered case branch"), or crash. Of course in a real implementation, one would cover the negative case, and maybe raise tailor-made and helpful error-messages ("condition of if-construct at line ... expected to be bool, but is int instead"). That's typically suppressed in presentations like this, keeping them **concise**.

But now, do the rules from Table 2.3, interpreted in a goal-directed manner **describe an algorithm?**

The answer is: Almost, where it not for one situation that **breaks it**, that's the treatment of function. So it's two rule that don't work: Fn and Fun. But they suffer from the same problem, so we focus on Fn. The rule tries to type-check $\Gamma \vdash \texttt{fn}\, x \Rightarrow e$. That involves a recursive call responsible to type-check the body e. That recusive call is done with an **extended context** $\Gamma, x{:}T_1$. And there's the problem: Where does T_1 come from? When seeing the rules as standard derivation rules (in a non-goal-directed manner), the problem is ignored. Some derivations work, for instance some choice for T_1 works, some doesn't, but it does not matter, the rule system is a **specification.** If a derivation exists, the expression is well-typed (and if not then not), and the specification is clear enough.

When interested in an type checking algorithm, the Fn rule is not useful. The "algorithm" cannot start **guessing**. First try a choice for T_1, to see if it works, if not, try another one. That's completely impractical. There are infinitely many types. So in case a program is ill-typed, the "algorithm" may continue trying new choice in the hope that the next one might work out, meaning the type checker will **not terminate** with the information that the program is ill-typed. That's quite undesirable. And it's horribly inefficient anyway, since there might not just be one Fun-situation, but multiple, which leads to a combinatorial explosion.

Monomorphism and polymorphism The fact that the "algorithm" has to do some guessing has another consequence. The explanation might have given the impression that, for well-typed programs, there is one correct guess, and (may) incorrect ones. But that's not true. There are programs, for which there are multiple correct guessing. Another better way of saying it is that there are programs that have not just one types, but multiple. The easiest and classical example for that is the identity function:

$$\texttt{fn}\, x \Rightarrow x \tag{2.10}$$

It has infinitely many types: $\mathsf{int} \to \mathsf{int}$, $\mathsf{bool} \to \mathsf{bool}$, $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ .... So it's of type $T \to T$, **for all concrete types** T!

A type system where every well-typed program has one type (in other words where every program has at most one type) is called **monomorphic.** The oppositive where a program can have more than one type is called **polymorphic.** So technically, our Curry-style formulation is polymorphic. However, it's polymorphic in a very crippled and useless way, maybe one should not even call it polymorphic. If we take the identity function from equation (2.10), sure, the type system is able to derive infinitely many types, reflecting the fact that the function gives back its input independent of it types, the code works for all types. This is true, but not really reflected in the type, because the type system must assign **one concrete type** (of many possible alternatives) to that function, maybe $\mathsf{int} \to \mathsf{int}$. And that's a **guess**, so when using the function on some argument, this type may be ok, but if not one would derive a nother type (but that's a general problem of the current formulation). Being forced to assign a concrete type means also that the function can be used in a program only for one type of input. If we'd give the name *id* to the

function, as in $\texttt{let}\, id = \texttt{fn}\, x \Rightarrow x\, \texttt{in} \ldots$. Then id can be use with exactly one choice for $T \to T$, which prevents the identity function to be used on integers at one place and on booleans at another place. The code from equation (2.10) would be perfectly fine to operate on various inputs, but the type system disallows it. So the identity function is not really polymorphic in a meaningful way

Different froms of polymorphism and monomorphic resp. polymorphic let. We mentioned that technically the type system in its Curry-formulation could be called polymorphic, but not in a meaningful or useful sense. Problematic is that type type system cannot represent polymorphism in its types. When writing $T \to T$, T is not a type, but the non-terminal form from the grammar equation (2.7). Related to that is the problem of **guessing:** the type system has to commit itself when checking to one particular type.

The standard way to remedy this is to have the type system reflect polymorphism. So far the type system has **no** notation to reflect polymorphic functions (like the identity function). The type system allows to derive infinitely many different types (which can be seen indirectly as sign that it's polymorphic). But there is no type that says "the function from equation (2.10) is the polymorphic identity". The standard way to do that is to incorporate **type variables** in a type system. Let's use α and similar to represent them, and with this the identity function would have the type $\alpha \to \alpha$.

We will encounter that later, when talking about **type inference**, where the treatment of type variables is crucial. However, we restrict ourselves there and treat the let-construct *monomorphically*. That refers to a very central contribution in type inference, namely the so-called "polymorphic let". So we do something slightly simpler here. Type variables are characteristic for one form of polymorphism

There are various forms of polymorphism, according to a classical classification by Cardelli and Wegner [1], there are 4 main variants. Arguably the two most important or most interesting are *parametric polymorphism* and *inclusion polymorphism* (both together called in [1] also as *universal polymorphism*). Inclusion polymorphism is also known as *subtype polymorphism* (as made popular by object-oriented languages). The two remaining ones are called *coercion* and *overloading* (both being *ad-hoc* polymoprhism).

Let-polymoprhism is of the *parametric* kind, and we might encounter it later. But as said: not right now, the calculus restricts itself to *monomorphic* lets. From a practical point of view, supporting only monomorphic lets is a terrible restriction, it invalidates basically all advantages of polymorphic functions. In other words, there is not much of a point to have polymorphic functions but not polymorphic lets. It's like supporting polymorphic functions, but without being able to naming them by a let-construct while preserving its polymorphic nature. As a consequence, one could make only monomorphic *re-use* of polymorphically defined functions. I.e., if one wants to use a function two or more times, it must be in situations involving the same type. For using a function one time only, the distinction between monomorphism and polymorphism becomes meaningless, there is also no *re-use* of the function (and one might not even bother to introduce a name for the function, for a singleton used, one could simply use the function anonymously). Of course, there still remain some use for monomorphic lets for polymorphic functions: if part of a library, one could define polymorphic functions and offer them at the interface with let-style bindings. That restricts them for monomorphic *use*. So each client code base

can use the function only monomorphically, but different code bases can make different decisions.

For fairness sake: there *are* uses of monomorphic lets. Especially within the Haskell community, a case has been made [5] that monomorphic lets have their place, and actually since version 7, GHC treats *local* let definitions monomorphically. Local means, in particular, the restriction does not apply to functions in library-interfaces. The motivation is efficiency of type inference. See for instance here or the discussion here in the context of *ocaml*.

At any rate, our type inference later will cover monomorphic lets. The key issues can be well understood under this restriction, though the system deviates in the discussed aspect from what is known as the Hindley-Milner-Damas type inference.

Subtype or inclusion polymorphism will probably *not* be covered in this lecture, at least not as far as the underlying type system is concerned. On the other hand: when dealing with the "non-standard" part of the type system (the annotations, the effects etc.) then there will be *"inclusion polymorphism"*. That is related to the "lattice treatment" for instance when dealing with data flow information as in the monotone frameworks (except that it was not formulated making use of type-theoretic notions).

Typed λ-calculis vs. untyped The syntax of our calculus is some variation of "the λ-calculus". There is, however, not exactly one version of that calculus. To the very least, there is the untyped λ-calculus, the purist one, a very minimalistic system originally developed for providing a foundation for math and logics etc., not so much for computer science. Typed variants came slightly later.

Anyway, in that form, it is *really* purist, there are only three constructs, *variables*, *abstractions* (i.e. functions), and *application* of functions. Unlike in our version, there are no such things as basic constants (like integers, booleans etc), there are no conditionals, and there is no syntax for recursive functions. That (untyped) λ-calculus is known to be a fundamental computational mechanism, "equivalent" to Turing machines (and other computational mechanisms). Those are equivalent insofar that Turing machines can encoded "programs"in the λ-calculus and vice versa. There are other such formalism; since there are all equivalent in the mentioned sense that each can encode each other, it's taken as granted that "this is what can be computed mechanically" (Church's thesis), and *computable* functions are just defined as those being encoded by such a formalism, which one is, does not matter (from a fundamental point of view).

Being computationally complete also means, one can encode everything representable on a computer. That includes numbers, booleans, conditionals, etc., in particular the constructs that we included in the syntax of our not-too-purist calculus. In particular, one can *encode* recursion and *non-terminating* functions or programs in the classical λ-calculus.

So, why do we then use a syntax supporting basic data types, conditional, and recursion, when those can be encoded? There are two reasons for that.

One is: one can encode things like numbers, conditionals, recursion etc. into the purist calculus (and there are even more than way one can do that). However, the **structure** of the construct is no longer visible and available for the analysis. Somehow, the specific

nature has been compiled or encoded away. For instance, one can encode booleans in the λ-calculus. Actually, it's not so hard to do encodings like that systematically. It's not pensum, but perhaps interesting. For natural numbers, such an encoding is known as **Church-numerals**. The analogous encoding for boolean —one could call them Church-Booleans— simply looks as follows: the two functions

$$\lambda x.\lambda y.x \quad \text{and} \quad \lambda x.\lambda y.y \tag{2.11}$$

represent the two boolean values true and false. Actually, to be more precise, when used in a disciplined manner, they behave as the values true and false. Equivalently they may also be used (by following a different disciplined usage) as false and true, i.e., the other way around. And finally they are just some functions, namely something that corresponds to the first projection (throwing away y) and the second projection.

But any type system cannot distinguish the Boolean-like nature of that values, it's just functions. Actually, it's characteristic for the purist, untyped λ)-calculus, that it *only* consists of function: literally every (closed) expression is interpreted as a function (and a value at the same). I say closed expression, because a variable in itself is, well, a variable, so one cannot say, a variable "is" or is interpreted as a function in itself. Being untyped means, an expression (being a function) can be applied to *any* other expression. So, being untyped means, any syntactally correct (closed) expression is a legal program. It's compariable with the trivial type system for the while-language. In that context we say, the type system is *trivial* since the type system accepts *all* syntactically correct programs, so the type system has not purpose at all (on top of the parser, so to say). A language with a trivial type system can also be said to be *untyped*, of course.

Since in that untyped calculus *anything goes* and there us no use for a type system, one can also not check whether a boolean value true resp. false is *actually* used as intended in the mind of the programmer cannot be captured by a type system. This information simply does not exist at that level of abstraction. In the same way, one cannot check that at the standard machine instruction level. At the lowest level of standard memory, there is only a contiguous stream of 0's and 1's and at that level alone, it makes no sense to ask whether it's an integer, a command, a boolean. If we want to analyse statically programs dealing with booleans etc., we need to do it at a level of abstraction, where these things exist structurally, i.e., syntactically. The same for conditonals and recursion.

That's one argument why our calculus supports those constructs and data structure. The second one has to do with the question: how does a type system *restrict* the calculus. That it indeed restrict the allowed programs is obvious, otherwise, as said, it would be a trivial type system and that's not different from being untyped. Static type system always classify the syntactically correct programs into the well-typed ones and the ill-typed ones. However, *how* restrictive a type discipline is, well, it depends. At this point, we want to discuss what is know as *simply-typed* λ calculus. It's like having available as types the ones mentioned in equation (2.7): at any rate, functional types, plus some basic types, here integers and booleans. Note: the calculus supports *higher-order* functions, but still it is called simply-typed.

One important question is **imposing a type system** on the untyped syntax (Curry-style) or having a type system operating with a syntax that includes types already (Church-style): how **restictive** is imposing a simply-typed discipline?

The answer is: it's a **radical** restriction. The untyped calculus is Turing complete, the typed ones are not! In particular, all well-typed program terminate. So, it's not really a good model for a programming language. Actually, also things we mentioned before, that Booleans and natural number etc. can be encoded, that's no longer true with the typed version. One can still "program" some functions like from the Church booleans from equation (2.11) and same for the Church numerals. Those, one cannot use those functions so that they behave as one wants them to, like one can do arithmetic on the numbers. The type system forbids that (at least the simple one we use).

Is that caused by the fact that we use the *simple* type discpline of the simply typed λ-calculus? Well, it depends. We can go for a more expressive type system, adding type variable, adding polymorphism etc. If we do that in the standard way, we would still *not* be able to get non-terminating functions. The type discipline *would still enforce termination.* That would apply to the system with type variables later, that would apply to what is known as *polymorphic* (or second-order) λ-calculus; System F is another name. And it would apply to even more expressive system. So recursion would still *not* be encodable. That why it's a good idea to simply add it to the syntax. Corresponding systems are called "simply-typed λ-calculus with recursion".

For the boolean encoding and the Church-numerals, actually going to something like the second-order calculus would allow some form of encoding in the way we showed and being still well-typed. But we are *not* going all the way to system F, we will be dealing with a weaker system. It's kind of like in between the STL ("first-order") and System F ("second order"), namely right where type inference is still decidable!

As a side remark: the word "first-order" resp. "second-order" don't refer to the functions, also the STL supports higher-order functions.

2.2.2.1 Church style

At the end, let's also have a short look at the Church-formulation. The corresponding rules from Table 2.4 very much look like the Curry-ones from Table 2.3, and Table 2.4 shows only the ones with relevant differences.

The syntax for abstractions and for let now specifies which type the relevant variable(s) should have. As a consequence, the rules are now **syntax-directed,** i.e., they directly correspond to a type-checking **algorithm**. Related to that, there is no problem of **type inference** as the programmer indicates the types, when introducing variables.

2.3 Type inference

Type inference is one core problem or technique when doing static analysis using types, i.e., a core problem for type checkers in general. Terminolgy-wise, one may distinguish type checking versus type inference. As hinted at, the word type **inference** is perhaps

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash e_2 : T_2}{\Gamma \vdash \ \mathtt{let}\, x{:}T_1 = e_1\, \mathtt{in}\, e_2 : T_2}\ \textsc{Let}$$

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \ \mathtt{fn}_\pi x{:}T_1 \Rightarrow e : T_1 \to T_2}\ \textsc{Fn} \qquad \frac{\Gamma, x{:}T_1, f{:}T_1 \to T_2 \vdash e : T_2}{\Gamma \vdash \ \mathtt{fun}_\pi(f{:}T_1 \to T_2)\ x{:}T_1 \Rightarrow e : T_1 \to T_2}\ \textsc{Fun}$$

Table 2.4: Church-style typing

a bit infortunate or misleading. The type system are often specified in the form of an inference system or a *deduction* system. We have seen instances in the two versions of a typed λ-calculus, and also earlier. An inference system is a bunch of rules, specifying how to "derive" or "infer" (in a step-by-step manner). In our context, deriviving whether a program is well-typed or not, resp. which type(s) it should have, if well-typed. In both cases of what is conventionally called type **checking** as well as what is called type **inference**, the system is given in the form of an *inference system*, which makes the name "type inference" a bit misleading. We mean by type inference or type reconstruction that we start with a *Curry-style* formulation: the user does not give the type when introducing a variable, hence the corresponding type must be *reconstructed* or *inferred*. But also in the Church-formulation, the problem is given by an inference system, using inference rules. Type reconstruction and corresponding classical techniques for that are the focus in this section.

Anyway, type checking vs. reconstruction is anyway a gray area. In the Church formulation, the type for variable declarations is given by the user, so it needs not to be reconstructed. Though an argument can be made: while the variable declarations are already equipped with a type, usages are not, and neither are other constructs, like functions applications. In those cases, the type needs to be "reconstructed" as well. Only of each and every substructure of a program is already annotated by a type (from the user), only then one should speak of type *checking*, checking whether all the type annotations are done correct by the user.

In the current set-up (standard types, not too complex calculus), no language would require that the user annotated *all* constructs. Also that's not needed, since annotative all variables basically does the job. Especially, if variables have to be introduced *first* before they are used, as is the case in calculus. That means, if the user introduces a variable first, then the corresponding type, given by the user can be remembers. In our formalization in the type environment Γ. Concretely that may be a symbol table. When

2.3.1 Type inference problem

Next we tackle what is called *type inference*. The terminology of type inference is standard (it goes back to Milner etc.), but some prefer to call it type reconstruction, as we discussed.

We discussed the challenge of type inference a bit already. The Curry-rules, when interpreted in a goal-directed manner, are not syntax-directed and thus are not an algorithm.

Trying to use them in a goal-directed manner involves guessing (or foresight), and guessing wrong would lead to backtracking, which is not a feasible approach.

What is required is required: mechanism leave decisions open: When the type for a variable is not (yet) clear, the system needs to mark it as such, and only later, when the variable is used, there needs to be a mechanism to **refine** the information about possible types.

It is useful here to revisit the previous rules of the Curry-style type system and reflect on *why* those rules are not directly an algorithm. The basic culprit is the rule for abstraction FN. In the formulation here, where we have also a rule for *recursive* function abstraction, also that rule is not "algorithmic" (rule FUN).

To get the right mental picture, one needs to be clear about, in which way the rules at intended to describe an algorithm, at least ultimately. Rules can be interpreted as a "logical" derivation process: given all the premises, the conclusion can be derived. That's ok, but when it comes to the question how to establish a judgment, one better switches perspective: namely: to establish the conclusion, one has to establish the premises first. Nothing really changes, of course, but thinking in this *goal-directed* way ("what do I need to do to establish a goal") may lead to a recursive procedure. Concretely in our setting and our rule, though, we need to be explicit about what *is* actually the goal? Of course, we have judgements of the form $\Gamma \vdash e : T$. With that interpretation, the goal to establish is a boolan question ("yes/no"). Is it the case or not that e has type T (given a context). For type inference, though, the question is $\Gamma \vdash e : ?$, which means: "is e well-typed, and if so, infer me the type". It's the *latter* way that we think about the judgements.

With this interpretation and concentrating here on FN: the culprit is the fact that the premise of the rule has to **guess** type T_1. What is the status of T_1? Well, T_1 is a symbol we use to represent types (a non-terminal in the grammar for types). It's not a type itself. Another terminology is that it's a **meta-variable** representing types. That T is a meta-variable for types means, we use that symbol to "speak about" the type system, but it's of course not itself a type. Especially it's not a variable in or of the type system.

The notion of "meta-variable" sounds esotheric, but actually it's not; think of how one would do an implementation. Let's try a recursive check procedure, assuming that the type rules where an implementation in the sense of a (non-deterministic) recursive procecure. The conclusion would be a call to a function, say `tcheck` (for do-a-type-check), with input a context, an expression, and a type) and a boolean return. That's a simplifying assumption, as we typically want to have the context Γ and expression e as input, and the type as output, but let's use the simpler "boolean" yes/no problem for illustration. That would mean, the type-check procedure could look like (in some unspecified programming language):

```
tcheck(gamma : Context, exp : Exp, type : Type) =
  if exp = app (exp1 exp2 ) ....
```

Now, `type` is a variable in the unspecified programming language, the formal parameter of the procedure, and is of type `Type`, which, one the one hand, is some concrete type in that said programming language and, on the other hand, is used to implement the types of the language whose type system we intend to implement. At any rate, `type` is a variable,

but *not* of the language we are implementing, but of the language we use to implement it. Thus it's a *meta*-variable, a variable of the language use to implement the target language (or the language used to "speak about" the target language).

One *core* "enabler" to make type inference work is to *internalize* the notion of "variable" from the meta-level to the level of the languaage we are dealing with. That means that the concept of types needs to be extended to include **type variables**. That will be called (here) *augmented types*.

Augmented types fancy name for: "we have added type variables"

$$\begin{array}{lcll} T & \in & \textbf{AType} & \text{augmented types} \\ \alpha & \in & \textbf{TVar} & \text{type variables} \end{array} \tag{2.12}$$

$$\begin{array}{lcl} T & ::= & \mathsf{int} \mid \mathsf{bool} \mid T \to T \mid \alpha \\ \alpha & ::= & \mathsf{'a} \mid \mathsf{'b} \mid \ldots \end{array} \tag{2.13}$$

With type variables added, what has changed? The allowed types, obviously, have changed, now that we have added type variable in equation (2.13). Now there are "more" types available, at least syntactically. On the other hand, the set of ground types has not changed, it's still integers, booleans, and functions, not more. We discuss the consequences of adding variables mostly for the Curry-formulation, because that is what we need to do the classical type inference. But we will shortly look at "Church" as well, to look around a bit broader.

On general, a type containing a variable represent many types, actually infinitely many types. One may call them *instances* of that types, i.e., special cases in the sense that choices are made for type variables. When concrete choices are made for all type variables insfar that the resulting instance no longer contains variables, it's called *ground* instance of that type. In the same line, a type without type variables is called *ground* type (and more generally, a *term* without (free) variables may be called *ground* term).

What does a type with variables stand for? It represents all its instances. That means a program with a type containing type variables has many types, infinitely many actually, and hence, that's a case of *polymorphism!*

Curry formulation Let's start by observing that here, the set of programs is unchanged by the augmentation, since the abstract syntax does *not mention types* (unlike for the Church-style formulation). If we take another look at the type system from Table 2.3. Note: the type system there *does not change at all* (apart from the fact, that the meta-variables T, T_1 etc. mentioned in the rules now refer to augmented types).

What changes then, it the type system still has the same rules (only "more types", but still the same ground types)? As just mentioned, the presence of type variables in the type system represents *polymorphism*. However, as discussed in connection with the rules of Table 2.3 *without* type variables, the type system already captured a polymorphic type

discipline! For instance, the function $\texttt{fn}\, x \Rightarrow x$ as canonical illustration, has infinitely many types, like

$$\mathsf{int} \to \mathsf{int}, \mathsf{bool} \to \mathsf{bool}, (\mathsf{int} \to \mathsf{bool}) \to (\mathsf{int} \to \mathsf{bool}), \ldots \tag{2.14}$$

Each one is derivable by the previous system: the crucial rule for that example is T-Fn, which mentions the **meta**-variable T_1, which can be chosen as an *arbitrary ground type* in the previous setting.

Now that we have type variables, we can choose arbitrary types, including non-ground types. For instance, we could give that identity function $\alpha \to \alpha$ (or $'\mathtt{a} \to '\mathtt{a}$).

Why is that better? The system *still* allows to derive infinitely many types, in particular, all the ground types from equation (2.14) are still derivable. However, now that we have variables, **also** $\alpha \to \alpha$ is derivable, and that non-ground type represents all its instances, including all its ground instances. So, the original system was polymorphic in the technical sense, adding type variables makes the type system "aware" of that fact.

So are we done then? Not really, the problem of type inference has not been solved. The type system is non-deterministic. Though the type $\alpha \to \alpha$ can be derived now, the other types are derivable as well. So, the *system* is not an algorithm (being non-deterministic). The goal will be to turn it into one. In this particular case of the identity function, the goal is to get a formulation that derives $\alpha \to \alpha$ a being the **"best"** type. It's the best in the sense that it captures all other potential types as special cases, i.e., as instances.

That all fit together in the end depends on the fact that such a most general type 1) always exists (for typable programs) and 2) that one can find it, and find it halfway efficiently.

Both cases brings us back to the notion of *lattice*...

2.3.2 Substitution, unification, and unification constraints

As informally mentioned earlier, one key to obtaining an algorithm is to leave decisions open (with the help of type variables) when introducing a variable, and later, when using the variable in a specific way, refining or specializing the information. For instance, when a variable x (with a current type of α, i.e. with no restrictions at all) is used in that it's applied to some e, then it's clear that x is more specifically of functional type, say $\alpha_1 \to \alpha_2$, and if at that point it's known that e is of, say, type int, we even more specifically know that x has to be of type $\mathsf{int} \to \alpha_2$.

That's what we mean we need a mechanism to refine or specialize initially general types to more specific one; in the example from the complete undetermined α to the more specific $\mathsf{int} \to \alpha_2$. Technically, the process of specialization is based on **substitutions** and **unification**.

2.3.2.1 Substitutions

Substitutions and unification are *general* concepts and techniques, with applications far beyond type inference. Substitution very generally is "replacement", and in the (standard) form as presented here, it's a purely syntactic operation, replacing variables (which is "syntax") by other "syntax". In the literature, when dealing with (the simplest form of) unification, the syntax is often called *terms* and one speaks of *term unification*. In our type-inference settings the syntax (or terms) are the augmented types, and the variables are the type variables. Apart from that, the setting will be plain vanilla term unification, the most basic form of unification.

What's then a substitution? It replaces variables by terms/types.

Definition 2.3.1 (Substitution)**.** A *substitution* is a finite mapping from type variables to augmented types, i.e.

$$\theta : \mathbf{TVar} \rightarrow_{fin} \mathbf{AType}$$

We consider only finite substitutions, i.e., finite functions. We write $dom(\theta)$. for the domain of a substitution. In general, a substitution can map to types that contain type variables. A substitution that maps to *ordinary* types only (= without variables) is called a **ground substitution**.

According to Definition 2.3.1, a substitution maps variables to types.[1] Typically, one wants to apply substitutions not just on type variables, but on types. What is mean is to use the substitution to replace all variables inside the type accordingly. So we use a substitution θ (with the same symbol) also as a function of type

$$\theta : \mathbf{AType} \rightarrow_{fin} \mathbf{AType}$$

One says, the underlying variable substitution is *lifted* to types/terms. By *lifting*, one simply means; if one knows the effect of a substitution on variables, then it's straightforward to use the substitution *also* as a mapping from terms to terms (here types to types), simply by replacing all the variables inside a term one by one. It's a simple recursive algorithm. Substitutions can be composed (and we write $\theta_1 \circ \theta_2$ for that, or just $\theta_2\theta_1$).

Above we defined substitutions as finite functions, i.e., function with a finite domain of variables. Sometimes, θ's are also considered as *total functions* (over "all" variables), setting $\theta(\alpha) = \alpha$ when $\alpha \notin dom(\theta)$.

2.3.2.2 Algorithm: basic idea and unification

We have discussed some basic ingredient of the algorithmic solution already, we need to avoid guessing, when introducing variables in the Curry-formulation (for abstractions and for let), and we do that with type variables. The analysis will have to specialize the initially

[1] From now on I just say types when I mean types which may contain variables. When particulary talking about types without type variables, I will use **ground types**.

unconstrainted type for a variable depending on how it it used. The specialization is done using substitutions.

That sounds plausible, but one question remains unanswered: which substitution to use for specialization? In general, if a type contains type variables, there are infinitely many ways how to specialize the type. If one does not have a strategy how to specialize, we are not much further than before, because guessing the type may lead to an over-specialization, and thus backtracking.

But we don't specialize types arbitrarily, we only specialize them when we **use** a variable. And we we use a variable, we specialize it as careful as possible, i.e., choose a substitution that specializes the type not more as required by the current situation. A program variable can be used multiple times, and in different situations. So after specializing the type accordingly to the first usage, the second use of the variable may call for a further specialization. So during the run of the inference algorithm the type of some program variable may gradually be specialized. It may also happen, that a use impose contradictory constraints on the type: in the first situation, the program variable must be of functional types (being applied to something), and in the second sitution later, the program variable needs to be an integers (maybe being used in some addition). That's of course **incompatible**, and the type inference will fail.

replace:

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : T_1 \to T_2}\ \textsc{Fn}$$

---

by

$$\frac{\Gamma, x{:}\alpha \vdash e : T_2}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e : \alpha \to T_2}\ \textsc{Fn}$$

- $x{:}\alpha$ when α is fresh (otherwise unused) means: type of x is completely arbitrary.
- syntax-directed now?
- T_1: meta-variable for types
- α: (still meta variable for) type variables

α's completely arbitrary?

Consider body

$$e = x\ g$$

for $\mathtt{fn}_\pi x \Rightarrow e$

$\Rightarrow$

- a function type: $\alpha = \beta \to \gamma$
- fit together with type of $g \Rightarrow$ condition or constraint on β

- judments "give back" not just the type, but also "restrictions" on type variables.
- represented as constraint[2]
- $\Rightarrow$

$$\Gamma \vdash e : (T, C)$$

Under the assumptions Γ (which might "assign" to (program) variables: type variables), program e possesses type T (again potentially containing type variables) *and* imposes the restrictions "embodied" by C on the type variables.

2.3.2.3 Constraint generation algorithm

The presentention here *deviates* from that in [2] (and from some previous years). It's partly a presentational issue, as the rules become quite more readable. And the presentation is more flexible resp. general in the sense that it can be used also for other problems formulated as type system, for instance some type-and-effect systems (though for classical type inference, the additional flexibility is not needed).

Anyway the two presentations amount to the same type inference algorithm. The difference is the following: generating constraints in the typing rules, as done here, is kind of a **2-phase approach**. In the first phase, in the syntax-directed rules, one generates constraints, and afterwards, in a second phase, one tries to solve them (in our setting, by unification, see below). The alternative, which one often finds in the literature (for instance the original papers about type inference) is a **one-phase-approach**: one *eagerly* not just generates constraints while traversing the abstract syntax tree in the typing rule, one *solves them eagerly at the same time* (here by unification).

It should be noted that the "solve-the-constraint-while-you-go" works fine for the situation here, based on unification. For more involved settings, though, it's not just a matter of presentation to do a 2-phase-approach and postpone the solution of the constraints. In more complex situations, the "on-the-fly" solution may no longer work. By more complex situations I mean not necesserily type inference for more complex languages, but type and effect systems.

The intuitive reason for that is as follows: The typing rules traverse the syntax tree in a particular manner, collecting information bottom-up. As it turns out, for unification and for collecting information about "the most general type", that works fine. If the information being collected is more complex, this "bottom-up" or "on-the-fly" treatment of *solving* the constraints may not work anymore (collecting them bottom-up is just fine). An example would be **data-flow analysis** (take reaching definitions as examples) phrased as type-and-effect system. Solving them involves (as we know from the corresponding chapter) a fixpoint algorithm, due to loops in the program or recursion. Type systems, however, typically don't go "up-*and*-down" a syntax tree or in cycles,[3] it's often a "one-sweep" process. That "sweep" could be used to collect thoss flow constraints, and in a second phase, they could be solved (which could involve fixpoint iteration).

[2] In the book, what is given back is a substitution instead.
[3] There are, however, the notion of *local* or *bidirectional* type inference, things we don't look at.

2.3.2.4 Constraints

Before generating constraints for type inference, it may be worthwhile to reflect a bit on what constraints are. In principle we should have an impression as we have dealt with constraints extensively in connection with **data flow** analysis.

Constraints and constraint solving is a **very broad field** in itself, i.e., very many different forms of constraints have been studied, with particular constraint solving algorithms. Very generally speaking,

> a constraint is nothing else than a formula with (free) variables.

When we mean by "formula" could be an equation, but also other involving inequations or other relations. In the context of data flow analysis we were also dealing with subset relations $\subseteq$. Also a constraint may involve logical connectives, in particular $\wedge$. Indeed, typically one has multiple constraints, a **constraint set** or constraint system. In that situation, the constraints are meant as a solution must satisfy all constraints of the set. I.e., a set of constrain can equivalently seen as one single constraint, being a *conjunction* of the individual constraintion. Indeed, when dealing with constraints, even if there are no other logical connectives involved, there is at least implicitly **conjunction**, if nothing else beyond atomic constraints.

For completeness sake: also the (non-)constraint "true" typically is seen as allowed constraints, so a minimal "constraint logic" supports "true" and $\wedge$. Note, "false" or other connective like $\vee$ (or $\neg \ldots$) are not necessarily part of a constraint set-up, though they may: according to out very general definition a constraint is just some formula(s) with free variable, where we don't fix any allowed or disallowed syntax. Though (at least implicitly) $\wedge$ and "true" are a must-have. Obvously, the more complex the formulas, the more complex the problem of constraint solving becomes.

The allowed set of connectives and other symbols fix the syntax of the constraints. But that's not all, one needs also to agree upon, over which **domain** the chosen constraints are **interpreted.** Like from which set of values to choose from for the free variables to solve the constraints. Additionally, how to interpret the relations and other symbols. For the monotone frameworks, we were dealing with constraints using the symbol $\sqsubseteq$ in the general description, and that was interpreted as $\subseteq$ resp. $\supseteq$ depending on whether we were doing a may resp. a must analysis. Often the interpretation of symbols is understood, so when writing $=$, we mean equality, or when writing $+$, it's addition, but generally, it makes sense to clear about the separation of *syntax* (how to write don't constraints or other things syntacitically) and *semantics* (what they are intended to mean).

With syntax and semantics fixed, it's also fixed what a **solution** of a constraint set is. It's a choice of values for all the variables that make all the constraints true. This choice **solves** the constraints, or one can also say it **satisfies** the constraints. If such a solution exist, one calls the constraints **satisfiable.**

Depending on the forms of constraints and the solution domain, different solution algorithms (or semi-algorithms) exists. Maybe the most well-known and elementary satisfiability problem is known as *SAT*, solving Boolean constraints.

Mostly, one is not literally interested in satisfiability, in the sense of a yes/no answer of whether a solution exists. One is interested in a concrete solution, (or maybe many or all, if possible), i.e., choices of values for the constraint variables. For instance, due to their shape and their solution domain (lattice), data flow constraints are always satisfiable. So plain satisfiability is a non-issue, one wants **the best solution** that satisfies the constraint, for instance the smallest reachable definition sets for each program point. In some sense (in that setting), the best, here smallest, solution implicitly actually **represents all solutions** as well: all variable assignments that are $\supseteq$ compared to the smallest solutions are solutions as well. That's due to the particular structure of the domain (being a lattice). That allows that one does not need to calculate all solutions, the smallest one (for may analysis) contains all the information we need. For other constraint settings, that may not be the case. For instance, for `SAT`, having one solution does not help (much) to have other solutions as well. In the worst case, the only approach is to combinatorially check all combinations of boolean assignments, which leads to an exponential effort for satisfiability checking. This makes *SAT* probably the most famous NP-complete problem, [4] a problem for which no efficient solution exists (under current consensus). It does not mean that SAT-solving does not profit from solution strategies (it does), and implementations will exploit one ore more heuristics and strategies, vastly improving performance. It's only that it's provable that in the worst-case, the complexity will fall into the "NP-complete" class, which means, as said, that it's not efficiently solvable (according to mathematical consensus, though it's not actually proven that it's categorically impossible).

Unification constraints Type inference, the current topic, relies on **unification**, resp. **unification constraints**. Since we base our presentation on a two-phase approach — generating constraints first, solving them afterwards— the constraints are clearly visible: the type system **generates** a constraint set, a set of unification constraints.

More precisely, one should talk about **term unification** and **term unification constraints.** That's more precise, insofar there exist also the (more complex) problem of unification on "other things" than plain terms. Terms basically means **syntax** (often of a simple form, for instance without binders or quantifiers). Characteristically for syntax resp. terms is the particularly simply notion of **equality.** When are two terms **equal?** The answer is trivial: when they are **syntactically equal**. i.e. when they are written the same way.

Generally, one is used to more complex equalities. For instance, one would expect the following equality $1 + x = x + 1$, as one interprets it as an equality over numbers, and one interprets the syntax $+$ as addition (and 1 as the number pronounced as "one"). That this equation is expected to hold depends on this standard **interpretation** of the equation constraint and how the syntax is interpreted. With the expected interpretation of $+$ as addition etc., one would expect $x + y$ being equal to $y + x$ (commutativity). One could generalize the unification problem relative to some such "laws" (maybe equational laws such as commutativity for some pieces of syntax). When going down that road, two terms that are written differently would still be considered equal. Consequently one would no longer consider them (as far as the constraint problem is concerned) as terms, but as

[4] And the first one in the sense that the NP-problem and related one was introduced first in connection with *SAT*).

interpreted terms or terms modulo some "laws". Thus, one would be facing no longer vanilla **term unification** but something more complex. But we don't need that for type inference.

For type inference, the unification will work on types containing type variables. So we are still using term unification, where the types play the role of terms.

- notation $T_1 =^? T_2$ - many other forms of "constraints" systems exists with specialized solving techniques - here: term *unification*

2.3.2.5 Constraint generation

$$\frac{}{\Gamma \vdash c : (T_c, \emptyset)}\ \text{T-Const} \qquad \frac{}{\Gamma \vdash x : (\Gamma(x), \emptyset)}\ \text{T-Var}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x{:}\alpha \vdash e_0 : (T_0, C_0)}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e_0 : (\alpha \to T_0, C_0)}\ \text{T-Fn}$$

$$\frac{\alpha, \alpha_0 \text{ fresh} \quad \Gamma, f{:}\alpha \to \alpha_0, x{:}\alpha \vdash e_0 : (T_0, C_0) \quad C_1 = \{T_0 =^? \alpha\}}{\Gamma \vdash \mathtt{fun}_\pi f\ x \Rightarrow e_0 : (\alpha \to T_0, C_0, C_1)}\ \text{T-Fun}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (T_1, C_1) \quad \Gamma \vdash e_2 : (T_2, C_2) \quad \alpha \text{ fresh} \\ C_3 = \{T_1 =^? (T_2 \to \alpha)\}\end{array}}{\Gamma \vdash e_1\ e_2 : (\alpha, C_1, C_2, C_3)}\ \text{T-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : (T_0, C_0) \qquad \Gamma \vdash e_1 : (T_1, C_1) \qquad \Gamma \vdash e_2 : (T_2, C_2) \\ C_4 = T_0 =^? \mathsf{bool} \qquad C_5 = T_1 =^? T_2\end{array}}{\Gamma \vdash \mathtt{if}\, e_0\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : (T_2, C_1, C_2, C_3, C_4, C_5)}\ \text{If}$$

$$\frac{\Gamma \vdash e_1 : (T_1, C_1) \qquad \Gamma, x{:}T_1 \vdash e_2 : (T_2, C_2)}{\Gamma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : (T_2, C_1, C_1)}\ \text{Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (T_1, C_1) \qquad \Gamma \vdash e_2 : (T_2, C_2) \\ C = \{T_1 =^? T^1_{\mathsf{op}}, T_2 =^? T^2_{\mathsf{op}}\}\end{array}}{\Gamma \vdash e_1\, \mathtt{op}\, e_2 : (T_{\mathsf{op}}, C_1, C_2, C)}\ \text{Op}$$

We are a bit sloppy here with the notation (but the rules are intended to be precise nonetheless). The sloppyness refers to the fact that constraints C are seen as sets of elements of the form $T_1 =^? T_2$, but we do not always write it as set $\{T_1 =^? T_2\}$. Also when we put two constrain sets together (by building set-union), we simply write C_1, C_2, as opposed to $C_1 \cup C_2$. Instead of seeing it as sets of basic unification constraints, could also see it as a *conjunction* (for instance using $\wedge$), but all of that is just notation. The intention is: a set of constraint is statisfied by a solution, if *all* the constraints inside the constraint set are satisfied, and that makes it a conjuction of basic constraints ("and").

2.3.3 Unification

In this section we cover something thoroughly classical, namely "unification". However, we should be careful, what we are doing is the most basic, classical notion of unification, also called *term unification*. So when we in the following talk about "unification" that's what we mean. Besides that, there are many variations and extensions of this very basic and fundamental concept (semi-unification, unification-modulo-this-and-that, higher-order unification ...). Most of them are outside the scope of the lecture (though not necessarily for type-based analysis or static analysis). However later, we will touch upon unification *modulo* some equivalences, that will be in the context of augmented type systems (for control flow analysis or other problems).

But what is characteristic for term unification? Well, it's unification for terms. Unification means generally "to make equal", namely here by **substitution**. Substitution means replacing variables, here replacing variables in terms by terms (which in turn may contain variables). That leads to a notion of equivalence: two terms are **unifiable**, if they can be made equal by unification, i.e., by finding a substitution that, when applied to the two terms, makes them equal.

Variables in a term stand for arbitrary terms, so a term containing variables represents infinitely many terms, for the different choices for the variable. For precision's sake, infinitely many only if the pool of available terms is infinite, which is basically always the case except for degenerated languages. Note that substitutions replace variables by terms which can contain variables themselves. Terms without variables are called *ground terms* and substitutions that replace variables by ground terms are called *ground substitutions*.

Replacing variables in a term can be interpreted as /specializing/ or /refining/ the term, making it more concrete. Instead of leaving some places in the term completely open, replacable by all possible terms, a substitution restricts the represented terms. Applying a ground substitutions, replacing all variable, obviously fixed the term completely.

2.3.3.1 Terms

Early we started with asking what is characteristic for *term* unification, and we focused on discussing unification. So still the question remains: what is characteristic for *terms?* Terms are a *syntactic* concept. They may be *interpreted*, i.e., one may attach a meaning or semantics to them, but that's something extra. Terms as such stand for themselves, they are nothing else than the syntax, and that also means, two terms are equal when they are (syntactically) equal. That's the strictest form of equality, everything is identical to itself, but unequal to anything else, (and that's what was meant, when saying terms stand just for themselves). At least that's the case focusing on ground term, but see later the discussion about the role of variables.

There may be variations on the notion of terms. For instance one may work with "typed" version, which are called *multi-sorted* terms. Or single-sorted, i.e., where there is only one sort. The latter is what we need. A specific choice of a (single-sorted) term language is

defined by a alphabet or set of symbols (function symbols, term formers, term constructors), each with a fixed arity. The arity is how many argument a symbol takes. Symbols with arity zero are called constant symbol.

One can construct new terms from old terms by applying a function symbol to terms, respecting the arity. That make terms ordered (finite) *trees* (with nodes marked by the symbols from the alphabet and in accordance with the agreed arity of the symbols). So: terms are trees. By way of terminology: an alphabet together with fixing arities for the symbol is called (single-sorted) term *signature*, but it's not crucial for us.

Actually, there is a different way of fixing a language of terms (or trees), and that's the good old context-free grammars (for us conventionally written in BNF).

We are interested in **types**, which are considered as terms for the purpose of term unification. They are given by the BNF from equation (2.12) (there called "augmented"). Also the abstract syntax of the calculus is given as grammar, in equation (2.1). There, the non-terminal e is even called terms (or expressions). However, there are aspects of that grammar which makes it slightly problematic to see it *only* as "terms" in the sense described, namely the use of bound variables. Typically, with bound variables, syntax is considered equal *up-to* consistent renaming of bound variables (known as α-equivalence). That means, it is often wished to treat such expression with a weaker notion of equality than the one that is characteristic for terms, namely that terms are equal only to themselves. Still, expression in the λ-calculus are often called terms. Even though our version of augmented types (see equation (2.12) has no *bound* variables, only free ones. But also there, the issue of "does the actual choice of variable names matter or rather not?" shows up (and the better answer is, type are better considered up to renaming also for free variables). In general, the notion of strict identity for terms applies only for ground terms, adding variables complicate matters.

So sum up, we are dealing with types (containing type variables). Terms which may contain variables (but not bound ones) and with identity as the only equivalence on *ground* terms is what is covered by *term unification*. The fact that there are free variables (whose exact identity does not matter), is part of that setting, and the free variables are more or less uproblematic. One has just to keep that in mind sometimes, but it's pretty natural, and we will remark on that fact in the context of the unification algorithm.

Non-terms and non-term unification? Unification is making things equal and term unification is doing that for terms, which are specific by they strict interpretation of equality: two ground terms are equal if they are identical. More general forms of unification still make things equal, though with a looser interpretation of what it means to be equal.

As ground terms, the expressions $5 + 3$ and $5 + 3$ are different terms (where $+$ is a binary function symbol written in infix). However, which this choice of symbols, one plausibly has a specific interpretation mind and with that interpretation comes non-trivial equalities. In the example, one presumable considers both terms equal (and equal also to a term 8 and many others as well). Considers are as terms, one does not interpret the two expression (or interprets them "by themselved"). One sometimes also speaks of *uninterpreted functions* (i.e., not particular meaning or interpretation attached).

Anyway, moving away from the syntactic, uninterpreted notion of terms so more loose notions of equality complicates matters. In particular, unification becomse more tricky. Actually, one may easily end up in a setting where unification becomes *undecidable*.

For the simple form of type inference on the underlying type system here, it is not relevant, there one does term unification. But later, we will encouter additional information on the types (augmented types, perhaps with flow information, or augmented by effects). At that point, we are no longer in the plain term-based setting.

Sometimes one can still do unification modulo the relevant congruence, but we will not do that. Instead, we move to a slightly different formulation using "simple types" and which will involved constraints. In the classicaly type inference, we presented a constraint based formulation as alternative to an eager presentation, where unification was done at the point when needed. We said, it's partly a matter of taste, but the more complex the analysis becomes, the move usefule becomes that freedom from separating the collecting-the-constraints phase from the solving-the-constraints phase.

Unifier Before talking about unification, we need to talk about **unifiers**. It's based on the notion of substitutions (cf. Definition 2.3.1). A unifier of two terms is nothing else than a substitution that makes two terms equal (i.e., syntactically equal).

Definition 2.3.2 (Unifier)**.** A **unifier** of two types (or terms in general) T_1 and T_2 is a substitution is a substitution θ such that

$$\theta(T_1) = \theta(T_2) \ .$$

Two types T_1 and T_2 are called **unifiable** if there *exists* a unifier for T_1 and T_2.

The definition can be used analogously for multiple types/terms. Note that some pairs of types are unifiable and some are not. However, if two types are unifiable, there exists typically not just one unifier, but many. In fact, infinitely many for our syntax of types.

If there exist many ways how to unify two unifiable type, the obvious question is, which one is the "right" one, i.e., which of the many possible unifiers should one use? The good news is, there is a natural choice: choose the substitution that makes enough choices in substituting variable to unify the two type, but don't make the choices more specific than necessary. For instance, a substitution $\theta = [\alpha \mapsto \mathsf{int} \to \alpha_1]$ is a choice for α and $\theta' = [\alpha \mapsto \mathsf{int} \to (\mathsf{bool} \to \alpha_2)]$ is a different one. And it's easy to see that if θ is a unifier for two types, then so is θ', it's only that it's more specific in that it specializes α_1 from θ to the more specific $\mathsf{bool} \to \alpha_2$ for θ_2. In that situation θ_1 is preferable (as more specific). This has to do with the way sets of unification constraints are solved, namely gradually. First one faces a unification problem (for instance in a type inference situation), which can be solved by θ, but later there will be additional constraints (for instance analysing the program further, or solving a set of unification constraints). If one used θ' instead of θ, when both are possible, it may turn out that the addition choice of $[\alpha_1 \mapsto \mathsf{bool} \to \alpha_2]$ will contradict other constraints later, being too specific, and thus one would be forced to **backtrack**, trying the more general θ instead. Of course it would have been better not to over-specialize and try θ from the beginning instread. Of course there is no guarantee

that θ won't not also run into troubles. However, if one can guarantee that θ is **the most general** unifier, it makes exactly those choices to unify two type but not more, then it's the right choice: one continues to solve the constraints and if it works in the end, it's fine, if not, then it's unsovable, because θ was the most general at that point anyway, and if it does not lead to a solution, no more specific one will solve the system either. So no need for back-tracking (otherwise unification would be unpractical).

That may sound plausible, but if course it relies on something that we have not established, namely that **exist a most general unifier**. It's imaginable that there exist two unifiers of two types, each of which cannot be generalized, but none of them more general that the other, so both incomparable. There do exist incomparable unifiers, for instance when having $T_1 = \alpha_1$ and $T_2 = \alpha_2$ (to take a trivial example), the unifiers

$$\theta_1 = [\alpha_1, \alpha_2 \mapsto (\alpha_3 \to \mathsf{bool})] \quad \text{and} \quad \theta_2 = [\alpha_1, \alpha_2 \mapsto (\mathsf{int} \to \alpha_4)]$$

are incomparable, neither θ_1 is more general than θ_2 nor the other way around. However, there exists in this situation

$$\theta = [\alpha_1, \alpha_2 \mapsto \alpha]$$

which is more general than both θ_1 and θ_2. The good news is that it can be proven that for two unifiable types (or terms), there exists a most general unifier, and it's unique[5] That's why one also speaks of "the" most general unifier (which is the one the unification algorithm gives back, see later).

As a side remark: the set of unifiers of two types form a kind of lattice. The (partial) order relation on unifiers (or substitution) is "being more general" and the most general unifier corresponds to a *least upper bound* of all possible unifiers.[6]

For concreteness sake, we should spell out what it means for a substitution or unifier to be more specific or more general than another.

Definition 2.3.3 (Ordering of substitutions)**.** A substitution θ_1 is more specific (or less general) than θ_2, written $\theta_1 \lesssim \theta_2$ if $\theta_1 = \theta\theta_2$ (for some θ)

In other words, θ_1 is more specific than θ_2 because one can do additional choices represented by θ such that the effect of θ_1 can be achieved by applying the more general substitution θ_2 first and afterways do the remaining specialization via θ.

This relation is almost a partial order, but not quite. It's straightforward to see that $\lesssim$ is reflexive and transitive. But how about **anti-symmetry?** Unfortunately it does not hold! Take as trivial example two unifiers for $T_1 = \alpha_1$ and $T_2 = \alpha_2$. There are unifiers $\theta_1 = [\alpha_1 \to \alpha_2]$, $\theta_2 = [\alpha_2 \to \alpha_1]$, or $\theta_3 = [\alpha_1 \to \alpha, \alpha_2 \to \alpha]$. None of which is intuitively more specific than the others, none makes any concrete choices for syntax. They just juse different variables to express the fact that α_1 and α_2 have to be the same but otherwise unconstrained. Technically, it's $\theta_1 \lesssim \theta_2$ and $\theta_2 \lesssim \theta_1$, and the same for comparisons involving θ_3. All of them are equally generally, and they just differ in their choice of variables.

[5] Some qualifying remarks on the uniqueness a bit later.

[6] Since "most specific unifiers" are uninteresting and don't exist anyway, it's not a lattice in the sense we introduced earlier, but only "half" of a lattice, or semi-lattice. But that's good enough for our purposes.

And that's the condition up-to which one should interpret unifiers: the choice of variable names should not matter. Thus the most general unifier exists **uniquely up to the choice of variables**. Since after a while that's understood and goes without saying, one simply says "there exists a unique MGU" and one speaks about "the" MGU of two types.

What kind of (order) relation are we dealing with? That the binary relation $\lesssim$ is reflexive and transitive. That straightforwardly from the fact that the identity function is a substitution, and that the composition of two substitutions is again a substitution. Those two properties, reflexivity and transitivity, makes $\lesssim$ to a /pre-order/.

What about /anti-symmetry/? If that would hold as well, $\lesssim$ would be a /partial order/. Anti-symmetry would mean that the following implication were true

$$\text{if} \quad T_1 \lesssim T_2 \quad \text{and} \quad T_2 \lesssim T_1 \quad \text{then} \quad T_1 = T_2 \ .$$

Does that hold? The immediate reaction is: no. The simplest example is trying to unify two variables, α_1 and α_2. Then $\theta_1 = [\alpha_1 \mapsto \alpha_2]$ and $\theta_2 = [\alpha_2 \mapsto \alpha_1]$ are /both/ unifiers; actually there are "others" as well, namely those of the form $\theta' = [\alpha_1 \mapsto \alpha', \alpha_2 \mapsto \alpha']$. Assuming that α_1 and α_2 are different variables, all those substitutions are the same, in a way. They take two separate variables α_1 and α_2 and replace them by a common one (they unify them by a common choice for a variable). But which variable we choose for that is immaterial. So we are back to the discussion, which role variables play for terms (and here now for substitutions). Respectively, it's a question of when are two terms (and here now substitutions) "equal". And a good answer is: *equality up-to renaming* is the adequate notion. If we take equality up-to variable renmaing as equality, then $\lesssim$ /is indeed/ a partial ordered.

Remark 2.3.4 (Some fine-print on the notion of renaming)**.** *As said, for types and substitutions, the choice of particular variables is immaterial, that's what is meant by "up-to renaming". It's probably obvious (at least on reflection), that does* ***not mean****, one can choose variables* ***randomly****. For example, if one is given the term (= type here) $\alpha_1 \to \alpha_2$, then one cannot rename α_1 to α_2. In other words, the two types $\alpha_1 \to \alpha_2$ and $\alpha_2 \to \alpha_2$ are obviously not equivalent. It's also not renaming, neither is $\alpha_1 \to \alpha_2$ a renamed version of $\alpha_2 \to \alpha_2$, nor the other way around.*

However, $\alpha_2 \to \alpha_2 \lesssim \alpha_1 \to \alpha_2$: there is a /substitution/ that /specializes/ the second type into the first, but the other way around it does not work. Renamings actually indeed /specific/ substitutions. Renaming is, unlike instantiation, i.e., $\lesssim$, something that one can reverse or invert. So for a renaming in one direction, there is an inverse renaming. After all "up-to renaming" is an equivalence relation, which implies symmetric. On can define T_1 is equal up-to renaming with T_2 as $T_1 \lesssim T_2$ and $T_2 \gtrsim T_2$ (one could write $T_1 \gtrless T_2$). It involves two substitutions which are inverses of each other. And that implies that the substitutions are /injective/. In the above illustration, identifying α_1 with α_2 by $[\alpha_1 \mapsto \alpha_2, \alpha_2 \mapsto \alpha_2]$ is non-injective, i.e., non-invertible and thus a substitution which is not a renaming.

Unification constraints and unification We know what a unifiers of two (or more) types are, and **unification** is the problem of finding them if they exist, resp. finding the most general unifier, as that's the "useful" one, if it exists. That can be understood as a constraint problem. Notationally we write

$$T_1 =^? T_2 \tag{2.15}$$

for the problem "find me a unifier for T_1 and T_2". Unification constraints are constraints on types, where the solution domain are types (resp. terms) and the equality relation $=^?$ is "syntactic equality after substitution". The solution of unification constraints are unifiers, and we are interested in the most general one, which we know to exist.

The constraint solving algorithm for that kind of constraints (called "unification") is a very "classical" methods (and actually very simple) and dates back to the mid-60ies Robinson [4]. It has many applications, for instance it's an "enabler" for all forms of theorem proving, also Prolog etc.

In general we don't deal with just one unification constraint with two types as in equation (2.15), but many of them, as *constraint set* (or a big conjuction of many individual constraints. We write C of such a constraint system. Notationally we write

$$\theta \models C$$

for θ satisfiers or solves the constraints C, in other words θ is a unifier for C (unifying all the individual unification constraints in C at the same time.

After all this clarifications and background information, now to the actual algorithm, that computes the most general unfifier (here just for pairs). After all this built-up, it's maybe a bit of a let-down, as it's really simple and straightforward:

Unification algorithm for types:

$$\begin{array}{rcl}
\mathcal{U}(\mathsf{int},\mathsf{int})) & = & id \\
\mathcal{U}(\mathsf{bool},\mathsf{bool})) & = & id \\
\mathcal{U}(T_1 \to T_2, T_1' \to T_2') & = & \texttt{let} \quad \theta_1 = \mathcal{U}(T_1, T_1') \\
 & & \qquad\; \theta_2 = \mathcal{U}(\theta_1 T_2, \theta_1 T_2') \\
 & & \texttt{in} \quad \theta_2 \circ \theta_1 \\
\mathcal{U}(T,\alpha) & = & \begin{cases} [\alpha \mapsto T] & \text{if } \alpha \text{ does not occur in } T \\ & \text{or if } \alpha = T \\ \text{fail} & \text{else} \end{cases} \\
\mathcal{U}(\alpha,T) & = & \text{symmetrically} \\
\mathcal{U}(T_1,T_2) & = & \text{fail} \qquad \text{in all other cases}
\end{array}$$

An interesting case is the one involving variables. First things first, for that case. In the cases with one argument being a type variable α, and the second one some type T, it's perfectly fine if T is a type variable α'. So the cases are not interpreted like T is α or a non-variable term. Maybe that is clear, but just to make sure, I point that out.

Now the interesting part, namely the condition that α does not occur in T. That is known as ***occur check**. This check is needed for unification, like here for type inference. It's also needed in theorem proving, and logic programming, in languages like Prolog. However, for efficency reason, Prolog implementations may decide to omit that check.

2.3.3.2 1-phase type inference algorithm

We presented type inference as a two-phase approach: the type rules generate a set of (term unification) constraints, which then are to be solved in a subsequent phase. That unification phase may fail, in case of contradictory constraints.

In many presentations, the solving of the constraints is interleaved with the typing rules. We show a corresponding formulation next, but apart from the "eagerness" when doing unification, it's analogous. Often, the presentation is also not written by using rules, by a function, like $\mathcal{W}'$, as shown on the slide. As discussed, the algo would not the famous algorithm $\mathcal{W}$, since we left out polymorphic lets.

- formulated here as /rule system/ - immediate correspondence to a /recursive/ function:

$$\mathcal{W}'(\Gamma, e) = (T, \theta)$$

instead of

$$\Gamma \vdash e : (T, \theta)$$

- not 2-phase, giving back a set of unification constraints C

$$\frac{}{\Gamma \vdash c : (T_c, id)}\ \text{T-Const} \qquad \frac{}{\Gamma \vdash x : (\Gamma(x), id)}\ \text{T-Var}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x{:}\alpha \vdash e_0 : (T_0, \theta_0)}{\Gamma \vdash \ \mathtt{fn}_\pi x \Rightarrow e_0 : (\theta_0\alpha \to T_0, \theta_0)}\ \text{T-Fn}$$

$$\frac{\alpha, \alpha_0 \text{ fresh} \quad \Gamma, f{:}\alpha \to \alpha_0, x{:}\alpha \vdash e_0 : (T_0, \theta_0) \quad \theta_1 = \mathcal{U}(T_0, \theta_0\alpha_0)}{\Gamma \vdash \mathtt{fun}_\pi f\ x \Rightarrow e_0 : (\theta_1\theta_0\alpha \to \theta_1(T_0), \theta_1 \circ \theta_0)}\ \text{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : (T_1, \theta_1) \quad \theta_1\Gamma \vdash e_2 : (T_2, \theta_2) \quad \alpha \text{ fresh} \quad \theta_3 = \mathcal{U}(\theta_2 T_1, T_2 \to \alpha)}{\Gamma \vdash e_1\ e_2 : (\theta_3\alpha, \theta_3\theta_2\theta_1)}\ \text{T-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : (T_0, \theta_0) \qquad \theta_0\Gamma \vdash e_1 : (T_1, \theta_1) \qquad \theta_1\theta_0\Gamma \vdash e_2 : (T_2, \theta_2) \\ \theta_3 = \mathcal{U}(\theta_2\theta_0 T_0, \mathsf{bool}) \qquad \theta_4 = \mathcal{U}(\theta_3 T_2, \theta_3\theta_2 T_1)\end{array}}{\Gamma \vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : (\theta_4\theta_3 T_2, \theta_4\theta_3\theta_2\theta_1\theta_0)}\ \text{If}$$

$$\frac{\Gamma \vdash e_1 : (T_1, \theta_1) \qquad \theta_1\Gamma, x{:}T_1 \vdash e_2 : (T_2, \theta_2)}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : (T_2, \theta_2\theta_1)}\ \text{Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (T_1, \theta_1) \qquad \theta_2\Gamma \vdash e_2 : (T_2, \theta_2) \\ \theta_3 = \mathcal{U}(\theta_2 T_1, T^1_{\mathtt{op}}) \qquad \theta_3 = \mathcal{U}(\theta_3 T_2, T^2_{\mathtt{op}})\end{array}}{\Gamma \vdash e_1\ \mathtt{op}\ e_2 : (T_{\mathtt{op}}, \theta_4\theta_3\theta_2\theta_1)}\ \text{Op}$$

2.3.3.3 Concluding remarks: "Classic" type inference

In the presentation here, we did **not** look at the **full** well-known **Hindley-Damas-Milner** type inference algorithm. what is missing here is what is known as **polymorphic let**. Instead our treatment could be called monomoprhic let, which is "almost useless" in practice. However, we wanted to focus on unification and constraints. With polymorphic let, the algo would need some extra tricks, not hard, but distracting for the unification problem.

To get at least a feeling of what is missing. In our setting we could define define *id* as the identify function as follows:

$$\texttt{let}\ id = \texttt{fun}\ x \to x\ \ \texttt{in} \ldots$$

id could get the most general type $\alpha \to \alpha$. So the type is **polymorphic** in a way, as it mentions a type variable. That sounds good, until one realizes that it's not really useful. Having defined *id* like that, we can use it on integers, on booleans or on any other type as input (and the same type as output). However, **we cannot use it on integers at one place and later on booleans at another**. So that would introduce contradictory unification constraints. So the whole thing is not really useful.

Now the missing ingredient is that the type system does not only introduce type variables, but also **universal quantification**. The identity function then would carry the type

$$\forall\alpha.\alpha \to \alpha$$

not just $\alpha \to \alpha$. In concrete implementations that use this kind of type system, the type would still be written $\alpha \to \alpha$ (or `'a -> 'a`) when typing *id* in into an interactive environment or REPL, with the understanding that all type variables are **implicitly prenex quantified** (quantified in front of the type). That's a restriction over the free use of such quantifiers, as for instance a type like

$$(\forall\alpha.\alpha \to \mathsf{int}) \to (\forall\alpha'.T) \tag{2.16}$$

would **not** be allowed (not being prenex-quantified, but having also quantifiers "in the middle"). That's a restriction on the language. The type of equation (2.16) describes a function that takes a polymorphic function as argument and return a polymorphic function. That cannot be captured by the types that rely on *prenex* quantification. It means, the classical Hindley-Milner-Damas type inference system support polymorphic functions (like *id* but they cannot capture polymorphic functions as argument or as returned from function. One could say, the language supports higher-order function, but the type system restricts polymorphism in that there is no higher-order polymorphic functions. When handed over as argument to another function *id* would have to be used monomorphically in the body of the other function. In the same way that on our even more restricted presentation, *id* is not even usable in a polymorphic way (as we did not introduce $\forall$ at all).

Why do languages like ML, Haskell etc impose that restriction? It's for practical reasons: Allowing "higher-order polymorphic functions" makes **type inference undecidable**, and the compromise formulated with prenex quantification and polymorphic let could be seen as "sweet spot" for type inference: still algorithmically doable, and giving practically useful form of polymorphism

Bibliography

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

[2] Nielson, F., Nielson, H.-R., and Hankin, C. L. (1999). *Principles of Program Analysis*. Springer Verlag.

[3] POPL'88 (1988). *Fifteenth Symposium on Principles of Programming Languages (POPL)*. ACM.

[4] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41.

[5] Vytiniotis, D., Peyton Jones, S., and Schrijvers, T. (2010). Let should not be generalised. In *Proceedings of TDLI'10*.

Index

λ-calculus, 4

asubstitution
    ground, 23
augmented type, 15

BNF, 24

Church-style type system, 5
context-free grammar, 24
Curry-style type system, 5

ground substitution, 17, 23
ground term, 23

monomorphism, 9, 10

polymorphic let, 9
polymorphism, 9, 10
pre-order, 27

recursion, 4
referential transparency, 3

signature, 24
substitution, 23
    ground, 17
syntactic sugar, 12

term, 23
type
    augmented, 15
type inference, 12
type system, 1
    Church-style, 5
    Curry-style, 5

unification, 22, 28