15-819: Resource Aware Programming Languages

Lectures 5 and 6: Type Inference

Jan Hoffmann

September 29, 2020

1 Introduction

Before we start to develop a type-based approach to resource analysis, we study type inference. Given an expression e our goal is to find a context Γ and a type τ such that $\Gamma \vdash e : \tau$, or report that e is not typeable if such a pair (Γ, τ) does not exist. Type inference serves as a blue print for resource bound inference, which corresponds to a type inference for a more complex type system. Moreover, type inference provides a good motivation for introducing *let polymorphism*, which will shed light on some design choices in type systems that we study later in the course.

2 A Monomorphic Type System

In this lecture, we use a similar language as in the previous lectures on cost semantics. The only difference is that we add let bindings, which will come in handy when we talk about let polymorphism.

$$\begin{array}{lcll}
e & ::= & x & x \\
 & & \mathrm{app}(e_1; e_2) & e_1(e_2) \\
 & & \mathrm{fun}\{\tau, \tau'\}(f, x.e) & \mathsf{fun}\ f\,x = e \\
 & & \mathrm{triv} & \langle\rangle \\
 & & \mathrm{let}(e_1; x.e_2) & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}$$

A type is either a type variable $(t, u, t_1, u_1, \ldots)$, an arrow type $\tau_1 \to \tau_2$, or the unit type $\mathbf{1}$.

$$\begin{array}{lcll}
\tau & ::= & t & t \\
 & & \mathrm{arr}(\tau_1; \tau_2) & \tau_1 \to \tau_2 \\
 & & \mathrm{unit} & \mathbf{1}
\end{array}$$

Figure 1 contains the type rules of the monomorphic type system. The rules define the judgement $\Gamma \vdash^m e : \tau$ stating that expression e has type τ in context Γ. As usual, a context Γ is a finite mapping from variables to types.

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau$$

The order in which variables appear in a context is irrelevant and a well-formed context contains each variable only once. Especially, if we write $\Gamma' = \Gamma, x : \tau$ then $\Gamma'(x) = \tau$.

A type variable can be viewed as an unspecified atomic type or a placeholder for one concrete type without type variables. Type variables do not enable any form of polymorphism. For example, in the rule M:APP for function application, the argument type τ' has to exactly match the type of e_2 and there is no instantiation of type variables. Therefore, a function type $t_1 \to t_1$ cannot be used with an argument of type $\mathbf{1}$ or $t_2 \to t_2$ but only with arguments of type t_1.

We skip the definition of the dynamic semantics, as it is not needed for the discussion of type inference. However, we could define a structural dynamic semantics and prove type soundness as discussed earlier in lecture.

$\boxed{\Gamma \vdash^m e : \tau \qquad \text{“expression } e \text{ has type } \tau \text{ in context } \Gamma\text{”}}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^m x : \tau}\ (\text{M:Var}) \qquad \frac{}{\Gamma \vdash^m \mathrm{triv} : \mathrm{unit}}\ (\text{M:Unit}) \qquad \frac{\Gamma, f : \tau \to \tau', x : \tau \vdash^m e : \tau}{\Gamma \vdash^m \mathrm{fun}\{\tau, \tau'\}(f, x.e) : \tau \to \tau'}\ (\text{M:Fun})$$

$$\frac{\Gamma \vdash^m e_1 : \tau' \to \tau \qquad \Gamma \vdash^m e_2 : \tau'}{\Gamma \vdash^m \mathrm{app}(e_1; e_2) : \tau}\ (\text{M:App}) \qquad \frac{\Gamma \vdash^m e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash^m e_2 : \tau}{\Gamma \vdash^m \mathrm{let}(e_1; x.e_2) : \tau}\ (\text{M:Let})$$

Figure 1: Monomorphic type rules.

3 Type Inference

When types get more involved, it can be elaborate to write down the types of expressions or to come up with the right types of a function. Therefore, functional programming languages like SML and OCaml have a mechanism for *type inference* or *type reconstruction*, which automatically fills in the type annotations to enable automatic type checking.

To define the type inference problem for our simple monomorphic language, we first need to deal with some formalities. Let $\mathcal{T}$ be the set of monomorphic types and let $\mathcal{X}$ be the set of type variables. In the monomorphic type system, we can view type variables as placeholders for other types. This view is justified by the following fact: If we replace a type variable $t \in \mathcal{X}$ with a type $\tau_0 \in \mathcal{T}$ in judgement $\Gamma \vdash^m e : \tau$ then we obtain a derivable judgement $\Gamma' \vdash^m e : \tau'$. To make this formal, we use *type substitutions*, (partial) finite functions $\sigma : \mathcal{X} \to \mathcal{T}$ from type variables to types. For a substitution σ, we inductively define $[\sigma] : \mathcal{T} \to \mathcal{T}$.

$$\begin{array}{lcl} [\sigma]t & = & \begin{cases} \tau & \text{if } \sigma(t) = \tau \\ t & \text{if } t \notin \mathrm{dom}(\sigma) \end{cases} \\ [\sigma]\mathrm{unit} & = & \mathrm{unit} \\ [\sigma]\mathrm{arr}(\tau_1; \tau_2) & = & \mathrm{arr}([\sigma]\tau_1; [\sigma]\tau_2) \end{array}$$

We sometimes just write $[\tau'/t]\tau$ for $[\sigma]\tau$ if $\mathrm{dom}(\sigma) = \{t\}$ and $\sigma(t) = \tau'$. We extent type substitutions to contexts $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ by applying the substitution pointwise to each type, that is, $[\sigma]\Gamma = x_1 : [\sigma]\tau_1, \ldots, x_n : [\sigma]\tau_n$. Similarly, we define type substitution for expressions.

$$\begin{array}{lcl} [\sigma]x & = & x \\ [\sigma]\mathrm{app}(e_1; e_2) & = & \mathrm{app}([\sigma]e_1; [\sigma]e_2) \\ [\sigma]\mathrm{fun}\{\tau, \tau'\}(f, x.e) & = & \mathrm{fun}\{[\sigma]\tau, [\sigma]\tau'\}(f, x.[\sigma]e) \\ [\sigma]\mathrm{triv} & = & \mathrm{triv} \\ [\sigma]\mathrm{let}(e_1; x.e_2) & = & \mathrm{let}([\sigma]e_1; x.[\sigma]e_2) \end{array}$$

Lemma 1. *Let* $\Gamma \vdash^m e : \tau$ *and let* σ *be a type substitution. Then* $[\sigma]\Gamma \vdash^m [\sigma]e : [\sigma]\tau$.

The type inference problem for our monomorphic language is defined as follows.

Given: An expression e

Question: Find a type τ and a type substitution σ such that $[\sigma]e : \tau$ if such a τ exists or report a type error otherwise.

So inferring a type includes to fill in the right types for function arguments and results. Note that the functions in the expression e are already annotated with types. However, a fresh type variable in the position of a type can be used to indicate a missing type. To infer types for an expression “without types”, we would simply annotate each function abstraction with unique type variables t_i and t'_i.

Example 1. *Recall the abbreviation for lambda abstraction. To infer a type for the expression $\lambda(x_1)\lambda(x_2)(x_1(x_2))(x_2(\langle\rangle))$, we first insert fresh type variables into the function abstractions to obtain*

$$\lambda(x_1 : t_1)\lambda(x_2 : t_2)(x_1(x_2))(x_2(\langle\rangle))$$

We then infer that $\sigma(t_2) = \mathbf{1} \to t_3$, $\sigma(t_1) = (\mathbf{1} \to t_3) \to t_3 \to t_4$, and

$$\tau = ((\mathbf{1} \to t_3) \to t_3 \to t_4) \to (\mathbf{1} \to t_3) \to t_4 \,.$$

We are also interested in finding the most general typing $[\sigma_0]e : \tau_0$ for an expression e in the following sense: For every substitution σ and type τ such that $[\sigma]e : \tau$ there exists a substitution σ' such that $\tau = [\sigma]\tau_0$ and $[\sigma]e = [\sigma']([\sigma_0]e)$. We will see that such a most general typing exists, which is not a priori clear.

4 Constraint-Based Typing

We will solve the type inference problem using constraint-based typing judgements and a unification algorithm. The constraint-based typing judgments can be used to derive a set of constraints from a expression and unification derives a most general typing (if such a typing exists) by solving the constraints.

Type Constraints First, we introduce type constraints. A type constraint is a pair

$$\langle \tau_1, \tau_2 \rangle$$

where τ_1 and τ_2 are monomorphic types.

Definition. *A type substitution σ* solves *the type constraint $\langle \tau_1, \tau_2 \rangle$ if $[\sigma](\tau_1)$ is syntactically equal to $[\sigma]\tau_2$. Let C be a set of type constraints. The substitution σ solve C if σ solves every type constraint in C. The substitution σ is then called a* solution *or* unifier *for C.*

We define $\mathcal{U}(C)$ to be the set of all unifiers.

We often write a constraint set as a list $\langle t_1, t_1' \rangle, \ldots, \langle t_n, t_n' \rangle$.

Example 2. *The constraint set $\langle u, t_1 \to t_2 \rangle, \langle t_1, t_3 \to t_4 \rangle, \langle t_2, \mathbf{1} \rangle$ has the solution*

$$\begin{array}{lcl} t_2 & \mapsto & \mathbf{1} \\ t_1 & \mapsto & t_3 \to t_4 \\ u & \mapsto & (t_3 \to t_4) \to \mathbf{1} \end{array}$$

The constraint set $\langle t, t_1 \to t_2 \rangle, \langle t_1, t \rangle$ has no solution.

When we define type substitutions, we have to be careful not to use type variables on the right-hand side that also appear on the left-hand side. Otherwise, the result of the substitution might not be what we expect.

Lemma 2. *A set C of type constraints has either 0, 1, or an infinite number of solutions.*

At first, Lemma 2 might look surprising. However, it is intuitively clear that either a solution exists or not. Furthermore, there is solution that maps one of the type variables to a type variable or not. If this is the case, then there is an infinite number of solutions and otherwise there is only one solution. You can prove the lemma first for one constraint $\langle \tau_1, \tau_2 \rangle$ by nested induction on τ_1 and the by induction on the number of constraints.

$\boxed{\Gamma \vdash^c e : \tau \mid C \qquad \text{Expression } e \text{ has type } T \text{ in context } \Gamma \text{ under constraints } C.}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^c x : \tau \mid \cdot}\ (\text{C:Var}) \qquad \frac{}{\Gamma \vdash^c \text{triv} : \text{unit} \mid \cdot}\ (\text{C:Unit})$$

$$\frac{\Gamma, x : \tau, f : \tau \to \tau' \vdash^c e : \tau_0 \mid C}{\Gamma \vdash^c \text{fun}\{\tau, \tau'\}(f, x.e) : \tau \to \tau' \mid \langle \tau_0, \tau' \rangle, C}\ (\text{C:Fun})$$

$$\frac{\Gamma \vdash^c e_1 : \tau_1 \mid C_1 \qquad \Gamma \vdash^c e_2 : \tau_2 \mid C_2 \qquad t \text{ fresh}}{\Gamma \vdash^c \text{app}(e_1; e_2) : t \mid \langle \tau_1, \tau_2 \to t \rangle, C_1, C_2}\ (\text{C:App})$$

$$\frac{\Gamma \vdash^c e_1 : \tau_1 \mid C_1 \qquad \Gamma, x{:}\tau_1 \vdash^c e_2 : \tau \mid C_2}{\Gamma \vdash^c \text{let}(e_1; x.e_2) : \tau \mid C_1, C_2}\ (\text{C:Let})$$

Figure 2: Constraint based typing rules.

Constraint-Based Typing Figure 2 contains the type rules for a constraint-based type judgement $\Gamma \vdash^c e : \tau \mid C$. It states that under context Γ, expression e has type τ under the constraints C. The intuitive meaning is that we can derive a monomorphic typing $\Gamma' \vdash^m e : \tau'$ if we substitute the variables in Γ and τ, so that the constraints in C are satisfied. This intuition is formalized in Theorem 1.

In the type rules, it might seem arbitrary at what points we introduce new variables and type constraints. For example, in the rule C:Fun, we introduce the type constraint $\langle \tau_0, \tau \to \tau' \rangle$ instead of requiring $\Gamma, x : \tau, f : \tau \to \tau' \vdash^c e : \tau \to \tau'$ in the premise. The reason is that we take an algorithmic view of the type rules. We assume we have a procedure, that takes a context and an expression and returns a type and a constraint set. So the premises can be seen as recursive calls to this procedure and we cannot (directly) control the return value. We call such an algorithmic view the *mode* of a judgment.

Theorem 1 and Theorem 2 show that the constraint based typing rules are sound and complete with respect to the monomorphic type system.

Theorem 1 (Soundness)**.** *Let* $\Gamma \vdash^c e : \tau \mid C$. *If* σ *is a type substitution that solves* C *then we have* $[\sigma]\Gamma \vdash^m [\sigma]e : [\sigma]\tau$.

Proof. It is convenient to rename e (in the theorem) to $\bar{e}$ in the proof. We show the statement by induction on the type derivation of $\Gamma \vdash^c \bar{e} : \tau \mid C$. Let $\bar{\Gamma} = [\sigma]\Gamma$ and let $\bar{\tau} = [\sigma]\tau$.

C*ase* C:Var. *Then* $\bar{e} = x$, $C = \emptyset$, *and* $\Gamma(x) = \tau$. *Thus* $\bar{\Gamma}(x) = \bar{\tau}$ *and* $\bar{\Gamma} \vdash^m x : \bar{\tau}$ *by rule* M:Var.

C*ase* C:Fun. *Then* $\bar{e} = \text{fun}\{\tau, \tau'\}(f, x.e)$ *and* $\bar{\tau} = \tau_0$. *By induction we have* $\bar{\Gamma}, x : \bar{\tau}, f : \bar{\tau} \to \bar{\tau}' \vdash^m e' : \bar{\tau}_0$ *where* $e' = [\sigma]e$ *and* $\bar{\tau}_0 = [\sigma]\tau_0$. *Moreover,* $\langle \tau_0, \tau \to \tau' \rangle \in C$ *and thus* $\bar{\tau}_0 = \bar{\tau} \to \bar{\tau}'$ *where* $\bar{\tau} \to \bar{\tau}' = [\sigma](\tau \to \tau')$. *Consequently, we can apply the rule* M:Fun *to derive* $\bar{\Gamma} \vdash^m \text{fun}\{\bar{\tau}, \bar{\tau}'\}(f, x.e') : \bar{\tau} \to \bar{\tau}'$.

C*ase* C:App. *Then* $\bar{e} = \text{app}(e_1; e_2)$ *and* $\tau = t$ *for a fresh type variable* t. *By induction, we have* $\bar{\Gamma} \vdash^m e_1' : \bar{\tau}_1$ *and* $\bar{\Gamma} \vdash^m e_2' : \bar{\tau}_2$ *where* $e_i' = [\sigma]e_i$. *Furthermore,* $\langle \tau_1, \tau_2 \to t \rangle \in C$. *Thus* $\bar{\tau}_1 = \bar{\tau}_2 \to \bar{t}$ *and we can use* M:App *to derive* $\bar{\Gamma} \vdash^m \text{app}(e_1; e_2) : \bar{t}$.

The other cases are similar. □

Theorem 2 (Completeness)**.** *Let* e *be an expression. If there are* Γ *and* $\bar{\tau}$ *such that* $\Gamma \vdash^m e : \bar{\tau}$ *then* $\Gamma \vdash^c e : \tau \mid C$ *for some* τ *and there exists a type substitution* σ *that solves* C *such that* $[\sigma]\tau = \bar{\tau}$.

Proof. The proof proceeds by rule induction on the type derivation $\Gamma \vdash^m \bar{e} : \bar{\tau}$ (we rename e to $\bar{e}$.)

Case M:VAR. *We simply use* C:VAR *to derive* $\bar{\Gamma} \vdash^c x : \bar{\tau} \mid \cdot$ *and set* $\sigma = \emptyset$.

Case M:ABS. *Then* $\bar{e} = \mathrm{lam}\{\bar{\tau}_1\}(x.e)$, $\bar{\tau} = \bar{\tau}_1 \to \bar{\tau}_2$. *By induction we have* $\Gamma, x{:}\bar{\tau}_1 \vdash^c e : \tau_2 \mid C$ *and* σ *that solves* C *such that* $[\sigma]\tau_2 = \bar{\tau}_2$. *We use* C:ABS *to derive* $\Gamma \vdash^c \mathrm{lam}\{\bar{\tau}_1\}(x.e) : \bar{\tau}_1 \to \tau_2 \mid C$. *The type substitution* σ *is as required.*

Case M:APP. *Then* $\bar{e} = \mathrm{app}(e_1; e_2)$ *and by induction we have* $\Gamma \vdash^c e_1 : \tau_2 \to \tau \mid C_1$, $\Gamma \vdash^c e_2 : \tau_2 \mid C_2$, σ_1, *and* σ_2 *such that*

$$\begin{array}{l} \sigma_1 \textit{ solves } C_1 \\ \sigma_2 \textit{ solves } C_2 \\ [\sigma_1]\tau_1 = \bar{\tau}_2 \to \bar{\tau} \\ [\sigma_2](\tau_2) = \bar{\tau}_2 \end{array}$$

From the freshness requirement we derive that $t \notin \mathit{dom}(\sigma_1) \cup \mathit{dom}(\sigma_2)$. *We use the rule* C:APP *to derive* $\Gamma \vdash^c \mathrm{app}(e_1; e_2) : t \mid \langle \tau_1, \tau_2 \to t \rangle, C_1, C_2$ *We define* $\sigma = \sigma_1 \cup \sigma_2 \cup \{t \mapsto \bar{\tau}\}$.

The other cases are similar. □

The Inference Problem To solve the type-inference problem we will first generate a constraint typing $\Gamma \vdash^c e : \tau \mid C$. And then solve the constraints in a second step called unification. The generation of the constraints can be implemented efficiently.

Theorem 3. *Given a context* Γ *and an expression* e, *there is a linear-time algorithm (linear in* $|e|$*) that produces a valid judgement* $\Gamma \vdash^c e : \tau \mid C$. *Moreover,* $|C| \le |e|$.

Proof. The algorithm is given by the rules in Figure 2. The rules are syntax-directed, that is, there is exactly one rule for each syntactic form. Moreover, each rule adds at most one constraint to the constraint set. □

5 Unification

I the following, we study the second part of the type-inference problem: Given a constraint-based type judgement

$$\Gamma \vdash^c e : \tau \mid C$$

we want to compute a type substitution that solves the type constraints C. This problems is often called the *unification problem* and the prevailing technique for finding such a type substitution is called *unification.* We first study unification at a high declarative level and then consider efficient ways of implementing it.

First, we extend type substitutions (point-wise) to type constraints and sets of type constraints.

Definition. *Let* σ *be a type substitution and let* $\langle \tau_1, \tau_2 \rangle$ *be a type constraint. We define* $[\sigma]\langle \tau_1, \tau_2 \rangle = \langle [\sigma]\tau_1, [\sigma]\tau_2 \rangle$. *Similarly, if* C *is a set of constraints, we define* $[\sigma]C = \bigcup_{(\langle \tau_1, \tau_2 \rangle) \in C} [\sigma]\langle \tau_1, \tau_2 \rangle$.

Definition. *A type substitution* ρ *is a* most general unifier (MGU) *of a set of type constraints* C *if* $\rho \in \mathcal{U}(C)$ *and for every* $\sigma \in \mathcal{U}(C)$ *there exists a type substitution* σ' *such that* $\rho = [\sigma'] \circ \sigma$.

MGUs always exist are unique up to renaming of variables.

Theorem 4. *Every unifiable set* C *of type constraints has an MGU* σ. *If* ρ *and* ρ' *are MGUs of* C *then* $\rho' = [\sigma] \circ \rho$ *where* $\sigma : \mathcal{X} \to \mathcal{X}$.

We will revisit this theorem later and discuss the proof. First, we discuss the declarative unification algorithm. The strategy of the algorithm is to transform all constraints successively into the form $\langle t, \tau \rangle$ for a type variable t that does not appear anywhere else in the constraint set. The transformation ensures that the unifiers of the transformed constraint set are exactly the unifiers of the original constraint set. After the transformation, we can simply read off the MGU as described in Lemma 3.

$C \Longrightarrow C'$	Constraint set C unifies to constraints set C'
$C \Longrightarrow \bot$	Constraint set C is not unifiable

$$\frac{}{\langle c(\tau_1,\ldots,\tau_n), c(v_1,\ldots,v_n)\rangle, C \Longrightarrow \langle \tau_1, v_1\rangle,\ldots,\langle \tau_n, v_n\rangle, C}\ (\text{U:D}\textsc{ec})$$

$$\frac{c \neq d}{\langle c(\tau_1,\ldots,\tau_n), d(v_1,\ldots,v_m)\rangle, C \Longrightarrow \bot}\ (\text{U:M}\textsc{is}) \qquad \frac{\tau \notin \mathcal{X}}{\langle \tau, t\rangle, C \Longrightarrow \langle t, \tau\rangle, C}\ (\text{U:F}\textsc{lip})$$

$$\frac{t \in \mathrm{FV}(\tau) \qquad \tau \neq t}{\langle t, \tau\rangle, C \Longrightarrow \bot}\ (\text{U:C}\textsc{yc}) \qquad \frac{}{\langle t, t\rangle, C \Longrightarrow C}\ (\text{U:E}\textsc{lim}1)$$

$$\frac{\sigma = t \mapsto \tau \qquad t \notin \mathrm{FV}(\tau) \qquad t \in \mathrm{FV}(C)}{\langle t, \tau\rangle, C \Longrightarrow \langle t, \tau\rangle, [\sigma]C}\ (\text{U:E}\textsc{lim}2)$$

Figure 3: Declarative unification rules.

Definition. *An type constraint $\langle t,\tau\rangle$ is in* solved form *in a constraints set C if t is a type variable that does not occur anywhere else in C; in particular $t \notin FV(\tau)$. A set of constraints C is in solved form if every type constraint in C is in solved form in C.*

Lemma 3. *Let $C = \{\langle t_1, \tau_1, ,\rangle \ldots, \langle t_n, \tau_n, \}\rangle$ be a set of type constraints in solved form. Then $\sigma_C = \{t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n\}$ is an indempotent MGU of C, that is, $[\sigma] \circ \sigma = \sigma$.*

We write σ_C for σ in the previous lemma.

The rules in Figure 3 define the transformation steps of unification. They define the judgment $C \Longrightarrow C'$, which indicates that the constraint set C is transformed into the constraint set C' in one step. The rules also define the judgment $C \Longrightarrow \bot$, which states that the constraint set C is not unifiable. The rules are declarative in the sense that they can be applied in any order and to any matching constraint in the set until the constraint set is saturated, that is, no rules can be applied anymore. To implement unification, we have to pick a specific strategy for apply the rules.

In our simple language, we only have the two type formers unit and $\mathrm{arr}(\tau_1;\tau_2)$. However, we formulate the rules more generally for arbitrary type formers $c(\tau_1,\ldots,\tau_n)$ in the rules U:Dec and U:Mis. In our case, we have $c, d \in \{\mathrm{arr}, \mathrm{unit}\}$. In the rule U:Elim2, the premise $t \in \mathrm{FV}(C)$ ensures that the rule as an effect and we make progress when applying it.

Example 3. *Below is a sequence of steps that transform a constraint set into solved form.*

$$\begin{aligned}
& \{\langle u, t_1 \to t_2\rangle, \langle t_1, t_3 \to t_4\rangle, \langle t_2, \mathrm{unit}\rangle\} \\
\Longrightarrow\ & \{\langle u, t_1 \to \mathrm{unit}\rangle, \langle t_1, t_3 \to t_4\rangle, \langle t_2, \mathrm{unit}\rangle\} \\
\Longrightarrow\ & \{\langle u, (t_3 \to t_4) \to \mathrm{unit}\rangle, \langle t_1, t_3 \to t_4\rangle, \langle t_2, \mathrm{unit}\rangle\}
\end{aligned}$$

We define $\mathcal{U}(\bot) = \emptyset$.

Lemma 4. *If $C \Longrightarrow C'$ then $\mathcal{U}(C) = \mathcal{U}(C')$.*

Proof. The proof is by rule induction on the rules in Figure 3. Rules U:Dec, U:Mis, U:Elim1, and U:Cyc are straightforward.

Assume that $\langle t,\tau\rangle, C \Longrightarrow \langle t,\tau\rangle, [\sigma](C)$ by use of rule Elim2. That is $\sigma = \{t \mapsto \tau\}$ and $t \notin \mathrm{FV}(\tau)$.

$$\begin{aligned}
& \rho \in \mathcal{U}(\langle t,\tau\rangle, C) \\
\Longleftrightarrow\ & \rho(t) = \rho(\tau) \text{ and } \rho \in \mathcal{U}(C) \\
\Longleftrightarrow\ & \rho(t) = \rho(\tau) \text{ and } [\sigma] \circ \rho \in \mathcal{U}(C) \\
\Longleftrightarrow\ & \rho(t) = \rho(\tau) \text{ and } \rho \in \mathcal{U}([\sigma](C)) \\
\Longleftrightarrow\ & \rho \in \mathcal{U}(\langle t,\tau\rangle, [\sigma](C))
\end{aligned}$$

□

Theorem 5 (Soundness). *If $C \Longrightarrow^* C'$ with C' in solved form then $\sigma_{C'}$ is a MGU of C.*

Proof. By induction using Lemma 4 we have $\mathcal{U}(C) = \mathcal{U}(C')$. Moreover, $\sigma_{C'}$ is a MGU of C'). □

The completeness theorem states that a constraint set C is unifiable if and only if every irreducible constraint system derivable from C is in solved form.

Theorem 6 (Completeness). *Let C be a constraint set. Either every irreducible constraint set C' such that $C \Longrightarrow^* C'$ is in solved form or $C \Longrightarrow^* \perp$ and there is no such C'.*

Proof. The proof of the theorem has to parts.

1. We show that we can make progress if and only if C is not in solved form, that is, either $C \Longrightarrow C'$ or $C \Longrightarrow \perp$ or C is in solved form.

2. We show that we always reach $\perp$ or a constraint set in solved form after a finite number of steps.

To prove the first part, we simply perform a case distinction on the definition of solved form and the rules that define the step relation $\Longrightarrow$.

To prove the second part, we define the measure $\mu(C) = (m, n, k)$ where m is the number of unsolved variables in C, n is the sum of the sizes of the terms in C, and k is the number of constraints that are not of the form $\langle t, \tau\rangle$. We observe that every rule in Figure 3 decreases this lexicographic order if C is not in solved form. The last component k is needed because of the rule U:FLIP. □

Efficient Implementation of Unification Unification can be implemented in linear time [MM82]. However, many existing implementations do not implement the linear-time algorithm but a simpler version that uses a *union-find* data structure. The performance of these simpler algorithms is sufficient in practice.

Theorem 7 (See [MM82]). *The unification problem is solvable in linear time.*

In this lecture, we are not discussing the linear-time algorithm but the simpler one based on union-find. Union-find is a disjoint-set data structure that provides the following operations.

union(C_1,C_2) Combine two disjoint sets C_1 and C_2.

find(τ) For a given element τ, find the representative of the set that contains τ.

singleton(τ) Create a singleton set for a given element τ.

Union-find can be implemented so that the cost of the operation *singleton* is constant and the cost of the other operations is bounded by $\alpha(n)$ where n is the size of the set and α is the inverse Ackermann function, which is sometimes referred to as *near-constant-time* since the Ackermann function grows very fast.

The unification algorithm applies the procedure *unify* (defined below) to every type constraint in the constraint set. The complexity of the algorithm is quasi-linear, that is, $O(n \cdot \alpha(n))$. In the operation *find*, we left it unspecified which element is the representative of a set with multiple elements. The procedure *unify* maintains the invariant that each disjoint set contains at most one type that is not a variable, that is, at most one type τ_0 of the form $c(\tau_1, \ldots, \tau_n)$ for a type constructor c. We assume that the *union* operation is implemented so that this type τ_0 is the representative of the set. If no such τ_0 exists then an arbitrary type variable t_0 is the representative. Before running the *unify* procedure, we have to create a singleton set for each type τ that appears in the constraint set.

```
unify($\tau$,$\upsilon$):
  let $\tau$ = find($\tau$) in
  let $\upsilon$ = find($\upsilon$) in
  match $\tau$,$\upsilon$ with
  | c($\tau_1$,...,$\tau_n$), c($\upsilon_1$,...,$\upsilon_n$) → for i = 1 to n: unify($\tau_i$,$\upsilon_i$)
  | c(_)      , t          → union($\tau$,t)
  | t         , c(_)       → union(t,$\upsilon$)
  | _         , _          → error 'types do not match'
```

If no error occurred, the unifier can be obtained by repeatedly using the operation *find* to replace the occurrences of all variables.

As you may have noticed, the algorithm is not quite correct yet because there could be circularities in the constraints. So we could have an unsolvable constraint like $\langle t, \mathrm{arr}(t;\tau)\rangle$ in which the same type variable appears on the left and right-hand side of the constraint. Such a cycle can be present indirectly when t is in the set represented by $\mathrm{arr}(u;\tau_1)$ and u is in the set represented by $\mathrm{arr}(t;\tau_2)$.

To fix the unsound algorithm, we add a separate circularity check. To keep the algorithm efficient, we check for circularity in a second step after the procedure *unify* as been applied to all type constraints. This can be done for example by maintaining an order of type variables while deriving the unifier from the union-find data structure.

6 Polymorphic Types

So far, we have worked with a monomorphic type system. However, it can lead to code duplication if we can use a function only with one type. For instance, we have to define the identity function for every argument type that we are using it with in a program. Instead, we would like to implement a functions once and then use it with different types. In other words, we would like to have *polymorphic types*, that is, types that correspond to a set of monomorphic types. Of course, we still want to be able to perform type inference using unification.

6.1 System F

We are not studying the polymorphic lambda calculus (also known as *System F*) in this course. However, we will discuss some aspects of the System F's polymorphic type system. Details can be found in PFPL [Har12].

Polymorphic types are given by the following grammar.

$$\tau ::= t \mid \tau_1 \to \tau_2 \mid \forall t.\tau$$

The expressions of System F are defined as follows.

$$\begin{array}{rcl} e & ::= & x \\ & & e_1(e_2) \\ & & \lambda(x:\tau)e \\ & & \Lambda(t)e \\ & & e[\tau] \end{array}$$

The two new syntactic forms are type abstraction $\Lambda t.e$, which introduces a polymorphic type, and type application $e[\tau]$, which eliminates a polymorphic type. In a polymorphic type system like System F, polymorphic types τ appear at every type position (every type can be polymorphic). So we have type judgements of the form

$$\Delta;\Gamma \vdash e : \tau$$

where

$$\Delta ::= \cdot \mid \Delta, t$$

is the list of free type variables that appear in the judgment and Γ is a context that maps variables to polymorphic types. Instantiation and application of polymorphic types is explicit as given by the following type rules.

$$\frac{\Delta, t; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda(t) e : \forall t.\tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall t.\tau' \qquad \Delta \vdash \tau}{\Delta; \Gamma \vdash e[\tau] : [t/\tau]\tau'}$$

We write $\Delta \vdash \tau$ to indicate that τ well-formed under Δ. We inductively define this judgment using the following rules.

$$\frac{}{\Delta, t \vdash t} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{\Delta, t \vdash \tau}{\Delta \vdash \forall t.\tau}$$

System F is very powerful while still being normalizing (i.e., every evaluation terminates). However, System F and its extensions are not used as the basis of most popular programming languages. The reason is that type inference for System F is undecidable. If we erase type abstractions and applications from terms then we cannot restore them again automatically in general. More formally, if we define

$$\begin{array}{lcl} \text{erase}(x) & = & x \\ \text{erase}(e_1\ (e_2)) & = & \text{erase}(e_1)\ (\text{erase}(e_2)) \\ \text{erase}(\lambda(x:\tau)\ e) & = & \lambda(x)\ \text{erase}(e) \\ \text{erase}(\Lambda(t)\ e) & = & \text{erase}(e) \\ \text{erase}(e[\tau]) & = & \text{erase}(e) \end{array}$$

Theorem 8 ((Wells, 1994))**.** *Let e be an expression of the untyped lambda calculus. It is undecidable whether there is a well-typed expression e′ in System F such that erase*$((e') = e$.

Moreover, most partial type reconstruction problems for System F are undecidable as well. As a result, weaker polymorphic systems have been studied and implemented.

6.2 Let Polymorphism

The kind of polymorphism that has been most successful in practice has been introduced by Hindley and Milner. It occupies a sweet spot between expressivity and efficiency of inference and is usually called *let polymorphism*.

We will work the same expressions *e* as in the monomorphic system. The first step in defining a type system for let polymorphism is to restrict the form of polymorphic types. In System F, quantified types can appear at any position at which a type can appear, for example, as argument types of functions. In let polymorphism, we allow quantification at the beginning of a type only.

$$\begin{array}{lcll} \tau & ::= & t & t \\ & & \text{arr}(\tau_1;\tau_2) & \tau_1 \to \tau_2 \\ & & \text{unit} & \mathbf{1} \\ \\ \rho & ::= & \text{mty}(\tau) & \tau \\ & & \text{pty}(t.\rho) & \forall t.\rho \end{array}$$

So every type is for the form $\forall t_1 \ldots. \forall t_n.\tau$ where τ does not contain type quantifiers.

The second step in defining let polymorphism is to control the positions in which polymorphic types ρ can appear. Our type judgements will have the form

$$\Delta; \Gamma \vdash^p e : \tau$$

where τ is a monomorphic type,

$$\Gamma ::= \cdot \mid \Gamma, x : \rho$$

is a polymorphic context, and Δ represents the set of free type variables in the judgment.

$\boxed{\Delta;\Gamma \vdash^p e : \tau \qquad \text{Expression } e \text{ has monomorphic type } \tau \text{ in context } \Gamma}$

$$\frac{\Delta \vdash \tau_1 \cdots \Delta \vdash \tau_n \qquad \Gamma(x) = \forall t_1 \ldots. \forall t_n.\tau' \qquad \tau = [t_1/\tau_1, \ldots, t_n/\tau_n]\tau'}{\Delta;\Gamma \vdash^p x : \tau} \text{(P:VAR)}$$

$$\frac{}{\Delta;\Gamma \vdash^p \text{triv} : \text{unit}} \text{(P:UNIT)} \qquad \frac{\Delta \vdash \tau' \to \tau \qquad \Delta;\Gamma, f : \tau' \to \tau, x : \tau' \vdash^p e : \tau}{\Delta;\Gamma \vdash^p \text{fun}\{\tau,\tau'\}(f, x.e) : \tau' \to \tau} \text{(P:FUN)}$$

$$\frac{\Delta;\Gamma \vdash^p e_1 : \tau' \to \tau \qquad \Delta;\Gamma \vdash^p e_2 : \tau'}{\Delta;\Gamma \vdash^p \text{app}(e_1; e_2) : \tau} \text{(P:APP)}$$

$$\frac{\Delta, t_1, \ldots, t_2;\Gamma \vdash^p e_1 : \tau_1 \qquad \rho = \forall t_1 \ldots. \forall t_n.\tau_1 \qquad \Delta;\Gamma, x{:}\rho \vdash^p e_2 : \tau}{\Delta;\Gamma \vdash^p \text{let}(e_1; x.e_2) : \tau} \text{(P:LET)}$$

Figure 4: Rules of the polymorphic type system.

The type rules in Figure 4 are mostly identical to the rules of the monomorphic type system. The only differences are in the rules P:LET and P:VAR. In the rule P:LET, we generalize all type variables that appear free in the monomorphic type τ_1 but not in the context Γ. Symmetrically, we ensure in the rule P:VAR that a (potentially) generalized type gets instantiated with a substitution σ. In this way, we again obtain a monomorphic type τ.

Example 4. *Consider the following expressions.*

```
let f = λ(x:t) x in
let y = f(<>) in
f (f)
```

We cannot type this program using the monomorphic type system. However, we can type it using let polymorphism as follows.

```
let (f : ∀t.t → t) = λ(x:t) x : t → t in
let y = (f : 1 → 1) (<>) in (* [t/unit](t → t)) *)
(f : (u → u) → u → u) (f : u → u)
```

In contrast, we cannot type the following program.

```
λ(f : t → t)
  let y = f(<>) in
  f (f)
```

The problem is the rule P:ABS*. Other than in* P:LET*, we do not generalize the type τ' of the function argument. This means that f has a monomorphic type $t \to t$ and can only be applied to arguments of type t.*

It is remarkable that these severe restrictions to polymorphism do not make programming in languages like ML unpleasant. For some reason, most programs naturally use higher-order arguments only with one type in a function body.

Type Inference It is straightforward to extend constraint-based type checking to let polymorphism. To see why, consider the following type rule for let expressions. All other rules, including C:VAR, remain unchanged.

$$\frac{\Gamma \vdash^c e_1 : \tau_1 \mid C_1 \qquad \Gamma \vdash^c [e_1/x]e_2 : \tau \mid C_2}{\Gamma \vdash^c \text{let}(e_1; x.e_2) : \tau \mid C_1, C_2} \text{(CP:LET)}$$

The reason that we have the precondition $\Gamma \vdash^c e_1 : \tau_1 \mid C_1$ is to also type e_1 when x does not appear free in e_2. In this case, we would not type e_1 at all and thus potentially not derive a valid judgement of the let-polymorphic type system.

Example 5. *Consider again the previous example.*

```
let f = λ(x:t) x in
let y = f(<>) in
f (f)
```

Using, the rule CP:LET, *we would basically transform the expression into the following and then use the monomorphic (constraint-based) type system to infer the types.*

```
let f = λ(x:t) x in
let y = (λ(x:t_1) x)(<>) in
(λ(x:t_2) x) (λ(x:t_3) x)
```

In this expression, we have replaced variables that are with their definitions but left the let bindings in place to ensure type checking in the case in which bindings are not used (e.g., variable y).

For efficiency reasons it is not advisable to use the rule CP:LET for type checking in practice. The problem is that this copying of terms can cause an exponential blowup. Consider for instance the following expression.

```
let x1 = let x2 = ... let xn = λ(x:t) x in ... in x2(x2) in x1(x1)
```

After the first substitution, we would have the following expression. When we type-check the subexpressions of the application, we have to substitute four occurrences of *x2* etc.

```
(let x2 = ... let xn = λ(x:t) x in ... in x2(x2))
  (let x2 = ... let xn = λ(x:t) x in ... in x2(x2))
```

Complexity of Type Checking Let Polymorphism In practice, efficient implementations perform type inference for a let expression let$(e_1; x.e_2)$ by first performing type inference for e_1 and then assigning the generalized (polymorphic) type of e_1 to x in the type inference for e_2. In more detail, the steps are as follows.

1. Use the constraint typing rules to derive the judgement $\Gamma \vdash^c e_1 : \tau_1 \mid C_1$.
2. Use unification to find a most general unifier σ for the constraints C_1 and compute $[\sigma'](\tau_1)$ where σ' is σ without the type variables in Γ.
3. Generalize $[\sigma](\tau_1)$ to a polymorphic type ρ as in the rule C:LET and perform type inference for e_2 using the context $\Gamma, x : \rho$.

We then instantiate the quantified types in the rule for variable application. At every use of x, we then generate a new copy of the quantified type variables in the type of x and emit the respective type constraints with respect to these variables.

While this strategy is efficient in practice, the complexity of type inference remains exponential [KTU90, Mai90]. Consider for instance the following example.

```
let f0 = λ(x:t_0) (x,x) in
let f1 = λ(y:t_1) f0(f0 y) in
let f2 = λ(y:t_2) f1(f1 y) in
let f3 = λ(y:t_3) f2(f2 y) in
...
```

The types of the functions after type inference grow exponentially in the size of the expression. Modulo parenthesis for the nested tuples, we have

$$f_i : t \to t * \cdots * t \text{ where the product has size } 2^{i+1}\,.$$

As a result, every known algorithm for inference takes exponential time in the worst case.[1]

[1] It is also possible to construct such a program using function arrows instead of tuples.

6.3 Side-Effects and Value Restriction

Another infamous issue with let polymorphism is its interaction with side effects. Consider a functional language with references like OCaml. If we use the rule P:LET then we can type a program that has a clearly wrong behavior.

```
let r = ref (fun x → x ) in
let _ = r := (fun x → not x) in
(!r) (true,true)
```

Assume that the type of the reference r is generalized to $\forall t.t \rightarrow t$. Then it can be instantiated to bool → bool in the assignment and also to (bool ∗ bool) → (bool ∗ bool) in the look-up. However, we apply the function *not* : bool → bool to the value *(true,true)*, which is clearly not type safe.

The widely adopted solution to this problem is the so called *value restriction*. The value restriction ensures that the type of an expression e_1 in the rule P:LET is only generalized to a polymorphic type if the expression e_1 is a value.

There have been many other proposals for soundly combining let polymorphism and side effects. Note however that it is not sufficient just avoid polymorphic generalization of mutable data structures like references.

References

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[KTU90] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Ml typability is dextime-complete. In *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, CAAP ’90, pages 206–220, Berlin, Heidelberg, 1990. Springer-Verlag.

[Mai90] Harry G. Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’90, pages 382–401, New York, NY, USA, 1990. ACM.

[MM82] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.