A modern eye on ML type inference

– Old techniques and recent developments –

François Pottier
INRIA

September 2005

Abstract

Hindley and Milner's type system is at the heart of programming languages such as Standard ML, Objective Caml, and Haskell. Its expressive power, as well the existence of a type inference algorithm, have made it quite successful. Traditional presentations of this algorithm, such as Milner's Algorithm $\mathcal{W}$, are somewhat obscure. These short lecture notes, written for the APPSEM'05 summer school, begin with a presentation of a more modern, constraint-based specification of the algorithm, and explain how it can be extended to accommodate features such as algebraic data types, recursion, and (lexically scoped) type annotations. Then, two chapters, **yet to be written**, review two recent proposals for incorporating more advanced features, known as arbitrary-rank predicative polymorphism and generalized algebraic data types. These proposals combine a traditional constraint-based type inference algorithm with a measure of local type inference.

Contents

Chapter 1

Introduction

Typechecking is a discipline that assigns specifications, or *types*, to programs. A type typically is a term whose size is small. Typechecking distinguishes itself from other program analyses by its relative simplicity, which stems in part from its compositionality: the type ascribed to a program fragment depends only on the types ascribed to the sub-fragments out of which it is composed.

Typechecking is interesting on multiple grounds. First, a *type soundness* theorem, established by the type system designer, guarantees that a well-typed program cannot "crash:" its execution cannot fail unexpectedly. This result is obtained in a *static* manner, that is, before the program is executed. It provides users with a partial robustness guarantee, as well as with a limited *security* property, on top of which more advanced properties can be enforced.

Next, some more ambitious type systems allow static enforcement of more advanced security policies, such as *access control* or *information flow control* policies. Others encode *data flow* analyses, which usually are of little interest to programmers, but enable more aggressive optimizations during compilation.

Last, thanks to its compositional nature, and thanks to the related notions of parametric *polymorphism* and type *abstraction*, typechecking encourages *modularity*, that is, the decomposition of programs into independent and complementary units. This activity is central in the development of complex software systems.

One often wishes for type systems to be as transparent as possible, that is, for the typechecking process to be automatic or almost automatic. Indeed, even if the type discipline is known to the programmer, it is desirable not to burden him with extra work, that is, to only require minimal help out of him. Thus, the programmer is typically asked to provide a specification for each program module, but, if possible, nothing more. Furthermore, when the results of a type-based analysis are intended for use by a compiler, it is desirable for the analysis to be fully automatic. This leads to studying the *type inference* problem, that is, the problem of determining which types a program or program fragment admits.

Because typechecking is compositional, type inference problems also admit a natural decomposition. In other words, the type inference problem associated with a program fragment admits a solution if and only if each of the sub-problems associated with its sub-fragments also admits a solution and if these solutions are consistent with respect to one another, that is, if the sub-fragments admit complementary types.

As a consequence, a language for expressing type inference problems must offer *conjunction*, which allows combining several sub-problems; la *existential quantification*, which allows introducing a *type variable* that denotes a type to be determined; and *predicates* over types, such as the equality predicate, which allows requiring two types to match. In other words, type inference is naturally reduced to the satisfaction of logical formulæ, or *constraints*.

Experience suggests that, even when the programming language of interest is rich, the constraint language required to express its type inference problems remains relatively modest. Thus, reducing type inference problems to constraint satisfaction problems allows a significant restriction of the universe of discourse, and forms a useful first step.

Types and *constraints* are among the most important objects manipulated in this document. The document begins with an overview of Hindley and Milner's type system and of a constraint-based version of its type inference algorithm. It then discusses its extension with data structures, recursion, and optional type annotations. Two more advanced chapters, **yet to be written**, discuss further extensions where *mandatory type annotations* help and where a *local type inference* algorithm is layered on top of the traditional, constraint-based algorithm.

Chapter 2

Type inference for ML

The programming languages of the ML family, whose most influential members are Standard ML, Objective Caml, and Haskell, are based on Hindley and Milner's type system (also known as Damas and Milner's type system).

Hindley [1] solved the type inference problem for the simply-typed λ-calculus by showing, in a constructive way, that every expression admits a *principal type*. His algorithm relies on first-order unification, and can be (but was not at the time) presented as the combination of constraint generation and constraint solving phases.

The simply-typed λ-calculus is a *monomorphic* type system, where an expression e cannot *simultaneously* admit several distinct types. Nevertheless, the principal type for e, produced by Hindley's algorithm, can contain *type variables*, indicating that e in fact admits several types (an infinite number of them). Thus, the study of type inference reveals a form of *parametric polymorphism*, a notion identified by Strachey [2].

Milner [3] suggested internalizing the notion of principal type by introducing *type schemes* and by effectively allowing an expression to simultaneously have several types. He proposed a type inference algorithm that also relies on first-order unification, but that initially appeared more difficult to state in terms of constraints. The connection between type inference and constraint solving became stronger during the following two decades, thanks in particular to proposals for extending Hindley and Milner's type system with more complex features such as subtyping. Gradually, constraints became more systematically used in the formulation of type systems and type inference algorithms.

This chapter gives a constraint-based presentation of type inference for Hindley and Milner's type system and for some of its extensions. Its material is drawn from a book chapter by Pottier and Rémy [4]. I begin with the simply-typed λ-calculus (§2.1), move on to Hindley and Milner's type system (§2.2), then discuss more language features (§2.3).

2.1 The simply-typed λ-calculus

The definition of simply-typed λ-calculus appears in figure 2.1. It defines an inductive predicate whose general form is $\Gamma \vdash e : \tau$, where Γ is an *environment*, e is an *expression*, and τ is a *type*. Expressions are given by the grammar

$$e ::= x \mid \lambda x.e \mid e\ e$$

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma; x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Figure 2.1: The simply-typed λ-calculus

$$\begin{array}{rcl} [\![\Gamma \vdash x : \tau]\!] & = & \Gamma(x) = \tau \\ [\![\Gamma \vdash \lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2.([\![\Gamma; x : \alpha_1 \vdash e : \alpha_2]\!] \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\ [\![\Gamma \vdash e_1\ e_2 : \tau]\!] & = & \exists \alpha.([\![\Gamma \vdash e_1 : \alpha \rightarrow \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!]) \end{array}$$

Figure 2.2: Constraint generation for the simply-typed λ-calculus

where x denotes a *variable*. Types are given by the grammar

$$\tau ::= \alpha \mid \tau \rightarrow \tau$$

where α denotes a *type variable*. An environment is a partial mapping of variables to types. A triple $\Gamma \vdash e : \tau$ is a *pre-judgement*. It is a *judgement* if can be derived from the rules of figure 2.1. A pair (Γ, τ) is a *typing* of e if and only if $\Gamma \vdash e : \tau$ is a judgement. It is a *principal* typing of e if and only if every other typing of e is obtained out of it by substituting types for type variables. An expression e is *typable* if and only if it admits a typing.

In the simply-typed λ-calculus, every typable expression e admits a principal typing. This fact was proved, in the setting of combinatory logic, independently by Curry and by Hindley [1]. Curry's proof was direct, while Hindley's relied on Robinson's first-order unification algorithm [5]. Yet, Hindley does not explicitly use equality constraints: instead, he uses the unification algorithm as a black box, which, when supplied with two arbitrary type schemes, returns their least upper bound with respect to the instantiation ordering, when it exists. This least upper bound is known as their *highest common instance*. Thus, Hindley's algorithm implicitly mixes constraint generation and constraint solving.

It is difficult to tell precisely when type inference for the simply-typed λ-calculus was considered as the combination of distinct constraint generation and constraint solving phases. Such a view appears to have gradually emerged during the 1980s, at a time when more general constraints, such as subtyping constraints [6], were being introduced. This view is found, in a more or less informal fashion, in papers by Clément, Despeyroux, Despeyroux, and Kahn [7] or by Cardelli [8]. The first explicit reduction of type inference for the simply-typed λ-calculus to satisfaction of equality constraints seems due to Wand [9].

2.1.1 A reduction of type inference to constraint solving

The constraints generated by Wand's algorithm are made up solely of type equations and conjunctions. For this reason, Wand can only reason about *fresh* type variables in an informal way. Following Jouannaud and Kirchner [10], I extend the constraint language with existential quantification, which allows dealing with this notion in a formal and elegant way. Thus, the constraint language is as follows:

$$C ::= \tau = \tau \mid C \wedge C \mid \exists \alpha.C$$

Constraints are made up of type equations, conjunction, and existential quantification. They are interpreted in a Herbrand universe, that is, in a finite tree model. One could also interpret them in a regular tree model, yielding a type inference algorithm for an extension of the simply-typed λ-calculus with (equi-)recursive types. Constraint solving consists in determining whether a constraint C is satisfiable. A constraint solving algorithm is best presented as a rewrite system [10] whose *solved forms* are isomorphic to *most general unifiers*.

To every pre-judgement $\Gamma \vdash e : \tau$, where the domain of Γ contains the free variables of e, one associates a constraint, written $[\![\Gamma \vdash e : \tau]\!]$. The definition, which is by induction over the structure of the expression e, appears in figure 2.2.

It is implicitly agreed that the type variables α_1, α_2, and α must be chosen *fresh* with respect to Γ and τ, that is, must not appear free in Γ or τ. This could be made explicit by adding the side condition $\alpha_1, \alpha_2, \alpha \notin \mathrm{ftv}(\Gamma, \tau)$. This freshness criterion is formal, because it is local: we do not request that the new type variables be chosen just *"fresh,"* which does not mean anything, but fresh *with respect to a few specific objects*, namely Γ and τ. Note that every type variable that occurs free in $[\![\Gamma \vdash e : \tau]\!]$ must necessarily occur free in Γ or τ.

One can establish the following properties:

Theorem 2.1.1 *ϕ is a solution of $[\![\Gamma \vdash e : \tau]\!]$ if and only if $(\phi\Gamma, \phi\tau)$ is a typing of e.* $\diamond$

Corollary 2.1.2 *Let α be an arbitrary type variable. Let the environment Γ map the free variables of e to type variables that are pairwise distinct and distinct from α. Then, e is typable if and only if $[\![\Gamma \vdash e : \alpha]\!]$ is satisfiable. Furthermore, if ϕ is a principal solution of $[\![\Gamma \vdash e : \alpha]\!]$, then $(\phi\Gamma, \phi\alpha)$ is a principal typing of e.* $\diamond$

This result means that the type inference problem, which is to determine whether a term is typable, can be reduced to the constraint satisfaction problem, which is to determine whether a constraint is satisfiable. Furthermore, it implies that the simply-typed λ-calculus has *principal typings*:

Corollary 2.1.3 *If e is typable, then e admits a principal typing.* $\diamond$

This property means that it is possible to analyze an expression e *independently of its context* and to infer not only its type, but also the requirements that it imposes upon its environment. Such a property has applications to separate analysis and separate compilation [11, 12]. Unfortunately, it does not hold of ML's type system; I come back to this point later (§2.2.1).

It is easy to check that the constraint $[\![\Gamma \vdash e : \alpha]\!]$ of corollary 2.1.2 has size $O(n)$, where n is the size of e, and can be built in time $O(n)$ or $O(n \log n)$, depending on how variables and environments are represented. A standard unification algorithm, such as Huet's [13], based on Tarjan's *union-find* data structure [14], allows determining whether such a constraint is satisfiable in time $O(n\alpha(n))$. (The linear-time unification algorithm by Paterson and Wegman [15] isn't of interest here, because it is not incremental and cannot be used for ML.) To conclude, the time and space complexity of type inference for simply-typed λ-calculus is bounded by $O(n \log n)$.

2.1.2 An alternate reduction of type inference to constraint solving

In the previous paragraphs, I have decomposed the type inference problem in such a way that the notion of an environment, as well as the process of building and looking up environments, are local to the first phase, that is, constraint generation. The second phase, namely constraint solving, is not concerned with environments, since this notion does not appear in the grammar of constraints. Yet, it is possible, if desired, to decompose the problem in a slightly different way, so that the notion of an environment instead becomes local to the second phase.

To this end, I enrich the syntax of constraints:

$$C ::= \ldots \mid x = \tau \mid \textbf{\textit{def}}\ x : \tau\ \textbf{\textit{in}}\ C$$

I now allow variables x to appear free within constraints. For this reason, constraints are now interpreted not only with respect to a valuation ϕ, which maps every type variable α to an element of the model in which types are interpreted, but also with respect to a second valuation ψ, which maps every variable x to such an element. I introduce two new constraint forms and equip them with the following interpretation. The equation $x = \tau$ is satisfied by the valuations ϕ and ψ if and only if ψx and $\phi\tau$ coincide. The constraint ***def*** $x : \tau$ ***in*** C is satisfied by ϕ and ψ if and only if C is satisfied by ϕ and $\psi[x \mapsto \phi\tau]$.

The expressiveness of the constraint language is not fundamentally affected by this extension. Indeed, one can check that the equivalence law

$$\textbf{\textit{def}}\ x : \tau\ \textbf{\textit{in}}\ C \equiv [\tau/x]C$$

is valid in this interpretation. (Two constraints are equivalent if and only if they are satisfied by the same valuations.) In other words, the ***def*** construct is an explicit substitution form. It can be considered, to a certain extent, as a form of syntactic sugar: indeed, if a constraint C has no free variables, then the above law, oriented from left to right and viewed as a rewrite rule, allows rewriting C into an equivalent constraint, expressed in the initial syntax. A slight gain in expressiveness stems from the existence of constraints with free variables: for instance, $\exists\alpha.(x_1 = \alpha \wedge x_2 = \alpha)$ expresses the fact that the variables x_1 and x_2 should have a common type. On can also write $x = \tau_1 \wedge x = \tau_2$, which superficially appears

$$\begin{array}{rcl} [\![x : \tau]\!] & = & x = \tau \\ [\![\lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2.(\textbf{\textit{def}}\ x : \alpha_1\ \textbf{\textit{in}}\ [\![e : \alpha_2]\!] \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\ [\![e_1\ e_2 : \tau]\!] & = & \exists \alpha.([\![e_1 : \alpha \rightarrow \tau]\!] \wedge [\![e_2 : \alpha]\!]) \end{array}$$

Figure 2.3: Constraint generation for the simply-typed λ-calculus (variant)

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma; x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \bar{\alpha} \# \operatorname{ftv}(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}.\tau} \qquad \frac{\Gamma \vdash e : \forall \bar{\alpha}.\tau}{\Gamma \vdash e : [\vec{\tau}/\vec{\alpha}]\tau}$$

Figure 2.4: Hindley and Milner's type system

reminiscent of *intersection types* [16, 17]; yet, according to the above interpretation, this constraint is equivalent to $x = \tau_1 \wedge \tau_1 = \tau_2$. Thus, we remain within a simply-typed setting, where every entity (variable or expression) is monomorphic.

One can now offer an alternate reduction of type inference for the simply-typed λ-calculus to constraint solving. It appears in figure 2.3, which defines a mapping of an expression e and a type τ to a constraint $[\![e : \tau]\!]$. Note that environments have disappeared! Environment lookup, which allowed producing an equation of the form $\Gamma(x) = \tau$, has been suppressed; instead, one now produces the equation $x = \tau$, where the name x isn't resolved. Environment extension, written $\Gamma; x : \alpha_1$ and exploited to associate type α_1 to the variable x when analyzing a λ-abstraction $\lambda x.e$, has also disappeared. Instead, analysis of the abstraction body e produces a constraint $[\![e : \alpha_2]\!]$ within which x can occur free. These free occurrences of x are given a meaning, a posteriori, by wrapping this constraint into the context ***def*** $x : \alpha_1$ ***in*** $[]$.

What are the advantages of this new presentation with respect to the initial approach? There are several, all of which are minor. First, the specification is now more abstract. The constraint solver can choose to eagerly eliminate all ***def*** forms via substitution, as suggested earlier, which essentially leads to the initial algorithm. It can also choose another rewrite strategy, for instance, a *bottom-up* one, leading to a different algorithm. Next, corollary 2.1.2 can be reformulated in a slightly simpler way, because it is no longer necessary to explicitly construct an environment Γ consisting of distinct type variables:

Theorem 2.1.4 *Let α be an arbitrary type variable. Then, e is typable if and only if $[\![e : \alpha]\!]$ is satisfiable.* $\diamond$

When $[\![e : \alpha]\!]$ is satisfiable, a principal typing of e can be reconstructed in a simple way out of a solved form of this constraint. For instance, a solved form of $[\![x + 1 : \alpha]\!]$ is $x = int \wedge \alpha = int$, whence one can deduce that a principal typing of $x + 1$ is $(x : int, int)$. Further details of this construction are omitted.

In the case of simply-typed λ-calculus, thus, the two reductions of type inference to constraint solving proposed above differ only in a cosmetic way. The true interest of the second one lies in the fact that it alone admits an elegant generalization to the case of ML (§2.2.2).

2.2 Hindley and Milner's type system

ML's type system, defined by Milner [3] and by Damas and Milner [18], extends the simply-typed λ-calculus by ascribing *type schemes*, that is, polymorphic types, to the variables whose definition is known, that is, to variables bound by a new ***let*** form:

$$e ::= \ldots \mid \textbf{\textit{let}}\ x = e\ \textbf{\textit{in}}\ e$$

In contrast, λ-bound variables, that is, formal function parameters, must remain monomorphic.

$$
\begin{array}{lcl}
\textbf{\textit{fresh}} & = & \textbf{\textit{do}}\ \alpha \in V \\
& & \textbf{\textit{do}}\ V \leftarrow V \setminus \{\alpha\} \\
& & \textbf{\textit{return}}\ \alpha \\
\mathcal{J}(\Gamma \vdash x) & = & \textbf{\textit{let}}\ \forall \alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \\
& & \textbf{\textit{do}}\ \alpha'_1, \ldots, \alpha'_n = \textbf{\textit{fresh}}, \ldots, \textbf{\textit{fresh}} \\
& & \textbf{\textit{return}}\ [\alpha'_i/\alpha_i]_{i=1}^n(\tau) \\
\mathcal{J}(\Gamma \vdash \lambda x.e_1) & = & \textbf{\textit{do}}\ \alpha = \textbf{\textit{fresh}} \\
& & \textbf{\textit{do}}\ \tau_1 = \mathcal{J}(\Gamma; x : \alpha \vdash e_1) \\
& & \textbf{\textit{return}}\ \alpha \rightarrow \tau_1 \\
\mathcal{J}(\Gamma \vdash e_1\ e_2) & = & \textbf{\textit{do}}\ \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
& & \textbf{\textit{do}}\ \tau_2 = \mathcal{J}(\Gamma \vdash e_2) \\
& & \textbf{\textit{do}}\ \alpha = \textbf{\textit{fresh}} \\
& & \textbf{\textit{do}}\ \phi \leftarrow \mathrm{mgu}(\phi(\tau_1) = \phi(\tau_2 \rightarrow \alpha)) \circ \phi \\
& & \textbf{\textit{return}}\ \alpha \\
\mathcal{J}(\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2) & = & \textbf{\textit{do}}\ \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
& & \textbf{\textit{let}}\ \sigma = \overline{\forall}\,\mathrm{ftv}(\phi(\Gamma)).\phi(\tau_1) \\
& & \textbf{\textit{return}}\ \mathcal{J}(\Gamma; x : \sigma \vdash e_2)
\end{array}
$$

Figure 2.5: Algorithm $\mathcal{J}$

A type scheme is a type where zero or more type variables are universally quantified:

$$\sigma ::= \forall \bar{\alpha}.\tau$$

Hindley and Milner's type system is obtained by extending the definition of simply-typed λ-calculus (figure 2.1) with the rules of figure 2.4.

2.2.1 Algorithms $\mathcal{W}$ and $\mathcal{J}$

The introduction of polymorphism makes the type inference problem more complex. First, from a theoretical point of view, its complexity is significantly increased: indeed, it becomes DEXPTIME-complete [19, 20]. Furthermore, from a more pragmatic point of view, it apparently remained difficult, for a long time, to give a clear description of the type inference algorithm for Hindley and Milner's type system. The two equivalent algorithms proposed by Milner [3], $\mathcal{W}$ and $\mathcal{J}$, are quite involved. Indeed, they mix calls to an underlying unification algorithm, compositions and applications of substitutions, and operations over type schemes, such as instantiation and generalization, the latter of which requires determining which type variables occur free in the current environment. Furthermore, for some unknown reason, $\mathcal{W}$ appears to have become more popular than $\mathcal{J}$, even though the latter is viewed—with reason!—by Milner as a simplification of the former. Jones' presentation of type inference for Haskell [21] does rest upon an implementation of $\mathcal{J}$.

As an illustration, here is a definition of $\mathcal{J}$. It is rather close to Milner's. It rests upon a function $\mathrm{mgu}(\cdot)$, which to a conjunction of equations associates an (idempotent) most general unifier or fails. I do not recall the classic definition of this function.

For enhanced readability, the algorithm is presented in an apparently imperative style: it relies on two global variables ϕ and V. ϕ is the *current substitution*: it initially is the identity and represents the most general unifier of the equations solved so far. V is an arbitrary, infinite set of type variables: whenever the algorithm requires a "fresh" type variable, it is drawn out of V. Formally speaking, one can consider ϕ and V as standing for two implicit parameters and two implicit results of the algorithm, which then becomes purely functional. The apparently imperative syntax that I use is in fact none other than Haskell's ***do*** notation [22].

The algorithm accepts an environment Γ and an expression e. It produces a type τ or fails. It satisfies the following invariants: (i) ϕ is of the form $\mathrm{mgu}(C)$, for some constraint C; (ii) no type variable in V

occurs free in C, Γ, or in the result τ. These invariants imply that ϕ is idempotent, on the one hand, and that V is fresh with respect to ϕ, on the other hand. The algorithm is defined in figure 2.5.

The manner in which the current substitution ϕ is updated, on the fourth line of the application case, is such that, if ϕ initially was the most general unifier of some constraint C, then ϕ is, after the update, the most general unifier of the conjunction $C \wedge \tau_1 = \tau_2 \rightarrow \alpha$. This suggests that, instead of maintaining a current substitution ϕ, one could maintain a current conjunction of equations C, and delay its resolution. In fact, when does one actually need to compute ϕ and to apply it?

The answer appears a few lines down, in the ***let*** case. There, a type scheme σ is built out of the type $\phi(\tau_1)$ by universally quantifying all of the type variables that do not occur free in $\phi(\Gamma)$. (The notation $\overline{\forall}\bar{\alpha}$ quantifies over all type variables *except* $\bar{\alpha}$.) It is difficult to imagine how to build σ without exhibiting ϕ, that is, without solving C. Perhaps for this reason, it was often considered that, in the case of ML, constraint generation and constraint solving were inherently intermixed and could not be separated entirely. Yet, there is really no reason to believe so (§2.2.2).

What about the proof of algorithm $\mathcal{J}$? One can establish the following properties in succession:

Theorem 2.2.1 (Correctness) *If $\mathcal{J}(\Gamma \vdash e)$ terminates in state (ϕ, V) and returns τ, then $\phi(\Gamma) \vdash e : \phi(\tau)$ is a judgement.* ◇

Theorem 2.2.2 (Completeness) *Let Γ be an environment. Let (ϕ_0, V_0) be a state that satisfies the algorithm's invariant. Let θ_0 and τ_0 be such that $\theta_0\phi_0(\Gamma) \vdash e : \tau_0$ is a judgement. Then, the execution of $\mathcal{J}(\Gamma \vdash e)$ out of the initial state (ϕ_0, V_0) succeeds. Let (ϕ_1, V_1) be its final state and τ_1 be its result. Then, there exists a substitution θ_1 such that $\theta_0\phi_0$ and $\theta_1\phi_1$ coincide outside V_0 and such that τ_0 equals $\theta_1\phi_1(\tau_1)$.* ◇

Although the first statement above is simple, and easily proved by structural induction, the second one isn't. It is hard to decipher. As one might imagine, its proof is heavy, and, worse, does not, in my opinion, shed any light upon the algorithm.

It is historically interesting to note that, for many years, the only known completeness proof for $\mathcal{W}$ was the one by Damas [23], which never became widely available. Apparently only in the 1990s were completeness proofs more widely published [24] and mechanized [25–27].

One should also note that the proofs of algorithms $\mathcal{W}$, $\mathcal{J}$, or of their variants, such as $\mathcal{M}$ [28], are sufficiently different in their structure, even though they share identical ideas, to only share a few preliminary lemmas. The constraint-based approach described further on (§2.2.2) is superior in that the proof of the constraint generator is performed just once. The various classic algorithms, namely $\mathcal{W}$, $\mathcal{J}$, and $\mathcal{M}$, then merely correspond to various constraint solving *strategies*, whose correctness proof is not difficult.

Let us introduce a few more definitions. A typing (Γ', τ) is *relative to* Γ if and only if its first component Γ' is an instance of Γ. A typing of e is *principal relative to* Γ if and only if it is relative to Γ and every typing of e relative to Γ is an instance of it. Then, the two previous theorems lead to the following conclusion.

Corollary 2.2.3 (Relative principal typings) *The execution of $\mathcal{J}(\Gamma \vdash e)$ succeeds if and only if e admits a typing relative to Γ. Furthermore, if ϕ_1 and τ_1 are the algorithm's results, then $(\phi_1(\Gamma), \phi_1(\tau_1))$ is a typing of e and is principal relative to Γ.* ◇

It is instructive to compare this statement with that of corollary 2.1.3. In the case of the simply-typed λ-calculus, the inference algorithm expected just one argument, namely the expression e, and either produced a principal typing or failed. In the case of ML, algorithm $\mathcal{J}$ expects not only e but also an environment Γ, and produces a principal typing of e *relative to* Γ, that is, a typing that is principal only among those typings whose first component is an instance of Γ. Thus, the search is restricted to only part of the universe of all typings of e. For this reason, it is usually said that Hindley and Milner's type system does not have *principal typings*, but nevertheless has *principal types*. For further details, consult Jim [11] or Wells [12].

Of course, one could supply the algorithm with an environment Γ that consists, as in corollary 2.1.2, of pairwise distinct type variables. If every environment was an instance of such a Γ, as in the simply-typed λ-calculus, then the algorithm would produce a principal typing. Unfortunately, such is not the case. A nontrivial type scheme is not an instance of a type variable: for instance, the type scheme $\forall\beta.\beta \rightarrow \beta$ is not an instance of α. (The *type* $\beta \rightarrow \beta$ is an instance of α.) In other words, by supplying algorithm $\mathcal{J}$ with an environment that consists of type variables, one requires it to assign monomorphic types to all of e's free variables. A variable can have polymorphic behavior only if the initial environment maps it to a nontrivial type *scheme*. In short, the algorithm is unable to infer that a free variable should receive a polymorphic type.

2.2.2 A constraint-based approach

Algorithms $\mathcal{W}$ and $\mathcal{J}$ exploit the unification algorithm as a black box, which, out of two arbitrary types, produces a most general unifier. The algorithms are presented in such a way that a call to $\mathrm{mgu}(\cdot)$ appears necessary at least at every ***let*** node. Thus, Milner's presentation, just like Hindley's (§2.1), implicitly mixes constraint generation and constraint solving. Yet, for the sake of modularity, it would be desirable to separate these two phases.

It is interesting to note that the standard presentations of many constraint-based extensions of ML, such as HM(X) [29], suffer from the same flaw, although perhaps in a less obvious way. In HM(X), creating a type scheme apparently does not require solving the current constraint, because type schemes take the form $\forall\bar{\alpha}[C].\tau$, where the constraint C isn't necessarily a solved form. However, in reality, because the constraint C is *copied* whenever a fresh instance of the type scheme is taken, it is important that C be solved and simplified before the type scheme is created. For this reason, the original presentation of type inference for HM(X) mixed constraint generation and constraint solving [29].

The solution that I now present, drawn from work by Pottier and Rémy [4], exploits ***def*** constraints, analogous to those introduced in §2.1.2. It is applicable to Hindley and Milner's type system as well as HM(X). Here, I limit my interest to the former.

The idea is to enrich the constraint language considered in §2.1.2 by allowing a variable x to denote not just a type, but a (constrained) type scheme. The syntax of constraints and of constrained type schemes is now:

$$\begin{array}{lcl} C & ::= & \tau = \tau \mid C \wedge C \mid \exists\alpha.C \mid x \preceq \tau \mid \textbf{\textit{def}}\ x : \varsigma\ \textbf{\textit{in}}\ C \\ \varsigma & ::= & \forall\bar{\alpha}[C].\tau \end{array}$$

The logical interpretation of constraints is now relative to a valuation ϕ that maps every type variable α to an element of the model in which types are interpreted, as before, and to a valuation ψ that maps every variable x to a *set* of such elements. Indeed, a type scheme is interpreted as a set of types. The constraint $x \preceq \tau$, which might be read "type τ is an instance of the type scheme x," is satisfied by ϕ and ψ if and only if $\phi\tau$ is a member of ψx. The constraint ***def*** $x : \varsigma$ ***in*** C is satisfied by ϕ and ψ if and only if C is satisfied by ϕ and $\psi[x \mapsto {}^{\psi}_{\phi}(\varsigma)]$, where the interpretation ${}^{\psi}_{\phi}(\varsigma)$ of a constrained type scheme ς is defined as follows: if ς is $\forall\bar{\alpha}[C].\tau$, then its interpretation is the set of all $\phi'\tau$, where ϕ and ϕ' coincide outside $\bar{\alpha}$ and where ϕ' and ψ satisfy C.

Again, the effect of the above definitions, which can appear somewhat technical, is only to validate the equivalence law

$$\textbf{\textit{def}}\ x : \varsigma\ \textbf{\textit{in}}\ C \equiv [\varsigma/x]C$$

Thus, the ***def*** construct is an *explicit substitution* form. For the above law to make sense, though, one must define the meaning of the constraint $\varsigma \preceq \tau$, which might be read "type τ is an instance of the constrained type scheme ς." Indeed, such a constraint appears when x is replaced with ς in a constraint of the form $x \preceq \tau$. To define its meaning is to define its interpretation: $\varsigma \preceq \tau$ is satisfied by ϕ and ψ if and only if $\phi\tau$ is a member of ${}^{\psi}_{\phi}(\varsigma)$. Equivalently, this constraint can be considered syntactic sugar: indeed, if ς is $\forall\bar{\alpha}[C].\tau'$, and if $\bar{\alpha}$ is fresh with respect to τ, then $\varsigma \preceq \tau$ is equivalent to $\exists\bar{\alpha}.(C \wedge \tau = \tau')$. In other words, the type τ is an instance of the constrained type scheme $\forall\bar{\alpha}[C].\tau'$ if, for some assignment of the variables $\bar{\alpha}$ that satisfies C, τ' coincides with τ.

The extended constraint language offers constrained type schemes. Like in HM(X), this allows building type schemes without having to first perform a constraint solving step. Furthermore, the new language allows referring to a type scheme through a variable x, avoiding the need for *copying* a type scheme whenever a fresh instance of it is desired.

We can now express a constraint generation algorithm for Hindley and Milner's type system in the style of §2.1.2 (figure 2.3). The new algorithm appears in figure 2.6. The first three lines are identical

$$\begin{array}{rcl}
[\![x : \tau]\!] & = & x \preceq \tau \\
[\![\lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2.(\textbf{\textit{def}}\ x : \alpha_1\ \textbf{\textit{in}}\ [\![e : \alpha_2]\!] \wedge \alpha_1 \rightarrow \alpha_2 = \tau) \\
[\![e_1\ e_2 : \tau]\!] & = & \exists \alpha.([\![e_1 : \alpha \rightarrow \tau]\!] \wedge [\![e_2 : \alpha]\!]) \\
[\![\textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau]\!] & = & \textbf{\textit{let}}\ x : \forall \alpha[[\![e_1 : \alpha]\!]].\alpha\ \textbf{\textit{in}}\ [\![e_2 : \tau]\!]
\end{array}$$

Figure 2.6: Constraint generation for Hindley and Milner's type system

to those of figure 2.3, except $x = \tau$ is replaced with $x \preceq \tau$, reflecting the fact that x now denotes a type scheme and that τ can be an arbitrary instance of it. The main novelty lies in the fourth line. Upon first reading, one can view the let form, which hasn't been defined yet, as synonymous for ***def***. The algorithm first builds the constrained type scheme $\varsigma_1 = \forall\alpha[[\![e_1 : \alpha]\!]].\alpha$, where α is an arbitrary type variable. This is a *principal* constrained type scheme for e_1. The constraint that expresses the fact that e_2 is well-typed, namely $[\![e_2 : \tau]\!]$, can contain free occurrences of x. It is placed in the context ***let*** $x : \varsigma_1$ ***in*** $[]$, so that these free occurrences denote the constrained type scheme ς_1.

In fact, if ***let*** was defined as synonymous for ***def***, this constraint generation algorithm would not be quite correct. In the particular case where x does not occur free in e_2, the constraint ***def*** $x : \forall\alpha[[\![e_1 : \alpha]\!]].\alpha$ ***in*** $[\![e_2 : \tau]\!]$ is equivalent to $[\![e_2 : \tau]\!]$, which does not guarantee that e_1 is well-typed. Thus, we introduce ***let*** $x : \varsigma$ ***in*** C as syntactic sugar for ***def*** $x : \varsigma$ ***in*** $(\exists\alpha.x \preceq \alpha \wedge C)$. One can then check that the constraint ***let*** $x : \forall\alpha[[\![e_1 : \alpha]\!]].\alpha$ ***in*** $[\![e_2 : \tau]\!]$ implies $\exists\alpha.[\![e_1 : \alpha]\!]$, which guarantees that e_1 is well-typed. This is important because ML has call-by-value semantics: the expression e_1 is evaluated even when x does not occur free in e_2.

One can establish the correctness and completeness of this constraint generation algorithm with respect to the specification of Hindley and Milner's type system. Here is a statement of both properties:

Theorem 2.2.4 (Correctness and completeness) *Let* Γ *be an environment whose domain is* $\mathrm{fv}(e)$. *The expression* e *is well-typed relative to* Γ *if and only if* ***def*** Γ ***in*** $\exists\alpha.[\![e : \alpha]\!]$ *is satisfiable.* $\diamond$

It is important to note that the constraint generation algorithm has linear complexity. In other words, the modular decomposition into constraint generation and constraint solving can be effectively exploited as an implementation technique, with no asymptotic overhead.

The use of the ***def*** or ***let*** constructs is precisely intended to allow building a constraint of linear size. (Indeed, let us stress that eliminating them, via a naïve expansion, would cause an exponential blowup in the size of the constraint.) This technique appears due to Müller [30]. It was independently used by Gustavsson and Svenningsson [31], in a setting where the only base predicate was a subtype relationship between variables that denote atoms. Gustavsson and Svenningsson show that, in this specific setting, the strategy that consists in simplifying the left-hand side of a ***let*** definition before duplicating it leads to an algorithm of cubic complexity. The exponential blowup is thus avoided altogether. Unfortunately, this result does not hold in the case where constraints involve equations between types. Indeed, in that case, constraint solving is necessarily DEXPTIME-hard, since type inference for Hindley and Milner's type system can be reduced to it. Yet, McAllester [32] offers an interesting complexity result: under the double hypothesis that inferred types have bounded size and that the (left-) nesting depth of ***let*** definitions is bounded, a constraint can be generated and solved in linear time. Whether this hypothesis is reasonable is subject to debate; nevertheless, this result appears to explain why type inference for Hindley and Milner's type system is deemed "efficient in practice."

One might think that the constraint-based approach is a lot of fuss with little benefit by arguing that the programming language only has four constructs (variable, abstraction, application, and ***let***), where as the constraint language has more (equation, conjunction, existential quantification, instantiation, and ***let***). Does this mean that a constraint-based encoding sheds no light on the type inference problem? Not at all. As we shall see (§2.3), the introduction of products and sums, algebraic data types, references, exceptions, which are some of the features of a full-fledged incarnation of ML, require no extensions to the constraint language. Polymorphic recursion as well as "rigid" type annotations require universal quantification in the constraint language. Thus, in the case of a realistic programming language, the reduction to constraint solving represents a significant simplification of the initial problem.

2.2.3 A few words of constraint solving

A constraint solving algorithm is usually presented as a rewrite system. To demonstrate its correctness, one establishes three properties: (i) every rewrite step preserves the constraint's logical interpretation; (ii) the rewrite system is strongly normalizing; and (iii) it is trivial to decide whether a normal form (also known as *solved* form) is satisfiable.

One of the strong points of the constraint-based approach is that it offers the opportunity of defining several distinct constraint solving algorithms, which often correspond to distinct strategies within a single rewrite system. These algorithms can then share a single correctness proof.

The constraint language considered here is made up of a kernel language, which consists of type equations, conjunction and existential quantification, enriched with an explicit substitution mechanism, represented by the ***let*** construct and by instantiation constraints. Thus, it is natural to define the rewrite system, in a modular way, as the combination of a solver for the kernel language and of a separate rule set for dealing with explicit substitutions. The former is none other than a first-order unification algorithm, and can be arbitrary; only the structure of its solved forms must be agreed upon. The latter performs operations known as "generalization" et "instantiation" in classic implementations of ML.

The most natural way of dealing with ***let*** constraints is the one implicitly employed by Milner's algorithms $\mathcal{W}$ and $\mathcal{J}$, as well as by Rémy [33], Müller [30], or Gustavsson and Svenningsson [31]. When faced with the constraint ***let*** $x : \varsigma$ ***in*** C, one first simplifies the constrained type scheme ς, so as to make it as compact as possible. Then, one eliminates the ***let*** construct by replacing all occurrences of x by ς within C. Simplifying ς prior to duplication saves effort. Simplifying a constrained type scheme consists, at least, in solving the constraint that it contains. One can also go further. For instance, one can decrease the number of its universal quantifiers, then float part of its constraint outside of the ***let*** construct, again avoiding some duplication. These ideas, due to Rémy [33], are developed in detail in [4, §8]. An efficient implementation of these techniques requires associating an integer *rank* with every type variable. This mechanism, imagined by Rémy [33] and rediscovered by McAllester [32], can be understood in logical terms: the rank of a type variable tells where it is bound. Decreasing its rank amounts to floating its binder up within the current constraint, that is, to performing scope extrusion.

An alternative strategy consists in dealing with ***let*** constraints in two distinct ways, depending on whether the variable x of interest is λ-bound or ***let***-bound in the program. In the former case, the constraint ***def*** $x : \tau$ ***in*** C is dealt with by eliminating the ***def*** construct *after* the constraint C is solved. In the latter case, the constraint ***let*** $x : \varsigma$ ***in*** C is dealt with by simplifying ς and eliminating the ***def*** construct *before* C is analyzed, as above. This alternative approach corresponds to Mitchell's PTL algorithm [34], and was used in other works as well [35–37]. One of its strong points, according to Chitil [37], is to facilitate an interactive search for type errors, by preventing the constraint solver from propagating information *sideways* with respect to the program's tree structure.

2.3 Extensions

This section briefly discusses a few basic extensions that must be made to the calculus considered so far—a pure λ-calculus—in order to turn it into a reasonable programming language. I focus on three features: data structures, recursion, and optional type annotations, because each provides an opportunity of discussing *explicit type annotations*, a topic that becomes central later in this document when arbitrary-rank polymorphism and generalized algebraic data types are discussed. I omit a discussion of many other features, among which *references*, *exceptions*, *pattern matching*, *objects*, and *modules*.

2.3.1 Data structures

Products and sums

Introducing so-called *structural* products and sums is straightforward. One extends the grammar of types as follows:

$$\tau ::= \ldots \mid \tau \times \tau \mid \tau + \tau$$

The type $\tau_1 \times \tau_2$ describes a pair whose left (resp. right) component has type τ_1 (resp. τ_2). The type $\tau_1 + \tau_2$ describes either the application of the left injection to a value of type τ_1, or the application of the right injection to a value of type τ_2. These products and sums are sometimes referred to as *anonymous* because pair components and injections are unnamed: they are referred to via the predefined labels "left" and "right."

Like many simple language features, products and sums can be viewed either as new language constructs or simply and as new constants, together with some syntactic sugar. This choice does not have significant impact. Here, we adopt the second approach. Thus, we extend the calculus with the following constants:

$$\begin{array}{rcl} (\cdot,\cdot) & : & \forall \alpha_1\alpha_2.\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \times \alpha_2 \\ \pi_i & : & \forall \alpha_1\alpha_2.\alpha_1 \times \alpha_2 \rightarrow \alpha_i \\ \textbf{\textit{inj}}_i & : & \forall \alpha_1\alpha_2.\alpha_i \rightarrow \alpha_1 + \alpha_2 \\ \textbf{\textit{case}} & : & \forall \alpha_1\alpha_2\alpha.\alpha_1 + \alpha_2 \rightarrow (\alpha_1 \rightarrow \alpha) \rightarrow (\alpha_2 \rightarrow \alpha) \rightarrow \alpha \end{array}$$

Since our emphasis is not on dynamic semantics, we omit the reduction rules that come with these constants. We also omit the definition of the syntactic sugars that usually accompany the $(\cdot,\cdot)$ and ***case*** constants. In terms of type inference, which is the main topic of interest, the impact of introducing new constants is null. Indeed, constants are dealt with just like variables: it is only a matter of extending the initial environment with the above bindings.

Structural products and sums are simple, but they alone do not lead very far. Indeed, a type built out of products, sums, and base types (such as the ***unit*** type or the type of machine integers) can only describe values of *bounded* size. They do not allow describing lists, trees, or other data structures of unbounded size. For this, some form of *recursive* types is needed. Indeed, the informal definition: "a list is either empty or a pair of an element and a list" is recursive.

Equi-recursive types

The equation

$$\alpha = \textbf{\textit{unit}} + \tau \times \alpha$$

can be viewed as a characterization of the type of lists whose elements have type τ. Indeed, it paraphrases the informal definition above. The most obvious way of introducing recursive types into a type system is to ensure that this equation admits a solution. This can be done by extending the grammar of types with syntax for regular trees:

$$\tau ::= \ldots \mid \mu\alpha.\tau$$

This is finite syntax for the regular tree obtained by "infinite unfolding." (Well-formedness conditions rule out meaningless "types," such as $\mu\alpha.\alpha$, whose infinite unfolding isn't well-defined.) Two types τ_1 and τ_2 are said to be *interconvertible* when their infinite unfoldings coincide; we then write $\tau_1 =_\mu \tau_2$. Two interconvertible types can be used interchangeably: this is formalized by introducing a new typing rule.

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 =_\mu \tau_2}{\Gamma \vdash e : \tau_2}$$

This rule is not syntax-directed, so it can be applied at any point in a program. As a result, its impact on type inference is not local, but pervasive. None of the constraint generation rules is affected, but the *interpretation* of constraints changes: equations must now be interpreted in a *regular tree* model, instead of a finite tree model, so that the equation $\alpha = \textbf{\textit{unit}} + \tau \times \alpha$ is viewed as satisfiable. Its solution is the regular tree denoted by $\mu\alpha.\textbf{\textit{unit}} + \tau \times \alpha$. In a finite tree model, it is unsatisfiable. In terms of constraint solving, this requires moving from unification of *finite* first-order terms to unification of *regular* first-order terms. The unification algorithm is almost unchanged [13]: the only difference is that the *occurs check* is removed.

This approach to recursive types is known as the *equi-recursive* approach [38, 39], because equality modulo infinite unfolding is placed at the heart of the type system. One of its strong points is to not require any explicit type annotations or declarations, so that full type inference is preserved. For this reason, it is exploited, for instance, in the object-oriented subsystem of Objective Caml [40]. Its main disadvantage is that, in the presence of equi-recursive types, many apparently meaningless programs have types. For instance, self-application $\lambda x.(x\ x)$ has principal type $\forall\beta.(\mu\alpha.\alpha \to \beta) \to \beta$. Yet, self-application is rarely used in practice. It is likely that the programmer intended to write something else—perhaps $\lambda x.(x + x)$. Thus, it would be better to immediately reject this program, instead of accepting it and assigning it a baroque type.

Iso-recursive types

The *iso-recursive* approach to recursive types addresses this problem. In this alternate approach, the equation

$$\alpha = \textbf{\textit{unit}} + \tau \times \alpha$$

is again viewed as unsatisfiable—that is, constraints are again interpreted in a finite tree model. Instead, the user is allowed to *declare* that a new type constructor, say ***list***, satisfies the *isomorphism*

$$\textbf{\textit{list}}\,\alpha \approx \textbf{\textit{unit}} + \alpha \times \textbf{\textit{list}}\,\alpha$$

or, more generally,

$$T\,\vec{\alpha} \approx \tau$$

where T is the user-defined type constructor and τ can refer to T as well as to the type parameters $\vec{\alpha}$. Declarations of iso-recursive types can in fact be mutually recursive: every equation can refer to a type constructor introduced by any other equation.

By *isomorphism*, it is meant that $T\,\vec{\alpha}$ and τ are distinct types, but that it is possible to convert one into the another via the *explicit* application of a constant whose dynamic semantics is the identity. Two constants are introduced for this purpose:

$$\begin{array}{rcl} \textbf{\textit{fold}}_T & : & \forall\bar{\alpha}.\tau \to T\,\vec{\alpha} \\ \textbf{\textit{unfold}}_T & : & \forall\bar{\alpha}.T\,\vec{\alpha} \to \tau \end{array}$$

In the iso-recursive approach, converting from $T\,\vec{\alpha}$ to its unfolding τ, or vice-versa, requires an explicit use of $\textbf{\textit{fold}}_T$ or $\textbf{\textit{unfold}}_T$, that is, an explicit annotation, whereas, in the equi-recursive approach, the conversion was implicit. As a result, full type inference is no longer available. In return, exotic terms such as $\lambda x.(x\ x)$ are rejected, which pragmatically is good. This can be viewed as a situation where *mandatory type annotations* are helpful.

If the ***list*** type constructor is declared as above, then the empty list is written

$$\textbf{\textit{fold}}_{\textbf{\textit{list}}}\ (\textbf{\textit{inj}}_1\ ())$$

A list l of type $\textbf{\textit{list}}\,\alpha$ is deconstructed by

$$\textbf{\textit{case}}\ (\textbf{\textit{unfold}}_{\textbf{\textit{list}}}\ l)\ (\lambda n.\ldots)\ (\lambda c.\textbf{\textit{let}}\ hd = \pi_1\,c\ \textbf{\textit{in}}\ \textbf{\textit{let}}\ tl = \pi_2\,c\ \textbf{\textit{in}}\ \ldots)$$

A common idiom is to *fold* when *constructing* data and to *unfold* when *deconstructing* it.

Algebraic data types

In ML and Haskell, structural products and sums are fused with iso-recursive types, yielding so-called *algebraic data types* [41]. The idea is to avoid requiring both a (type) *name* and a (field or tag) *number*, as in

$$\textbf{\textit{fold}}_{\textbf{\textit{list}}}\ (\textbf{\textit{inj}}_1\ ())$$

Indeed, this is verbose. The use of a numeric tag is unpleasant, as it is difficult to remember the meaning of tags, and fragile: programs are likely to break when the definition of lists evolves. Instead, it would be desirable to mention *a single name*, as in

$$\textit{Nil}\ ()$$

This is permitted by algebraic data type declarations.

An algebraic data type constructor T is introduced via a *record type* or *variant type* definition:

$$T\,\vec{\alpha} \approx \prod_{i=1}^{k} \ell_i : \tau_i \qquad \text{or} \qquad T\,\vec{\alpha} \approx \sum_{i=1}^{k} \ell_i : \tau_i$$

Labels ℓ are used as names for fields or tags. (Field labels and tag labels could be taken in two distinct syntactic categories.) The record labels used in all algebraic data type declarations must be pairwise distinct, so that every record label can be uniquely associated with a type constructor T and with an index i. A similar requirement bears on tag labels.

The record type definition

$$T\,\vec{\alpha} \approx \prod_{i=1}^{k} \ell_i : \tau_i$$

introduces the constants

$$\begin{array}{rcll} \ell_i & : & \forall\bar{\alpha}.T\,\vec{\alpha} \rightarrow \tau_i & \qquad i \in \{1, \ldots, k\} \\ \boldsymbol{make}_T & : & \forall\bar{\alpha}.\tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow T\,\vec{\alpha} & \end{array}$$

Some syntactic sugar is in order. In concrete syntax, we write $e.\ell$ for $(\ell\ e)$. When $k > 0$, we write $\{\ell_i = e_i\}_{i=1}^{k}$ for $(\boldsymbol{make}_T\ e_1\ \ldots\ e_k)$.

The variant type definition

$$T\,\vec{\alpha} \approx \sum_{i=1}^{k} \ell_i : \tau_i$$

introduces the constants

$$\begin{array}{rcll} \ell_i & : & \forall\bar{\alpha}.\tau_i \rightarrow T\,\vec{\alpha} & \qquad i \in \{1, \ldots, k\} \\ \boldsymbol{case}_T & : & \forall\bar{\alpha}\gamma.T\,\vec{\alpha} \rightarrow (\tau_1 \rightarrow \gamma) \rightarrow \ldots (\tau_k \rightarrow \gamma) \rightarrow \gamma & \end{array}$$

In concrete syntax, we write $\boldsymbol{case}\ e\ [\ell_i : e_i]_{i=1}^{k}$ for $(\boldsymbol{case}_T\ e\ e_1\ \ldots\ e_n)$ when $k > 0$.

One can now declare

$$\boldsymbol{list}\,\alpha \approx \mathit{Nil} : \boldsymbol{unit} + \mathit{Cons} : \alpha \times \boldsymbol{list}\,\alpha$$

This gives rise to

$$\begin{array}{rcl} \mathit{Nil} & : & \forall\alpha.\boldsymbol{unit} \rightarrow \boldsymbol{list}\,\alpha \\ \mathit{Cons} & : & \forall\alpha.\alpha \times \boldsymbol{list}\,\alpha \rightarrow \boldsymbol{list}\,\alpha \\ \boldsymbol{case}_{\boldsymbol{list}} & : & \forall\alpha\gamma.\boldsymbol{list}\,\alpha \rightarrow (\boldsymbol{unit} \rightarrow \gamma) \rightarrow (\alpha \times \boldsymbol{list}\,\alpha \rightarrow \gamma) \rightarrow \gamma \end{array}$$

Then, the empty list is written

$$\mathit{Nil}\ ()$$

A list l of type $\boldsymbol{list}\,\alpha$ is deconstructed by

$$\begin{array}{l} \boldsymbol{case}\ l\ [\\ \quad \mathit{Nil} : \lambda n.\ldots \\ \quad \mid \mathit{Cons} : \lambda c.\boldsymbol{let}\ \mathit{hd} = \pi_1\,c\ \boldsymbol{in}\ \boldsymbol{let}\ \mathit{tl} = \pi_2\,c\ \boldsymbol{in}\ \ldots \\ ] \end{array}$$

This yields concrete syntax that is more pleasant, and more robust, than that obtained when viewing structural products and sums and iso-recursive types as two orthogonal language features. This explains the success of algebraic data types.

2.3.2 Recursion

The ability to define data structures of unbounded size, such as lists, is only one side of the coin. Indeed, it is also necessary to be able to define *functions* that manipulate such data structures. The definition of these functions typically requires *recursion*.

Monomorphic recursion

In the pure λ-calculus, recursion needs not be introduced as an additional language feature: indeed, it is possible, within the language, to define a *fixpoint combinator*, which allows encoding arbitrary recursive function definitions. However, this is mostly a theoretical consideration. In practice, it is just as simple to view this combinator as a constant, together with an *ad hoc* reduction rule. The type scheme ascribed to this constant is

$$\textbf{\textit{fix}} : \forall\alpha\beta.((\alpha \to \beta) \to (\alpha \to \beta)) \to \alpha \to \beta$$

and the reduction rule, in a call-by-value setting, is

$$\textbf{\textit{fix}}\, v_1\, v_2 \to v_1\, (\textbf{\textit{fix}}\, v_1)\, v_2$$

This is perhaps not very enlightening at first sight. The idea is that a simple recursive function definition, of the general form

$$\textbf{\textit{letrec}}\ f = \lambda x.e_1\ \textbf{\textit{in}}\ e_2$$

can then be viewed as syntactic sugar for

$$\textbf{\textit{let}}\ f = \textbf{\textit{fix}}\ (\lambda f.\lambda x.e_1)\ \textbf{\textit{in}}\ e_2$$

In short, an application of ***fix*** yields an anonymous recursive function, which, via a ***let*** definition, receives the name f within e_2. This syntactic sugar allows deriving not only the *dynamic semantics* of ***letrec***, but also its *static semantics*. Indeed, in the setting of Hindley and Milner's type system, a simple analysis of the expression ***let*** $f =$ ***fix*** $(\lambda f.\lambda x.e_1)$ ***in*** e_2 shows that it admits type τ_2 under environment Γ if and only if one can prove

$$\begin{array}{c}\Gamma; f : \tau \to \tau'; x : \tau \vdash e_1 : \tau' \\ \bar{\alpha} \# \mathrm{ftv}(\Gamma) \\ \Gamma; f : \forall\bar{\alpha}.\tau \to \tau' \vdash e_2 : \tau_2\end{array}$$

for appropriate values of τ, τ', and $\bar{\alpha}$. That is, the typechecking rule for ***letrec*** that arises out this syntactic sugar is

$$\frac{\Gamma; f : \tau \to \tau'; x : \tau \vdash e_1 : \tau' \qquad \bar{\alpha} \# \mathrm{ftv}(\Gamma) \qquad \Gamma; f : \forall\bar{\alpha}.\tau \to \tau' \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{\textit{letrec}}\ f = \lambda x.e_1\ \textbf{\textit{in}}\ e_2 : \tau_2}$$

The first premise checks that the recursive function definition is consistent: f and $\lambda x.e_1$ must agree upon a common type $\tau \to \tau'$. The second and third premises allow this type to be generalized, that is, turned into a type scheme $\forall\bar{\alpha}.\tau \to \tau'$, which is assigned to f when typechecking e_2.

This rule is peculiar in that the variable f must be assigned a *monomorphic* type while typechecking the *body* of the definition, that is, within e_1. It is assigned a *polymorphic* type only while typechecking the definition's *uses*, that is, within e_2. This is a weakness, and is sometimes a problem in practice. At best, the problem can be circumvented via code duplication; in some cases, no workaround exists, so there is a loss in expressive power [42].

In terms of type inference, the constraint

$$[\![\textbf{\textit{letrec}}\ f = \lambda x.e_1\ \textbf{\textit{in}}\ e_2 : \tau]\!]$$

is equivalent to

$$\textbf{\textit{let}}\ f : \forall\alpha\beta[\textbf{\textit{let}}\ f : \alpha \to \beta; x : \alpha\ \textbf{\textit{in}}\ [\![e_1 : \beta]\!]].\alpha \to \beta\ \textbf{\textit{in}}\ [\![e_2 : \tau]\!]$$

Because α and β denote *types*, the variable f is considered *monomorphic* while typechecking e_1. It receives a *polymorphic* type scheme only while typechecking e_2.

Polymorphic recursion

[43] suggested resolving this issue by introducing a more symmetric rule for ***letrec*** definitions:

$$\frac{\Gamma; f : \sigma \vdash \lambda x.e_1 : \sigma \qquad \Gamma; f : \sigma \vdash e_2 : \tau}{\Gamma \vdash \textbf{\textit{letrec}}\ f = \lambda x.e_1\ \textbf{\textit{in}}\ e_2 : \tau}$$

Here, σ is an arbitrary type scheme. The body of the definition and its uses are typechecked within a common environment, where it is permitted for f to receive a type scheme. This rule is strictly more expressive, yet safe. It is known as *polymorphic recursion*.

Unfortunately, type inference in the presence of this feature becomes problematic. Informally speaking, it seems that the rule requires *guessing a type scheme*, which first-order unification cannot do. In fact, the problem was shown by Henglein [44] and by Kfoury, Tiuryn, and Urzyczyn [45] to be inter-reducible with *semi-unification*, an undecidable problem. Several semi-algorithms are folklore and reportedly work well [46]. Still, having to rely on a (potentially unpredictable) semi-algorithm is somewhat discomforting.

Fortunately, type inference in the presence of polymorphic recursion becomes a simple problem again if one is willing to rely on a *mandatory type annotation*. The type system's specification is modified as follows:

$$\frac{\Gamma; f : \sigma \vdash \lambda x.e_1 : \sigma \qquad \Gamma; f : \sigma \vdash e_2 : \tau}{\Gamma \vdash \textbf{\textit{letrec}}\ f : \sigma = \lambda x.e_1\ \textbf{\textit{in}}\ e_2 : \tau}$$

Here, σ is no longer guessed: it is provided by the programmer. (For simplicity, let us assume, for the time being, that σ is a closed type scheme. This limitation can easily be removed; see §2.3.3.)

It can be shown that the constraint generation rule that corresponds to this typechecking rule is

$$\begin{gathered} [\![\textbf{\textit{letrec}}\ f : \sigma = \lambda x.e_1\ \textbf{\textit{in}}\ e_2 : \tau]\!] \\ = \\ \textbf{\textit{let}}\ f : \sigma\ \textbf{\textit{in}}\ ([\![\lambda x.e_1 : \sigma]\!] \wedge [\![e_2 : \tau]\!]) \end{gathered}$$

Here, it is clear that f is assigned the type scheme σ both *inside and outside* of the recursive definition. For this constraint generation rule to make sense, though, there remains to define the notation $[\![e : \sigma]\!]$, which is new. This requires extending the constraint language with *universal quantification*:

$$C ::= \ldots \mid \forall \alpha.C$$

The interpretation of this new constraint form is standard. Constraint solving now amounts to *first-order unification under a mixed prefix*, a simple extension of standard first-order unification, for which there is no established reference—consult, for instance, Pottier and Rémy [47, §1.10].

It should be intuitively clear that e admits the type scheme $\forall \alpha.\alpha \to \alpha$ if and only if e has type $\alpha \to \alpha$ *for every possible instance of* α, or, equivalently, for an abstract α. To reflect this, one defines

$$[\![e : \forall \bar{\alpha}.\tau]\!]$$

as syntactic sugar for

$$\forall \bar{\alpha}.[\![e : \tau]\!]$$

The need for universal quantification arises when polymorphism is *asserted* by the programmer—as opposed to *inferred* by the system.

2.3.3 Optional type annotations

Polymorphic recursion is a situation where a *mandatory* type annotation *helps*. This is in contrast with the *optional* type annotations allowed by Standard ML, Objective Caml, or Haskell, which can only *restrict* the set of valid types for a program.

What's the point of providing explicit type annotations, when types can be inferred anyway? I can think of two answers. First, type annotations serve as machine-checked documentation. Second, type annotations allow partitioning a large type inference problem into several independent sub-problems, which makes it easier to report accurate type error locations when a program is ill-typed.

What's the point of explicitly making a program's principal type less general than it could otherwise be? Perhaps the program is more versatile than intended, and the type annotations, which serve as documentation, document its intended use only.

The treatment of optional type annotations has remained part of the folklore for a long time, and has only rather recently been put in print by Peyton Jones and Shields [48] and by Pottier and Rémy [47]. When expressed in terms of constraints, it is particularly simple.

Optional type annotations are introduced by the rule

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e : \tau) : \tau}$$

Here, the type τ is provided by the programmer as part of the annotation. It must be a ground type, because we have not (yet) introduced any means of *binding* type variables in expressions. Constraint generation is extended accordingly:

$$[\![(e : \tau) : \tau']\!] = [\![e : \tau]\!] \wedge \tau = \tau'$$

It is not difficult to check that $[\![(e : \tau) : \tau']\!]$ entails $[\![e : \tau']\!]$, which means that the presence of the annotation makes the generated constraint *more specific* (more demanding). This is why type annotations restrict the set of valid types for a program.

Now, what about *non-ground* type annotations? Does it make sense for a type annotation to mention (the name of) a type variable? It does, provided this name has been explicitly *bound* earlier in the program. That is, we now need one (or several) program constructs for introducing type variables.

When the programmer introduces a new type variable, he presumably intends that variable to denote an unknown type. But does the programmer mean that the program should be well-typed for *some* instance of this variable, or for *all* such instances? The two interpretations differ. Both are useful. As a result, it makes sense to introduce two program constructs for binding type variables, corresponding to *existential* and *universal* quantification, respectively.

Their typechecking rules are as follows:

$$\frac{\Gamma \vdash [\tau/\alpha]e : \sigma}{\Gamma \vdash \exists\alpha.e : \sigma} \qquad\qquad \frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash \forall\alpha.e : \forall\alpha.\sigma}$$

The rule for existential quantification requires e to be well-typed under *some* instantiation of α. Indeed, it suffices to find *one* type τ that satisfies the premise for the conclusion to hold. The substitution $[\tau/\alpha]$ is applied to the expression e, that is, to all type annotations within e. Thus, all references to α within e are replaced with references to τ in the premise.

The rule for universal quantification requires e to be well-typed under the *absence* of any hypothesis about α, that is, under the hypothesis that α is abstract. This is formalized by its second premise. The rule is in fact identical to the standard rule for generalization in Hindley and Milner's type system, except it is syntax-directed. References to α within e are not substituted out.

Constraint generation for existential quantification is straightforward:

$$[\![\exists\alpha.e : \tau]\!] = \exists\alpha.[\![e : \tau]\!]$$

The type annotations inside e potentially contain free occurrences of α. Thus, the constraint $[\![e : \tau]\!]$ itself can contain such occurrences. They are given meaning by the existential quantifier. It is clear, here, that it is up to type inference—more precisely, up to the constraint solver—to determine an appropriate value for α.

For instance, the expression

$$\lambda x_1.\lambda x_2.\exists\alpha.((x_1 : \alpha), (x_2 : \alpha))$$

has principal type scheme

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

Indeed, the generated constraint contains the pattern

$$\exists\alpha.(\llbracket x_1 : \alpha\rrbracket \wedge \llbracket x_2 : \alpha\rrbracket \wedge \ldots)$$

which causes x_1 and x_2 to share a common type. Without the annotation, its principal type would be more general. If α was universally quantified $(\lambda x_1.\lambda x_2.\forall\alpha.\ldots)$, the expression would be ill-typed, since it would contain an assertion that x_1 and x_2 have every type. If α was universally quantified *up front*, $(\forall\alpha.\lambda x_1.\lambda x_2.\ldots)$, the expression would again be well-typed.

Constraint generation for universal quantification is somewhat more subtle. A naïve definition *fails*:

$$\llbracket\forall\bar{\alpha}.e : \tau\rrbracket = \forall\bar{\alpha}.\llbracket e : \tau\rrbracket$$

This requires τ to be simultaneously equal to *all* of the types that e assumes when $\bar{\alpha}$ varies. This is not what was intended. The informal intended meaning was that e should be well-typed for *all* instances of $\bar{\alpha}$ and that its eventual type τ corresponds to *some* such instance. To reflect this, one can instead define

$$\llbracket\forall\bar{\alpha}.e : \tau\rrbracket = \forall\bar{\alpha}.\exists\gamma.\llbracket e : \gamma\rrbracket \wedge \exists\bar{\alpha}.\llbracket e : \tau\rrbracket$$

This definition is correct. The trouble with it is that e is duplicated, which means that constraint generation no longer has linear time or space complexity. This can be avoided with a slight extension of the ***let*** constraint form [47].

Chapter 3

Conclusion

These lecture notes are currently **incomplete**. Two more chapters were planned and were to describe situations where *mandatory type annotations help* and where a *local type inference* algorithm is layered on top of a traditional, constraint-based type inference algorithm so as to reduce the amount of required annotations—in effect, turning some mandatory type annotations into optional type annotations again!

One chapter should discuss type inference in the presence of arbitrary-rank polymorphism. References are early attempts to marry type inference for Hindley and Milner's type system with first-class polymorphism, via a feature that one could refer to as "iso-universal types" [49, 50]; Läufer and Odersky's proposal for introducing arbitrary-rank polymorphism without relying on explicit type declarations [51]; Peyton Jones *et al.*'s suggestion of introducing an additional *local type inference* component [52]; and Rémy's reconstruction of their work [53]. Also relevant are papers by Garrigue and Rémy [54], Le Botlan and Rémy [55], and Vytiniotis, Weirich, and Peyton Jones [56].

One chapter should discuss type inference in the presence of generalized algebraic data types [57]. Recent references are papers by Simonet and Pottier [58], Peyton Jones, Washburn, and Weirich [59], Pottier and Régis-Gianas [60], and Sulzmann, Wazny, and Stuckey [61].

What lessons could be drawn from the papers discussed in these notes? In short,

- Constraint-based type inference is a versatile tool that can deal with many language features while relying on a single constraint solver. The solver's definition can be complex, but its behavior remains predictable because it is *correct* and *complete* with respect to the logical interpretation of constraints.

- In situations where it is difficult or impossible, due to intractability or undecidability issues, to design a complete constraint solver, the *constraint generation* process can be modified to take advantage of user-provided *hints*—typically type annotations.

- In situations where the necessary hints are so numerous that they become a burden, a *local type inference* algorithm can be used to automatically produce some of these hints. Although its design is usually *ad hoc*, it should remain predictable if it is sufficiently simple.

These ideas appear to be making their way into the next generation of programming languages equipped with type inference.

Bibliography

[1] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[2] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

[3] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[4] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[5] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[6] John C. Mitchell. Coercion and type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 175–185, January 1984.

[7] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 13–27, 1986.

[8] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[9] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticæ*, 10: 115–122, 1987.

[10] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. Technical Report 561, Université Paris-Sud, April 1990.

[11] Trevor Jim. What are principal typings and what are they good for? Technical Report MIT/LCS TM-532, Massachusetts Institute of Technology, August 1995.

[12] J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer Verlag, 2002.

[13] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris 7, September 1976.

[14] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[15] M. S. Paterson and M. N. Wegman. Linear unification. In *Annual ACM Symposium on Theory of Computing*, pages 181–186, 1976.

[16] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.

[17] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In J. Roger Hindley and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, 1980.

[18] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[19] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer Verlag, 1990.

[20] Harry G. Mairson, Paris C. Kanellakis, and John C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[21] Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999.

[22] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[23] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[24] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

[25] Catherine Dubois and Valérie Ménissier-Morain. Typage de ML: Spécification et preuve en Coq. In *Actes du GDR Programmation*, November 1997.

[26] Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3–4):319–346, November 1999.

[27] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[28] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[29] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[30] Martin Müller. A constraint-based recast of ML-polymorphism. In *International Workshop on Unification*, June 1994. Technical Report 94-R-243, CRIN, Nancy, France.

[31] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*. Springer Verlag, May 2001.

[32] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4): 512–537, July 2002.

[33] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, 1992.

[34] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[35] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, September 1996.

[36] François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170 (2):153–183, November 2001.

[37] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ACM International Conference on Functional Programming (ICFP)*, pages 193–204, September 2001.

[38] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 242–252, July 1996.

[39] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003.

[40] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[41] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 136–143, 1980.

[42] Joseph J. Hallett and Assaf J. Kfoury. Programming examples needing polymorphic recursion. Technical Report BUCS-TR-2004-004, Department of Computer Science, Boston University, January 2004.

[43] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer Verlag, April 1984.

[44] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[45] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[46] Michael I. Schwartzbach. Polymorphic type inference. Technical Report BRICS-LS-95-3, BRICS, June 1995.

[47] François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

[48] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

[49] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer Verlag, April 1994.

[50] Mark P. Jones. Using parameterized signatures to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 1996.

[51] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 54–67, January 1996.

[52] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. Manuscript, July 2005.

[53] Didier Rémy. Simple, partial type inference for system F based on type containment. In *ACM International Conference on Functional Programming (ICFP)*, September 2005.

[54] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Information and Computation*, 155(1):134–169, 1999.

[55] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system F. In *ACM International Conference on Functional Programming (ICFP)*, pages 27–38, August 2003.

[56] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: type inference for higher-rank types and impredicativity. Manuscript, April 2005.

[57] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 224–235, January 2003.

[58] Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[59] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Manuscript, July 2004.

[60] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. Submitted, July 2005.

[61] Martin Sulzmann, Jeremy Wazny, and Peter J. Stuckey. A framework for extended algebraic data types. Manuscript, July 2005.