Putting Type Annotations to Work

Martin Odersky
Department of Computer Science
University of Karlsruhe
76128 Karlsruhe, Germany
odersky@ira.uka.de

Konstantin Läufer
Department of Mathematical Sciences
Loyola University Chicago
Chicago, Illinois 60626, USA
laufer@math.luc.edu

Abstract

We study an extension of the Hindley/Milner system with explicit type scheme annotations and type declarations. The system can express polymorphic function arguments, user-defined data types with abstract components, and structure types with polymorphic fields. More generally, all programs of the polymorphic lambda calculus can be encoded by a translation between typing derivations. We show that type reconstruction in this system can be reduced to the decidable problem of first-order unification under a mixed prefix.

1 Introduction

Two of the most important cornerstones of type theory for programming languages are the Hindley/Milner system and the second-order polymorphic λ-calculus. This paper tries to explore some of the design space between them.

The Hindley/Milner system [Mil78] extends the simply-typed λ-calculus with polymorphic let-bound identifiers. It thus adds considerable expressive power yet retains the property that no type annotations in programs are needed, since most general types can be inferred [DM82]. This property has made the Hindley/Milner system very appealing as a basis of type systems for programming languages.

By contrast, the second-order polymorphic λ-calculus F_2 [Gir71, Rey74] allows polymorphic types everywhere, but requires explicit annotations of both argument types and type instantiations. The general problem of typechecking without type annotations is undecidable [Wel94] but there have been several approaches towards type reconstruction where some type information is given. These generally fall into two categories. Curry-style reconstruction fills in polymorphic abstractions and applications together with type annotations. This style of reconstruction is complicated by the lack of principal types in F_2. The proposed schemes all have rather complex inference rules with cumbersome conversions between declared and inferred types [McC84, OG89]. By contrast, Church-style reconstruction requires the position of type abstractions and applications to be indicated in the original source. This style of reconstruction (also called partial type reconstruction [Boe89]) was shown to be reducible to higher-order unification [Pfe88]. Even though Church-style reconstruction is thus undecidable in general this result opens up the possibility for semi-decision procedures that work well in practice. On the other hand, the position of a polymorphic application has to be indicated explicitly in the source, which leads to a rather unfamiliar coding style, at least for programmers used to the Hindley/Milner system.

Recently there have been several approaches towards extending the Hindley/Milner system with some form of embedded quantifiers without going all the way to the polymorphic λ-calculus. For instance, Launchbury and Peyton Jones have presented an elegant type system for syntactic control of interference [LPar] that uses second-order universal quantification. Perry [Per90] and Läufer and Odersky [LO94] have studied existential quantification in algebraic datatypes, which yields a Hindley/Milner style version of Mitchell and Plotkin's abstract types [MP88]. This style of existential quantification has been implemented in compilers for Hope [Per90], Haskell [Aug94] and CAML [MP93]. Rémy [Rém94] has extended Läufer and Odersky's system with universal quantification in datatypes, so that objects with polymorphic methods can be expressed. Jones [Jon95] has investigated record types with polymorphic elements as a way to capture essential aspects of module systems. A proposal along these lines has

Appeared in Proc. 23rd ACM Symposium on Principles of Programming Languages, January 1996.

been accepted for inclusion in Haskell 1.3.

It seems that a combination of all of the above systems, while feasible, would be rather unwieldy. Fortunately, it turns out that it is good enough to consider as a generalization a far simpler type system that captures the extensions' commonalities and expresses their differences via encodings. The extensions all have in common that some form of explicit type information is required. For instance, Läufer and Odersky's and Rémy's systems restrict existential quantification to the components of explicitly declared datatypes, while Jones restricts universal quantification to fields of explicitly declared record types.

Here we study a type system that allows (but does not require) explicit type scheme annotations for function arguments. The idea is that a formal function parameter is polymorphic only if annotated with a type scheme; otherwise the parameter is monomorphic, i.e. it has a type, not a type scheme. As an important special case we admit a rudimentary form of user-defined data type declaration that introduces a value constructor with a single, possibly polymorphic argument. Finally, we also allow type scheme annotations for expressions.

Note that this is roughly the kind of type annotations that most programming languages offer or require. The crucial extension of this paper is that annotations and declarations can refer to polymorphic type schemes instead of just types. The ramifications of this simple idea are quite substantial.

- We can express polymorphic function arguments by annotating the argument with a type scheme.
- We can express data types and record types by their usual Church encodings in a type-correct way.
- By slightly modifying these Church encodings, we can also express existentially or universally quantified component types of records and data types, thereby subsuming the type systems of Perry, Läufer and Odersky, Rémy, and Jones. The encodings give us principal type properties and type inference algorithms for these systems "for free".
- Unlike the situation in the simply typed λ-calculus [Mor68] or ML [Mil78], it is no longer possible to reduce type inference to a simple Herbrand unification problem. We need to consider instead the problem of finding a most general substitution that makes one type scheme an instance of another. We show here that this problem is reducible to the problem of first-order unification under a mixed prefix [Mil92], which is decidable. Decidability holds because we still admit only types and not type schemes in the range of substitutions — otherwise the problem would be equivalent to semi-unification, which is undecidable [KTU89].
- Unlike the situation in F_2, we still maintain a stratification between types and type schemes. A universally quantified variable can be instantiated only to types, never to type schemes. We get back the full power of F_2 in an indirect way, by allowing type schemes as components of explicitly declared data types. We show that we can encode all of F_2 by providing type declarations for all polymorphic types in a given F_2 program. This shows that our typing discipline provides essentially the same capabilities as F_2, even though the encoding in F_2 does not support a formal comparison of expressive power in the sense of Felleisen [Fel90] since it fails to be compositional.

Our typing discipline is a conservative extension of the Hindley/Milner system. Every typable program in that system continues to be typable. This holds also if type annotations in the style of ML or Haskell are added to Hindley/Milner. We were able to show principal type properties and soundness and completeness of type inference fully analogous to the ones stated by Damas and Milner [DM82]. Since the engineering issues of ML-like programming languages and type checkers are by now well understood, we believe that this makes our system promising as a practical kernel language on which type-systematic extensions of ML or Haskell can be based.

The rest of this paper is organized as follows. Section 2 presents our type system. Section 3 shows how previous polymorphic extensions of ML can be embedded in it. Section 4 discusses an encoding of the polymorphic λ-calculus. Section 5 states the most general instantiation problem and presents an algorithm to solve it. Section 6 presents a type inference algorithm. Section 7 concludes.

2 The Type System

Figure 1 presents the abstract syntax of our kernel language, $Exp{:}\sigma$. As in the Hindley/Milner system, we distinguish between types, which cannot contain quantification over type variables, and type schemes, which can. Compared to Milner's language Exp there are two extensions that can be considered independently, but that are most useful in combination. One extension considers type annotations for formal arguments and expressions; the other considers type declarations.

Type Scheme Annotations

Type scheme annotations can be applied to formal arguments in λ-abstractions $\lambda x{:}\,\sigma.e$ and to expressions $e{:}\,\sigma$. Annotations with types are common in programming languages that build on the Hindley/Milner system. For instance,

Variables	x, y, z			
Type Constructors	T			
Expressions	e	$=$	$x \mid \lambda x.e \mid e\, e' \mid \mathsf{let}\ x = e\ \mathsf{in}\ e'$	*Exp* terms
		$\mid$	$\lambda x{:}\sigma.e \mid e : \sigma$	annotated terms
		$\mid$	T	type injection
		$\mid$	T^{-1}	type projection
		$\mid$	$\mathsf{newtype}\ T\ \alpha_1\ \ldots\ \alpha_n = \sigma\ \mathsf{in}\ e$	type declaration
Type variables	α, β, γ			
Types	τ	$=$	$\alpha \mid \tau_1 \to \tau_2 \mid T\ \tau_1\ \ldots\ \tau_n$	
Type schemes	σ	$=$	$\tau \mid \sigma_1 \to \sigma_2 \mid \forall \alpha.\sigma$	

Figure 1: Abstract syntax.

$$(\tau)\quad \vdash \tau \leq \tau \qquad\qquad \frac{\vdash \sigma_1' \leq \sigma_1 \qquad \vdash \sigma_2 \leq \sigma_2'}{\vdash \sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2'}\ (\to)$$

$$(\forall \leq)\quad \frac{\vdash [\tau/\alpha]\sigma \leq \sigma'}{\vdash \forall \alpha.\sigma \leq \sigma'} \qquad\qquad \frac{\vdash \sigma \leq \sigma'}{\vdash \sigma \leq \forall \alpha.\sigma'}\ (\alpha \notin \mathrm{ftv}(\sigma)) \quad (\leq \forall)$$

Figure 2: Instance rules for type schemes.

$$(\mathrm{Taut})\quad \begin{array}{c} \Gamma, x : \sigma \vdash x : \sigma \\ \Gamma, T : \sigma \vdash T : \sigma \end{array} \qquad\qquad \frac{\Gamma \vdash T : \sigma \to T\ \overline{\tau}}{\Gamma \vdash T^{-1} : T\ \overline{\tau} \to \sigma}\quad (\mathrm{Proj})$$

$$(\mathrm{Gen})\quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\sigma}\ (\alpha \notin \mathrm{ftv}(\Gamma)) \qquad\qquad \frac{\Gamma \vdash e : \sigma \qquad \vdash \sigma \leq \sigma'}{\Gamma \vdash e : \sigma'}\quad (\mathrm{Sub})$$

$$(\mathrm{Lambda})\quad \frac{\Gamma_x.x{:}\tau \vdash e : \sigma}{\Gamma_x \vdash \lambda x.e : \tau \to \sigma} \qquad\qquad \frac{\Gamma \vdash e : \sigma \to \sigma' \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e\, e' : \sigma'}\quad (\mathrm{Apply})$$

$$(\mathrm{TypedLambda})\quad \frac{\Gamma_x.x{:}\sigma \vdash e : \sigma'}{\Gamma_x \vdash \lambda x{:}\sigma.e : \sigma \to \sigma'} \qquad\qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash (e : \sigma) : \sigma}\quad (\mathrm{Typed})$$

$$(\mathrm{Let})\quad \frac{\Gamma_x \vdash e : \sigma \qquad \Gamma_x.x{:}\sigma \vdash e' : \sigma'}{\Gamma_x \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \sigma'} \qquad\qquad \frac{\Gamma_T, T{:}\forall\overline{\alpha}.\sigma \to T\ \overline{\alpha} \vdash e : \sigma'}{\Gamma_T \vdash \mathsf{newtype}\ T\ \overline{\alpha} = \sigma\ \mathsf{in}\ e : \sigma'}\quad (\mathrm{Newtype})$$

Figure 3: Typing rules.

$$\mathsf{map} = \lambda \mathsf{f}\colon \mathsf{a} \to \mathsf{b}.\ \lambda \mathsf{xs}\colon [\mathsf{a}].\ \mathsf{case\ xs\ of} \ldots$$

declares the argument types of function map in terms of two type variables a and b. By generalizing over these type variables we then obtain the usual polymorphic type scheme for map:

$$\mathsf{map}\colon \forall \mathsf{a}.\forall \mathsf{b}.(\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}].$$

What is new here is the ability to annotate with type schemes instead of types. For instance, it is now possible to write

$$\mathsf{f}\ (\mathsf{g}\colon \forall \mathsf{c}.[\mathsf{c}] \to \mathsf{Int}) = \mathsf{g}\ [\text{"hello"}] + \mathsf{g}\ [1{,}2].$$

As a consequence, a type scheme may now form part of a larger type scheme. For instance, f's most general type scheme would be

$$(\forall \mathsf{c}.[\mathsf{c}] \to \mathsf{Int}) \to \mathsf{Int}.$$

We therefore have to give up Hindley/Milner's restriction that quantifiers may occur only at the outermost level of a type scheme and have to admit type schemes such as $\sigma_1 \to \sigma_2$.

An immediate consequence is that we have to refine the "generic instance" relation [DM82] if we want to get principal types for the system with annotations. Consider the function $\lambda \mathsf{x}{:}\mathsf{Int}.[\,]$. Two derivable type schemes for this function are

$$\forall \mathsf{a}.\mathsf{Int} \to [\mathsf{a}] \quad \text{and} \quad \mathsf{Int} \to \forall \mathsf{a}.[\mathsf{a}].$$

None of these type schemes is a generic instance of the other. Furthermore, there is no third type scheme that has both of these type schemes as generic instances. But using the relation $(\leq)$ defined in Figure 2, we get $\mathsf{Int} \to \forall \mathsf{a}.[\mathsf{a}]$ as the more general of both type schemes. The relation $(\leq)$ implements a form of subtyping for type schemes. Rule $(\forall \leq)$ together with subsumption is equivalent to the quantifier elimination rule of the Hindley/Milner system. Rule $(\leq \forall)$ allows us to requantify a type scheme. Functions over type schemes are handled by the standard contravariance rule $(\to)$. As usual, we identify type schemes that are instances of each other.

The relation $(\leq)$ is a subrelation of Mitchell's containment relation [Mit90] and hence is validated by all type inference models. For type schemes that have quantifiers only at the outermost level, $(\leq)$ is the inverse of the "generic instance" relation given by Damas and Milner [DM82]. We changed the direction of $(\leq)$ sign to stay in line with Mitchell's containment relation which corresponds to the semantic intuition of subtyping as subsetting.

$(\leq)$ has the following useful properties.

Proposition 2.1 Let σ and σ' be type schemes and let θ be a substitution. If $\vdash\ \sigma \leq \sigma'$ then $\vdash\ \theta\sigma \leq \theta\sigma'$.

Proposition 2.2 $(\leq)$ is transitive.

Proof Sketch: Assume that $\sigma_1 \leq \sigma_2$ and $\sigma_2 \leq \sigma_3$. We show $\sigma_1 \leq \sigma_3$ by an induction on the sum of the depths of the proof trees for $\sigma_1 \leq \sigma_2$ and $\sigma_2 \leq \sigma_3$. Proposition 2.1 is used for the case where the last rule in the proof of $\sigma_1 \leq \sigma_2$ is an application of rule $(\leq \forall)$. □

The typing rules, given in Figure 3, largely follow the Hindley/Milner system. The two main differences are both motivated by the possible occurrence of quantifiers at all levels in a type scheme. First, it is necessary to consider type schemes instead of types in the conclusion of each typing rule, since type schemes cannot always be reconstructed using generalization at the outermost level. Second, Hindley/Milner's elimination rule for outermost quantifiers is replaced by a more general subsumption rule, which takes into account the instance relation $(\leq)$ on type schemes.

Type annotations alone are sufficient for expressing polymorphic function arguments. But one shortcoming of this system remains: the resulting second-order polymorphic functions cannot be arguments of polymorphic functions themselves, since this would require an instantiation of a type variable to a type scheme. For instance, the following code would not be type-correct:

$$\mathsf{map\ f\ [length,\ const\ 0]}.$$

The problem is that the type variable a in map's type cannot be instantiated to the type scheme $\forall \mathsf{c}.[\mathsf{c}]{\to}\mathsf{Int}$. We circumvent this problem by providing a way to "package" a type scheme in an explicitly declared data type.

Type Declarations

A type declaration $\mathsf{newtype}\ T\ \alpha_1\ \ldots\ \alpha_n = \sigma\ \mathsf{in}\ e$ corresponds to a simple form of an algebraic data type declaration with a single unary constructor. Each type $T\ \tau_1\ \ldots\ \tau_n$ thus introduced is different from $[\tau_i/\alpha_i]\sigma$. The type constructor T may be used anywhere, including in the type scheme σ. We require that every type constructor is declared at most once in a program (this is not enforced by the typing rules). We often use the shorthand $\overline{\alpha}$ or $\overline{\tau}$ for vectors of type variables or types.

A similar declaration in Haskell would be

$$\mathsf{data\ T\ a_1\ \ldots\ a_n = T\ elemtype}.$$

We generalize Haskell in that elemtype may be an arbitrary type scheme instead of a type.

The Haskell syntax above makes explicit our convention that T doubles up as an injection function that maps values of the component type to values of type $T\ \overline{\tau}$. For every new type constructor T there is also a projection function T^{-1}, which is an inverse of the injection T. By contrast, projection in Haskell is implicit in the meaning of case expressions. Instead of Haskell's

$$\mathsf{case\ t\ of\ T\ x} \Rightarrow \mathsf{e}$$

we would write

```
let x = T⁻¹ t in e.
```

With the help of type declarations we can now code our problematic example as follows.

```
newtype ListFun = ∀c.[c] → Int
in let f g = let g' = ListFun⁻¹ g
             in g' ["hello"] + g' [1,2]
in map f [ListFun length, ListFun (const 0)].
```

But much more is possible. For once, newtype declarations are sufficient to express data types with general products and sums by their usual Church encodings, combined with explicit injection and projection operations. For instance, the type of pairs with a constructor mkpair and selectors fst and snd would be coded as follows.

```
newtype Pair a b = ∀c. (a → b → c) → c
in let mkpair x y = Pair λk.k x y
in let fst p      = Pair⁻¹ p λx.λy.x
in let snd p      = Pair⁻¹ p λx.λy.y
in    ...
```

Note that the Pair type expands into a type scheme, not a type. Therefore, we could not apply the same technique in languages like ML or Haskell, which admit only types on the right hand sides of data type declarations.

A second example encodes the list type, using the List type constructor recursively.

```
newtype List a = ∀b.b → (a → List a → b) → b
in let nil       = List λn.λc.n
in let cons x xs = List λn.λc.c x xs
in    ...
```

A case expression like

```
case xs of { nil ⇒ e₁ | cons y ys ⇒ e₂ }
```

would then be coded as

```
List⁻¹ xs e₁ λy.λys.e₂.
```

Of course, in an actual programming language we would assume that product and sum types are definable directly, without the need for Church encodings. The existence of the encodings ensures in this case that the additional language constructs require no essential additions to the type system — after all, we could typecheck by encoding first and then using our kernel language. In the next section, we apply this program to some polymorphic extensions of the Hindley/Milner system.

3 Extensions

In this section, we show how some previous extensions of Hindley/Milner with embedded quantifiers can be expressed in our system. In particular, we deal with Läufer and Odersky's version of abstract types [LO94] and with Jones' version of polymorphic structures [Jon95]. A system equivalent in expressiveness to Rémy's [Rém94] can then be obtained by combining both extensions.

Abstract Types

We consider a set of global *data type declarations*

$$\mathsf{data}\ D\ \overline{\alpha} \quad = \quad k_1\tau_1 \ \mid \ \ldots \ \mid \ k_n\tau_n \tag{1}$$

Here D is a data type constructor, and $k_1, \ldots, k_n$ are value constructors. Conceptually, a data type constructor is a special instance of a type constructor T, whereas value constructors k form a separate alphabet. As in [LO94] we adopt the convention that any type variables in one of the τ_i that do not appear in $\overline{\alpha}$ are existentially quantified. By contrast, in ML or Haskell such type variables would be disallowed.

Example 3.1 The following declares a type of lists with heterogeneous elements. Each element consists of some value and a function that maps this value to an integer key. The type of the value may vary from element to element.

```
data KeyList = KNil
             | KCons ((a, a → Int), KeyList)
```

A function that finds the maximal key can then be written as follows:

```
maxkey xs = case xs of
  { KNil ⇒ minint
  | KCons ((y, f), ys) ⇒ f y ‘max‘ maxkey ys }
```

Slightly modifying our treatment of lists in the last section, this program is translated into *Exp*: σ as follows.

```
newtype KeyList =
    ∀b.b→ (∀a.((a, a→Int), KeyList) → b) → b
in let KNil = KeyList
    λn.λc: ∀a.((a, a→Int), KeyList) → b. n
in let KCons x xs = KeyList
    λn.λc: ∀a.((a, a→Int), KeyList) → b. c (x, xs)
in let maxkey xs =
    KeyList⁻¹xs
           minint
           λ((y, f), ys).f y ‘max‘ maxkey ys
```

Note that the implied existential quantifier for the type variable a in the definition of KeyList turns into a second rank universal quantifier in KeyList's translation.

For the general case we augment our kernel language *Exp*: σ with value constructors and case expressions.

$$\begin{array}{lcl} e & ::= & \ldots \\ & | & k \\ & | & \mathsf{case}\ e\ \mathsf{of}\ \{k_1 x_1 \Rightarrow e_1 \mid \ldots \mid k_n x_n \Rightarrow e_n\} \end{array}$$

Let *Exp*: σ+$\exists$ be the term-language thus defined. Given a data type declaration (1), let $\overline{\beta_i} = \mathrm{ftv}(\tau_i) \backslash \overline{\alpha}$ for $i = 1, \ldots, n$. Then the following typing rules are equivalent to the treatment in [LO94].

$$\text{(AbsI)} \quad \Gamma \vdash k_i : \forall\overline{\alpha}.\forall\overline{\beta_i}.\tau_i \rightarrow D\ \overline{\alpha} \qquad (i = 1, \ldots, n)$$

$$\text{(AbsE)} \quad \frac{\begin{array}{c} \Gamma \vdash e : D\ \overline{\tau''} \\ \Gamma \vdash k_i : \forall\overline{\beta_i}.\tau_i' \rightarrow D\ \overline{\tau}'' \\ \Gamma \vdash \lambda x_i.e_i : \forall\overline{\beta_i}.\tau_i' \rightarrow \tau \\ \mathrm{ftv}(\tau) \cap \overline{\beta_i} = \emptyset \end{array} \quad (i = 1, \ldots, n)}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{k_1 x_1 \Rightarrow e_1 \mid \ldots \mid k_n x_n \Rightarrow e_n\} : \tau}$$

Let $\vdash^{\exists}$ be the relation that results from adding these rules to those in Figure 3. We now give an encoding $(^{\exists})$ of *Exp*: $\sigma + \exists$ in *Exp*: σ that preserves typeability. For the constructors and case expressions that correspond to a data type declaration (1), we define:

$$\begin{array}{lcl} k_i^{\exists} & = & \hat{k}_i \qquad (i = 1, \ldots, n) \\ & & \text{where each } \hat{k}_i \text{ is a new variable,} \\ \multicolumn{3}{l}{\mathsf{case}\ e\ \mathsf{of}\ \{k_1 x_1 \Rightarrow e_1 \mid \ldots \mid k_n x_n \Rightarrow e_n\}^{\exists}} \\ & = & D^{-1}\ e^{\exists}\ (\lambda x_1.e_1^{\exists})\ \ldots\ (\lambda x_n.e_n^{\exists}). \end{array}$$

We extend $(^{\exists})$ homomorphically to all other expressions. Finally, we add for every data type declaration of form (1) the global declarations below, where γ is a fresh type variable.

$$\begin{array}{l} \mathsf{newtype}\ D\ \overline{\alpha}\ =\ \forall\gamma.(\forall\overline{\beta_1}.\tau_1 \rightarrow \gamma) \rightarrow \ldots \rightarrow \\ \qquad\qquad\qquad (\forall\overline{\beta_n}.\tau_n \rightarrow \gamma) \rightarrow \gamma \\ \mathsf{in\ let}\ \hat{k}_i = \lambda x.D\ (\lambda y_1 : \forall\overline{\beta_1}.\tau_1 \rightarrow \gamma.\ \ldots\ . \\ \qquad\qquad\qquad \lambda y_n : \forall\overline{\beta_n}.\tau_n \rightarrow \gamma.y_i x) \\ \qquad\qquad (i = 1, \ldots, n) \end{array}$$

Then we have:

Proposition 3.2 For all typotheses Γ, terms e and type schemes σ in *Exp*: $\sigma + \exists$,

$$\Gamma \vdash^{\exists} e : \sigma \ \Leftrightarrow\ \Gamma \vdash e^{\exists} : \sigma.$$

Proof: An easy comparison of typing derivations. □

Polymorphic Structures

An analogous treatment lets us encode structures with polymorphic fields in *Exp*: σ. Consider a set of global *structure declarations*

$$\mathsf{struct}\ S\ \overline{\alpha}\ =\ \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad (2)$$

Here, S is a type constructor, and $l_1, \ldots, l_n$ are field labels. To keep the treatment simple, we assume that every label l occurs in at most one structure type declaration; hence structures do not have scopes of their own. A more flexible scheme, in which a label could be part of several structures, would be obtained by adding overloading to our type system [Jon92, OWW95]. In symmetry with our treatment of data types, we now adopt the convention that any type variables in one of the τ_i that do not appear in $\overline{\alpha}$ are *universally* quantified.

Example 3.3 We define a type for set objects that contain as a field a polymorphic map function.

```
struct Set a = { elem  : a → Bool,
                 union : Set a → Set a,
                 map   : (a → b) → Set b }
```

Note that the type variable b in map's signature does not appear on the left-hand side of the definition, and hence is considered to be universally quantified. This structure declaration could be expressed in *Exp*: σ as follows.

```
newtype Set a = ∀c.∀b.((a → Bool) →
                       (Set a → Set a) →
                       ((a → b) → Set b) → c)
                       → c.
```

More generally, let the term language *Exp*: $\sigma + \forall$ be obtained by adding structure expressions and selector functions to *Exp*: σ.

$$e \ ::= \ \ldots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid \#l$$

Given a structure type declaration (2), we add the following typing rules, where $\overline{\beta_i} = \mathrm{ftv}(\tau_i) \backslash \overline{\alpha}$ ($i = 1, \ldots, n$).

$$\text{(PolyI)} \quad \Gamma \vdash \#l_i : \forall\overline{\alpha}.S\ \overline{\alpha} \rightarrow \forall\overline{\beta_i}.\tau_i \qquad (i = 1, \ldots, n)$$

$$\text{(PolyE)} \quad \frac{\begin{array}{ll} \Gamma \vdash e_i : \forall\overline{\beta_i}.\tau_i' & (i = 1, \ldots, n) \\ \Gamma \vdash \#l_i : S\ \overline{\tau''} \rightarrow \forall\overline{\beta_i}.\tau_i' & (i = 1, \ldots, n) \end{array}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : S\ \overline{\tau''}}$$

Let $\vdash^{\forall}$ be the relation that results from adding these rules to those in Figure 3. To encode *Exp*: σ+$\forall$ in *Exp*: σ, define for every data type of form (2):

$$\begin{array}{lcl} (\#l_i)^{\forall} & = & \hat{l}_i \\ & & \text{where each } \hat{l}_i \text{ is a new variable,} \\ \multicolumn{3}{l}{\{l_1 = e_1, \ldots, l_n = e_n\}^{\forall}} \\ & = & S\ (\lambda k.k\ e_1^{\forall}\ \ldots\ e_n^{\forall}) \end{array}$$

Extend $(^{\forall})$ homomorphically to all other expressions and add for every declaration (2) the global declarations

$$\begin{array}{l} \mathsf{newtype}\ S\ \overline{\alpha} = \forall\gamma.\forall\overline{\beta_1}\ldots\forall\overline{\beta_n}.(\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \gamma) \rightarrow \gamma \\ \mathsf{in\ let}\ \hat{l}_i = \lambda x.S^{-1}\ x\ (\lambda y_1.\ \ldots\ \lambda y_n.y_i) \end{array}$$

Then the following proposition is shown by a comparison of typing derivations.

Proposition 3.4 For all typotheses Γ, terms e and type schemes σ in *Exp*: $\sigma+\forall$,

$$\Gamma \vdash^{\forall} e : \sigma \;\Leftrightarrow\; \Gamma \vdash e^{\forall} : \sigma.$$

Discussion. One shortcoming of the presented encodings is that the component types of data types and structures can have only one layer of quantifiers. The encodings share this property with the original proposals of Läufer and Odersky and Jones, but not with Rémy's system. A more powerful type system would admit arbitrary type schemes for the components. This would present no problems for data types, hence Rémy's system could be expressed by a straightforward combination of our encodings for data types and structures. But an analogous generalization would not work for structure types, since there the result of a selection is captured in a type variable, and therefore needs to have a type without quantifiers. (Of course, it is possible to re-quantify at the outermost level after the selection). Data types suffer a different shortcoming — albeit for a similar reason — in that each branch in a case-expression needs to have a type without quantifiers.

It is possible to lift both restrictions by considering product and sum types in the kernel language, with σ ranging over

$$\sigma \;\; ::= \;\; \tau \mid \sigma \to \sigma \mid \sigma \times \sigma \mid \sigma + \sigma \mid \forall\alpha.\sigma$$

Alternatively, one can also work around the restrictions by inventing intermediate data and structure types for each level of quantification.

4 Encoding F_2

In this section we present a translation of the second order polymorphic λ-calculus F_2 into our typing discipline. F_2 is given by the typing rules below.

$$\text{(Taut)} \quad \frac{\Gamma, x:\sigma \;\vdash^F\; x:\sigma}{}$$

$$(\to\text{I}) \quad \frac{\Gamma, x:\sigma \;\vdash^F\; M:\sigma'}{\Gamma \;\vdash^F\; \lambda x{:}\sigma.M : \sigma \to \sigma'}$$

$$(\to\text{E}) \quad \frac{\Gamma \;\vdash^F\; M:\sigma' \to \sigma \qquad \Gamma \;\vdash^F\; N:\sigma'}{\Gamma \;\vdash^F\; M\;N:\sigma}$$

$$(\forall\text{I}) \quad \frac{\Gamma \;\vdash^F\; M:\sigma}{\Gamma \;\vdash^F\; \Lambda\alpha.M : \forall\alpha.\sigma} \; (\alpha \notin \text{ftv}(\Gamma))$$

$$(\forall\text{E}) \quad \frac{\Gamma \;\vdash^F\; M:\forall\alpha.\sigma}{\Gamma \;\vdash^F\; M[\sigma'] : [\sigma'/\alpha]\sigma}$$

The crucial idea of the translation of F_2 into our typing discipline is that a polymorphic F_2 type $\forall\alpha.\sigma$ is mapped to a data type $T_\psi\;\tau_1\;\ldots\;\tau_n$ where the type constructor T_ψ is indexed by an n-ary type abstraction ψ and $(\psi, \tau_1, \ldots, \tau_n)$ is the α-lifting of σ's translation:

Definition. The α-lifting of a type τ consists of an n-ary type abstraction ψ and types $\tau_1, \ldots, \tau_n$ such that $\psi\;\tau_1\;\ldots\;\tau_n = \tau$ and $\tau_1, \ldots, \tau_n$ are maximal subterms of τ that do not contain α. We write in this case

$$\mathit{lift}_\alpha\;\tau = (\psi, \tau_1, \ldots, \tau_n).$$

We arrange such that for every translated type $(\forall\alpha.\sigma)^*$ there is in the translation a global type declaration

$$\textsf{newtype}\; T_\psi\;\beta_1\;\ldots\;\beta_n \;=\; \forall\alpha.\alpha \to \psi\;\beta_1\;\ldots\;\beta_n.$$

where $\mathit{lift}_\alpha\;\sigma^* = (\psi, \ldots)$.

For simplicity, we avoid variable renamings by assuming that all type variables in the F_2 source are mutually distinct. The encoding of F_2 types is then given by:

$$\begin{array}{lcl} \alpha^* & = & \alpha \\ (\sigma_1 \to \sigma_2)^* & = & \sigma_1^* \to \sigma_2^* \\ (\forall\alpha.\sigma)^* & = & T_\psi\;\tau_1\;\ldots\;\tau_n \\ & & \text{where } \mathit{lift}_\alpha\;\sigma^* = (\psi, \tau_1, \ldots, \tau_n). \end{array}$$

This encoding is stable under substitutions, as is shown in the following lemma.

Lemma 4.1 For all F_2 types σ_1, σ_2, type variables α,

$$([\sigma_1/\alpha]\sigma_2)^* \;=\; [\sigma_1^*/\alpha]\sigma_2^*.$$

Proof: By induction on the structure of σ_2. The case $\sigma_2 = \forall\alpha.\sigma_2'$ relies on the observation that if

$$\mathit{lift}_\alpha\;\tau \;=\; (\psi, \tau_1, \ldots, \tau_n)$$

then

$$\mathit{lift}_\alpha\;\theta\tau \;=\; (\psi, \theta\tau_1, \ldots, \theta\tau_n),$$

for any substitution θ that does not involve α. □

We extend $(^*)$ pointwise to type environments, defining

$$\{x_i : \sigma_i\}^* \;=\; \{x_i : \sigma_i^*\}.$$

We now come to the encoding of F_2 terms. Since this encoding depends on both a term and its type, which in turn depends on a type environment, we formulate $(^*)$ as a mapping from F_2's typing rules for type judgments $\Gamma \;\vdash^F\; M : \sigma$ to a different set of typing rules for type judgments $\Gamma^* \;\vdash^*\; M^* : \sigma^*$. We will then show in a second step that each $\vdash^*$ rule is valid as a $\vdash$ derivation in an augmented environment.

Rules (Taut), ($\to$I) and ($\to$E) are mapped by (*) to identical rules with $\vdash^*$ instead of $\vdash^F$. For the remaining two rules, we define:

$$(\forall \mathrm{I})^* \quad = \quad \frac{\begin{array}{c} \mathit{lift}_\alpha\ \sigma^* = (\psi, \tau_1, \ldots, \tau_n) \\ \alpha \notin \mathrm{ftv}(\Gamma^*) \\ \Gamma^* \vdash^* N : \sigma^* \end{array}}{\Gamma^* \vdash^* T_\psi\ (\lambda x_\alpha : \alpha.N) : T_\psi\ \tau_1\ \ldots\ \tau_n}$$

$$(\forall \mathrm{E})^* \quad = \quad \frac{\begin{array}{c} \mathit{lift}_\alpha\ \sigma^* = (\psi, \tau_1, \ldots, \tau_n) \\ \Gamma^* \vdash^* N : T_\psi\ \tau_1\ \ldots\ \tau_n \end{array}}{\Gamma^* \vdash^* T_\psi^{-1}\ N\ \langle \sigma' \rangle : [(\sigma')^*/\alpha]\sigma^*}$$

In rule ($\forall$E), the type argument $[\sigma]$ is translated to a *representative* $\langle\sigma\rangle$, which is a term with type σ^*. The mapping $\langle\cdot\rangle$ from F_2 types to representatives is defined below.

$$\begin{array}{lcl} \langle \alpha \rangle & = & x_\alpha \\ \langle \sigma_1 \to \sigma_2 \rangle & = & \lambda x : \sigma_1^* . \langle \sigma_2 \rangle \\ \langle \forall \alpha.\sigma \rangle & = & T_\psi\ (\lambda x_\alpha : \alpha . \langle \sigma \rangle) \\ & & \text{where } \mathit{lift}_\alpha\ \sigma^* = (\psi, \ldots) \end{array}$$

Definition. Given a type scheme σ, let

$$\Delta_\sigma \quad = \quad \{x_\alpha : \alpha \mid \alpha \in \mathrm{ftv}(\sigma)\}.$$

Analogously for an F_2 term M, let

$$\Delta_M \quad = \quad \{x_\alpha : \alpha \mid \alpha \in \mathrm{ftv}(M)\}.$$

Finally, for an F_2 derivation $\mathcal{D}$ with conclusion $\Gamma \vdash M : \sigma$, let $\mathcal{S}_\mathcal{D}$ be the set of all polymorphic types of form $\forall\alpha.\sigma$ occurring in the environment or type part of a typing judgment in $\mathcal{D}$. Then the type environment $\Delta_\mathcal{D}$ is given by

$$\begin{array}{rcl} \Delta_\mathcal{D} & = & \Delta_M \cup \{\, T_\psi : \forall\overline{\beta}.(\forall\alpha.\alpha \to \psi\ \overline{\beta}) \to T_\psi\ \overline{\beta} \\ & & \quad \mid \exists(\forall\alpha.\sigma) \in \mathcal{S}_\mathcal{D}.\mathit{lift}_\alpha\ \sigma = (\psi, \ldots)\}. \end{array}$$

Informally, $\Delta_\mathcal{D}$ contains a binding $x_\alpha : \alpha$ for every free variable α in σ, and it contains for every polymorphic type in the derivation $\mathcal{D}$ a corresponding type constructor T_ψ. $\Delta_\mathcal{D}$ can be produced by a *Exp*:σ context which consists of a series of type declarations of the form

$$\mathsf{newtype}\ T_\psi\ \beta_1\ \ldots\ \beta_n \ = \ \forall\alpha.\alpha \to \psi\ \beta_1\ \ldots\ \beta_n,$$

followed by a series of λ-abstractions of the form $\lambda x_\alpha : \alpha$.

Lemma 4.2 $\Delta_\sigma \vdash \langle\sigma\rangle : \sigma^*$.

Proof: Directly from the definition of $\langle\cdot\rangle$. □

The following proposition is shown by a straightforward induction on $\vdash^F$ derivations.

Proposition 4.3 Let $\mathcal{D}$ be a typing derivation in F_2 with conclusion $\Gamma \vdash^F M : \sigma$. Then there exists a unique term M^* and a $\vdash^*$-Proof with structure $\mathcal{D}^*$ which concludes with

$$\Gamma^*, \Delta_M \vdash^* M^* : \sigma^*.$$

It remains to be shown that each $\mathcal{D}^*$ derivation can be completed to a valid *Exp*:σ derivation. To show this, we need a standard property of *Exp*:σ, namely that type derivations are invariant under weakenings and additions of hypotheses. This is stated in the following lemma, which is shown by a straightforward induction on typing derivations.

Lemma 4.4 If $x \notin \mathrm{fv}(e)$ then $\Gamma, x : \sigma' \vdash e : \sigma$ iff $\Gamma \vdash e : \sigma$.

Theorem 4.5 If $\Gamma \vdash^F M : \sigma$ by an F_2 derivation $\mathcal{D}$ then $\Gamma^*, \Delta_\mathcal{D} \vdash M^* : \sigma^*$.

Proof: By an induction on the structure of $\mathcal{D}$. If the last step in the proof is an application of a (Taut) rule, the result follows immediately. If it is one of ($\to$I) or ($\to$E), the result follows by a simple inductive step. Assume now that the proof consists of a derivation $\mathcal{D}'$ of $\Gamma \vdash^F M : \sigma$, followed by an application of rule

$$(\forall\mathrm{I})\ \frac{\Gamma \vdash^F M : \sigma}{\Gamma \vdash^F \Lambda\alpha.M : \forall\alpha.\sigma}\ (\alpha \notin \mathrm{ftv}(\Gamma)).$$

By the induction hypothesis, $\Gamma^*, \Delta_{\mathcal{D}'} \vdash^F M^* : \sigma^*$. Let $\Delta' = \Delta_{\mathcal{D}'} \backslash \{x_\alpha : \alpha\}$. Assume first that $\alpha \in \mathrm{ftv}(M)$. Then $\Delta_{\mathcal{D}'}$ contains a binding $x_\alpha : \alpha$. By rule (Lambda),

$$\Gamma^*, \Delta' \vdash \lambda x_\alpha : \alpha.M^* : \alpha \to \sigma^*. \qquad (3)$$

On the other hand, if $\alpha \notin \mathrm{ftv}(M)$, (3) follows from the induction hypothesis, rule (Lambda), and Lemma 4.4. Then by rule (Gen), since α is free in Γ^*, Δ',

$$\Gamma^*, \Delta' \vdash \lambda x_\alpha : \alpha.M^* : \forall\alpha.\alpha \to \sigma^*. \qquad (4)$$

Furthermore, $\Delta_\mathcal{D}$ contains both Δ' and the binding

$$T_\psi : \forall\overline{\beta}.(\forall\alpha.\alpha \to \psi\ \overline{\beta}) \to T_\psi\ \overline{\beta}. \qquad (5)$$

It follows by rules (Taut), (Sub) that

$$\Gamma^*, \Delta_\mathcal{D} \vdash T_\psi : (\forall\alpha.\alpha \to \psi\ \overline{\tau}) \to T_\psi\ \overline{\tau}. \qquad (6)$$

It also follows from (4) and Lemma 4.4 that

$$\Gamma^*, \Delta_\mathcal{D} \vdash \lambda x_\alpha : \alpha.M^* : \forall\alpha.\alpha \to \sigma^*. \qquad (7)$$

Since $\sigma^* = \psi\ \overline{\tau}$ by assumption, the case then follows from (6), (7) and an application of (App).

Assume finally that the proof $\mathcal{D}$ consists of a derivation $\mathcal{D}'$ of $\Gamma \vdash^F M : \sigma$, followed by an application of rule

$$(\forall\mathrm{E})\ \frac{\Gamma \vdash^F M : \forall\alpha.\sigma}{\Gamma \vdash^F M[\sigma'] : [\sigma'/\alpha]\sigma}.$$

By the induction hypothesis, $\Gamma^*, \Delta_{\mathcal{D}'} \vdash M^* : (\forall\alpha.\sigma)^*$, where $(\forall\alpha.\sigma)^* = T_\psi\ \overline{\tau}$, for some type constructor T_ψ such that $\psi\ \overline{\tau} = \sigma^*$ and $\mathcal{D}$ contains the binding

$$T_\psi \quad : \quad \forall\overline{\beta}.(\forall\alpha.\alpha \to \psi\ \overline{\beta}) \to T_\psi\ \overline{\beta}. \qquad (8)$$

Then by (Taut), (Proj) and (Sub):

$$\Gamma^*, \Delta_{\mathcal{D}'} \quad \vdash \quad T_\psi^{-1} : T_\psi\ \overline{\tau} \to \forall\alpha.(\alpha \to \sigma^*). \qquad (9)$$

By rule (App),

$$\Gamma^*, \Delta_{\mathcal{D}'} \quad \vdash \quad T_\psi^{-1}\ M^* : \forall\alpha.(\alpha \to \sigma^*). \qquad (10)$$

Then by rule (Sub),

$$\Gamma^*, \Delta_{\mathcal{D}'} \quad \vdash \quad T_\psi^{-1}\ M^* : (\sigma')^* \to [(\sigma')^*/\alpha]\sigma^*. \ (11)$$

Since $\Delta_{\mathcal{D}} \supseteq \Delta_{\mathcal{D}'}$ it follows with Lemma 4.4 that

$$\Gamma^*, \Delta_{\mathcal{D}} \quad \vdash \quad T_\psi^{-1}\ M^* : (\sigma')^* \to [(\sigma')^*/\alpha]\sigma^*. \ (12)$$

Furthermore, since $\Delta_{\mathcal{D}} \supseteq \Delta_\sigma$, Lemma 4.2 with Lemma 4.4 implies that

$$\Gamma^*, \Delta_{\mathcal{D}} \quad \vdash \quad \langle\sigma'\rangle : (\sigma')^*. \qquad (13)$$

Then by (12), (13) and rule (App),

$$\Gamma^*, \Delta_{\mathcal{D}} \quad \vdash \quad T_\psi^{-1}\ M^*\ \langle\sigma'\rangle : [(\sigma')^*/\alpha]\sigma^*. \qquad (14)$$

Finally with Lemma 4.1,

$$\Gamma^*, \Delta_{\mathcal{D}} \quad \vdash \quad T_\psi^{-1}\ M^*\ \langle\sigma'\rangle : ([\sigma'/\alpha]\sigma)^*, \qquad (15)$$

which proves the case. □

Example 4.6 Consider the successor function on Church-numerals

$$\hat{n} \quad = \quad \Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f^n\ x,$$

which is given by:

$$\begin{aligned} \mathit{succ} \quad &: \quad (\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha) \to \\ &\qquad\qquad \forall\beta.(\beta \to \beta) \to \beta \to \beta \\ \mathit{succ} \quad &= \quad \lambda m : \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha. \\ &\qquad\qquad \Lambda\beta.\lambda f : \beta \to \beta.\lambda x : \beta. \\ &\qquad\qquad\quad m[\beta]\ f\ (f\ x). \end{aligned}$$

The liftings of *succ*'s argument and result type schemes with respect to their quantified type variables are:

$$\begin{aligned} \mathit{lift}_\alpha\ \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha \quad &= \quad (\alpha \to \alpha) \to \alpha \to \alpha \\ \mathit{lift}_\beta\ \forall\beta.(\beta \to \beta) \to \beta \to \beta \quad &= \quad (\beta \to \beta) \to \beta \to \beta. \end{aligned}$$

We thus need the following global type declarations:

$$\begin{aligned} \mathsf{newtype}\ S \quad &= \quad \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha \to \alpha \\ \mathsf{newtype}\ T \quad &= \quad \forall\beta.\beta \to (\beta \to \beta) \to \beta \to \beta. \end{aligned}$$

Translating the successor function results in

$$\begin{aligned} \mathit{succ}^* \quad = \quad & \lambda m{:}\ S.T(\lambda x_\beta : \beta.\lambda f : \beta \to \beta. \\ & \qquad \lambda x{:}\ \beta.S^{-1}\ m\ x_\beta\ f\ (f\ x)). \end{aligned}$$

Although S and T are identical and a single type declaration would be sufficient, the translation does not provide this simplification.

It might seem that the F_2 translation makes our previous encodings of abstract types and polymorphic structures superfluous, since these can clearly be expressed in F_2. However, unlike these previous encodings, which had only local transformation rules for terms, the translation of F_2 depends on the full typing derivation of an F_2 program. It is therefore not clear how to use the translation for validating typing rules for abstract types and polymorphic structures in *Exp*: σ, as we did in the last section.

5 Finding Most General Instantiators

In this section we study the problem of finding substitutions that make one type scheme an instance of another.

Preliminaries: Substitutions and Unifiers. A (type variable) *substitution* is an idempotent mapping from type variables to types that maps all but a finite number of type variables to themselves. Let $\mathrm{dom}(\theta) = \{\alpha \mid \theta\alpha \neq \alpha\}$. Substitutions are extended homomorphically to mappings on types and type schemes. When applying a substitution θ to a type scheme σ, we assume that the bound variables in σ are disjoint from $\mathrm{dom}(\theta)$. This can always be achieved by renaming bound variables in σ.

Let $\mathbf{1}$ be the identity substitution and let $[\tau/\alpha]$ be the mapping (idempotent or not) that replaces α by τ. Composition of substitutions ϕ and θ is written $\theta \circ \phi$. Let V be a set of type variables. Then $\theta|_V$ is the substitution that equals θ on all type variables in V and that is the identity on all other type variables. Conversely, $\theta\backslash_V$ is the substitution that equals θ except on V, where it is the identity.

Let U be a finite set of type variables. Usually we use U for the universe of type variables that are of interest in the situation at hand. We define $\theta_1 \leq_U^\phi \theta_2$ if $(\phi \circ \theta_1)|_U = \theta_2|_U$. We write $\theta_1 \leq_U \theta_2$ if $\exists\phi.\theta_1 \leq_U^\phi \theta_2$. Note that this makes the "more general" substitution the smaller element in the pre-order $\leq_U$. This choice, which reverses the usual convention in treatments of unification (e.g. [LMM87]), was made to stay in line with the semantic notion of type instance.

We make $\leq_U$ a partial order by identifying substitutions that are equal up to variable renaming, or equivalently, by defining $\phi =_U \theta$ iff $\phi \leq_U \theta$ and $\theta \leq_U \phi$. It follows from [LMM87][Theorem 7] that $\leq_U$ is a complete lower semi-lattice where least upper bounds, if they exist, correspond to unifications and greatest lower bounds correspond to anti-unifications.

The Instantiation Algorithm. We address here the following problem.

$$(\alpha)^I \qquad [\beta/\alpha] \ \vdash^I \ \alpha \leq \beta$$

$$[T\overline{\tau}/\alpha] \ \vdash^I \ \alpha \leq T\overline{\tau} \qquad (\alpha \not\in \mathrm{ftv}(\overline{\tau})) \qquad\qquad [T\overline{\tau}/\alpha] \ \vdash^I \ T\overline{\tau} \leq \alpha \qquad (\alpha \not\in \mathrm{ftv}(\overline{\tau}))$$

$$\frac{\theta \ \vdash^I \ \beta_1 \to \beta_2 \leq \sigma_1 \to \sigma_2 \qquad \beta_1,\, \beta_2 \text{ new} \qquad \alpha \not\in \mathrm{ftv}(\sigma_1, \sigma_2)}{(\theta \circ [\beta_1 \to \beta_2 \,/\, \alpha])\backslash_{\{\beta_1,\beta_2\}} \ \vdash^I \ \alpha \leq \sigma_1 \to \sigma_2}$$

$$\frac{\theta \ \vdash^I \ \sigma_1 \to \sigma_2 \leq \beta_1 \to \beta_2 \qquad \beta_1,\, \beta_2 \text{ new} \qquad \alpha \not\in \mathrm{ftv}(\sigma_1, \sigma_2)}{(\theta \circ [\beta_1 \to \beta_2 \,/\, \alpha]\backslash_{\{\beta_i,\beta_2\}}) \ \vdash^I \ \sigma_1 \to \sigma_2 \leq \alpha}$$

$$(\to)^I \qquad \frac{\theta_1 \ \vdash^I \ \sigma_1' \leq \sigma_1 \qquad \theta_2 \ \vdash^I \ \sigma_2 \leq \sigma_2' \qquad \theta = \theta_1 \sqcup \theta_2}{\theta \ \vdash^I \ \sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2'}$$

$$(T)^I \qquad \frac{\theta_i \ \vdash^I \ \tau_i \leq \tau_i' \quad (i = 1, \ldots, n) \qquad \theta = \theta_1 \sqcup \ldots \sqcup \theta_n}{\theta \ \vdash^I \ T\ \tau_1, \ldots, \tau_n \leq T\ \tau_1', \ldots, \tau_n'}$$

$$(\forall \leq)^I \qquad \frac{\theta \ \vdash^I \ [\beta/\alpha]\sigma \leq \rho \qquad \beta \text{ new}}{\theta\backslash_{\{\beta\}} \ \vdash^I \ \forall\alpha.\sigma \leq \rho}$$

$$(\leq \forall)^I \qquad \frac{\theta \ \vdash^I \ \sigma \leq [T\beta_1 \ldots \beta_n/\alpha]\sigma' \qquad T \text{ new} \qquad \{\beta_1 \ldots \beta_n\} = \mathrm{ftv}(\sigma, \forall\alpha.\sigma')}{\theta \ \vdash^I \ \sigma \leq \forall\alpha.\sigma'}$$

Figure 4: Algorithm MGI.

> (*Instantiating Substitution*). Given type schemes σ and σ', find the most general substitution $\theta = MGI(\sigma \leq \sigma')$ such that $\theta\sigma \leq \theta\sigma'$, provided θ exists; return failure otherwise.

This problem can be reduced to the unification under a mixed prefix problem [Mil92]. Unification under a mixed prefix involves finding a substitution U that solves a system of equations

$$Q_1\alpha_1 \ \ldots \ Q_m\alpha_m.s_1 = t_1 \wedge \ldots \wedge s_n = t_n$$

where the Q_i are $\exists\forall$-quantifiers and s_i and t_i are simply-typed λ-terms. We shall be concerned here only with the simpler problem where s_i and t_i are first-order terms, i.e. types. The domain of the substitution U are the existentially quantified variables in the prefix $Q_1\alpha_1 \ \ldots \ Q_m\alpha_m$. Let α_i be one such variable. Then $U\alpha_i$ can refer to any variable α_j with $j \leq i$, but not to any variable bound further to the right than α_i.

The reduction of the instantiation problem to a unification under a mixed prefix problem proceeds in three steps.

Step 1:. Decompose the instantiation problem to a system of equations with quantifier prefixes by applying the mapping $(^\circ)$ defined below.

$$\begin{array}{lcl}
(\forall\alpha.\sigma \leq \rho)^\circ & = & \exists\alpha.(\sigma \leq \rho)^\circ \quad \text{if } \alpha \not\in \mathrm{ftv}(\rho) \\
(\sigma \leq \forall\alpha.\sigma')^\circ & = & \forall\alpha.(\sigma \leq \sigma')^\circ \quad \text{if } \alpha \not\in \mathrm{ftv}(\sigma) \\
(\sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2')^\circ & = & (\sigma_1' \leq \sigma_1)^\circ \wedge (\sigma_2 \leq \sigma_2')^\circ \\
(\sigma_1 \to \sigma_2 \leq \alpha)^\circ & = & \exists\beta_1, \beta_2.\alpha = \beta_1 \to \beta_2 \wedge \\
 & & \quad (\beta_1 \leq \sigma_1)^\circ \wedge (\sigma_2 \leq \beta_2)^\circ \\
 & & \text{where } \beta_1, \beta_2 \text{ new.} \\
(\alpha \leq \sigma_1' \to \sigma_2')^\circ & = & \exists\beta_1, \beta_2.\alpha = \beta_1 \to \beta_2 \wedge \\
 & & \quad (\sigma_1' \leq \beta_1)^\circ \wedge (\beta_2 \leq \sigma_2')^\circ \\
 & & \text{where } \beta_1, \beta_2 \text{ new.} \\
(\tau \leq \tau')^\circ & = & \tau = \tau'
\end{array}$$

The meta-variable ρ in the first clause of this mapping is assumed to range over type schemes without quantifiers at the outermost-level:

$$\rho \ ::= \ \tau \mid \sigma \to \sigma'.$$

Step 2: Bring the resulting system into prefix form by applying the equations

$$\begin{array}{rcl}
\mathcal{E} \wedge (Q\alpha.\mathcal{E}') & = & Q\alpha.(\mathcal{E} \wedge \mathcal{E}') \\
(Q\alpha.\mathcal{E}) \wedge \mathcal{E}' & = & Q\alpha.(\mathcal{E} \wedge \mathcal{E}')
\end{array}$$

left-to-right as often as necessary.

Step 3: Let $\mathcal{E}_2(\sigma \leq \sigma')$ be the system resulting from Step 2. Then a unification under a mixed prefix problem

$\mathcal{E}(\sigma \leq \sigma')$ is obtained by existentially quantifying all free variables in $\mathcal{E}_2$.

$$\mathcal{E}(\sigma \leq \sigma') = \exists \mathrm{ftv}(\mathcal{E}_2).\mathcal{E}_2$$

Proposition 5.1 $\vdash \theta\sigma \leq \theta\sigma'$ iff θ is a solution to the problem $\mathcal{E}(\sigma \leq \sigma')$.

A more direct approach, which combines the transformation to a unification under a mixed prefix and the solution of this problem in a single algorithm, is shown in Figure 4. Algorithm, MGI is expressed as an inference system whose clauses are of the form

$$\theta \vdash^I \sigma \leq \sigma'.$$

Each derivation step takes as inputs two type schemes σ and σ'. It yields as output a substitution θ. We will show that θ is the most general substitution such that $\vdash \theta\sigma \leq \theta\sigma'$ holds.

The most interesting rule of the algorithm is $(\leq \forall)^I$. This rule has to enforce the side-condition $(\alpha \notin \mathrm{ftv}(\sigma))$ in the corresponding instance rule, $(\leq \forall)$. It does this by replacing α with a Skolem function T that has as arguments all other type variables in σ and $\forall\alpha.\sigma'$. This way, any substitution which would violate the side-condition by instantiating some type variable to α would lead to failure of an $(\alpha)^I$ rule in MGI due to a circular variable dependence (an "occurs check").

We now state soundness and completeness of algorithm MGI. The proofs for this and the following theorems proceed by standard inductions on derivations. Proofs are omitted here; they will be given in a forthcoming technical report [OL95].

Lemma 5.2 (Substitution) If $\vdash \sigma \leq \sigma'$ then $\vdash \theta\sigma \leq \theta\sigma'$.

Theorem 5.3 Let σ, σ' be type schemes, let θ be a substitution and let U be a finite set of type variables.

(Soundness) If $\theta \vdash^I \sigma \leq \sigma'$ then $\mathrm{dom}(\theta) \subseteq \mathrm{ftv}(\sigma, \sigma')$ and $\vdash \theta\sigma \leq \theta\sigma'$.

(Completeness) If $\vdash \theta\sigma \leq \theta\sigma'$ then there is a substitution $\phi \leq_U \theta$ such that $\phi \vdash^I \sigma \leq \sigma'$.

For type reconstruction we need a slightly different version of this algorithm that restricts the returned substitution to be the identity on some given variable set V. This algorithm is again given in logical form. For simplicity, we reuse the $\vdash^I$ symbol, writing

$$V, \theta \vdash^I \sigma \leq \sigma'.$$

The modified algorithm is obtained from MGI by skolemizing V, using the rule below.

$$\frac{T_1, \ldots, T_n \text{ new} \quad \phi = [T_i/\alpha_i]_{i=1,\ldots,n} \quad \theta \vdash^I \phi\sigma \leq \phi\sigma'}{\{\alpha_1, \ldots, \alpha_n\}, \phi^{-1} \circ \theta \vdash^I \sigma \leq \sigma'}$$

Corollary 5.4 Let σ, σ' be type schemes, let U and V be finite sets of type variables, and let θ be a substitution.

(Soundness) If $V, \theta \vdash^I \sigma \leq \sigma'$ then $\mathrm{dom}(\theta) \subseteq \mathrm{ftv}(\sigma, \sigma') \backslash V$ and $\vdash \theta\sigma \leq \theta\sigma'$.

(Completeness) If $\vdash \theta\sigma \leq \theta\sigma'$ and $\theta|_V = \mathbf{1}$ then there is a substitution $\phi \leq_U \theta$ such that $V, \phi \vdash^I \sigma \leq \sigma'$.

Proof: Direct from Theorem 5.3 and the definition of modified MGI. □

6 Type Reconstruction

Figure 5 explains the type reconstruction algorithm. Following [Rém89], it is expressed as an inference system, with clauses of the form

$$V, \theta\Gamma \vdash^W e : \sigma \quad \text{and} \quad V, \theta\Gamma \vdash^G e : \sigma.$$

Each derivation step takes as input a type variable set V, a typothesis Γ and an expression e. It yields as output a substitution θ and a type scheme σ. Informally, whenever a clause $V, \theta\Gamma \vdash^G e : \sigma$ is derivable, then θ is the identity on V and $\theta\Gamma \vdash e : \sigma$ holds. Furthermore, whenever $V, \theta\Gamma \vdash^W e : \sigma$ is derivable, then σ is the most general type scheme such that $\theta\Gamma \vdash e : \sigma$ holds. This will be made precise in the theorems below.

The purpose of the set of variables V is to prevent the computed substitution from touching type variables that occur free in annotations. For instance, given the function declaration

```
map = λf: a → b. λxs: [a]. case xs of ...
```

the body of map would be typechecked under assumptions f: a → b, xs: [a]. It is not OK to instantiate these variables when typechecking the body of map. Such an instantiation is prevented by including a and b in V.

The type reconstruction algorithm uses the auxiliary clause $\vdash^E \sigma \leq \sigma'$, which states that σ' is obtained from σ by instantiating generic type variables. The only derivation rule for this clause is $(\forall\mathrm{Elim})^W$. All $\vdash^W$ clauses have a derivation that ends in a $(\mathrm{Taut})^W$ and $(\mathrm{Gen})^W$ rule. All other rules in Figure 5 have a $\vdash^G$ conclusion. Informally, this forces a complete generalization of the result type scheme after each derivation step.

The most complex rules in the reconstruction algorithm have to do with function application. Two rules are needed, depending on whether type reconstruction for the function part of the application yields a function type or a type variable. In the first case, the rule computes a substitution instance of the result type scheme of the function. In the second case, a fresh type variable is created to hold the function result type, which corresponds to what is done in Hindley/Milner type reconstruction.

$$(\forall\text{Elim})^W \qquad \vdash^E \ \forall\overline{\alpha}.\sigma \leq [\overline{\beta}/\overline{\alpha}]\sigma \qquad \overline{\beta} \text{ new}$$

$$(\text{Taut})^W \qquad \begin{array}{l} V \ , \ \mathbf{1} \ (\Gamma, x : \sigma) \ \vdash^W \ x : \sigma \\ V \ , \ \mathbf{1} \ (\Gamma, T : \sigma) \ \vdash^W \ T : \sigma \end{array}$$

$$(\text{Gen})^W \qquad \frac{V \ , \ \theta\Gamma \ \vdash^G \ e : \sigma}{V \ , \ \theta \,|_{\text{ftv}\Gamma}\, \Gamma \ \vdash^W \ e : \forall \text{ftv}(\sigma) \backslash \text{ftv}(\theta\Gamma).\sigma}$$

$$(\text{Lambda})^W \qquad \frac{V \ , \ \theta(\Gamma_x.x{:}\,\alpha) \ \vdash^W \ e : \sigma \qquad \alpha \text{ new}}{V \ , \ \theta\Gamma_x \ \vdash^G \ \lambda x.e : \theta\alpha \to \sigma}$$

$$\frac{V \cup \text{ftv}(\sigma) \ , \ \theta(\Gamma_x.x{:}\,\sigma) \ \vdash^W \ e : \sigma'}{V \ , \ \theta\Gamma_x \ \vdash^G \ \lambda x{:}\,\sigma.e : \sigma \to \sigma'}$$

$$(\text{Apply})^W \qquad \frac{\begin{array}{c} V \ , \ \theta_1\Gamma \ \vdash^W \ e : \sigma \qquad \vdash^E \ \sigma \leq \sigma_1 \to \sigma_2 \\ V \ , \ \theta_2\Gamma \ \vdash^W \ e' : \sigma' \qquad V \ , \ \theta_3 \ \vdash^I \ \sigma' \leq \sigma_1 \qquad \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3 \end{array}}{V \ , \ \theta\Gamma \ \vdash^G \ e\,e' : \theta\sigma_2}$$

$$\frac{\begin{array}{c} V \ , \ \theta_1\Gamma \ \vdash^W \ e : \sigma \qquad \vdash^E \ \sigma \leq \alpha \qquad \beta \text{ new} \\ V \ , \ \theta_2\Gamma \ \vdash^W \ e' : \sigma' \qquad V \ , \ \theta_3 \ \vdash^I \ \alpha \leq \sigma' \to \beta \qquad \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3 \end{array}}{V \ , \ \theta\Gamma \ \vdash^G \ e\,e' : \theta\beta}$$

$$(\text{Typed})^W \qquad \frac{V \cup \text{ftv}(\sigma) \ , \ \theta_1\Gamma \ \vdash^W \ e : \sigma' \qquad V \cup \text{ftv}(\sigma) \ , \ \theta_2 \ \vdash^I \ \sigma' \leq \sigma \qquad \theta = \theta_1 \sqcup \theta_2}{V \ , \ \theta\Gamma \ \vdash^G \ (e : \sigma) : \sigma}$$

$$(\text{Let})^W \qquad \frac{V \ , \ \theta_1\Gamma_x \ \vdash^W \ e : \sigma \qquad V \ , \ \theta_2(\Gamma_x.x{:}\,\sigma) \ \vdash^W \ e' : \sigma' \qquad \theta = \theta_1 \sqcup \theta_2}{V \ , \ \theta\Gamma_x \ \vdash^G \ \mathsf{let}\ x = e \ \mathsf{in}\ e' : \theta\sigma'}$$

$$(\text{Proj})^W \qquad \frac{T : \sigma \in \Gamma \qquad \vdash^E \ \sigma \leq \sigma' \to T\,\overline{\tau}}{V \ , \ \mathbf{1}\,\Gamma \ \vdash^G \ T^{-1} : T\,\overline{\tau} \to \sigma'}$$

$$(\text{Newtype})^W \qquad \frac{\sigma'' = \forall\overline{\alpha}.\sigma \to T\,\overline{\alpha} \qquad V \cup \text{ftv}(\sigma'') \ , \ \theta(\Gamma_T, T{:}\,\sigma'') \ \vdash^W \ e : \sigma'}{V \ , \ \theta\Gamma_T \ \vdash^G \ \mathsf{newtype}\ T\,\overline{\alpha} = \sigma \ \mathsf{in}\ e : \sigma'}$$

Figure 5: Type reconstruction algorithm.

Lemma 6.1 (Substitution) If $\Gamma \vdash e : \sigma$ then $\theta\Gamma \vdash e : \theta\sigma$.

Theorem 6.2 Let Γ be a typothesis, let e be an expression, let σ be a type scheme. Let $V \supseteq \mathrm{ftv}(\Gamma) \cap \mathrm{ftv}(e)$ and U be finite sets of type variables and let θ be a substitution.

(Soundness) If $V, \theta\Gamma \vdash^W e : \sigma$ then $\mathrm{dom}(\theta) \subseteq \mathrm{ftv}(\Gamma) \backslash (V \cup \mathrm{ftv}(e, \sigma)$ and $\theta\Gamma \vdash e : \sigma$.

(Completeness) If $\theta\Gamma \vdash e : \sigma$ and $\theta|_{V \cup \mathrm{ftv}(e)} = \mathbf{1}$ then there is a substitution $\phi \leq_U^{\phi'} \theta$ and a type scheme σ' such that $V, \phi\Gamma \vdash^W e : \sigma'$ and $\phi'\sigma' \leq \sigma$.

Corollary 6.3 (Principal Types) Let Γ be a closed typothesis. If $\Gamma \vdash e : \sigma$ then there is a type scheme $\sigma' \leq \sigma$ such that $\emptyset, \mathbf{1}\Gamma \vdash^W e : \sigma'$ and $\Gamma \vdash e : \sigma'$.

7 Conclusion

We have presented a type system that generalizes several recent second-order polymorphic extensions of the Hindley/Milner system. The presented type system stays firmly in the tradition of Hindley/Milner in that all Hindley/Milner programs continue to be typable with the same types, and the essential theorems carry over.

To keep the present treatment simple we have kept the type system fairly small. When applied in a programming language, several extensions would be possible and maybe even desirable. We have already discussed polymorphic sum and product type schemes. As another possible extension, it is straightforward to add polymorphic recursion [Myc84], which is known to be undecidable in the absence of type declarations [Hen93, KTU93].

Starting with Hope [BMS80], many programming languages have supported polymorphic recursion when explicit declarations are given for polymorphically recursive functions. Nevertheless, we are not aware of a formal analysis of type reconstruction for these languages. Our system can be extended to polymorphic recursion by adding the typing rule below.

$$\text{(Letrec)} \quad \frac{\Gamma_x, x{:}\sigma \vdash e : \sigma \qquad \Gamma_x, x{:}\sigma \vdash e' : \sigma'}{\Gamma \vdash \mathsf{letrec}\ x{:}\sigma = e\ \mathsf{in}\ e' : \sigma'}$$

The corresponding clause for the type reconstruction algorithm is:

$$\text{(Letrec)}^W \quad \frac{\begin{array}{c} V \cup \mathrm{ftv}(\sigma)\ ,\ \theta_1(\Gamma_x, x{:}\sigma) \vdash^W e : \sigma'' \\ V \cup \mathrm{ftv}(\sigma)\ ,\ \theta_2 \vdash^I \sigma'' \leq \sigma \\ V \cup \mathrm{ftv}(\sigma)\ ,\ \theta_3(\Gamma_x, x{:}\sigma) \vdash^W e' : \sigma' \\ \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3 \end{array}}{V\ ,\ \theta\Gamma \vdash \mathsf{letrec}\ x{:}\sigma = e\ \mathsf{in}\ e' : \sigma'}$$

An extension of the soundness and completeness proofs for type reconstruction is straightforward.

As a more ambitious extension one could combine our system with subtyping. This is particularly intriguing since we already have a subsumption rule, albeit for type schemes, not for types. Moreover, the instance relationship on function type schemes uses the contravariance rule that is standard in subtyping systems. What is still missing is a definition of subtyping for types. An extension along these lines should yield a system in which parametric polymorphism is regarded as a special form of subtyping, which would lead to a closer integration of the two typing disciplines.

Acknowledgments

We'd like to thank Mark Jones, Benjamin Pierce, Didier Rémy and Phil Wadler for stimulating discussions. The idea of lifting out maximal subterms of polymorphic types in the F_2 encoding is due to Didier Rémy. Thanks also to Dilip Sequeira for helpful comments on an earlier version of the paper.

References

[Aug94] L. Augustsson. Haskell B. user's manual version 0.999.7, October 1994. Distributed with the HBC compiler.

[BMS80] Rod Burstall, David MacQueen, and Donald T. Sanella. Hope: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Redwood Estates, California, August 1980. The LISP Company.

[Boe89] Hans-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 192–206. ACM, ACM Press, June 1989.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, January 1982.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In Neil D. Jones, editor, *ESOP '90, European Symposium on Programming*, pages 134–151. Springer-Verlag, 1990. Lecture Notes in Computer Science 432.

[Gir71] J. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[Jon92] Mark P. Jones. *Qualified Types: Theory and Practice.* D.phil. thesis, Oxford University, September 1992.

[Jon95] Mark P. Jones. From Hindley-Milner types to first-class structures. In *Proc. Haskell Workshop, La Jolla*, pages 115–136, June 1995. Yale University Research Report YALEU/DCS/RR-1075.

[KTU89] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. Technical Report BUCS-89-010, Boston University, Oct. 1989. also in Proc. of Symp. on Theory of Computing, Baltimore, Maryland, May 1990.

[KTU93] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(1):290–311, April 1993.

[LMM87] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

[LPar] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, to appear.

[McC84] N. McCracken. The typechecking of programs with implicit type structure. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, pages 301–315. Springer-Verlag, June 1984. Lecture Notes in Computer Science 173.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[Mit90] John C. Mitchell. Polymorphic type inference and containment. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, chapter 8. Addison-Wesley Publishing Company, Inc., 1990.

[Mor68] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968. Technical Report MAC-TR-57.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[MP93] M. Mauny and F. Pottier. An implementation of Caml-Light with existential types. Technical report, INRIA, October 1993. Distributed with the Caml-Light system.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Symposium. on Programming, LNCS 167*, 1984.

[OG89] James William O'Toole and David K. Gifford. Polymorphic type reconstruction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–217. ACM, ACM Press, June 1989.

[OL95] Martin Odersky and Konstantin Läufer. Type reconstruction in the presence of type scheme annotations. Technical report, University of Karlsruhe, 1995. forthcoming.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–1469, June 1995.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages.* PhD thesis, Imperial College of Science, Technology, and Medicine, University of London, 1990.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, July 1988.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

[Rém94] Didier Rémy. Programming objects with ML-ART, and extension to ML with abstract and record types. In *Proc. Theoretical Aspects of Computer Software*, pages 321–346, April 1994. Springer LNCS 789.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *International Programming Symposium*, pages 408–425. Springer-Verlag, 1974. Lecture Notes in Computer Science 19.

[Wel94] J.B. Wells. Typability and type checking in the second order λ-calculus are equivalent and undecidable. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.