

First-class polymorphism with existential types

Daan Leijen

Microsoft Research

daan@microsoft.com

Draft, August 15, 2006

Abstract

Abstract datatypes can be typed conveniently using existential types. Even though it is a powerful abstraction mechanism, current type inference systems based on Hindley-Milner do not allow existential types as first-class citizens – every existential type must be explicitly declared, packed, and unpacked using a data constructor. We present an extension of the MLF type system with first-class existential types. Existential types are simply introduced by an annotation, and eliminated automatically in applications without the need for a special open construct. The system is fully implemented in the experimental Morrow interpreter.

1. Introduction

Type systems based on Hindley-Milner [5, 18] support polymorphism using universal quantification, where we can assign types to values that behave uniformly over all types. Here are two tuples with (an inferred) universal type:

$$\begin{aligned} \text{conI} &= (1, \text{id}) && :: \forall \alpha. (\text{Int}, \alpha \rightarrow \alpha) && \text{-- inferred} \\ \text{conL} &= ([1], \text{length}) && :: \forall \alpha. ([\text{Int}], [\alpha] \rightarrow \text{Int}) && \text{-- inferred} \end{aligned}$$

The types assigned to the tuples incomparable and we can not put them in a list for example. However, both tuples can be given an equivalent type using existential quantification. Informally, when an expression e has a typing $\exists \alpha. \sigma$ there exists some type α such that the expression e has type σ . We can give the following existential type to our tuples:

$$\begin{aligned} \text{absI} &= (1, \text{id}) && :: \exists \alpha. (\alpha, \alpha \rightarrow \text{Int}) \\ \text{absL} &= ([1], \text{length}) && :: \exists \alpha. (\alpha, \alpha \rightarrow \text{Int}) \end{aligned}$$

Using existential quantification over the type of the first component we have effectively made that type abstract. We can only manipulate this value by passing it as an argument to the second component. Since both values have the same type now, we can put them in a list:

$$[\text{absI}, \text{absL}]$$

There are many examples of the utility of existential types. In particular, we can give existential type signatures to abstract interfaces, and use different implementations that are instantiated to the abstract existential type. This is very useful for collection libraries for example. Furthermore, we can create heterogeneous lists where the differing implementation types are existentially quantified, as shown above.

Unfortunately, languages that support existential types severely restrict their use. Extended versions of Haskell and ML require all existential types to be packed explicitly in a datatype [7]. One reason for this restriction is that first-class existential quantification quickly leads to impredicative types where the quantifiers can appear anywhere in a type, which is not allowed in type systems based on Hindley-Milner. For example, the heterogeneous list in the pre-

vious paragraph has an impredicative type where the existential quantifier is inside the list constructor:

$$[\text{absI}, \text{absL}] :: [\exists \alpha. (\alpha, \alpha \rightarrow \text{Int})] \quad \text{-- inferred}$$

A more serious drawback of packing existential types in datatypes is that they must be opened explicitly through pattern matching (or a special ‘open’ construct). This seems innocent but an explicit open construct forces a rigid structure on a program where all terms that share a representation of an abstract type need to be used under a single scope. This was first described by MacQueen [16] in the context of ML modules as the ‘diamond import problem’ and is the main objection to using existential types as a foundation for first-class modules [14, 6, 24].

In this paper, we are going to remove these restrictions and present a type system where existential types first-class. In particular:

- Existential quantifiers can appear anywhere in a type. Since this requires impredicative types, we use the MLF type system as our foundation [10, 11]. This is a sophisticated type inference system with first-class polymorphism where types can be impredicative and of higher-rank.
- Existential types are simply introduced by a type annotation, and there is no need to artificially declare and pack existential types in a datatype.
- Existential types are eliminated automatically when applied to a function that is polymorphic in the existential type. For example, the following expression is accepted without any annotation:

$$(\lambda x \rightarrow (\text{snd } x) (\text{fst } x)) \text{absI} \quad :: \text{Int} \quad \text{-- inferred}$$

This is the key idea in this paper, and removes the need for a special open construct (like pattern matching) while making it easier for programmers to work with existential types in practice.

- To make type inference decidable, existential types are never ‘guessed’ by the type system, and always explicitly introduced by a type annotation. However, since the type system is impredicative, abstraction extends uniformly over existential types without need for extra type annotations. In particular, type variables can be instantiated to existential types, and polymorphic functions like id and map work unchanged on existential values too. For example, without any type annotation, the type system accepts:

$$\text{head } (\text{map } \text{id } [\text{absI}]) \quad :: \exists \alpha. (\alpha, \alpha \rightarrow \text{Int}) \quad \text{-- inferred}$$

- We give type rules for existential types without using skolemization. In many approaches, elimination of an existential quantifier introduces fresh skolem type constants. This is an efficient implementation technique for unification but is inconve-

nient when studying the type rules themselves. In particular, we can elegantly deal with ‘escaping’ existential variables which is usually disallowed in type systems based on skolemization. For example, the type system accepts:

$$\text{snd absI} \quad :: \exists \alpha. \alpha \rightarrow \text{Int} \quad \text{-- inferred}$$

The system is not without flaws. In particular, type inference is incomplete with respect to the natural type rules and we introduce a restricted version of the type rules to restore completeness (Section 6). Furthermore, the system is fragile under rewrites in the sense that when $f x$ is typable, it is not always the case that $\text{revapp } x f$ is typeable where revapp is reverse application. Still, we feel that this work is a significant step forward in type inference for first-class existential types. The system is fully implemented in Morrow [12] and experience on small but significant examples shows that it works well in practice. Indeed, all examples presented in this paper are valid Morrow programs.

2. Existential quantification

Universal quantification allows to assign types to values that behave uniformly independent of a particular type. For example, a function that determines the length of a list behaves uniformly over lists of all possible element types:

$$\text{length} \quad :: \forall \alpha. [\alpha] \rightarrow \text{Int}$$

As shown in the introduction, we can also assign meaning to existential quantification where values are defined for a particular but abstract type. Here are some further examples of existential types:

$$\begin{aligned} 1 & \quad :: \exists \alpha. \alpha \\ [1, 2] & \quad :: \exists \alpha. [\alpha] \\ [1, 2] & \quad :: [\exists \alpha. \alpha] \end{aligned}$$

As we can see from these examples, expressions can be assigned many different existential types. Actually, every expression has type $\exists \alpha. \alpha$ since for every value there exists a type that is the type of that value. We call this type ‘top’ and write it as \top . By definition:

$$\top \equiv \exists \alpha. \alpha$$

The dual of this type is the most polymorphic type bottom (\perp), which we can define as

$$\perp \equiv \forall \alpha. \alpha$$

The type $\exists \alpha. \alpha$ is not very useful since there is no way to manipulate it except for passing it around. A more interesting type expresses a relationship between existentially quantified types. Take for example the following definition of *Key* type:

$$\begin{aligned} \text{type } \text{Key } \alpha & = \{ \text{val} :: \alpha, \text{key} :: \alpha \rightarrow \text{Int} \} \\ \text{intKey} & = \{ \text{val} = 1, \text{key } x = x \} \quad :: \text{Key } \text{Int} \\ \text{listKey} & = \{ \text{val} = [1, 2], \text{key } xs = \text{length } xs \} \quad :: \text{Key } [\text{Int}] \end{aligned}$$

The *Key* record contains two fields *val* and *key*, where the *key* function can be applied to *val* value. By existentially quantifying the element type, we can define an abstract key:

$$\begin{aligned} \text{absKey} & :: \exists \alpha. \text{Key } \alpha \\ \text{absKey} & = \text{intKey} \quad \text{-- or } \text{listKey} \end{aligned}$$

Even though we have no information about the existentially quantified type α , we do know the relation of this type with the given operations: the type of *val* is the same as the domain of the *key* function, and thus we can apply them. This is a simple example, but in general we can view values of type *Key* as implementations of an abstract type where we can only manipulate the abstract

type using provided operations. Functions that use such functionality should be polymorphic in the abstract type. In particular, if we define a polymorphic *use* function:

$$\begin{aligned} \text{use} & :: \forall \alpha. \text{Key } \alpha \rightarrow \text{Int} \\ \text{use } k & = k.\text{key } k.\text{val} \end{aligned}$$

we can apply that to the abstract key:

$$\text{use absKey}$$

This is the key idea of the type inference system presented in this paper – since *use* is polymorphic in α , it can be applied to *any* value of *Key* α , including values of a type where α is abstract! The type inference algorithm effectively *opens* the existential type automatically. Note that this is not trivial in general as some existential type variables may need to be opened, while others should stay quantified. Take for example the application $f x$ where:

$$\begin{aligned} f & :: \forall \alpha. (\exists \beta. T \alpha \beta) \rightarrow T \alpha \alpha \\ x & :: \exists \alpha \beta. T \alpha \beta \end{aligned}$$

In this case, type inference only opens the existential type α , and accepts the application with type $\exists \alpha. T \alpha \alpha$. The example shows the key ingredients of our system:

- Any type can be existentially quantified. Note that f expects an existentially quantified type. This requires a higher-ranked impredicative type system where quantifiers can occur anywhere in a type. We return to this subject in Section 2.2.
- Existential types are automatically opened when applied to a function that is polymorphic in the existential types. When dealing with multiple first-class existential types, it is very convenient when the programmer is not burdened by opening and packing existentials explicitly.

2.1 Quantified component types

As an example of the reduced burden, we can look at how the application $f x$ is defined in systems based on Hindley-Milner. One way to support first-class quantified types in a Hindley-Milner type system is to package quantified types in datatypes [19, 9, 23, 7]. The introduction and elimination of quantifiers is combined with the usual datatype construction and pattern matching. This preserves type inference and is low in notational overhead. For example, in GHC we can express an abstract *Key* type as¹:

$$\text{data Key} = \exists \alpha. \text{Key} \{ \text{val} :: \alpha, \text{key} :: \alpha \rightarrow \text{Int} \}$$

where we use pattern matching to open the existential type. Unfortunately this quickly becomes a burden in practice. For example, if we want to express the previous application $f x$, we need to introduce two new data types just to express the types of f and x :

$$\begin{aligned} \text{data } X & = \exists \alpha \beta. X (T \alpha \beta) \\ \text{data } F \alpha & = \exists \beta. F (T \alpha \beta) \\ f & :: \forall \alpha. F \alpha \rightarrow T \alpha \alpha \\ x & :: X \end{aligned}$$

To apply f to x , we need to explicitly pattern match on x to open it, and pack it again using the F constructor. Furthermore, we need yet another datatype to express the result type:

$$\begin{aligned} \text{data } R & = \exists \alpha. R (T \alpha \alpha) \\ \text{apply} & = \text{case } x \text{ of } X \ x' \rightarrow R (f (F \ x')) \end{aligned}$$

The approach of packaging quantified types in datatypes is a simple extension to Hindley-Milner that preserves type inference and is

¹Confusingly, Haskell uses the `forall` keyword to existentially quantify types. In concrete syntax, we write the declaration of the *Key* type as `data Key = forall a. Key {val:: a, key :: a -> Int}`.

natural for the programmer. Unfortunately, as we can see from the above example, the introduction of a spurious datatype for every instance of an existential quantification severely limits abstraction over existentially quantified types and can be a real burden in practice.

More seriously though, the explicit opening enforces a strict block structure on programs where all terms that need to share the type of an abstract implementation need to be defined inside a single scope (‘the diamond import problem’). Suppose for example that we have the following implementation of a stack:

```
type Stack s α = { empty :: s α,
                  push  :: α → s α → s α,
                  top   :: s α → α }

listStack :: ∀α. Stack [] α
listStack = { empty = [],
             push  = λx xs → (x : xs),
             top   = λ(x : xs) → x }
```

We can treat the list implementation of the `listStack` as abstract and we create a new datatype with an existential quantifier:

```
data AbsStack α = ∃s. AbsStack (Stack s α)
absStack :: AbsStack α
absStack = AbsStack listStack
```

Suppose we have the following signature of collections:

```
type Coll c α = { fromList :: [α] → c α }
```

We can of course make a collection out of any stack:

```
newColl :: ∀sα. Stack s α → Coll s α
newColl s
  = { fromList = foldr s.insert s.empty }
```

Just like the abstract stack, we can also hide the implementation type of a collection, and create abstract collections from abstract stacks:

```
data AbsColl α = ∃c. AbsColl (Coll c α)
newAbsColl :: ∀α. AbsStack α → AbsColl α
newAbsColl (AbsStack s) = AbsColl (newColl s)
```

Since the collection implementation type is now existentially quantified, we can never actually use the result of `fromList!` The only way to use the abstract collection functionality is to open both the stack and the collection inside the same scope:

```
case absStack of
  AbsStack s → let coll = newColl s
               in s.top (coll.fromList [1])
```

In general, we need to open an existential datatype with a scope that covers all terms that need to share a particular implementation type. This enforces a strict block structure on the program, and prevents us from defining these terms in separate compilation units for example. This is one of the most important reasons that existential types are generally dismissed as the basis for abstract types in module systems [16, 14, 24].

In this paper, we are going to solve this issue head-on by making existential types first-class citizens where no spurious data types are necessary. As shown in the previous examples, a necessary ingredient is a higher-rank, impredicative type system where (existentially) quantified types can occur anywhere in the type. We do this by building on the MLF type system [10, 11] that already supports type inference for first-class polymorphic values. Before discussing on how to extend MLF with existential types, we first give a brief tour of the basic MLF system.

2.2 A tour of MLF

Hindley-Milner type inference requires that only values that are bound by a `let` construct can have a polymorphic type. For example, it is not possible for the argument of a function to be used with a polymorphic type. Here is an example of a function that takes a polymorphic argument:

```
f choose = (choose True False, choose 1 2)
```

The function `f` takes a function `choose` and calls it on two booleans and two characters in the body of the function. Neither Haskell, nor ML, would accept this definition, because in the Hindley-Milner type system, lambda-bound variables such as `choose` can only have monomorphic types, which is equivalent to the property that universal quantifiers can only appear at the outermost level of a type.

In contrast, explicitly typed languages such as system-F allow universal quantifiers to appear deep within a type. For example, if we know that `choose` has the polymorphic type $\forall\alpha. \alpha \rightarrow \alpha \rightarrow \alpha$, then the above definition makes sense, and `f` can be given the following type:

```
f :: (∀α. α → α → α) → (Bool, Char)
```

This is a rank-2 type, as it contains quantifiers to the left of the function arrow. The Haskell implementations GHC and Hugs support higher-rank polymorphism such as it occurs in the definition of `f` above. If `f` is equipped with a type signature, the compiler accepts the definition. However, even though higher-ranked, the implementations are still limited due to a second restriction of Hindley-Milner: quantified variables can only be instantiated with a monomorphic type. Systems with this restriction are called *predicative*.

The MLF type system is an extension of Hindley-Milner that fully supports first-class polymorphism: universal quantifiers can appear anywhere in a type, and quantified variables can be instantiated with a polymorphic type. Because of the last feature, MLF is an *impredicative* type system.

2.2.1 Impredicative types

Impredicative types are very important in a programming language since it extends abstraction uniformly over higher-ranked types. For example, we can have structures with quantified types, like a list with polymorphic identity functions:

```
[id] :: [∀α. α → α]
```

Actually, there is another type that we can assign to this expression, namely:

```
[id] :: ∀α. [α → α]
```

This is the type that would be inferred by a predicative type system like Hindley Milner. Unfortunately, neither type is better as the other. For example, we can append a list of type `[Int → Int]` if we assign the second type, but only if we assign the first type, we can pass it to a function that expects a list of truly polymorphic functions. Indeed, a naïve implementation of impredicative polymorphism does not have principal types.

2.2.2 Principal types

The MLF type system restores the principal type property even in the presence of impredicative types by extending type language with bounds. In particular, the principal type of `[id]` is in MLF:

```
[id] :: ∀(α ≥ ∀β. β → β). [α]
```

The quantification of α has a flexible bound $\forall\beta. \beta \rightarrow \beta$, meaning that it can be instantiated with any instance of $\forall\beta. \beta \rightarrow \beta$. Instan-

tiating with the trivial instance $\forall\beta. \beta \rightarrow \beta$, or $\gamma \rightarrow \gamma$ for a fresh γ leads to the two types given above.

Besides flexible bounds, we also have rigid bounds that bind a type variable to exactly the given type. For example:

$$g :: \forall(\alpha = \forall\beta. \beta \rightarrow \beta). [\alpha] \rightarrow (Int, Bool)$$

$$g [x] = (x 1, x True)$$

In this case, the function g expects a list with truly polymorphic elements; the type α must be at least as polymorphic as $\forall\beta. \beta \rightarrow \beta$, and can not be instantiated. The application is $g [id]$ is accepted, but the application $g [id, (+1)]$ would be rejected.

2.2.3 Presentation of MLF types

MLF types put us in a dilemma: they are nice to work with from a theoretician’s or an implementor’s point of view, but they are awkward to read for a programmer. The reason for both is that MLF types, although higher-ranked, collect all the qualifiers in a prefix, at the beginning of the type. For instance, the type of *runST* in GHC is:

$$\forall\alpha. (\forall s. ST s \alpha) \rightarrow \alpha$$

while the type in MLF is:

$$\forall\alpha. \forall(\beta = \forall s. ST s \alpha). \beta \rightarrow \alpha$$

In an earlier paper [13] we proposed to adopt a simple heuristic when presenting types to users in MLF-based systems:

- types are first converted to normal form,
- a flexible constraint ($\alpha \geq \sigma$) is inlined only if there is a single occurrence of α in the scope of the constraint, and when that occurrence is *positive* (i.e., to the right of a function arrow),
- a rigid constraint ($\alpha = \sigma$) is inlined only if there is a single occurrence of α in the scope of the constraint, and when that occurrence is *negative* (i.e., to the left of a function arrow).

For the purpose of this heuristic, only function arrows (but no other type constructors) influence the sign of a position. This heuristic is loss-free: the original MLF type can easily be recovered from the simplified type. As an example, the following explicit MLF type

$$\forall(\beta = \forall\alpha. \alpha \rightarrow \alpha). \forall(\gamma \geq \forall\alpha. \alpha \rightarrow \alpha). [\beta] \rightarrow [\gamma]$$

can be presented according to this heuristic as:

$$[\forall\alpha. \alpha \rightarrow \alpha] \rightarrow [\forall\alpha. \alpha \rightarrow \alpha]$$

The heuristic is used in the experimental Morrow compiler [12], which presents the above type and the type of *runST* in their simpler form. Experience to the present day shows that it makes MLF much easier to work with, because the bounded quantifications that remain are the ones where the expressed sharing actually plays a crucial role in the type. Most simple functions, however, get the same types as they would in ML or Haskell. In the remainder of this article we will make use of this convention.

2.2.4 Type annotations

Type inference in MLF never invents polymorphism: whenever a lambda-bound argument is used polymorphically, a type annotation is required. The same holds for the Haskell implementation of higher-ranked polymorphism.

All these systems, however, share the property that type annotations are only required for programs which actually make use of higher-ranked polymorphism. All other programs continue to work without type annotations. Therefore, switching the underlying type system of an ML-like language to MLF does not break any existing programs.

2.2.5 Abstraction

Due to the impredicative nature of MLF, abstraction extends uniformly over higher ranked values. For example, we can *map* our f function from Section 2.2 over a list of *choose* functions:

$$map f [\lambda x y \rightarrow x, \lambda x y \rightarrow y]$$

In a predicative type system like in GHC, this would not work for two reasons: one of the quantifiers of the *map* function needs to be instantiated with a polymorphic type, and the list contains elements of polymorphic functions.

Of course, as described in Section 2.1, we can simulate this function in a predicative system by hiding the polymorphism within a datatype. While theoretically no less expressive, in practice this can prevent programmers from adopting a solution that makes use of polymorphic types. The MLF type system encourages programmers to see polymorphic values as first-class values in every respect, that need not be avoided and require no special treatment.

Note that in the basic MLF type system, one can already model existential types using second-rank types. In particular, we can translate expression e with any existential type $\exists\alpha. \sigma$ as:

$$\llbracket e :: \exists\alpha. \sigma \rrbracket = (\lambda f \rightarrow f e) :: \forall\beta. (\forall\alpha. \sigma \rightarrow \beta) \rightarrow \beta$$

That is, we package up the value into a function that takes a ‘de-constructor’ function that is polymorphic in the existential quantifier. However, in practice this corresponds directly to using datatypes to package existential types and leads to the same kind of problems.

3. Extending MLF with existential quantification

Instead of using an encoding, we extend the basic MLF system directly with first-class existential types, called MLF_{\exists} . The syntax of MLF_{\exists} types is given in Figure 1.

The basic type language distinguishes monomorphic types τ (*monotypes*) from polymorphic types σ (*polytypes*). A monotype is either an applied constructor c , or a type variable α . We assume that the binary function space constructor (\rightarrow) is among the possible constructors. A polytype is either a monotype, the most polymorphic type bottom (\perp), or a polytype quantified with a bound $\forall(\alpha \geq \sigma)$. A type variable is either quantified existentially, or universally where it is constrained by a *flexible* (\geq), or *rigid* ($=$) bound.

Intuitively, a rigid bound can only be instantiated with *exactly* the type given, while a flexible bound can be instantiated with any *instance* of that type. In particular, a bound $\forall(\alpha \geq \perp)$ can be instantiated with any type since \perp is the most polymorphic type possible. For example, the full MLF type of the identity function is:

$$id :: \forall(\alpha \geq \perp). \alpha \rightarrow \alpha$$

We call a bound $\forall(\alpha \geq \perp)$ an *unconstrained bound* and usually shorten it to $\forall\alpha$. Indeed, in Hindley-Milner type systems all quantifiers have unconstrained bounds and are left implicit. The polytype \perp is defined equivalent to $\forall(\alpha \geq \perp). \alpha$.

Dually, flexible existential bounds can not be instantiated, but express that there exists some type that is an instance of the bound. The unconstrained existential bound $\exists(\alpha \geq \perp)$ quantifies a type α that can be any possible type, and we shorten it to $\exists\alpha$. The least polymorphic type \top (*top*) is made equivalent to $\exists(\alpha \geq \perp). \alpha$.²

The MLF_{\exists} type language lets us directly express existential quantification at any rank without a need for ‘spurious’ datatypes.

²To express the duality between universal and existential quantification better, we could also define existential \leq bounds and define $\top = \exists(\alpha \leq \top). \alpha$.

Monomorphic types	
$\tau ::= \alpha$	type variable
$ c \tau_1 \dots \tau_n$	constructor application
Polymorphic types	
$\sigma ::= \nabla(\alpha \diamond \sigma). \sigma$	quantified type
$ \tau$	mono type
$ \perp$	most polymorphic type
Quantifier	
$\nabla(\alpha \diamond \sigma) ::= \forall(\alpha \geq \sigma)$	universal with flexible bound
$ \forall(\alpha = \sigma)$	universal with rigid bound
$ \exists(\alpha \geq \perp)$	existential quantifier
Prefix	
$Q ::= q_1, \dots, q_n$	a prefix is a list of quantifiers
$q ::= \nabla(\alpha \diamond \sigma)$	quantifier
Syntactic sugar	
$\forall \alpha = \forall(\alpha \geq \perp)$	
$\exists \alpha = \exists(\alpha \geq \perp)$	
$\exists Q = \exists \alpha_1, \dots, \exists \alpha_n$	prefix of existential quantifiers
$\forall Q = \forall(\alpha_1 \diamond \sigma_1), \dots$	prefix of universal quantifiers
$Q. \sigma = q_1. \dots q_n. \sigma$	quantify using a prefix

Figure 1. MLF_∃ types.

For example, we can directly express a heterogeneous lists of existential values:

$$[\exists \alpha. \{ val :: \alpha, key :: \alpha \rightarrow Int \}]$$

which is a shorthand for:

$$\forall(\beta \geq \exists \alpha. \{ val :: \alpha, key :: \alpha \rightarrow Int \}). [\beta]$$

Since existentially quantified variables always have an unconstrained bound, we can always move them to the front of a type. In particular, we can write any type σ in the form $\exists Q_1. \forall Q_2. \tau$, where Q_1 just contains existential quantifiers, and Q_2 consists of universally quantified type variables. We will formalize this in Section 5.1 when discussing the equivalence relation on types.

We usually define relations on types under a *prefix* Q , which is a list of quantified type variables. Such prefix gives the bounds of free type variables of the types in the relation. In typical Hindley-Milner systems such prefix is usually implicit as all bounds are unconstrained. We assume that every quantified variable is unique in a prefix and that we can thus talk about the domain ($\text{dom}(Q)$) of a prefix. When two prefixes Q_1 and Q_2 are catenated ($Q_1 Q_2$), we implicitly assume that their domains are disjoint, written as $\text{dom}(Q_1) \not\cap \text{dom}(Q_2)$.

4. Typing rules

Figure 2 gives the syntax directed type rules of MLF_∃. We choose to present only syntax directed rules as our goal is to automatically open (and close) existentials when applied to functions that are polymorphic in the existential quantifiers. A syntax directed presentation gives us exact control over the scope of existentials in an application.

A derivation $(Q) \Gamma \vdash_s e : \sigma$ means that under a prefix Q and in a type environment Γ , the expression e can be assigned type σ . The type environment Γ associates types to variables in scope, and the prefix Q provides bounds to the free type variables in Γ , i.e. $\text{ftv}(\Gamma) \subseteq \text{dom}(Q)$.

Even though existential types are an expressive extension that require a sophisticated unification algorithm, the type rules are sur-

(S-VAR)	$\frac{x : \sigma \in \Gamma}{(Q) \Gamma \vdash_s x : \sigma}$
(S-LET)	$\frac{(Q) \Gamma \vdash_s e_1 : \sigma_1 \quad (Q) \Gamma, x : \sigma_1 \vdash_s e_2 : \sigma_2}{(Q) \Gamma \vdash_s \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \sigma_2}$
(S-LAM)	$\frac{\alpha \notin \text{ftv}(\tau) \quad \text{dom}(Q') \not\cap \text{ftv}(\Gamma) \quad (QQ') \Gamma, x : \tau \vdash_s e : \sigma}{(Q) \Gamma \vdash_s \lambda x. e : (Q', \forall(\alpha \geq \sigma)). \tau \rightarrow \alpha}$
(S-APP)	$\frac{(Q) \Gamma \vdash_s f : \exists Q_a. \sigma_1 \quad (Q) \Gamma \vdash_s e : \exists Q_b. \sigma_2 \quad (QQ_a Q_b) \sigma_1 \sqsubseteq \forall Q_3. \tau_3 \rightarrow \tau \quad (QQ_a Q_b) \sigma_2 \sqsubseteq \forall Q_3. \tau_3}{(Q) \Gamma \vdash_s f \ e : Q_a Q_b Q_3. \tau}$

Figure 2. Syntax directed typing rules

prisingly simple. The rule (S-VAR) assigns a type to variables that are bound in the environment. The rule (S-LET) nicely shows that we are working in a system with first-class polymorphism as it has no effect on types at all, and just extends the type environment Γ with a new binding. The rule (S-LAM) assigns types to lambda expressions. Just like MLF, it assigns a monotype to the argument and thus an annotation is needed for arguments that are used polymorphically.

The only rule that deals explicitly with existential types is (S-APP). The application rule uses the instance relation \sqsubseteq that we define formally in Section 5. Informally, the notation $(Q) \sigma_1 \sqsubseteq \sigma_2$ means that σ_2 is an instance of σ_1 , where Q provides bounds to the free type variables in σ_1 and σ_2 . The application rule uses the instance relation to instantiate the function type and the argument type such that the argument types are equal.

Here is a small example of the application rule in a derivation of the type of *id* 1:

$$(S-APP) \frac{(Q) \Gamma \vdash_s id : \forall \alpha. \alpha \rightarrow \alpha \quad (Q) \Gamma \vdash_s 1 : Int \quad (Q) \forall \alpha. \alpha \rightarrow \alpha \sqsubseteq Int \rightarrow Int \quad (Q) Int \sqsubseteq Int}{(Q) \Gamma \vdash_s id \ 1 : Int}$$

In this example, the prefixes Q_a, Q_b , and Q_3 are all empty.

4.1 Opening existential types

The main difference with the MLF type rules is that the MLF_∃ application rule can move existential quantifiers of the function and the argument type to the prefix used in the instance relation, effectively ‘opening’ the existential quantifiers Q_a and Q_b . The result type is generalized again over these existential prefixes Q_a and Q_b , and also over the instantiation prefix Q_3 . There are some cases where the choice of Q_b is ambiguous and we return to this issue later in Section 6 where we discuss principal types.

Actually, the rule for application is expressive enough to not require a special syntactical ‘open’ construct. For example, using the *Key* example from the introduction, we can already accept the application *use absKey*. As explained in the introduction, even though the type α in *absKey* $:: \exists \alpha. Key \ \alpha$ is abstract, the polymorphic *use* function can be applied to it since it works for any type, including any abstract type! By moving the existential type $\exists \alpha$ of *absKey* to the prefix Q_b , we are able to derive that the application *use intKey* is well typed:

$$\begin{array}{c}
(Q) \Gamma \vdash_s use : \forall \beta. Key \beta \rightarrow Int \\
(Q) \Gamma \vdash_s intKey : \exists \alpha. Key \alpha \\
(Q, \exists \alpha) \forall \beta. Key \beta \rightarrow Int \sqsubseteq Key \alpha \rightarrow Int \\
(Q, \exists \alpha) Key \alpha \sqsubseteq Key \alpha \\
\hline
(S-APP) \frac{}{(Q) \Gamma \vdash_s use intKey : \exists \alpha. Int}
\end{array}$$

Here, the type β is instantiated to α , and the type $\exists \alpha. Int$ is simplified to Int . We believe this is the first work that combines automatic existential elimination with automatic instantiation to existential types, removing the need for a special construct for opening existential types.

Note that we could not have instantiated the type of use directly to $\exists \alpha. Key \alpha \rightarrow Int$ since the result prefix Q_3 is required to consist of universally quantified type variables. Indeed, allowing existential quantifiers would be unsound in general, take for example the (unsound) application $f \text{ absKey}$ where f has type $\exists \alpha. Key \alpha \rightarrow Int$.

4.2 Existential arguments

If a function expects an existential argument no opening of existentials is necessary. Take for example the function g that expects an existentially quantified argument:

$$\begin{array}{l}
g :: (\exists \alpha. Key \alpha) \rightarrow Int \\
g \text{ key} = use \text{ key}
\end{array}$$

The application $g \text{ absKey}$ is well typed since the arguments match directly. In particular, as formalized in Section 5.3, the type of g can be ‘instantiated’ to the equivalent type $\forall (\beta = \exists \alpha. Key \alpha). \beta \rightarrow Int$, and the type of absKey to $\forall (\beta = \exists \alpha. Key \alpha). \beta$.

In general, only some existentials might be opened. For example, suppose we have a function f of type:

$$f :: \forall \alpha_1. (\exists \beta_1. \forall \gamma_1. T \alpha_1 \beta_1 \gamma_1) \rightarrow \alpha_1$$

and we supply an argument e of type:

$$e :: \exists \beta_2. \exists \alpha_2. \forall \gamma_2. T \alpha_2 \beta_2 \gamma_2$$

The application $f \ e$ is accepted with type $\exists \alpha. \alpha$. To accept this application, only the $\exists \alpha_2$ should be opened, while the $\exists \beta_2$ should stay quantified to match with the expected argument type. Since this can not be determined by a simple inspection of types, the type inference algorithm presented in Section 7 uses a sophisticated unification algorithm that opens outermost existentials only when necessary.

4.3 Skolemization

Since (S-APP) generalizes the result type over existential bounds, it can elegantly deal with ‘escaping’ existential types. For example, suppose absKey is passed to a function that returns the value field:

$$(\lambda k \rightarrow k.val) intKey$$

The usual treatment of existential types based on skolemization prevents such ‘escaping’ existential types, but in MLF_{\exists} , the application rule just moves opened existential types to and from the prefix, assigning the type $\exists \alpha. \alpha$ to the above expression. Of course, sharing of existential types is maintained, where the expression:

$$(\lambda k \rightarrow (k.val, k.key)) intKey$$

has type $\exists \alpha. (\alpha, \alpha \rightarrow Int)$.

4.4 Annotations and introducing existential types

Surprisingly, we have not given a rule for type annotations or rules for propagating type annotations. In MLF_{\exists} all type annotations can simply be treated as an application of primitives with specific types that behave like the identity function. We assume a set of primitive

annotation functions $\text{annot}\langle \sigma \rangle$ that assign a type σ to their argument. Every type annotation σ is translated into an application of $\text{annot}\langle \sigma \rangle$:

$$\llbracket e :: \sigma \rrbracket = \text{annot}\langle \sigma \rangle e$$

where we assign the following type to the annotation function:

$$\text{annot}\langle \sigma \rangle :: \sigma \rightarrow \sigma$$

where the above type is of course a shorthand for the more explicit type:

$$\text{annot}\langle \sigma \rangle :: \forall (\beta_1 = \sigma). \forall (\beta_2 \geq \sigma). \beta_1 \rightarrow \beta_2$$

Note that the argument β_1 is rigidly bound such that the type of the expression e must be at least as polymorphic as σ . The result type β_2 is has a flexible bound such that the annotated expression can be any instance of σ . Unfortunately, this definition is too restricted to handle existential types. For example, annotating $1 :: \exists \alpha. \alpha$ would fail as the application $\text{annot}\langle \exists \alpha. \alpha \rangle 1$ can not instantiate the type $(\exists \alpha. \alpha) \rightarrow (\exists \alpha. \alpha)$ to $Int \rightarrow (\exists \alpha. \alpha)$.

To deal with existential type annotations, we extend our definition of primitive type annotations to take existential quantifiers into account.

$$\text{annot}\langle \exists \bar{\alpha}. \sigma \rangle :: \forall \bar{\alpha}. \sigma \rightarrow \exists \bar{\alpha}. \sigma$$

where $\bar{\alpha}$ consists of the existential quantifiers of the type, and σ is of the form $\forall Q. \tau$. Again, the above type is a shorthand for:

$$\text{annot}\langle \exists \bar{\alpha}. \sigma \rangle :: \forall \bar{\alpha}. \forall (\beta_1 = \sigma). \forall (\beta_2 \geq \exists \bar{\alpha}. \sigma). \beta_1 \rightarrow \beta_2$$

When the annotation function is applied, it can instantiate the $\bar{\alpha}$ types to any type, while returning a type where these types are existentially quantified. In contrast, the universal quantifiers in σ are rigidly bound such that the type of e must be at least as polymorphic as σ .

Given these annotations, we can already type many examples from Section 2. For example:

$$1 :: \exists \alpha. \alpha$$

or:

$$\{ val = 1, key \ x = x \} :: \exists \alpha. \{ val :: \alpha, key :: \alpha \rightarrow Int \}$$

4.5 Monomorphic annotations and opening existentials

As a final refinement, we fully generalize type annotations to refer to types in the prefix. The *some* quantifier (\exists) stands for an unknown monomorphic type. A full annotation is translated as:

$$\text{annot}\langle \exists Q. \exists \bar{\alpha}. \sigma \rangle :: \forall Q. \forall \bar{\alpha}. \sigma \rightarrow \exists \bar{\alpha}. \sigma$$

This annotation makes the type annotation primitive polymorphic in Q . Such types can potentially be instantiated to types in the prefix. This form of annotation captures many scenarios of lexically scoped type variables [21] but without extending our type rules. Take for example the definition of $runST$:

$$\begin{array}{l}
runST :: \forall \alpha. (\forall s. ST \ s \ \alpha) \rightarrow \alpha \\
runST \ st = \dots
\end{array}$$

If we leave out the type signature, we need to annotate the parameter st with polymorphic type $\forall s. ST \ s \ \alpha$, where we somehow need to refer to the outer polymorphic type variable α . Using the *some* quantifier, we can write:

$$runST \ (st :: \exists \alpha. \forall s. ST \ s \ \alpha) = \dots$$

In particular we can use the ‘some’ quantifier at **let** bindings to refer to opened existential types. Take for example the expression

$$\text{let } x = intKey \text{ in } [x]$$

$$\begin{array}{l}
\text{(R-TRANS)} \quad \frac{(Q) \sigma_1 \oplus \sigma_2 \quad (Q) \sigma_2 \oplus \sigma_3}{(Q) \sigma_1 \oplus \sigma_3} \\
\text{(R-CONTEXT)} \quad \frac{(Q, \nabla(\alpha \diamond \sigma)) \sigma_1 \oplus \sigma_2 \quad \alpha \notin \text{dom}(Q)}{(Q) \nabla(\alpha \diamond \sigma). \sigma_1 \oplus \nabla(\alpha \diamond \sigma). \sigma_2}
\end{array}$$

Figure 3. General properties on type relations, where \oplus stands for \equiv (equivalence), Ξ (abstraction), or \sqsubseteq (instance).

Since the type rule for **let** bindings does nothing at all, the type assigned to this expression is $[\exists \alpha. \text{Key } \alpha]$. If we would have liked to open the existential binding we can use the ‘some’ quantifier to force it to be opened:

```

let  $x :: \exists \alpha. \text{Key } \alpha$ 
     $x = \text{intKey}$ 
in  $[x]$ 

```

A type annotation on a **let** binding is type checked as if it was written as **let** $x = \text{intKey}$ **in** $((\lambda x :: \exists \alpha. \text{Key } \alpha \rightarrow [x]) x)$.³ In this case, the type of the expression becomes $\exists \alpha. [\text{Key } \alpha]$ since the annotation forces the existential quantifier of *intKey* to be opened and moved to the prefix. Since both examples refer to unknown types, lexically scoped type variables can not be used these examples.

4.6 The diamond import problem revisited

As shown in the previous section, we can open existential bindings explicitly at top level using a ‘some’ annotation. In such case, existential quantifiers are moved to the ‘global’ prefix. This means that the existential type is still abstract, but that we can globally share the representation. This effectively solves the ‘diamond import problem’ shown in Section 2.1. For example, we can create an abstract stack from a *listStack* just by giving a type annotation:

```

absStack = listStack ::  $\exists s. \forall \alpha. \text{Stack } s \alpha$ 

```

Since existentials are automatically opened, we can create a collection by directly applying the *newColl* function:

```

newColl absStack ::  $\exists s. \forall \alpha. \text{Coll } s \alpha$  -- inferred

```

Unfortunately, this doesn’t do much good if we are unable to share the implementation type of the collection and stack. To globally ‘import’ an abstract stack, we need to give a type signature that opens the existential at top level:

```

stack ::  $\exists s. \forall \alpha. \text{Stack } s \alpha$ 
stack = absStack

```

The *stack* value gets type $\forall \alpha. \text{Stack } s \alpha$ where $\exists s$ is in the global prefix and can thus be globally shared to create a collection of the same implementation and use the collection and stack together:

```

coll = newColl stack
main = stack.top (coll.fromList [1])

```

Besides being clear and convenient for a programmer, it is appealing that we can treat first-class existential types, automatic opening, and shared existential types with just four simple type rules, and without special mechanisms like skolemization or type propagation rules.

$$\begin{array}{l}
\text{(EQ-CONTEXT)} \quad \frac{(Q) \sigma_1 \equiv \sigma_2}{(Q) \nabla(\alpha \diamond \sigma_1). \sigma \equiv \nabla(\alpha \diamond \sigma_2). \sigma} \\
\text{(EQ-REFL)} \quad (Q) \sigma \equiv \sigma \\
\text{(EQ-FREE)} \quad \frac{\alpha \notin \text{ftv}(\sigma)}{(Q) \nabla(\alpha \diamond \sigma_\alpha). \sigma \equiv \sigma} \\
\text{(EQ-COMM)} \quad \frac{\alpha_1 \notin \text{ftv}(\sigma_2) \quad \alpha_2 \notin \text{ftv}(\sigma_1)}{(Q) \nabla_1(\alpha_1 \diamond_1 \sigma_1) \nabla_2(\alpha_2 \diamond_2 \sigma_2). \sigma \equiv \nabla_2(\alpha_2 \diamond_2 \sigma_2) \nabla_1(\alpha_1 \diamond_1 \sigma_1). \sigma} \\
\text{(EQ-MONO)} \quad \frac{\nabla(\alpha \diamond \sigma_\alpha) \in Q \quad (Q) \sigma_\alpha \equiv \tau_\alpha}{(Q) \tau \equiv \tau[\tau_\alpha/\alpha]} \\
\text{(EQ-VAR)} \quad (Q) \forall(\alpha \diamond \sigma). \alpha \equiv \sigma
\end{array}$$

Figure 4. Equivalence between types.

$$\begin{array}{l}
\text{nf}(\tau) \quad \doteq \tau \\
\text{nf}(\perp) \quad \doteq \perp \\
\text{nf}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) \doteq \text{nf}(\sigma_\alpha) \quad \text{iff } \text{nf}(\sigma) = \alpha \\
\text{nf}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) \doteq \text{nf}(\sigma) \quad \text{iff } \alpha \notin \text{ftv}(\sigma) \\
\text{nf}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) \doteq \text{nf}(\sigma[\tau/\alpha]) \quad \text{iff } \text{nf}(\sigma_\alpha) = \tau \\
\text{nf}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) \doteq \nabla(\alpha \diamond \text{nf}(\sigma_\alpha)). \text{nf}(\sigma)
\end{array}$$

Figure 5. Normal form of types

5. Relations between types

In this section we define formally what it means when two types are in an instance relation (\sqsubseteq). The instance relation is build upon the abstraction relation (Ξ) and the equivalence (\equiv) relation on types.

These three relations share two common properties that are given in Figure 3. These two rules effectively represent six rules where the operator \oplus stands for the equivalence (\equiv), abstraction (Ξ), and instance relation (\sqsubseteq).

The rule (R-TRANS) rule defines these relations as transitive. Rule (R-CONTEXT) allow these relations to operate under the leading prefix of a type. For example, if two types are equivalent as $\sigma_1 \equiv \sigma_2$, we also expect $Q.\sigma_1 \equiv Q.\sigma_2$ to hold.

5.1 Equivalence

The equivalence relation between MLF_\exists types abstracts from syntactical artifacts like the order of quantifiers. For example, we would like to consider the type $\forall \alpha. \forall(\beta = [\alpha]). \beta$ to be equivalent to the type $\forall \alpha. [\alpha]$, or $\exists \alpha. \text{Int}$ to Int . The equivalence relation is defined formally by the rules in Figure 3 and Figure 4.

The first rule (EQ-CONTEXT) extend the equivalence relation into bounds. The rule (EQ-REFL) states that equivalence is reflexive, (EQ-FREE) discards unused quantifiers, (EQ-COMM) states that the order of independent binders does not matter, and (EQ-MONO) inlines monotype bounds. Note that (EQ-MONO) works under any quantifier and bound in our definition. In a system with an more elaborate instance relation over existential bounds, it might be reasonable to retain flexible monotype bounds.

The (EQ-VAR) rule only applies to universal bounds, which is somewhat surprising. If we would have chosen to introduce special existential bounds of the form \leq , we could have defined that a bound $\exists \alpha$ is a shorthand for $\exists(\alpha \leq \top)$. In this case, the

³In practice, we use an equivalent type rule for **let** annotations to handle recursive **let** bindings.

$$\begin{array}{l}
\text{(A-CONTEXT)} \quad \frac{(Q) \sigma_1 \sqsubseteq \sigma_2}{(Q) \forall(\alpha = \sigma_1). \sigma \sqsubseteq \forall(\alpha = \sigma_2). \sigma} \\
\text{(A-EQUIV)} \quad \frac{(Q) \sigma_1 \equiv \sigma_2}{(Q) \sigma_1 \sqsubseteq \sigma_2} \\
\text{(A-HYP)} \quad \frac{\forall(\alpha = \sigma) \in Q}{(Q) \sigma \sqsubseteq \alpha}
\end{array}$$

Figure 6. Type abstraction.

rule (EQ-VAR) would apply to existential quantifiers too where we could derive $\top \equiv \exists(\alpha \leq \top). \alpha$. However, we feel that the current presentation is closer to the original MLF type system and is better suited for this paper.

Since existential quantifiers always have unconstrained bounds, we can use rule (EQ-COMM) (and α -renaming) to always move all those quantifiers to the front of the prefix. Therefore, we have for any type σ that $\sigma \equiv \exists Q_1. \forall Q_2. \tau$ for some Q_1, Q_2 , and τ . When we write equivalence in such form, we assume without loss of generality that we use (EQ-VAR) and (EQ-FREE) to expose the true prefix, i.e. a type as $\forall(\alpha \geq \forall\beta\gamma. [\gamma]). \alpha$ is viewed as $\forall\gamma. [\gamma]$.

Using the equivalence relation, we can define a *normal form* of each type, and we can define an algorithm to compute the normal form of a type, as shown in Figure 5. The normal form algorithm basically simplifies *trivial* bounds away. From the definition we can see that there are three forms of trivial bounds: the quantified variable is bound by a monotype, the variable does not occur in the body of the type, or if the body of a universal quantified type variable is the variable itself.

THEOREM 1. *The normal form of a type is always equivalent to the type itself: $\text{nf}(\sigma) \equiv \sigma$*

As shown in Section 2.2.3, the normal form of a type is important in practice when presenting complicated MLF_{\exists} types to users.

5.2 Abstraction

In MLF, an annotation is needed for lambda bound variables that are used polymorphically. The lambda bound variable is assigned a type variable that has a rigid bound to a polymorphic type. In effect, the type variable is an *abstraction* of that polymorphic type. The abstraction relation allows us to reason formally about type abstractions.

The abstraction relation is defined in Figure 3 and Figure 6. We can read an abstraction $(Q) \sigma_1 \sqsubseteq \sigma_2$ as ‘ σ_2 is an abstraction of σ_1 ’. The rule (A-CONTEXT) extends the abstraction relation through universally quantified rigid bounds. Equivalence is included through (A-EQUIV), while the hypothesis rule (A-HYP) replaces a polytype by a type variable, provided that it is rigidly bound.

5.3 Instance

The instantiation relation (\sqsubseteq) defines when two types are instance of each other, for example $\forall\alpha. [\alpha] \sqsubseteq [\text{Int}]$. The instance relation is defined in Figure 3 and Figure 7.

The rule (I-CONTEXT) extends the instance relation through universally quantified flexible bounds. The rule (I-ABSTRACT) includes the abstraction relation. The hypothesis rule (I-HYP) instantiates flexible bounds, and (I-RIGID) instantiates flexible bound to rigid bounds. The rule (I-BOT) instantiates \perp to any type. Using (I-BOT) we can for example derive:

$$\begin{array}{l}
\forall(\alpha \geq \perp). \alpha \\
\equiv \{(\text{EQ-FREE})\}
\end{array}$$

$$\begin{array}{l}
\text{(I-CONTEXT)} \quad \frac{(Q) \sigma_1 \sqsubseteq \sigma_2}{(Q) \forall(\alpha \geq \sigma_1). \sigma \sqsubseteq \forall(\alpha \geq \sigma_2). \sigma} \\
\text{(I-ABSTRACT)} \quad \frac{(Q) \sigma_1 \sqsubseteq \sigma_2}{(Q) \sigma_1 \sqsubseteq \sigma_2} \\
\text{(I-HYP)} \quad \frac{\forall(\alpha \geq \sigma) \in Q}{(Q) \sigma \sqsubseteq \alpha} \\
\text{(I-RIGID)} \quad (Q) \forall(\alpha \geq \sigma_\alpha). \sigma \sqsubseteq \forall(\alpha = \sigma_\alpha). \sigma \\
\text{(I-BOTTOM)} \quad (Q) \perp \sqsubseteq \sigma
\end{array}$$

Figure 7. Type instance.

$$\begin{array}{l}
\forall\beta. \forall(\alpha \geq \perp). \alpha \\
\sqsubseteq \{(\text{R-CONTEXT}) \wedge (\text{I-CONTEXT}) \wedge (\text{I-BOT})\} \\
\forall\beta. \forall(\alpha \geq \beta \rightarrow \beta). \alpha \\
\equiv \{(\text{R-CONTEXT}) \wedge (\text{EQ-MONO}) \wedge (\text{EQ-FREE})\} \\
\forall\beta. \beta \rightarrow \beta
\end{array}$$

and:

$$\begin{array}{l}
\forall(\beta \geq \perp). \beta \rightarrow \beta \\
\sqsubseteq \{(\text{I-BOT}) \wedge (\text{R-CONTEXT-FLEXIBLE})\} \\
\forall(\beta \geq \text{Int}). \beta \rightarrow \beta \\
\equiv \{(\text{EQ-MONO}) \wedge (\text{EQ-FREE})\} \\
\text{Int} \rightarrow \text{Int}
\end{array}$$

There are some interesting cases to consider when using existential annotations. For example, suppose we have an expression e of type $[\exists\alpha. \alpha]$. Surprisingly, the annotated expression $e :: \exists\alpha. [\alpha]$ is actually well-typed. Since the annotation unifies the type of e with $\forall\alpha. [\alpha]$, we can derive:

$$\begin{array}{l}
\forall\alpha. [\alpha] \\
\equiv \{syntax\} \\
\forall(\alpha \geq \perp). [\alpha] \\
\sqsubseteq \{(\text{I-BOTTOM}) \wedge (\text{I-CONTEXT})\} \\
\forall(\alpha \geq \exists\beta. \beta). [\alpha] \\
\equiv \{syntax \wedge renaming\} \\
[\exists\alpha. \alpha]
\end{array}$$

Of course, this particular case is only sound since the existential type has no structure. If e had type $[\exists\alpha. (\alpha, \alpha \rightarrow \text{Int})]$ for example, the annotation $e :: \exists\alpha. [(\alpha, \alpha \rightarrow \text{Int})]$ is unsound and rejected (and we can not derive an instance this time).

5.4 Existential type instantiation

It seems surprising that we have only defined an equivalence relation on existential types – the abstraction and instance relation are not extended to existential quantifiers. However, by rule (EQ-REFL) we can handle equivalent existential types, and through rule (I-BOTTOM), we can still instantiate unconstrained universally quantified type variables to existential types. This proves to be ‘just enough’ in practice. In particular, for any type σ , we have:

$$\text{(I-EXISTS)} \quad \forall\alpha. \sigma \sqsubseteq \exists\alpha. \sigma$$

The derivation is:

$$\begin{array}{l}
\forall\alpha. \sigma \\
\equiv \{syntax\} \\
\forall(\alpha \geq \perp). \sigma \\
\equiv \{(\text{EQ-FREE})\} \\
\exists\beta. \forall(\alpha \geq \perp). \sigma \\
\sqsubseteq \{(\text{I-BOTTOM}) \wedge (\text{I-CONTEXT}) \wedge (\text{R-CONTEXT})\}
\end{array}$$

$$\begin{array}{c}
\text{(P-VAR)} \quad \frac{x : \sigma \in \Gamma}{(Q) \Gamma \vdash_p x : \sigma} \\
\text{(P-LET)} \quad \frac{(Q) \Gamma \vdash_p e_1 : \sigma_1 \quad (Q) \Gamma, x : \sigma_1 \vdash_p e_2 : \sigma_2}{(Q) \Gamma \vdash_p \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \sigma_2} \\
\text{(P-LAM)} \quad \frac{\alpha \notin \text{ftv}(\tau) \quad \text{dom}(Q') \not\cap \text{ftv}(\Gamma) \quad (QQ') \Gamma, x : \tau \vdash_p e : \sigma}{(Q) \Gamma \vdash_g \lambda x. e : (Q', \forall(\alpha \geq \sigma)). \tau \rightarrow \alpha} \\
\text{(P-APP)} \quad \frac{(Q) \Gamma \vdash_p f : \exists Q_a. \sigma_1 \quad (Q) \Gamma \vdash_p e : \exists Q_b. \sigma_2 \quad (QQ_a Q_b) \sigma_1 \sqsubseteq \forall Q_3. \tau_3 \rightarrow \tau \quad (QQ_a Q_b) \sigma_2 \sqsubseteq \forall Q_3. \tau_3}{(Q) \Gamma \vdash_a f \ e : (\exists Q_a Q_b, Q_3. \tau)} \\
\text{(P-EX)} \quad \frac{(Q) \Gamma \vdash_a f \ e : (\exists Q_a, \sigma) \quad \forall Q_0. \forall \sigma_0. (Q) \Gamma \vdash_a f \ e : (\exists Q_0, \sigma_0) \Rightarrow |Q_a| \leq |Q_0|}{(Q) \Gamma \vdash_g f \ e : Q_a. \sigma} \\
\text{(P-GEN)} \quad \frac{\forall \sigma_0. (Q) \Gamma \vdash_g e : \sigma_0 \Rightarrow (Q) \sigma \sqsubseteq \sigma_0 \quad (Q) \Gamma \vdash_g e : \sigma}{(Q) \Gamma \vdash_p e : \sigma}
\end{array}$$

Figure 8. Principal syntax directed typing rules

$$\begin{aligned}
& \exists \beta. \forall(\alpha \geq \beta). \sigma \\
& \equiv \{(\text{EQ-MONO}) \wedge \alpha\text{-rename}\} \\
& \exists \alpha. \sigma
\end{aligned}$$

6. Principal types

The MLF_{\exists} type rules can sometimes derive different types for the same expression. For example, for the expression $\lambda x \rightarrow x$ we can derive the type $\forall \alpha. \alpha \rightarrow \alpha$, but also $\text{Int} \rightarrow \text{Int}$, or $\text{Bool} \rightarrow \text{Bool}$. This poses a challenge for a type inference algorithm: what type should it infer for this expression? Fortunately, there is usually a ‘best choice’ that we can make. In particular, a Hindley-Milner style type inference algorithm always infers ‘most general types’.

In MLF_{\exists} however there are more choices to be made than in Hindley-Milner. In the application rule we can choose how many existentials to open in the argument as prefix Q_b . For example, suppose we have the following two values:

$$\begin{aligned}
\text{wrap} &:: \forall \alpha. \alpha \rightarrow [\alpha] \\
\text{key} &:: \exists \beta. \text{Key } \beta
\end{aligned}$$

For the application $\text{wrap } \text{key}$ we can derive the types:

$$[\exists \alpha. \text{Key } \alpha \mid$$

and

$$\exists \alpha. [\text{Key } \alpha \mid$$

In the first derivation, no existential is opened, and the type of wrap is simply instantiated to $\forall(\alpha \geq \exists \beta. \text{Key } \beta). \alpha \rightarrow [\alpha]$. In the second derivation the existential quantifier of key is opened and wrap is instantiated to $\text{Key } \beta \rightarrow [\text{Key } \beta]$ where β existentially bound in the prefix Q_b .

Both of the derived types are sound but they are not in an instance relation, i.e. the original type rules do not have principal types. For a type inference algorithm, we wish to give more restricted set of type rules where only one of the above types is derivable. Such restricted type rules can be used as a clear specification of a practical type inference algorithm.

Fortunately, there is still a ‘best choice’ that we can make. We argue that the first type is always preferred where no existentials are opened. If it is desired to open the existential type, we can give a type signature to force the second type, as in $(\text{wrap}::\forall \alpha. \text{Key } \alpha \rightarrow [\text{Key } \alpha]) \ \text{key}$. Indeed, if we choose the other way around, we would never be able to derive the first type. Intuitively, we want to treat existentials as abstract as long as possible, just as we keep types polymorphic as long as possible. In effect, MLF_{\exists} ‘only opens existentials when necessary’.

We can enforce this rule by adding a side condition to (S-APP) requiring that Q_b is always the shortest possible prefix, effectively only opening existentials when necessary.

Before we formalize the new requirement, we need to make one more refinement to restore principal types. Take the following application where the definition of wrap is expanded: $(\lambda x \rightarrow [x]) \ \text{key}$. If we derive the type $\forall \alpha. \alpha \rightarrow [\alpha]$ for the sub expression $(\lambda x \rightarrow [x])$, the new side condition ensures we can only derive the type $[\exists \alpha. \text{Key } \alpha]$ the expression. However, the type rules also allow derivations of *less general* types, and we can also derive $(\lambda x \rightarrow x) : \forall \alpha. \text{Key } \alpha \rightarrow [\text{Key } \alpha]$, leading to the type $\exists \alpha. [\text{Key } \alpha]$ again.

The solution is to modify the typing rules such that only most general types can be derived. The final revised principal type rules are given in Figure 8. The new \vdash_p relation derives most general principal types. The rules for variables and let bindings are unchanged from the original syntax directed rules. The rule for lambda abstractions derives a type using the \vdash_g relation. Such type may not be most general, and the new rule (S-GEN) ensures that only most general types that can be derived resulting in a \vdash_p relation again. The application rule (S-APP) derives a type using \vdash_a . The existential rule (S-EX) takes such derivation and ensures that existentials are only opened when necessary. Since (S-EX) derives a \vdash_g relation, the rule (S-GEN) is used again to ensure most general types. Since we only added restrictions, the principal rules are a sound (but incomplete) extension of the original type rules.

We are not the first to suggest restrictions on the type rules to restore completeness of type inference: Vytiniotis, Weirich and Peyton Jones use the same solution for boxy type inference [26] as implemented in GHC. Similar ideas were also used by Garrigue and Rémy in their extension of ML with semi-explicit first-class polymorphism [4], and by Leroy and Mauny in the context of dynamics in ML [15].

The addition of the rules (P-EX) and (P-GEN) rule has complicated the type rules, and one might ask in how far this is justified. Would an explicit open construct be better where no ambiguity can occur? We argue that this is not the case. Even though an explicit open construct leads to more natural type rules, it is hard to program with and, as shown earlier, forces a rigid block structure on programs. For a programmer it is more convenient when the type inferencer opens existentials automatically, corresponding intuitively with automatic instantiation of polymorphic types.

Furthermore, the new conditions may be complicated from a theoretical perspective, but are easy to explain in practice: ‘ MLF_{\exists} only opens existential types when necessary’. Our goal here is to find a set of syntax directed type rules that give a programmer a clear understanding of the type system without knowledge of a particular inference algorithm.

6.1 Properties

Note that, as described in Section 5.1 we assume without loss of generality that rules (EQ-VAR) and (EQ-FREE) have been applied to expose the structure of a prefix when writing an equivalence $\sigma \equiv \exists Q_1. \forall Q_2. \tau$.

(INF-VAR)	$\frac{x : \sigma \in \Gamma}{(Q) \Gamma \vdash x : (Q, \sigma)}$
(INF-LET)	$\frac{(Q) \Gamma \vdash e_1 : (Q_1, \sigma_1) \quad (Q_1) \Gamma, x : \sigma_1 \vdash e_2 : (Q_2, \sigma_2)}{(Q) \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : (Q_2, \sigma_2)}$
(INF-LAM)	$\frac{\text{fresh } (\alpha, \beta) \quad (Q, \forall(\alpha \geq \perp)) \Gamma, x : \alpha \vdash e : (Q_1, \sigma) \quad (Q_2, Q_3) = Q_1 \uparrow \text{dom}(Q)}{(Q) \Gamma \vdash \lambda x. e : (Q_2, (Q_3, \forall(\beta \geq \sigma)). \alpha \rightarrow \beta)}$
(INF-APP)	$\frac{\text{fresh } (\alpha_1, \alpha_2, \beta) \quad (Q) \Gamma \vdash e_1 : (Q_1, \sigma_1) \quad (Q_1) \Gamma \vdash e_2 : (Q_2, \sigma_2) \quad Q' = (Q_2, \forall(\alpha_1 \geq \sigma_1), \forall(\alpha_2 \geq \sigma_2), \forall(\beta \geq \perp)) \quad (Q', ()) \alpha_1 \sim \alpha_2 \rightarrow \beta : (Q_3, ()) \quad (Q_4, Q_5) = Q_3 \uparrow \text{dom}(Q)}{(Q) \Gamma \vdash e_1 e_2 : (Q_4, Q_5, \beta)}$

Figure 9. Type inference.

THEOREM 2. *In rule (P-APP), the choice for Q_a is unique and contains all existential quantifiers, i.e. $\exists Q_a. \sigma_1 \equiv \exists Q_a. \forall Q_1. \tau_1$ for some Q_1 and τ_1 .*

PROOF. We have that $(Q Q_a Q_b) \sigma_1 \sqsubseteq \forall Q_3. \tau_3 \rightarrow \tau$. Suppose that Q_a would not contain all existential quantifiers but that σ_1 has the form $\exists \alpha. \sigma$. This means we must either remove the existential bound, or put it under a universal bound in order to instantiate to $\forall Q_3. \tau_3 \rightarrow \tau$. Since the result is not a type variable, we can not use the rules (EQ-VAR), (A-HYP), or (I-HYP). The only applicable rule that can discard the existential quantifier is (EQ-FREE). This rule only applies of $\alpha \notin \text{ftv}(\sigma_1)$, and in that case it would not matter if we had moved it to Q_a . Therefore, there is a unique best choice for Q_a and it must contain all existential quantifiers. \square

The following lemma states that the instantiation relation can only introduce outer existential quantifiers.

LEMMA 1. *If $(Q) \sigma_1 \sqsubseteq \sigma_2$ where $\sigma_1 \equiv \exists Q_a. \forall Q_1. \tau_1$, implies that $\sigma_2 \equiv \exists Q_a. \exists Q_b. \forall Q_2. \tau_2$, for some Q_b .*

PROOF. By inspection on all possible instantiation rules \square

THEOREM 3. *In rule (P-APP), there exists a unique shortest possible prefix Q_b .*

PROOF. See Appendix B. \square

7. Type inference

Type inference in MLF_{\exists} is possible since all existential and polymorphic types are introduced through type annotations. Since these are presented as primitive operations in the initial typing environment, the type inference algorithm never ‘guesses’ existential or polymorphic types, but only propagates known type information. The type inference algorithm is presented in Figure 9.

The rules for variables and **let** bindings just propagate types. The inference rule for lambda expressions introduces a fresh type α for the argument type, and generalizes over the result type using the split algorithm (\uparrow), see appendix A and [11, 13] for details. The application rule is the most interesting, and unifies the type of the argument with the expected argument type. Finally, it generalizes over the result type using the split algorithm.

(U-VAR)	$(Q, \mathcal{E}) \alpha \sim \alpha : (Q, \mathcal{E})$
(U-CON)	$\frac{(Q_{i-1}, \mathcal{E}_{i-1}) \tau_i \sim \tau'_i : (Q_i, \mathcal{E}_i) \text{ for } i \in 1..n}{(Q_0, \mathcal{E}_0) c \tau_1 .. \tau_n \sim c \tau'_1 .. \tau'_n : (Q_n, \mathcal{E}_n)}$
(U-MVAR-L)	$\frac{\forall(\alpha \diamond \sigma) \in Q \quad \sigma \in \mathcal{V} \quad (Q, \mathcal{E}) \tau \sim \text{nf}(\sigma) : (Q', \mathcal{E}')}{(Q, \mathcal{E}) \tau \sim \alpha : (Q', \mathcal{E}')}$
(U-MONO-L)	$\frac{\forall(\alpha \diamond \sigma) \in Q \quad \tau \notin \mathcal{V} \quad \sigma \notin \mathcal{V} \quad \alpha \notin \text{dom}(Q / \tau) \quad (Q, \mathcal{E}) \tau \sim_p \sigma : (Q', -)}{(Q, \mathcal{E}) \tau \sim \alpha : (Q' \triangleleft \forall(\alpha = \tau), \mathcal{E})}$
(U-EXISTS-L)	$\frac{\exists \alpha_1 \in Q \quad \forall(\alpha_2 \geq \sigma_2) \in Q \quad \text{cf}(\sigma_2) = \perp}{(Q, \mathcal{E}) \alpha_1 \sim \alpha_2 : (Q \triangleleft \forall(\alpha_2 = \alpha_1), \mathcal{E})}$
(U-EXISTS)	$\frac{\exists \alpha_1 \in Q \quad \exists \alpha_2 \in Q \quad \alpha_1 \neq \alpha_2 \quad \mathcal{E}' = \mathcal{E} \setminus \{\alpha_1, \alpha_2\}}{(Q, \mathcal{E}) \alpha_1 \sim \alpha_2 : (Q \triangleleft \forall(\alpha_1 = \alpha_2), \mathcal{E}')}$
(U-POLY)	$\frac{\forall(\alpha_1 \diamond_1 \sigma_1) \in Q \quad \forall(\alpha_2 \diamond_2 \sigma_2) \in Q \quad \alpha_1 \neq \alpha_2 \quad \sigma_1 \notin \mathcal{V} \quad \sigma_2 \notin \mathcal{V} \quad \alpha_1 \notin \text{dom}(Q / \sigma_2) \quad \alpha_2 \notin \text{dom}(Q / \sigma_1) \quad (Q, \mathcal{E}) \sigma_1 \sim_p \sigma_2 : (Q_3, \mathcal{E}', \sigma_3) \quad Q' = Q_3 \triangleleft \forall(\alpha_1 \diamond_1 \sigma_1) \triangleleft \forall(\alpha_2 \diamond_2 \sigma_2) \triangleleft (\alpha_1 \wedge \alpha_2)}{(Q, \mathcal{E}) \alpha_1 \sim \alpha_2 : (Q', \mathcal{E}')}$

Figure 10. Monotype unification. All the left (-L) rules have a corresponding right (-R) rule with the arguments swapped.

The opening of existential types is performed by the unification algorithm, and not by the application rule directly. The reason is that existentials should only be opened as necessary, and this can only be determined during unification. Using the example from Section 4, suppose we have a function f of type:

$$f :: \forall \alpha_1. (\exists \beta_1. \forall \gamma_1. T \alpha_1 \beta_1 \gamma_1) \rightarrow \alpha_1$$

and we supply an argument e of type:

$$e :: \exists \beta_2. \exists \alpha_2. \forall \gamma_2. T \alpha_2 \beta_2 \gamma_2$$

than, the application $f e$ is accepted with type $\exists \alpha. \alpha$. To accept this application, only the $\exists \alpha_2$ should be opened, while the $\exists \beta_2$ should stay quantified to match with the expected argument type. Since this can not be determined by a simple inspection of types, we leave the opening of existentials to the unification algorithm, where we are careful to only open existentials when they occur on the outermost level of a type (as required by (S-APP)).

7.1 Unification

The unification algorithm extends the one in MLF. In particular, for monotype unification we replace the last rule by the last three rules in Figure 10 that handle all combinations of unifying universal and existential quantifiers. The polytype unification algorithm is extended with an extra check on ‘escaping’ existentials (Figure 11).

The rules (U-EXISTS-L) and (U-EXISTS-R) unify existential quantifiers with unconstrained universal quantifiers. These rules basically implement the (I-BOTTOM) relation on existentially quantified type variables. As we see in (U-EXISTS-L), the unconstrained universal bound $\forall(\alpha_2 \geq \perp)$ is updated to equal the existential bound, $\forall(\alpha_2 = \alpha_1)$. The update algorithm \triangleleft is specified in Appendix A.

$$\begin{array}{l}
\text{(POLY-BOT-L)} \quad (Q, \mathcal{E}) \perp \sim_p \sigma : (Q, \mathcal{E}, \sigma) \\
\text{(POLY-BOT-R)} \quad (Q, \mathcal{E}) \sigma \sim_p \perp : (Q, \mathcal{E}, \sigma) \\
\frac{\begin{array}{l} \not\exists \bar{\alpha}_1. \forall Q'_1 \equiv Q_1 \quad \not\exists \bar{\alpha}_2. \forall Q'_2 \equiv Q_2 \\ (Q_1 Q_2, (\mathcal{E}, \bar{\alpha}_1 \leftrightarrow \bar{\alpha}_2)) \tau_1 \sim \tau_2 : (Q_3, (\mathcal{E}', -)) \\ (Q_4, Q_5) = Q_3 \uparrow \text{dom}(Q) \\ \mathcal{E} = () \vee (\bar{\alpha}_1 \cup \bar{\alpha}_2) \not\exists \text{dom}(Q_4) \end{array}}{\text{(POLY)} \quad \frac{(Q, \mathcal{E}) (Q_1. \tau_1) \sim_p (Q_2. \tau_2) : (Q_4, \mathcal{E}', Q_5. \tau_1)}{}}
\end{array}$$

Figure 11. Polytype unification. The arguments are assumed to be in constructed form.

$$\begin{array}{l}
\frac{\alpha_1 \in A_1 \quad \alpha_2 \in A_2}{(\mathcal{E}, A_1 \leftrightarrow A_2) \setminus \{\alpha_1, \alpha_2\} \doteq (\mathcal{E}, (A_1 - \alpha_1) \leftrightarrow (A_2 - \alpha_2))} \\
\frac{\alpha_2 \in A_1 \quad \alpha_1 \in A_2}{(\mathcal{E}, A_1 \leftrightarrow A_2) \setminus \{\alpha_1, \alpha_2\} \doteq (\mathcal{E}, (A_1 - \alpha_2) \leftrightarrow (A_2 - \alpha_1))} \\
\frac{\{\alpha_1, \alpha_2\} \not\exists (A_1 \cup A_2) \quad \mathcal{E}' = \mathcal{E} \setminus \{\alpha_1, \alpha_2\}}{(\mathcal{E}, A_1 \leftrightarrow A_2) \setminus \{\alpha_1, \alpha_2\} \doteq (\mathcal{E}', A_1 \leftrightarrow A_2)}
\end{array}$$

Figure 12. Existential association. Fails if none of the above rules apply.

The rule (U-EXISTS) unifies two existential types. Of course, two arbitrary existential quantifiers can not unify (try for example $\exists \alpha. [T \ \alpha]$ and $[\exists \alpha. T \ \alpha]$). In particular, we only have an equivalence relation on existential types, and a unification of existential quantifiers should correspond to a renaming. A sufficient condition is that existential quantifiers can be unified if they occur at the same level and if they are in a one-to-one correspondence. For example, we can unify α_1 and α_2 in:

$$\exists \alpha_1. T \ \alpha_1 \equiv \exists \alpha_2. T \ \alpha_2$$

but not in:

$$\exists \alpha_1. T \ (\alpha_1, \alpha_1) \not\equiv \exists \alpha_2. \exists \beta. T \ (\alpha_2, \beta)$$

We implement this by passing an existential association environment \mathcal{E} to both monotype and polytype unification. This is a list of tuples where each tuple contains two existential type variable sets that are allowed to unify with each other. Rule (POLY) in Figure 11 extends this environment with the existential quantifiers of the unified polytypes. The rule (U-EXISTS) in Figure 10 uses this environment to check if two existential quantifiers are allowed to unify with each other and returns an environment where these two quantifiers are removed, ensuring that the existential quantifiers unify one-to-one. The algorithm that checks the association is given in Figure 12.

The rule (POLY) for polytype unification is structured like the one in plain MLF but also checks if all existentials are directly quantified and do not escape. However, the (S-APP) rule allows existentials at the outer level to be opened, and (POLY) therefore allows existentials to escape if the existential association environment is empty ($\mathcal{E} = ()$), which is only true at the outermost level when unification is invoked from (INF-APP).

The update algorithm in Appendix A makes use of the abstraction algorithm ($\Xi?$). It is beyond the scope of this article to discuss this algorithm in detail and the interested reader is referred to the thesis of LeBotlan [10]. However, the abstraction check calculates of weight polynomes over polytypes and this weight calculation

needs to take existential quantifiers into account. Since there is no abstraction or instantiation over existential quantifiers it is sufficient to assign Z weights to existential bounds. This corresponds nicely to the church encoding of existential types:

$$\llbracket \exists \alpha. \sigma \rrbracket = \forall \beta. (\forall \alpha. \sigma \rightarrow \beta) \rightarrow \beta$$

where the existential quantifier always gets a Z weight since it is multiplied by both a flexible and rigid bound: $((w \star =) \star \geq) = Z$.

7.2 Soundness and completeness

Soundness and completeness of type inference relies on soundness and completeness results for unification. Since the algorithm is extremely similar to the one in MLF, most of the proofs for MLF [10] carry over directly to MLF_{\exists} with small variations.

LEMMA 2. *Unification terminates.*

PROOF. The termination of the unification algorithm follows directly from the proof of the MLF unification algorithm and inspection of the added rules. For the added rules it suffices to notice that for (U-EXISTS-L) we have that $\alpha_2 \notin \text{dom}(Q / \alpha_1)$ since $\text{cf}(\sigma_2) = \perp$, and the same for (U-EXISTS). \square

Soundness of unification is proved by showing that if the unification algorithm returns a prefix, it is an instance of the initial prefix and unifies the input types. For polytype unification, we show that the returned prefix is an instance of the initial prefix, and that the returned polytype is an instance of both input types, and thus the least upper bound of both.

LEMMA 3. *Unification is sound:*

$$\begin{array}{l}
(Q, \mathcal{E}) \tau_1 \sim \tau_2 : (Q', \mathcal{E}') \Rightarrow Q \sqsubseteq Q' \wedge (Q') \tau_1 \equiv \tau_2 \\
(Q, \mathcal{E}) \sigma_1 \sim_p \sigma_2 : (Q', \mathcal{E}', \sigma) \\
\Rightarrow Q \sqsubseteq Q' \wedge (Q') \sigma_1 \sqsubseteq \sigma \wedge (Q') \sigma_2 \sqsubseteq \sigma
\end{array}$$

PROOF. See Appendix B. \square

LEMMA 4. *Unification is complete: for two types τ_1, τ_2 , and a prefix Q , if there exists a prefix Q' such that $(Q') \tau_1 \equiv \tau_2$ where $Q \sqsubseteq Q'$, we have $(Q, -) \tau_1 \sim \tau_2 : (Q'', -) \wedge Q'' \sqsubseteq Q'$.*

We have not done a formal proof of completeness yet. However since MLF unification is complete, and since there is no instantiation relation on existential types, the only source of incompleteness would be in the choice of which existentials escape, i.e. Q_a and Q_b in (P-APP). By following the proofs of Theorems 2 and 3, we can see that there is in all cases either a unique or a best choice. In particular, we should always unify with existential types when possible (instead of letting them escape). This is exactly the case with rules (U-EXISTS-L), (POLY-BOT-R), and (POLY-BOT-L) that all unify with existential types whenever possible. Therefore, we have good evidence to believe that the unification algorithm is complete with respect to the principal syntax directed rules.

The soundness and completeness of the type inference algorithm with respect to the syntax directed type rules follows almost directly from the above lemmas and the similar proofs of MLF since the type inference rules are equivalent.

LEMMA 5. *Type inference is sound; for any Q, Γ , and e , where $(Q) \Gamma \vdash e : (Q', \sigma)$, we have:*

$$Q \sqsubseteq Q' \wedge (Q') \Gamma \vdash_p e : \sigma$$

LEMMA 6. *Type inference is complete; if $(Q) \Gamma \vdash_p e : \sigma$, then $(Q) \Gamma \vdash e : (Q', \sigma')$ succeeds and returns a principal solution, $Q'. \sigma' \sqsubseteq Q. \sigma$.*

8. Related work

Cardelli en Wegner [2] wrote a comprehensive tutorial paper that explored the interaction between existential and universal quantification in 1985. Mitchell and Plotkin considered abstract types modeled as existential types [19]. Later Läufer and Odersky [9], Jones [7], Remy [23], and Simonet [25] describe how data types can be used to combine type inference with existential types, as shown in Section 2.1. This approach has been widely adopted in compilers for Haskell and ML [17]. Läufer shows that existential types combine well with qualified types [8].

MacQueen [16] observed that existential types with an explicit elimination construct is impractical for modeling abstract types in ML modules and proposes to use a system based on dependent types. Cardelli and Leroy propose a ‘dot notation’ for existential types to overcome some of these problems [1]. Later Leroy describes a system that models abstract types using ‘manifest types’ [14] which does not require dependent types. Jones [6] argues that the dot-notation does not behave well under simple rewriting, and proposes a solution based on parameterized signatures combined with existential types. We believe that our type inference system can be used as a foundation to implement many of the ideas put forward in that paper. Shields and Peyton-Jones investigate first-class modules for Haskell [24] and describe a type system with predicative existential types.

Extending to Hindley-Milner [5, 18] to practical higher-rank type inference was described by Odersky and Läufer [20] and Peyton-Jones *et al.* [22]. LeBotland and Remy first describe type inference for impredicative higher-rank type systems in the context of MLF [10, 11], which was extended to support qualified types by Leijen and Löh [13]. Recent work by Dijkstra [3] describes a practical type inference system with higher-ranked types that is extended with existential types. To our knowledge, this is the only work that also allows existential quantifiers as first-class citizens, but typically requires more type annotations.

9. Conclusion

We have shown how to extend MLF with first-class existential types. Existential quantifier introduction is explicit through type annotations and existential quantifier elimination is inferred. We believe that this makes working with existential types more natural and hope it leads to new abstractions and programming styles.

Acknowledgments

We wish to thank Conal Elliott, Simon Peyton Jones, Andres Löh, Benjamin Pierce, Didier Remy, Wolfram Schulte, and Wouter Swierstra for their valuable comments on an earlier version of this paper.

References

- [1] L. Cardelli and X. Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990. Also available as research report 56, DEC Systems Research Center.
- [2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
- [3] A. Dijkstra. *Stepping through Haskell*. PhD thesis, Universiteit Utrecht, Nov. 2005.
- [4] J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Journal of Information and Computation*, 155:134–169, 1999.
- [5] J. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.
- [6] M. Jones. Using parameterized signatures to express modular structure. In *Proc. of the Twenty Third Annual ACM Symposium on Principles of Programming Languages*, Jan. 1996.
- [7] M. P. Jones. First-class polymorphism with type inference. In *Proceedings of the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 1997.
- [8] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.
- [9] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *Trans. on Programming Languages and Systems*, 1994.
- [10] D. Le Botlan. *ML^F: Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, INRIA Rocquencourt, May 2004. Available in English.
- [11] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003)*, Uppsala, Sweden, pages 27–38. ACM Press, aug 2003.
- [12] D. Leijen. *Morrow: a row-oriented programming language*. <http://www.equational.org/morrow>, Jan. 2006.
- [13] D. Leijen and A. Löh. Qualified types for MLF. In *The International Conference on Functional Programming (ICFP’05)*. ACM Press, Sept. 2005.
- [14] X. Leroy. Manifest types, modules, and separate compilation. In *POPL ’94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122. New York, NY, USA, 1994. ACM Press.
- [15] X. Leroy and M. Mauny. Dynamics in ML. In *ACM conference on Functional Programming and Computer Architecture (FPCA’91)*. Springer-Verlag, 1991. volume 523 of LNCS.
- [16] D. MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 277–286, 1986.
- [17] M. Mauny and F. o. Pottier. An implementation of Caml-light with existential types. Technical report, INRIA-Rocquencourt, Oct. 1993.
- [18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:248–375, Aug. 1978.
- [19] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
- [20] M. Odersky and K. Läufer. Putting type annotations to work. In *23th ACM Symp. on Principles of Programming Languages (POPL’96)*, pages 54–67, Jan. 1996.
- [21] S. Peyton Jones and M. Shields. Lexically scoped type variables. Unpublished, Mar. 2004.
- [22] S. Peyton-Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. Submitted to the Journal of Functional Programming (JFP), 2006.
- [23] D. Remy. Programming objects with ML-ART, an extension to ml with abstract and record types. In *TACS ’94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 321–346. London, UK, 1994. Springer-Verlag.
- [24] M. Shields and S. Peyton-Jones. First-class modules for Haskell. In *Ninth International Conference on Foundations of Object-Oriented Languages (FOOL 9)*, Portland, Oregon, Dec. 2001.
- [25] V. Simonet. An extension of HM(X) with first-class existential and universal datatypes. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 39–50. ACM Press, Aug. 2005.
- [26] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy types: type inference for higher-rank types and impredicativity. In *The International Conference on Functional Programming (ICFP’06)*, Sept. 2006.

A. Supplemental algorithms

Useful domain:

$$\begin{aligned} \text{dom}(Q / \sigma) &\doteq \text{dom}(Q / \text{ftv}(\sigma)) \\ \alpha \in \text{dom}(Q / \beta_1 \dots \beta_n) \\ &\Leftrightarrow \\ Q &= (Q_1, \nabla(\alpha \diamond \sigma), Q_2) \wedge \alpha \in \text{ftv}(Q_2. \beta_1 \rightarrow \dots \rightarrow \beta_n \rightarrow ()) \end{aligned}$$

Constructed forms:

$$\begin{aligned} \text{cf}(\tau) &\doteq \tau \\ \text{cf}(\perp) &\doteq \perp \\ \text{cf}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) &\doteq \text{cf}(\sigma_\alpha) \quad \text{iff } \text{nf}(\sigma) = \alpha \\ \text{cf}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) &\doteq \nabla(\alpha \diamond \sigma_\alpha). \text{cf}(\sigma) \end{aligned}$$

Free type variables:

$$\begin{aligned} \text{ftv}(\alpha) &\doteq \{\alpha\} \\ \text{ftv}(c \tau_1 \dots \tau_n) &\doteq \text{ftv}(\tau_1) \cup \dots \cup \text{ftv}(\tau_n) \\ \text{ftv}(\perp) &\doteq \emptyset \\ \text{ftv}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) &\doteq (\text{ftv}(\sigma) - \{\alpha\}) \cup \text{ftv}(\sigma_\alpha) \quad \text{iff } \alpha \in \text{ftv}(\sigma) \\ \text{ftv}(\nabla(\alpha \diamond \sigma_\alpha). \sigma) &\doteq \text{ftv}(\sigma) \quad \text{iff } \alpha \notin \text{ftv}(\sigma) \end{aligned}$$

Splitting a prefix. The split algorithm takes a prefix Q and a set of type variables $\bar{\alpha}$, and splits Q in two parts (Q_1, Q_2) such that the domain of Q_1 is the domain of Q relevant to $\bar{\alpha}$.

$$\begin{aligned} () \uparrow \bar{\alpha} &\doteq ((), ()) \\ \frac{\alpha \in \bar{\alpha} \quad (Q_1, Q_2) = Q \uparrow (\bar{\alpha} - \alpha) \cup \text{ftv}(\sigma)}{(Q, \nabla(\alpha \diamond \sigma)) \uparrow \bar{\alpha} &\doteq ((Q_1, \nabla(\alpha \diamond \sigma)), Q_2)} \\ \frac{\alpha \notin \bar{\alpha} \quad (Q_1, Q_2) = Q \uparrow \bar{\alpha}}{(Q, \nabla(\alpha \diamond \sigma)) \uparrow \bar{\alpha} &\doteq (Q_1, (Q_2, \nabla(\alpha \diamond \sigma)))} \end{aligned}$$

Update a prefix with a new bound:

$$\text{(UPDATE)} \quad \frac{(Q_1, (Q_3, \nabla_0(\alpha \diamond \sigma_0), Q_4)) = Q \uparrow \text{ftv}(\sigma) \quad (\diamond = (\geq)) \vee ((Q) \sigma_0 \text{E? } \sigma)}{(Q) \triangleleft \nabla(\alpha \diamond \sigma) \doteq (Q_1 Q_3, \nabla(\alpha \diamond \sigma), Q_4)}$$

B. Proofs

B.1 Unique prefixes

Theorem 3: *In rule (P-APP), there exists a unique shortest possible prefix Q_b .*

PROOF. We prove this by induction on the function type σ_1 . In each case, we show that we either of a unique choice, or that we can make a unique best choice.

Starting with the instantiation of the function type, we have $(Q Q_a Q_b) \sigma_1 \sqsubseteq \nabla Q_3. \alpha \rightarrow \tau$. First we can use equivalence to bring σ_1 in normal-form: $\text{nf}(\sigma_1) = \nabla Q_1. \tau_1$ (using Theorem 2). If τ_1 is a type variable α , and we have a bound $\nabla(\alpha \geq \perp)$ (by the definition of normal form) and we use rule (I-BOTTOM) to instantiate to the full type of the argument e such that Q_b is empty. Otherwise, τ_1 has the form $\tau_4 \rightarrow \tau_5$. If τ_4 has the form $c \dots$, we can use Lemma 1 to show that in that case Q_b must contain all existential quantifiers of the argument type. Finally, τ_4 can be a type variable α . If it is bounded as $\exists \alpha$ it must be that $\alpha \in \text{dom}(Q Q_a)$, and thus Q_b must contain again all quantifiers.

This leaves us with the bound $\nabla(\alpha \diamond \sigma)$ where $\text{nf}(\sigma_1) \equiv \nabla Q_1. \alpha \rightarrow \tau_5$. Since we work on the normal form, σ is either \perp or a quantified polytype. If we have $\nabla(\alpha \geq \perp) \in Q_1$ we can again instantiate to the type of the argument e such that Q_b is empty by Lemma 1. If we have a rigid bound $\nabla(\alpha = \sigma)$, or a flexible bound in the outer prefix $\nabla(\alpha \geq \sigma) \in (Q Q_a)$, we can not instantiate further. By (A-HYP) or (I-HYP) we must have that $(Q Q_a Q_b) \sigma_2 \sqsubseteq \sigma$.

We have by Lemma 1 that $\sigma \equiv \exists Q_c. \exists Q_d. \forall Q_6. \tau_6$ where $\sigma_2 \equiv \exists Q_c. \forall Q_d. \forall (Q_2). \tau_2$, where Q_d has only unconstrained bounds. Since σ is fixed, this determines Q_c uniquely by the shapes of the types, and therefore Q_b is uniquely determined.

Finally, we handle the case where the bound is instantiable $\nabla(\alpha \geq \sigma) \in Q_1$. We have that $\sigma \equiv \exists Q_x. \forall Q_d. \forall Q_6. \tau_6$ for some Q_d of unconstrained bounds. By rule (I-EXISTS), we can instantiate this to $\exists Q_x. \exists Q_e. \forall Q_6. \tau_6$. By Lemma 1 and the instance relations, we have that $Q_x Q_e = Q_c Q_d$, where $\sigma_2 \equiv \exists Q_c. \forall Q_d. \forall (Q_2). \tau_2$, where Q_d and Q_e only have unconstrained bounds. Since Q_x is fixed, we can now choose Q_e to be a large as possible using rule (I-EXISTS) to maximize Q_c . Since this choice is uniquely determined by the shape of the types, we have a unique minimal Q_b . This last notion can be formalized using the concept of skeletons as shown in the thesis of Le Botlan [10]. \square

B.2 Soundness

Lemma 3: *Unification is sound:*

$$\begin{aligned} (Q, \mathcal{E}) \tau_1 \sim \tau_2 &: (Q', \mathcal{E}') \Rightarrow Q \sqsubseteq Q' \wedge (Q') \tau_1 \equiv \tau_2 \\ (Q, \mathcal{E}) \sigma_1 \sim_p \sigma_2 &: (Q', \mathcal{E}', \sigma) \\ &\Rightarrow Q \sqsubseteq Q' \wedge (Q') \sigma_1 \sqsubseteq \sigma \wedge (Q') \sigma_2 \sqsubseteq \sigma \end{aligned}$$

In contrast to MLF we also propagate the existential association environment \mathcal{E} . We extend the instance relation on prefixes [10] to an instance relation on (Q, \mathcal{E}) pairs in the obvious way:

LEMMA 7. $Q_1 \sqsubseteq Q_2 \Rightarrow (Q_1, \mathcal{E}) \sqsubseteq (Q_2, \mathcal{E})$

The existential association environment ensures that unification between existential quantifiers only happens when it corresponds to a renaming since we only have equivalence on existential types through (EQ-REFL). Rule (POLY) determines those existential types that can potentially be aliased as $(\mathcal{E}, \bar{\alpha}_1 \leftrightarrow \bar{\alpha}_2)$, where it is used by (U-EXISTS). In a sense this is a delayed choice and we define:

$$\text{(U-ENV)} \quad \mathcal{E}' = \mathcal{E} \setminus \{\alpha_1, \alpha_2\} \Rightarrow (Q, \mathcal{E}) \sqsubseteq (Q \triangleleft \nabla(\alpha_1 = \alpha_2), \mathcal{E}')$$

Using this property we can prove soundness of unification:

PROOF. Soundness is proved by induction over the cases of the unification algorithm. We can reuse the MLF proofs using Lemma 7, except for the additional rules (U-EXISTS-L) and (U-EXISTS). For (U-EXISTS-L) we assume $Q' = Q \triangleleft \nabla(\alpha_2 = \alpha_1)$. This update is well-defined as $\alpha_2 \notin \text{dom}(Q / \alpha_1)$ since $\text{cf}(\sigma_2) = \perp$, and succeeds by hypothesis. Using (I-BOTTOM) we can derive that $Q \sqsubseteq Q'$ and by Lemma `reflemma:ref:instext` $(Q, \mathcal{E}) \sqsubseteq (Q', \mathcal{E})$. Also, $(Q') \alpha_1 \equiv \alpha_2$ holds by (EQ-MONO).

For rule (U-EXISTS), we can derive from property (U-ENV) that $(Q, \mathcal{E}) \sqsubseteq (Q \triangleleft \nabla(\alpha_1 = \alpha_2), \mathcal{E} \setminus \{\alpha_1, \alpha_2\})$. The update is well defined since $\alpha_1 \notin \text{dom}(Q / \alpha_2)$ and $\alpha_1 \neq \alpha_2$. By (EQ-MONO), we have that $(Q \triangleleft \nabla(\alpha_1 = \alpha_2)) \alpha_1 \equiv \alpha_2$.

This leaves the extended (POLY) unification rule. By definition of the split algorithm and the hypothesis, we have that $(Q, \mathcal{E}) \sqsubseteq (Q_4, \mathcal{E}')$. We can reuse the MLF proof for `polyunify` (lemma 4.4.1 in [10]) directly but must be careful when concluding $(Q_4) \sigma_1 \sqsubseteq Q_5. \tau_1$. This only holds if all outer existential quantifiers in σ_1 exist in Q_5 as stated by Lemma 1. Otherwise, an existential quantifier could ‘escape’ by unifying with an outer polymorphic bound through rule (U-EXISTS-L) (take for example $\forall \alpha. T (T \alpha)$ and $\forall(\alpha \geq \exists \beta. T \beta). T \alpha$). Fortunately, the condition $(\bar{\alpha}_1 \cup \bar{\alpha}_2) \not\uparrow \text{dom}(Q_4)$ guarantees that no existential quantifiers escape and thus $(Q_4) \sigma_1 \sqsubseteq Q_5. \tau_1$. The same applies to σ_2 and we can further reuse the standard MLF proofs to show soundness. The condition where the association environment is empty $\mathcal{E} = ()$ only holds when called initially from (INF-APP). In that case, it is sound to let existentials escape and be moved to the prefix Q_4 since those correspond to the outer existentials Q_a and Q_b in (S-APP) and (P-APP). \square