An Extension of ML with First-Class Abstract Types

In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992

Konstantin Läufer,[*] New York University, laufer@cs.nyu.edu

Martin Odersky,[†] Yale University, odersky@cs.yale.edu

1 Introduction

Many statically-typed programming languages provide an abstract data type construct, such as the package in Ada, the cluster in CLU, and the module in Modula2. In these languages, an abstract data type consists of two parts, interface and implementation. The implementation consists of one or more representation types and some operations on these types; the interface specifies the names and types of the operations accessible to the user of the abstract data type.

ML [MTH90] provides two distinct constructs for describing abstract data types:

- The (obsolete) **`abstype`** mechanism is used to declare an abstract data type with a single implementation. It has been superseded by the module system.
- The ML module system provides signatures, structures, and functors. Signatures act as interfaces of abstract data types and structures as their implementations; functors are essentially parametrized structures. Several structures may share the same signature, and a single structure may satisfy several signatures. However, structures are not first-class values in ML for type-theoretic reasons discussed in [Mac86] [MH88]. This leads to considerable difficulties in a number of practical programming situations.

Mitchell and Plotkin show that abstract types can be given existential type [MP88]. By stating that a value v has the existential type $\exists\alpha.\tau$, we mean that for some fixed, unknown type $\hat{\tau}$, v has type $\tau[\hat{\tau}/\alpha]$. This paper presents a semantic extension of ML, where the component types of a datatype may be existentially quantified. We show how datatypes over existential types add significant flexibility to the language without even changing ML syntax; in particular, we give examples demonstrating how we express

- first-class abstract types,
- multiple implementations of a given abstract type,
- heterogeneous aggregates of different implementations of the same abstract type, and
- dynamic dispatching of operations with respect to the implementation type.

We have a deterministic Damas-Milner inference system [DM82] [CDDK86] for our language, which leads to a syntactically sound and complete type reconstruction algorithm. Furthermore, the type system is semantically sound with respect to a standard denotational semantics.

Most previous work on existential types does not consider type reconstruction. Other work appears to be semantically unsound or does not permit polymorphic instantiation of variables of existential type. By contrast, in our system such variables are **`let`**-bound and may be instantiated polymorphically.

We have implemented a Standard ML prototype of an interpreter with type reconstruction for our core language, Mini-ML [CDDK86] extended with recursive datatypes over existentially quantified component types. All examples from this paper have been developed and tested using our interpreter.

[*] Supported by the Defence Advanced Research Project Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-91-5-1472

[†] Supported by the Defence Advanced Research Project Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-91-J-4043

2 ML Datatypes with Existential Component Types

In ML, datatype declarations are of the form

$$\texttt{datatype}\ [\mathit{arg}]\ T\ \texttt{=}\ K_1\ \texttt{of}\ \tau_1\ \mid\ \ldots\ \mid\ K_n\ \texttt{of}\ \tau_n$$

where the *K*'s are value constructors and the optional prefix argument *arg* is used for formal type parameters, which may appear free in the component types τ_i. The value constructor functions are universally quantified over these type parameters, and no other type variables may appear free in the τ_i's.

An example for an ML datatype declaration is

```
datatype 'a Mytype = mycons of 'a * ('a -> int)
```

Without altering the syntax of the datatype declaration, we now give a meaning to type variables that appear free in the component types, but not in the type parameter list. We interpret such type variables as existentially quantified.

For example,

```
datatype Key = key of 'a * ('a -> int)
```

describes a datatype with one value constructor whose arguments are pairs of a value of type **`'a`** and a function from type **`'a`** to **`int`**. The question is what we can say about **`'a`**. The answer is, nothing, except that the value is of the same type **`'a`** as the function domain. To illustrate this further, the type of the expression

```
key(3,fn x => 5)
```

is **`Key`**, as is the type of the expression

```
key([1,2,3],length)
```

where **`length`** is the built-in function on lists. Note that no argument types appear in the result type of the expression. On the other hand,

```
key(3,length)
```

is not type-correct, since the type of **`3`** is different from the domain type of **`length`**.

We recognize that **`Key`** is an abstract type comprised by a value of some type and an operation on that type yielding an **`int`**. It is important to note that values of type **`Key`** are first-class; they may be created dynamically and passed around freely as function parameters. The two different values of type **`Key`** in the previous examples may be viewed as two different implementations of the same abstract type.

Besides constructing values of datatypes with existential component types, we can decompose them using the **`let`** construct. We impose the restriction that no type variable that is existentially quantified in a **`let`** expression appears in the result type of this expression or in the type of a global identifier. Analogous restrictions hold for the corresponding **`open`** and **`abstype`** constructs described in [CW85] [MP88].

For example, assuming **`x`** is of type **`Key`**, then

```
let val key(v,f) = x in
    f v
end
```

has a well-defined meaning, namely the **`int`** result of **`f`** applied to **`v`**. We know that this application is type-safe because the pattern matching succeeds, since **`x`** was constructed using constructor **`key`**, and at that time it was enforced that **`f`** can safely be applied to **`v`**. On the other hand,

```
let val key(v,f) = x in
    v
end
```

is not type-correct, since we do not know the type of **`v`** statically and, consequently, cannot assign a type to the whole expression.

Our extension to ML allows us to deal with existential types as described in [CW85] [MP88], with the further improvement that decomposed values of existential type are `let`-bound and may be instantiated polymorphically. This is illustrated by the following example,

```
datatype 'a t = k of ('a -> 'b) * ('b -> int)
let val k(f1,f2) = k(fn x => x,fn x => 3) in
    (f2(f1 7),f2(f1 true))
end
```

which results in `(3,3)`. In most previous work, the value on the right-hand side of the binding would have to be bound and decomposed twice.

3 Some Motivating Examples

Minimum over a heterogeneous list

Extending on the previous example, we first show how we construct heterogeneous lists over different implementations of the same abstract type and define functions that operate uniformly on such heterogeneous lists. A heterogeneous list of values of type `Key` could be defined as follows:

```
val hetlist =
    [key(3,fn x => x), key([1,2,3,4],length), key(7,fn x => 0),
     key(true,fn x => if x then 1 else 0), key(12,fn x => 3)]
```

The type of `hetlist` is `Key list`; it is a homogeneous list of elements each of which could be a different implementation of type `Key`. We define the function `min`, which finds the minimum of a list of `Key`'s with respect to the integer value obtained by applying the second component (the function) to the first component (the value).

```
fun min [x] = x
  | min ((key(v1,f1))::xs) =
        let val key(v2,f2) = min xs in
            if f1 v1 <= f2 v2 then key(v1,f1) else key(v2,f2)
        end
```

Then `min hetlist` returns `key(7,fn x => 0)`, the third element of the list.

Stacks parametrized by element type

The previous examples involved datatypes with existential types but without polymorphic type parameters. As an example for a type involving both, we show an abstract stack parametrized by element type.

```
datatype 'a Stack = stack of {value   : 'b,
                              empty   : 'b,
                              push    : 'a * 'b -> 'b
                              pop     : 'b -> 'a * 'b
                              top     : 'b -> 'a,
                              isempty : 'b -> bool}
```

An implementation of an `int Stack` in terms of the built-in type `list` can be given as

```
stack{value = [1,2,3], empty = [], push = op ::,
      pop = fn xs => (hd xs,tl xs), top = hd, isempty = null}
```

An alternative implementation of `Stack` could be given, among others, based on arrays. Different implementations could then be combined in a list of stacks. To facilitate dynamic dispatching, constructors of stacks of different implementations can be provided together with stack operations that work uniformly across implementations. These “outer” operations work by opening the stack, applying the intended “inner” operation, and encapsulating the stack again, for example

```
fun makeliststack xs = stack{value = xs, empty = [],push = op ::,
        pop = fn xs => (hd xs,tl xs), top = hd, isempty = null}
fun makearraystack xs = stack{...}
fun push a (stack{value = v, push = pu, empty = e,
                  pop = po, top = t, isempty = i}) =
             stack{value = pu(a,v), push = pu, empty = e,
                  pop = po, top = t, isempty = i}
map (push 8) [makeliststack [2,4,6], makearraystack [3,5,7]]
```

4 Type-Theoretical Aspects

A deterministic type inference system for our language is given in the appendix; it leads directly to a syntactically sound and complete type reconstruction algorithm to compute principal types. Our type system is semantically sound with respect to a standard denotational semantics. Moreover, it is a conservative extension of ML. That is, for a program in our language whose declarations introduce no existentially quantified type variables, our type reconstruction algorithm and the ML type reconstruction algorithm compute the same type. A comprehensive treatment of polymorphic type inference with existential types is found in [Lä92].

5 Related Work

Hope+C

The only other work known to us that deals with Damas-Milner-style type inference for existential types is [Per90]. However, the typing rules given there are not sufficient to guarantee the absence of runtime type errors, even though the Hope+C compiler seems to impose sufficient restrictions. The following unsafe program, here given in ML syntax, is well-typed according to the typing rules, but rejected by the compiler:

```
datatype T = K of ''a
fun f x = let val K z = x in z end
f(K 1) = f(K true)
```

XML$^+$

The possibility of making ML structures first-class by implicitly hiding their type components is discussed in [MMM91] without addressing the issue of type inference. By hiding the type components of a structure, its type is implicitly coerced from a strong sum type to an existential type. Detailed discussions of sum types can be found in [Mac86] [MH88].

Haskell with existential types

Existential types combine well with the systematic overloading polymorphism provided by Haskell type classes [WB89]; this point is further discussed in [LO91]. Briefly, we extend Haskell’s data declaration in a similar way as the ML datatype declaration above. In Haskell [HPW91], it is possible to specify what type class a (universally quantified) type variable belongs to. In our extension, we can do the same for existentially quantified type variables. This lets us construct heterogeneous aggregates over a given type class.

Dot notation

MacQueen [Mac86] observes that the use of existential types in connection with an elimination construct (`open`, `abstype`, or our `let`) is impractical in certain programming situations; often, the scope of the elimination construct has to be made so large that some of the benefits of abstraction are lost. A formal treatment of the dot notation, an alternative used in actual programming languages, is found in [CL90]. An extension of ML with an analogous notation is described in [Lä92].

Dynamics in ML

An extension of ML with objects that carry dynamic type information is described in [LM91]. A dynamic is a pair consisting of a value and the type of the value. Such an object is constructed from a value by applying the constructor **`dynamic`**. The object can then be dynamically coerced by pattern matching on both the value and the runtime type. Existential types are used to match dynamic values against dynamic patterns with incomplete type information. Dynamics are useful for typing functions such as **`eval`**. However, they do not provide type abstraction, since they give access to the type of an object at runtime. It seems possible to combine their system with ours, extending their existential patterns to existential types. We are currently investigating this point.

Acknowledgments

We would like to express our thanks to Ben Goldberg, Fritz Henglein, Ross Paterson, Nigel Perry, Benjamin Pierce, and Phil Wadler, for helpful suggestions and stimulating discussions.

A Formal Discussion of the Extended Language

In this appendix, we describe the formal language and the type system underlying our extension of ML. The typing rules and auxiliary functions translate to the type reconstruction algorithm given below.

A.1 Syntax

Language syntax

Identifiers	x
Constructors	K
Expressions	$e ::= x \mid (e_1, e_2) \mid e\ e' \mid \lambda x.e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e' \mid$ $\mathbf{data}\ \forall\alpha_1\ldots\alpha_n.\chi\ \mathbf{in}\ e \mid K \mid \mathbf{is}\ K \mid \mathbf{let}\ K\ x = e\ \mathbf{in}\ e'$

In addition to the usual constructs (identifiers, applications, λ-abstractions, and **`let`** expressions), we introduce desugared versions of the ML constructs that deal with datatypes. A **`data`** declaration defines a new datatype; values of this type are created by applying a constructor K, their tags can be inspected using an **`is`** expression, and they can be decomposed by a pattern-matching **`let`** expression. The following example shows a desugared definition of ML's list type and the associated length function.

```
data ∀α. (μβ.Nil unit + Cons α × β)  in
    let length = fix λlength.λxs.
                        if (is Nil xs)
                           0
                           (let Cons ab = xs in + (length(snd ab)) 1)
    in
        length(Cons(3,Cons(7,Nil())))
```

Type syntax

Type variables	α
Skolem functions	κ
Types	$\tau ::= \mathit{unit} \mid \mathit{bool} \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \to \tau' \mid \kappa(\tau_1, \ldots, \tau_n) \mid \chi$

Recursive types	$\chi ::=$	$\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ if $i \neq j$
Existential types	$\eta ::=$	$\exists\alpha.\eta \mid \tau$
Type schemes	$\sigma ::=$	$\forall\alpha.\sigma \mid \tau$
Assumptions	$a ::=$	$\sigma/x \mid \forall\alpha_1\ldots\alpha_n.\chi/K$

Our type syntax includes recursive types χ and Skolem type constructors κ; the latter are used to type identifiers bound by a pattern-matching `let` whose type is existentially quantified. Explicit existential types arise only as domain types of value constructors. Assumption sets serve two purposes: they map identifiers to type schemes and constructors to the recursive type schemes they belong to. Thus, when we write $A(K)$, we mean the σ such that $\sigma = \forall\alpha_1\ldots\alpha_n.\ldots + K\eta + \ldots$. Further, let $\Sigma[K\eta]$ stand for sum type contexts such as $K_1\eta_1 + \ldots + K_m\eta_m$, where $K_i = K$ and $\eta_i = \eta$ for some i.

A.2 Type Inference

Instantiation and generalization of type schemes

$\forall\alpha_1\ldots\alpha_n.\tau \geq \tau'$	iff there are types $\tau_1, \ldots\tau_n$ such that $\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$
$\exists\alpha_1\ldots\alpha_n.\tau \leq \tau'$	iff there are types $\tau_1, \ldots\tau_n$ such that $\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$
$gen(A, \tau)$	$= \forall(FV(\tau) \setminus FV(A)).\tau$
$skolem(A, \exists\gamma_1\ldots\gamma_n.\tau)$	$= \tau[\kappa_i(\alpha_1, \ldots\alpha_k)/\gamma_i]$ where $\kappa_1\ldots\kappa_n$ are new Skolem type constructors such that $\{\kappa_1, \ldots, \kappa_n\} \cap FS(A) = \emptyset$, and $\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1\ldots\gamma_n.\tau) \setminus FV(A)$

The first three auxiliary functions are standard. The function *skolem* replaces each existentially quantified variable in a type by a unique type constructor whose actual arguments are those free variables of the type that are not free in the assumption set; this reflects the "maximal" knowledge we have about the type represented by an existentially quantified type variable. In addition to FV, the set of free type variables in a type scheme or assumption set, we use FS, the set of Skolem type constructors that occur in a type scheme or assumption set.

Inference rules for expressions

The first five typing rules are essentially the same as in [CDDK86].

(VAR) $$\frac{A(x) \geq \tau}{A \vdash x : \tau}$$

(PAIR) $$\frac{A \vdash e_1 : \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

(APPL) $$\frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash e\ e' : \tau}$$

(ABS) $$\frac{A[\tau'/x] \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

(LET) $$\frac{A \vdash e : \tau \qquad A[gen(A, \tau)/x] \vdash e' : \tau'}{A \vdash \texttt{let } x = e \texttt{ in } e' : \tau'}$$

The new rules DATA, CONS, TEST, and PAT are used to type datatype declarations, value constructors, **`is`** expressions, and pattern-matching **`let`** expressions, respectively.

(DATA) $$\frac{\begin{array}{c}\sigma = \forall \alpha_1 \ldots \alpha_n . \mu\beta . K_1\eta_1 + \ldots + K_m\eta_m \\ FV(\sigma) = \varnothing \qquad A[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau\end{array}}{A \vdash \texttt{data } \sigma \texttt{ in } e : \tau}$$

The DATA rule elaborates a declaration of a recursive datatype. It checks that the type scheme is closed and types the expression under the assumption set extended with assumptions about the constructors.

(CONS) $$\frac{A(K) \geq \mu\beta.\Sigma[K\eta] \qquad \eta[\mu\beta.\Sigma[K\eta]/\beta] \leq \tau}{A \vdash K : \tau \to \mu\beta.\Sigma[K\eta]}$$

The CONS rule observes the fact that existential quantification in argument position means universal quantification over the whole function type; this is expressed by the second premise.

(TEST) $$\frac{A(K) \geq \mu\beta.\Sigma[K\eta]}{A \vdash \texttt{is } K : (\mu\beta.\Sigma[K\eta]) \to bool}$$

The TEST rule ensures that **`is`** K is applied only to arguments whose type is the same as the result type of constructor K.

(PAT) $$\frac{\begin{array}{c}A \vdash e : \mu\beta.\Sigma[K\eta] \qquad FS(\tau') \subseteq FS(A) \\ A[gen(A, skolem(A, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x] \vdash e' : \tau'\end{array}}{A \vdash \texttt{let } K\ x = e \texttt{ in } e' : \tau'}$$

The last rule, PAT, governs the typing of pattern-matching **`let`** expressions. It requires that the expression e be of the same type as the result type of the constructor K. The body e' is typed under the assumption set extended with an assumption about the bound identifier x. By definition of the function *skolem*, the new Skolem type constructors do not appear in A; this ensures that they do not appear in the type of any identifier free in e' other than x. It is also guaranteed that the Skolem constructors do not appear in the result type τ'.

Relation to the ML Type Inference System

Theorem 1 [Conservative extension] Let Mini-ML' be an extension of Mini-ML with recursive datatypes, but not with existential quantification. Then, for any Mini-ML' expression e, $A \vdash e : \tau$ iff $A \vdash_{\text{Mini-ML'}} e : \tau$.

Proof: By structural induction on e.

Corollary 2 [Conservative extension] Our type system is a conservative extension of the Mini-ML type system described in [CDDK86], in the following sense: For any Mini-ML expression e, $A \vdash e : \tau$ iff

$A \vdash_{\text{Mini-ML}} e : \tau$.

Proof: Follows immediately from the previous theorem.

A.3 Type Reconstruction

The type reconstruction algorithm is a straightforward translation from the deterministic typing rules, using a standard unification algorithm [Rob65] [MM82]. We conjecture that its complexity is the same as that of algorithm W.

Auxiliary functions

In our algorithm, we need to instantiate universally quantified types and generalize existentially quantified types. Both are handled in the same way.

$inst_{\forall}(\forall\alpha_1\ldots\alpha_n.\tau) = \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ where $\beta_1, \ldots, \beta_n$ are fresh type variables

$inst_{\exists}(\exists\alpha_1\ldots\alpha_n.\tau) = \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ where $\beta_1, \ldots, \beta_n$ are fresh type variables

The functions *skolem* and *gen* are the same as in the inference rules, with the additional detail that *skolem* always creates fresh Skolem type constructors.

Algorithm

Our type reconstruction function takes an assumption set and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

```
$TC(A, x)$                              $= (Id, inst_{\forall}(A(x)))$

$TC(A, (e_1, e_2))$                     $=$ let  $(S_1, \tau_1) = TC(A, e_1)$
                                               $(S_2, \tau_2) = TC(S_1A, e_2)$
                                           in  $(S_2S_1, S_2\tau_1 \times \tau_2)$

$TC(A, ee')$                            $=$ let  $(S, \tau) = TC(A, e)$
                                               $(S', \tau') = TC(SA, e')$
                                               $\beta$ be a fresh type variable
                                               $U = mgu(S'\tau, \tau' \to \beta)$
                                           in  $(US'S, U\beta)$

$TC(A, \lambda x.e)$                    $=$ let  $\beta$ be a fresh type variable
                                               $(S, \tau) = TC(A[\beta/x], e)$
                                           in  $(S, S\beta \to \tau)$

$TC(A,$ let $x = e$ in $e')$           $=$ let  $(S, \tau) = TC(A, e)$
                                               $(S', \tau') = TC(SA[gen(SA, \tau)/x], e')$
                                           in  $(S'S, \tau')$
```

```
TC(A, data σ in e)        = let  $\forall\alpha_1\ldots\alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma$  in
                                  if  $FV(\sigma) = \varnothing$  then
                                      $TC(A[\sigma/K_1, \ldots, \sigma/K_m], e)$

TC(A, K)                  = let  $\tau = inst_\forall(A(K))$
                                  $\mu\beta.\ldots + K\eta + \ldots = \tau$
                            in  $(Id, (inst_\exists(\eta[\tau/\beta])) \to \tau)$

TC(A, is K)               = let  $\tau = inst_\forall(A(K))$
                            in  $(Id, \tau \to bool)$

TC(A, let K x = e in e')  = let  $\hat{\tau} = inst_\forall(A(K))$
                                $\mu\beta.\ldots + K\eta + \ldots = \hat{\tau}$
                                $(S, \tau) = TC(A, e)$
                                $U = (mgu(\hat{\tau}, \tau))S$
                                $\tau_\kappa = skolem(UA, U(\eta[\hat{\tau}/\beta]))$
                                $(S', \tau') = TC(UA[gen(UA, \tau_\kappa)/x], e')$
                            in
                                if  $FS(\tau') \subseteq FS(S'UA) \wedge$
                                      $(FS(\tau_\kappa) \setminus FS(U(\eta[\hat{\tau}/\beta]))) \cap FS(S'UA) = \varnothing$
                                then  $(S'U, \tau')$
```

Theorem 3 [Syntactic Soundness and Completeness] The type reconstruction algorithm TC is sound and complete with respect to the type inference relation $\vdash$.

Proof: We extend the proof given in [CDDK86] to deal with the new constructs.

A.4 Semantics

We give a standard denotational semantics. The evaluation function E maps an expression $e \in Exp$ to some semantic value v, in the context of an evaluation environment $\rho \in Env$. An evaluation environment is a partial mapping from identifiers to semantic values. Runtime type errors are represented by the special value wrong. Tagged values are used to capture the semantics of algebraic data types.

We distinguish between the three error situations, runtime type errors (wrong), nontermination, and a mismatch when an attempt is made to decompose a tagged value whose tag does not match the tag of the destructor. Both nontermination and mismatch are expressed by $\perp$.

Our type inference system is sound with respect to the evaluation function; a well-typed program never evaluates to wrong. The formal proof for semantic soundness is given below.

It should be noted that we do not commit ourselves to a strict or non-strict evaluation function. Therefore, our treatment of existential types applies to languages with both strict and non-strict semantics. For either case, appropriate conditions would have to be added to the definition of the evaluation function.

Semantic domain

Unit value	$U = \{\mathsf{unit}\}_\perp$
Boolean values	$B = \{\mathsf{false}, \mathsf{true}\}_\perp$
Constructor tags	C
Semantic domain	$V \cong U + B + (V \to V) + (V \times V) + (C \times V) + \{\mathsf{wrong}\}_\perp$

In the latter definition of V, $+$ stands for the coalesced sum, so that all types over V share the same $\perp$.

Semantics of expressions

The semantic function for expressions,

$$E : Exp \to Env \to V,$$

is defined as follows:

$$
\begin{array}{ll}
E[\![x]\!]\rho & = \rho(x) \\
E[\![(e_1, e_2)]\!]\rho & = \langle E[\![e_1]\!]\rho, E[\![e_2]\!]\rho\rangle \\
E[\![e\,e']\!]\rho & = \mathsf{if}\ E[\![e]\!]\rho \in V \to V\ \mathsf{then} \\
 & \qquad (E[\![e]\!]\rho)\,(E[\![e']\!]\rho) \\
 & \quad \mathsf{else}\ \mathsf{wrong} \\
E[\![\lambda x.e]\!]\rho & = \lambda v \in V. E[\![e]\!](\rho[v/x]) \\
E[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]\rho & = E[\![e']\!](\rho[E[\![e]\!]\rho/x]) \\
E[\![\mathtt{data}\ \sigma\ \mathtt{in}\ e]\!]\rho & = E[\![e]\!]\rho \\
E[\![K]\!]\rho & = \lambda v \in V. \langle K, v\rangle \\
E[\![\mathtt{is}\ K]\!]\rho & = \lambda v \in V. \mathsf{if}\ v \in \{K\} \times V\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false} \\
E[\![\mathtt{let}\ K\ x = e\ \mathtt{in}\ e']\!]\rho & = E[\![e']\!](\rho[\mathsf{if}\ E[\![e]\!]\rho \in \{K\} \times V\ \mathsf{then} \\
 & \qquad\qquad \mathsf{snd}\,(E[\![e]\!]\rho) \\
 & \qquad\quad \mathsf{else}\ \perp/x])
\end{array}
$$

Semantics of types

Following [MPS86], we identify types with *weak ideals* over the semantic domain V. A type environment $\psi \in TEnv$ is a partial mapping from type variables to ideals and from Skolem type constructors to functions between ideals. The semantic interpretation of types,

$$T : TExp \to TEnv \to \Im(V)$$

is defined as follows.

$$
\begin{aligned}
T[\![unit]\!]\psi &= U \\
T[\![bool]\!]\psi &= B \\
T[\![\alpha]\!]\psi &= \psi(\alpha) \\
T[\![\tau_1 \times \tau_2]\!]\psi &= T[\![\tau_1]\!]\psi \times T[\![\tau_2]\!]\psi \\
T[\![\tau \to \tau']\!]\psi &= T[\![\tau]\!]\psi \to T[\![\tau']\!]\psi \\
T[\![\kappa(\tau_1, \ldots, \tau_n)]\!]\psi &= (\psi(\kappa))\,(T[\![\tau_1]\!]\psi, \ldots, T[\![\tau_n]\!]\psi) \\
T[\![\mu\beta. \textstyle\sum K_i\eta_i]\!]\psi &= \mu(\lambda I \in \Im(V). \textstyle\sum \{K_i\} \times T[\![\eta_i]\!](\psi[I/\beta])) \\
T[\![\forall\alpha.\sigma]\!]\psi &= \bigcap_{I \in \Re} \lambda I \in \Im(V). T[\![\sigma]\!](\psi[I/\alpha]) \\
T[\![\exists\alpha.\eta]\!]\psi &= \bigsqcup_{I \in \Re} \lambda I \in \Im(V). T[\![\eta]\!](\psi[I/\alpha])
\end{aligned}
$$

The universal and existential quantifications range over the set $\Re \subseteq \Im(V)$ of all ideals that do not contain wrong. Note that the sum in the definition of recursive types is actually a union, since the constructor tags are assumed to be distinct. It should also be noted that our interpretation does not handle ML's nonregular, mutually recursive datatypes; it appears that the PER model described in [BM92] would provide an adequate interpretation.

Theorem 4 The semantic function for types is well-defined.

Proof: As in [MPS86]. We observe that $\lambda I \in \Im(V). \sum \{K_i\} \times T[\![\eta_i]\!](\psi[I/\alpha])$ is always contractive, since cartesian product and sum of ideals are contractive; therefore, the fixed point of such a function exists.

Lemma 5 Let ψ be a type environment such that for every $\alpha \in \mathrm{Dom}\,\psi$, wrong $\notin \psi(a)$. Then for every type scheme σ, wrong $\notin T[\![\sigma]\!]\psi$.

Proof: By structural induction on σ.

Lemma 6 [Substitution] $T[\![\sigma[\sigma'/\alpha]]\!]\psi = T[\![\sigma]\!](\psi[T[\![\sigma']\!]\psi/\alpha])$.

Proof: Again, by structural induction on σ.

Definition 1 [Semantic type judgment] Let A be an assumption set, e an expression, and σ a type scheme. We define $\models_{\rho,\psi} A$ as meaning that $\mathrm{Dom}\,A \subseteq \mathrm{Dom}\,\rho$ and for every $x \in \mathrm{Dom}\,A$, $\rho(x) \in T[\![A(x)]\!]\psi$; further, we say $A \models_{\rho,\psi} e : \sigma$ iff $\models_{\rho,\psi} A$ implies $E[\![e]\!]\rho \in T[\![\sigma]\!]\psi$; and finally, $A \models e : \sigma$ means that for all $\rho \in Env$ and $\psi \in TEnv$ we have $A \models_{\rho,\psi} e : \sigma$.

Theorem 7 [Semantic Soundness] If $A \vdash e : \tau$ then $A \models e : \tau$.

Proof: By induction on the size of the proof tree for $A \vdash e : \tau$. We need to consider each of the cases given by the type inference rules. Applying the inductive assumption and the typing judgments from the preceding steps in the type derivation, we use the semantics of the types of the partial results of the evaluation. In each of the cases below, choose ψ and ρ arbitrarily, such that $\models_{\rho,\psi} A$. We include only the nonstandard cases. Lemma 6 will be used with frequency.

$A \vdash \texttt{data}\ \forall\alpha_1\ldots\alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m\ \texttt{in}\ e : \tau$

The premise in the type derivation is $A[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau$, where $\sigma = \forall\alpha_1\ldots\alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$. Since by definition, $\models_{\rho,\psi} A[\sigma/K_1, \ldots, \sigma/K_m]$, we can use the inductive assumption to obtain $E[\![\texttt{data}\ \forall\alpha_1\ldots\alpha_n.\chi\ \texttt{in}\ e]\!]\rho = E[\![e]\!]\rho \in T[\![\tau]\!]\psi$.

$A \vdash K : \tau \to \mu\beta.\Sigma[K\eta]$

The last premise in the type derivation is $\eta\ [\mu\beta.\Sigma[K\eta]/\beta] \le \tau$, where $\eta = \exists\gamma_1\ldots\gamma_n.\hat{\tau}$. By definition of instantiation of existential types, $\tau = \hat{\tau}[\tau_j/\gamma_j, \mu\beta.\Sigma[K\eta]/\beta]$ for some types $\tau_1, \ldots, \tau_n$. First, choose an arbitrary $v \in T[\![\tau]\!]\psi$ and a finite $a \le v$. Now,

$$\begin{aligned}
a &\in (T[\![\hat{\tau}[\tau_j/\gamma_j, \mu\beta.\Sigma[K\eta]/\beta]]\!]\psi)^\circ \\
&= (T[\![\hat{\tau}[\mu\beta.\Sigma[K\eta]/\beta]]\!]\ (\psi[T[\![\tau_j]\!]\psi/\gamma_j]))^\circ \\
&\subseteq \bigcup_{J_1,\ldots,J_n \in \mathfrak{R}} (T[\![\hat{\tau}\ [\mu\beta.\Sigma[K\eta]/\beta]]\!]\ (\psi[J_j/\gamma_j]))^\circ \\
&= (\bigsqcup_{J_1,\ldots,J_n \in \mathfrak{R}} T[\![\hat{\tau}\ [\mu\beta.\Sigma[K\eta]/\beta]]\!]\ (\psi[J_j/\gamma_j]))^\circ \\
&= (T[\![\eta\ [\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi)^\circ.
\end{aligned}$$

Hence, $v = \bigsqcup\{a \mid a \text{ finite and } a \le v\} \in T[\![\eta\ [\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi$, by closure of ideals under limits. Consequently,

$$\begin{aligned}
\langle K, v\rangle &\in \{K\} \times T[\![\eta\ [\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi \\
&\subseteq \ldots + \{K\} \times T[\![\eta\ [\mu\beta.\Sigma[K\eta]/\beta]]\!]\psi + \ldots \\
&= \ldots + \{K\} \times T[\![\eta]\!]\ (\psi[T[\![\mu\beta.\Sigma[K\eta]]\!]\psi/\beta]) + \ldots \\
&= T[\![\mu\beta.\Sigma[K\eta]]\!]\psi.
\end{aligned}$$

Hence $E[\![K]\!]\rho \in T[\![\tau \to \mu\beta.\Sigma[K\eta]]\!]\psi$.

$A \vdash \texttt{is}\ K : (\mu\beta.\Sigma[K\eta]) \to bool$

Choose an arbitrary $v \in T[\![\mu\beta.\Sigma[K\eta]]\!]\psi$. Clearly, $(E[\![\texttt{is}\ K]\!]\rho)v \in B$, whence $E[\![\texttt{is}\ K]\!]\rho \in T[\![(\mu\beta.\Sigma[K\eta]) \to bool]\!]\psi$.

$A \vdash \texttt{let}\ K\ x = e\ \texttt{in}\ e' : \tau'$

We follow the proof in [MPS86]. The first premise in the type derivation is $A \vdash e : \tau$, where $\tau = \mu\beta.\Sigma[K\eta]$ and $\eta = \exists\gamma_1\ldots\gamma_n.\hat{\tau}$. Let $\{\alpha_1, \ldots, \alpha_k\} = FV(\tau) \setminus FV(A)$. Then, for every $I_1, \ldots, I_k \in \mathfrak{I}(V)$, $\models_{\rho,\psi[I_i/\alpha_i]} A$ holds, since none of the α_i's are free in A.

Let $v = E[\![e]\!]\rho$; by the inductive assumption, $v \in T[\![\tau]\!]\ (\psi[I_i/\alpha_i])$. Consequently,

$$v \in \bigcap_{I_1,\ldots,I_k \in \mathfrak{R}} T[\![\tau]\!]\ (\psi[I_i/\alpha_i])$$

$$= \bigcap_{I_1, \ldots, I_k \in \Re} T[\![\mu\beta.\Sigma[K\eta]]\!](\psi[I_i/\alpha_i])$$

$$= \ldots + \{K\} \times \bigcap_{I_1, \ldots, I_k \in \Re} T[\![\eta]\!](\psi[I_i/\alpha_i, T[\![\tau]\!](\psi[I_i/\alpha_i])/\beta]) + \ldots.$$

First, consider the case $\mathsf{fst}(v) \neq K$. Then, by definition, $E[\![\mathbf{let}\ K\ x = e\ \mathbf{in}\ e']\!]\rho = \bot$, and we are done, since $\bot \in T[\![\tau']\!]\psi$.

In the more interesting, second case, $\mathsf{fst}(v) = K$. Then

$$\mathsf{snd}(v) \in \bigcap_{I_1, \ldots, I_k \in \Re} \bigsqcup_{J_1, \ldots, J_n \in \Re} T[\![\hat{\tau}]\!](\psi[I_i/\alpha_i, J_j/\gamma_j, T[\![\tau]\!](\psi[I_i/\alpha_i])/\beta])$$

Let $\alpha_1, \ldots, \alpha_h$, $h \leq k$, be those variables among $\alpha_1, \ldots, \alpha_k$ that are free in $\hat{\tau}[\tau/\beta]$.

We now choose a finite a such that $a \leq \mathsf{snd}(v)$, thus

$$a \in \bigcap_{I_1, \ldots, I_h \in \Re} \bigcup_{J_1, \ldots, J_n \in \Re} (T[\![\hat{\tau}[\tau/\beta]]\!](\psi[I_i/\alpha_i, J_j/\gamma_j]))^\circ.$$

By definition of set union and intersection, there exist functions $f_1, \ldots, f_n \in \Im(V)^h \to \Im(V)$, such that

$$a \in \bigcap_{I_1, \ldots, I_h \in \Re} (T[\![\hat{\tau}[\tau/\beta]]\!](\psi[I_i/\alpha_i, f_j(I_1, \ldots, I_h)/\gamma_j]))^\circ$$

$$\subseteq \bigcap_{I_1, \ldots, I_h \in \Re} T[\![\hat{\tau}[\tau/\beta]]\!](\psi[I_i/\alpha_i, f_j(I_1, \ldots, I_h)/\gamma_j])$$

$$= \bigcap_{I_1, \ldots, I_h \in \Re} T[\![\hat{\tau}[\kappa_j(\alpha_1, \ldots, \alpha_h)/\gamma_j, \tau/\beta]]\!](\psi[I_i/\alpha_i, f_j/\kappa_j])$$

$$= T[\![\forall\alpha_1 \ldots \alpha_h.\hat{\tau}[\kappa_j(\alpha_1, \ldots, \alpha_h)/\gamma_j, \tau/\beta]]\!](\psi[f_j/\kappa_j])$$

$$= T[\![\mathit{gen}(A, \mathit{skolem}(A, \eta[\tau/\beta]))]\!](\psi[f_j/\kappa_j]),$$

assuming that the κ_j's are the ones generated by $\mathit{skolem}(A, \eta[\tau/\beta])$.

Since by definition of skolem, none of the κ_j's are free in A, $\models_{\rho, \psi[f_j/\kappa_j]} A$ holds and we can extend A and ρ, obtaining $\models_{\rho[a/x], \psi[f_j/\kappa_j]} A[\mathit{gen}(A, \mathit{skolem}(A, \eta[\tau/\beta]))/x]$.

We now apply the inductive assumption to the last premise,

$$A[\mathit{gen}(A, \mathit{skolem}(A, \eta[\tau/\beta]))/x] \vdash e' : \tau',$$

and obtain

$$E[\![e']\!](\rho[a/x]) \in T[\![\tau']\!](\psi[f_j/\kappa_j]) = T[\![\tau']\!]\psi,$$

since $FS(\tau') \subseteq FS(A)$. Finally,

$$\begin{aligned} E[\![\mathbf{let}\ K\ x = e\ \mathbf{in}\ e']\!]\rho &= E[\![e']\!](\rho[\mathsf{snd}(E[\![e]\!]\rho)/x]) \\ &= \bigsqcup\{E[\![e']\!](\rho[a/x]) \mid a \text{ finite and } a \leq \mathsf{snd}(E[\![e]\!]\rho)\}, \end{aligned}$$

by the continuity of E. The latter expression is in $T[\![\tau']\!]\psi$ by the closure of ideals under limits.

■

Corollary 8 [Semantic Soundness] If $A \vdash e : \tau$, then $E[\![e]\!]\rho \neq \mathsf{wrong}$.

Proof: We apply Lemma 5 to the previous theorem.

References

[BM92] K. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 316–327, January 1992.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[CL90] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pages 466–491, Sea of Galilee, Israel, April 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[HPW91] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.1. Technical Report YALEU/DCS/RR-777, Dept. of Computer Science, Yale University, New Haven, Conn., August 1991.

[Lä92] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, Department of Computer Science, 1992. In preparation.

[LM91] X. Leroy and M. Mauny. Dynamics in ML. In *Proc. Functional Programming Languages and Computer Architecture*, pages 406–426. ACM, 1991.

[LO91] K. Läufer and M. Odersky. Type classes are signatures of abstract types. In *Proc. Phoenix Seminar and Workshop on Declarative Programming*, November 1991.

[Mac86] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, Jan. 1986.

[MH88] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, Jan. 1988.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1991.

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

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.