Polymorphic Type Inference and Abstract Data Types

KONSTANTIN LÄUFER
Loyola University
and
MARTIN ODERSKY
Universität Karlsruhe

Many statically typed programming languages provide an abstract data type construct, such as the module in Modula-2. However, in most of these languages, implementations of abstract data types are not first-class values. Thus, they cannot be assigned to variables, passed as function parameters, or returned as function results. Several higher-order functional languages feature strong and static type systems, parametric polymorphism, algebraic data types, and explicit type variables. Most of them rely on Hindley-Milner type inference instead of requiring explicit type declarations for identifiers. Although some of these languages support abstract data types, it appears that none of them directly provides light-weight abstract data types whose implementations are first-class values. We show how to add significant expressive power to statically typed functional languages with explicit type variables by incorporating first-class abstract types as an extension of algebraic data types. Furthermore, we extend record types to allow abstract components. The components of such abstract records are selected using the dot notation. Following Mitchell and Plotkin, we formalize abstract types in terms of existentially quantified types. We give a syntactically sound and complete type inference algorithm and prove that our type system is semantically sound with respect to standard denotational semantics.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages;* D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types; modules; packages;* F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics;* F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: Dynamic dispatching, existentially quantified types, first-class abstract types, polymorphism, type inference, universally quantified types

A preliminary version of this paper was presented at the ACM SIGPLAN Workshop on ML and its Applications, San Francisco, June 1992.
Authors' addresses: K. Läufer, Department of Mathematical Sciences, Loyola University, 6525 North Sheridan Road, Chicago, IL 60626; email: laufer@math.luc.edu; M. Odersky, Institut für Programmstrukturen and Datenorganisation, Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany; email: odersky@ira.uka.de.

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 Modula-2. 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. However, in most such languages, implementations of abstract data types are not first-class values. Thus, they cannot be assigned to variables, passed as function parameters, or returned as function results.

Several higher-order functional languages, such as Haskell [Hudak et al. 1992], Hope [Burstall et al. 1980], Miranda [Turner 1985], and ML [Milner 1990], feature strong and static type systems, parametric polymorphism, algebraic data types, and explicit type variables. Most languages in this group rely on Hindley-Milner type inference instead of requiring explicit type declarations for identifiers. Although some of these languages support abstract data types, it appears that none of them directly provides light-weight abstract data types whose implementations are first-class values. Instead, they provide the following distinct constructs that can be used to express abstract data types:

(1) Tuples or records of closures can be used to model abstract data types [Odersky 1991]. The hidden bindings shared between the closures correspond to the representation; the closures themselves correspond to the operations; and the type of the tuple or record corresponds to the interface. The shortcoming of this approach is the complete encapsulation of the internal representation, which makes it hard to add operations to the abstract type or to implement efficient binary operations [Läufer 1992].

(2) Modules provide a mechanism for separate compilation and data abstraction. A module in Haskell consists of an interface and an implementation of that interface. The Standard ML module system generalizes modules by allowing signatures (interfaces) and structures (implementations) as independent entities: several structures may share the same signatures, and a single structure may satisfy several signatures. Furthermore, Standard ML provides parameterized structures called functors. For type-theoretic reasons, first-class structures would entail a type of all types, leading to inconsistencies in the language [Meyer et al. 1986; Mitchell and Harper 1990]. Therefore, structures are not treated as first-class values. This causes considerable difficulties in a number of practical programming situations [Läufer 1992]. Some recent proposals that do treat structures as first-class values are discussed toward the end of this section.

(3) The abstype construct in Standard ML and Miranda allows the declaration of abstract data types, but admits only one implementation per type. Haskell emulates this construct by exporting an algebraic data type without its constructors from a module; so, this requires a single implementation for each type as well. Since the abstype construct can also be

emulated in Standard ML within the module system, the former has largely been superseded by the latter.

On the type-theoretic side, Mitchell and Plotkin [1988] and subsequently Cardelli and Wegner [1985] have shown that abstract types can be represented as existentially quantified types. By stating that a value v has the existentially quantified type $\exists\alpha.\tau$, we mean that v has type $[\tilde{\tau}/\alpha]\tau$ for some fixed, but private type $\tilde{\tau}$.

This article demonstrates how light-weight abstract data types with first-class implementations can be conveniently integrated into any functional language with a static, polymorphic type system, explicit type variables, and algebraic data type declarations. The key idea of our work is to allow existentially quantified component types in algebraic data types. To be concrete, our proposal is presented as an extension to ML. It applies equally to other languages with similar type systems, such as Haskell, Hope, or Miranda. Furthermore, our proposed extension is independent of strictness considerations. We show how data types with existential component types add significant flexibility to a language without even changing its syntax. In particular, we give examples demonstrating how we express the following:

—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 representation type.

We present a deterministic type inference system in the style of Damas and Milner [1982] for our language, which leads to a syntactically sound and complete type inference algorithm. Furthermore, the type system is semantically sound with respect to a standard denotational semantics. We then extend record types to allow abstract components. The components of such abstract records are selected using the familiar dot notation. The semantic soundness of this extension is shown by a type-preserving translation [Läufer 1992] to the first extension.

Our proposal has been implemented by Leroy and Mauny [1992] in the Caml Light compiler for ML. All examples from this article have been developed and tested using this compiler and are given in Caml syntax. Most other work on existential types does not consider type inference or permit polymorphic instantiation of identifiers that have existential type. By contrast, such identifiers are let-bound in our system and may be instantiated polymorphically, as illustrated in Section 2.

Hope+C [Perry 1990] is the only prior work known to us that includes Damas-Milner-style type inference for existential types. However, the typing rules given there are not sufficient to guarantee the absence of run-time type errors, even though the Hope+C compiler seems to impose sufficient restrictions. The following unsafe program, given here in ML syntax, is well typed

according to the typing rules, but rejected by the compiler. The type variable 'a is existentially quantified.

```
type T = K of 'a
let f x = let K z = x in z
f(K 1) = f(K true)
```

Existential types combine well with the systematic overloading polymorphism provided by Haskell type classes [Wadler and Blot 1989]. We extend Haskell's data declaration similarly to the ML datatype declaration [Läufer and Odersky 1991; Läufer 1992]. In Haskell, 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 allows us to construct heterogeneous aggregates over a given type class.

Existential types are also beneficial in relation with dynamic types. Leroy and Mauny [1991] propose an extension of ML with *dynamics*, pairs consisting of a value and its type. Dynamics admit pattern matching on both the value and the run-time type. Existential types are used to match dynamic values against dynamic patterns with incomplete type information. This makes dynamics useful for typing functions such as eval. However, dynamics do not provide type abstraction because they give access to the type of an object at run-time. It seems possible to combine Leroy and Mauny's system with ours, extending their existential patterns to existential types. We are currently investigating this possibility.

Pierce and Turner [1993] describe an object-oriented language based on existential quantification instead of recursive record types. Their language is based on an extension of F_ω that includes subtyping and seems sufficiently powerful to model most features found in typical object-oriented languages, including class inheritance, reference to the methods of a superclass, and private instance variables. However, their language is explicitly typed, and algorithmic type inference is not considered.

Starting with earlier work by Mitchell and Plotkin [1988] and MacQueen [1986], there has been an ongoing discussion whether abstract types should be replaced by an advanced module system such as the one found in Standard ML [Milner et al. 1990]. Since modules are not first-class values for type-theoretic reasons, their use as abstract data types is limited. Mitchell et al. [1990] describe the possibility of treating modules as first-class values but do not address the issue of type inference. By hiding the type components of a structure, the type of the structure itself is implicitly coerced from a strong (dependent) sum type to a weak (existentially quantified) sum type. Harper and Lillibridge [1994] and independently Leroy [1994] further explore this idea in a new treatment of the Standard ML module system. In their approach, structures have weak sum types and act as first-class values. Thus, stratification of types into different universes of "small" types and "large" strong sum types is no longer necessary. Furthermore, signatures may contain *manifest type* specifications that express constraints on types in structures or functors. This treatment simplifies the sharing constraint mech-

anism of the Standard ML module system and supports true separate compilation.

In the remainder of this article, Section 2 describes an extension of algebraic data types with existential quantification. Section 3 presents a system of abstract record types with a dot notation for field selection. Section 4 contains a collection of examples. Section 5 introduces the underlying formal language, ExML. Sections 6 and 7 discuss a type system and a type inference algorithm for ExML. Section 8 presents a denotational semantics ExML, and Section 9 concludes.

2. MAKING ALGEBRAIC DATA TYPES ABSTRACT

This section illustrates how abstract data types can be provided in the form of algebraic data types with existentially quantified component types. While our extension can be applied to any language based on a polymorphic type system with algebraic data types and explicit type variables, it has been implemented in the Caml Light compiler for ML [Leroy and Mauny 1992], and all examples are given in Caml syntax. See Harper [1990] for an introduction to ML.

An algebraic data type declaration is of the form

type [*args*] $T = K_1$ of $\tau_1 \mid \ldots \mid K_k$ of τ_k

where the Ks are value constructors, and the optional prefix argument *args* is used for formal type parameters that 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.

The extension we propose works as follows: without altering the type declaration syntax, we 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, the type declaration

```
type KEY = Key of 'a * ('a → int)
```

describes a data type with one value constructor whose components are pairs of a value of type 'a and a function from type 'a to int. The question about what we can say regarding the existentially quantified type variable 'a remains. The answer is, nothing, except that it ensures that the type of value is the same as the domain of the function. To illustrate this further, the type of the expression

```
Key(3, fun x → 5)
```

is KEY, as is the type of the expression

```
Key([1; 2; 3], list_length)
```

where list_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, list_length)
```

is not type-correct because the type of 3 is different from the domain type of list_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 data types with existential component types, we can decompose them using a let expression with pattern matching. We impose the restriction that a type variable that is existentially quantified in a let expression must not appear in the result type of the expression or in the type of a global identifier. Analogous restrictions hold for the corresponding open and abstype constructs for existential types (See Cardelli and Wegner [1985] and Mitchell and Plotkin [1988] for further discussion.)

For example, assuming x is of type KEY, then

```
let (Key(v, f)) = x in f v
```

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

```
let (Key(v, f)) = x in v
```

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

Our extension allows us to deal with existential types, with the further improvement that decomposed values of existential type are let bound and may be instantiated polymorphically. This is illustrated by the example

```
type 'a T = K of ('a → 'b) * ('b → int)
let (K(f, g)) = K((fun x → x), (fun x → 3)) in
  g(f true) = g(f 7)
```

which results in true. In most prior work, the value on the right-hand side of the binding would have to be bound and decomposed twice.

3. ABSTRACT RECORDS AND THE DOT NOTATION

MacQueen [1986] observed 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. In particular, the lowest-level entities have to be opened at the outermost level. These are the traditional disadvantages of block-structured languages as compared to modular ones.

To overcome these problems, Cardelli and Leroy [1990] proposed a dot notation for existential types. We use this notation in our proposal and show that it can be combined with polymorphic type inference. We model abstract types as record types with existentially quantified component types. Values with abstract components are created by record construction and decomposed

by record field selection. This mechanism provides comparable expressiveness to modules in Modula-2 with the crucial difference that records are first-class values. (See Läufer [1992] for a formal treatment of a "dotless" dot notation in ML.)

Informally, fields selected from the same record identifier are always given compatible abstract types. We can extend this rule to nested records; fields selected from identical access paths are then given compatible abstract types. However, we disallow field selection from arbitrary record expressions because we cannot determine statically when two abstract types have compatible representations. This point is further discussed by Leroy [1994].

The following examples illustrate the dot notation in ML syntax. We start with a record type with existentially quantified component types,

```
type KEY = {x : 'a; f : 'a → int}.
```

In the first expression

```
let z = {x = 3, f = fun x → x + 2} in
  z.f z.x
```

the existential type variable in the type of f is the same as the one in the type of x, and the function application produces a result of type int. This follows from the fact that both f and x are selected from the same record identifier z. Consequently, their types must be compatible, and the whole expression is type correct.

On the other hand, the following expressions are not type correct. For instance,

```
let z = {x = 3, f = fun x → x + 2} in
  z.f
```

is incorrect because the existential type variable in the type of f escapes the scope of z. The expression

```
let z = {x = 3, f = fun x → x + 2} in
let y = z in
  z.f y.x
```

is also incorrect because different identifiers are given different private types. Since we cannot determine statically that they hold the same values in this case, we must assume that the values have different types.

Our last example involves nested records:

```
type NEST = {k1, k2 : KEY}
let z = {x = 3, f = fun x → x + 2} in
let n = {k1 = z, k2 = z} in
```

While the application n.k1.f n.k1.x would be type correct in the context of these definitions, the similar expression n.k1.f n.k2.x would not because we cannot guarantee statically that both n.k1 and n.k2 have the same representation type.

4. EXAMPLES

The following examples have been developed and tested using the Caml Light system [Leroy and Mauny 1992]. See Läufer [1992] for additional examples.

Minimum over a Heterogeneous List

Given the type declaration from Section 2,

```
type KEY = Key of 'a * ('a → int),
```

we define a heterogeneous list whose elements are of type KEY along with some auxiliary functions. Caml uses semicolons to separate list elements, as in the expression

```
let I x = x
let b2i b = if b then 1 else 0
let ks = [Key(7, I); Key([1; 2; 3], list_length); Key(true, b2i)]
```

We then define a function that takes a value of type KEY and applies the second component (the function) to the first component (the value):

```
let key(Key(x, f)) = f x.
```

Finally, we define a function that returns the smallest element of a list of KEYs with respect to the integer obtained by applying the function key to the elements:

```
let rec min = fun [x]          → x
                | (x :: xs)    → let y = min xs in
                                 if key x < key y then x else y
```

Then, the expression key (min ks) evaluates to 1.

Multiple Existentially Quantified Type Variables

It is permitted to have more than one existentially quantified type variable in the component type of a value constructor, as illustrated by the following example:

```
type MULTI = Multi of 'a * 'b * ('a → 'b → int)
let multi(Multi(x, y, f)) = f x y
let multiList =
    [Multi (3,        4,        prefix +);
     Multi([1; 2; 3], [4; 5],   (fun x y → list_length (x @ y)));
     Multi([7; 8; 9], 10,       (fun x y → list_length x + y))].
```

The application map multi multiList then results in the list [7; 5; 13]. The expression prefix + in Caml syntax is equivalent to op + in Standard ML and turns the operator + into a prefix function symbol.

Lists of Composable Functions

The algebraic data types in the preceding examples have only one constructor each. Data types with several constructors are possible as well; any existentially quantified type variables are local to the component type of the constructor in which they appear. The following type describes lists of functions,

in which the type of each function would allow it to be composed with the next:

```
type ('a, 'b) FUNLIST = FunCons of ('a → 'c) * ('c, 'b) FUNLIST
                      | FunNil of 'a → 'b.
```

This type combines universal and existential quantification. The universally quantified type variables 'a and 'b correspond to the argument type of the first and the result type of the last function, respectively. The existentially quantified type variable 'c represents the intermediate types arising during the composition of two functions. We can now construct lists of composable functions:

```
let twice x = 2 * x
let equal x y = x = y
let double x = (x, x)
let funNil = FunNil (fun x → x)
let fl = FunCons(twice, FunCons(equal 4, FunCons(double, funNil))).
```

We would like to write a function that applies a list of functions to an argument. The first, naive attempt fails because the type of apply in the recursive call is different from the type of apply on the left-hand side. This form of polymorphic recursion is not permitted in ML:

```
let rec apply = fun (FunNil f) x          → f x
                  | (FunCons (f, fl))  x → apply fl (f x).
```

We can overcome this problem by encapsulating the function list and its argument in another abstract type:

```
type 'b FUNAPPL = FunAppl of ('a, 'b) FUNLIST * 'a.
```

Thus, the recursion in the following definition of apply′ is now monomorphic as both occurrences have type 'b FUNAPPL → 'b. We then define apply in terms of apply′:

```
let rec apply' =
   fun (FunAppl (FunNil f, x))        → f x
     | (FunAppl (FunCons(f, fl), x)) → apply' (FunAppl (fl, f x))
let apply fl x = apply' (FunAppl(fl, x)).
```

So, the evaluation of the expression apply f1 2 results in (True, True).

Stacks Parameterized by Element Type

This example demonstrates how universal and existential quantification can be combined in abstract container types. We first define an abstract record type STACK with existentially quantified component types. The advantage over a tuple type is that we can refer to the components by name, as in

```
type 'a STACK =
     {Self : 'b;
      Push : 'a → 'b → 'b;
      Pop : 'b → 'b;
      Top : 'b → 'a;
      Null : 'b → bool}.
```

An on-the-fly implementation of an int STACK in terms of the built-in type list can be given as

```
{Self = [1; 2; 3]; Push = (fun x xs → x :: xs);
 Top = hd; Pop = tl; Null = (fun xs → xs = [ ])}.
```

For the systematic implementation of stacks, we provide a constructor function for each implementation; one is based on lists,

```
let makeListStack xs =
   {Self  = xs;
    Push= (fun x xs → x :: xs);
    Top  = hd;
    Pop  = tl;
    Null  = (fun xs → xs = [ ])}
```

and one on arrays,

```
let makeArrayStack xs =
   {Self  = vect_of_list (rev xs);
    Push= (fun x v → concat_vect v [|x|]);
    Top  = (fun v → vect_item v (vect_length v − 1));
    Pop  = (fun v → sub_vect v 0 (vect_length v − 1));
    Null  = (fun v → vect_length v = 0)}.
```

For dynamic dispatching, we write stack functions that work uniformly across implementations. These "wrapper" functions work by decomposing a value of type STACK, applying the intended "inner" operation to the Self component, and constructing a new value with an updated Self component:

```
let push x {Self = s; Push = p; Pop = o; Top = t; Null = n} =
   {Self = p x s; Push = p; Pop = o; Top = t; Null = n}
```

When the result type of an operation is not abstract, no encapsulation is necessary:

```
let top {Self = s; Push = p; Pop = o; Top = t; Null = n} = t s.
```

We can combine different implementations in a heterogeneous list of stacks and apply the wrapper functions to each element in the list. For example, the expression

```
map top (map (push 1) [makeListStack[2; 3; 4]; makeArrayStack[5; 6; 7]]
```

evaluates to [1; 1].

Parameterized Stacks in the Dot Notation

The dot notation lets us express the stack wrapper functions much more elegantly. We rewrite the push wrapper function to update the Self component by applying the inner Push operation. Similarly, the new top wrapper function applies the inner Top operation to the Self component. The keyword with is not part of Caml, but we use it here to express *component-wise, nondestructive* record update.

```
let push x s = s with {Self = s.Push x s.Self}
let top s    = s.Top s.Self
```

5. THE LANGUAGE ExML

ExML is an extension of Mini-ML [Clement et al. 1986] with user-defined algebraic data types. In addition to the usual constructs (identifiers, applications, λ-abstractions, and let expressions), we introduce sugar-free versions of the ML constructs that deal with data types. A data declaration introduces a new recursive data type. 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. Names z, needed in the definition of type environments, include identifiers x and value constructors K. The syntax of ExML expressions is given in Figure 1.

The syntax of ExML types includes recursive algebraic data types χ and Skolem types κ; the latter are used to type identifiers that are bounded by a pattern-matching let expression and whose type is existentially quantified. Explicit existential types η arise only as domain types of value constructors.

The match expression in source-level Caml syntax corresponds to nested if expressions with is expressions as conditions and pattern-matching let expressions that match the patterns of different cases. The following ML example

```
type 'a T = K of 'a | L of int * 'a T | M
...
match x with K y     → 1
           | L(y, z) → y
           | M       → 0
```

can be written in ExML as follows, assuming that type int is defined:

```
data ∀α.μβ.Kα + L (int × β) + M unit
in  ...
    if is K x then      1
    else if is L x then let L z = x in fst z
    else                0
```

ExML lacks special syntax for mutually recursive type declarations because mutual recursion in algebraic data types does not add any expressive power to a language that already supports ordinary μ-recursion. This is an application of Bekić's theorem, which states that a group of mutually recursive declarations can be replaced with several μ-recursive declarations by successive elimination (see Winskel [1993] for details). The following example illustrates this transformation; the source-level ML type declarations

```
type S = SNil | SCon of S | SMut of T
type T = TNil | TCon of T | TMut of S
```

translate to the following equivalent ExML declarations:

```
data μβS.SNil unit + SCon βS
       + SMut ( μβT.TNil unit + TCon βT + TMut βS) in
data μβT.TNil unit + TCon βT
       + TMut ( μβS.SNil unit + SCon βS + SMut βT) in ...
```

Type variables	α, β
Skolem types	κ
Recursive types	$\chi = \mu\beta.K_1\eta_1 + \ldots + K_k\eta_k$
Types	$\tau = unit \mid bool \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \chi \mid \kappa(\tau_1, \ldots, \tau_n)$
Existential types	$\eta = \exists\alpha.\eta \mid \tau$
Type schemes	$\sigma = \forall\alpha.\sigma \mid \tau$
Constructors	K
Identifiers	x, y
Names	$z = x, y, K$
Expressions	$e = x \mid (e_1 e_2) \mid \lambda x.e \mid$ **let** $x = e_1$ **in** e_2
	$\mid$ **data** σ **in** $e \mid K \mid$ **is** $K \mid$ **let** $K\ x = e_1$ **in** e_2

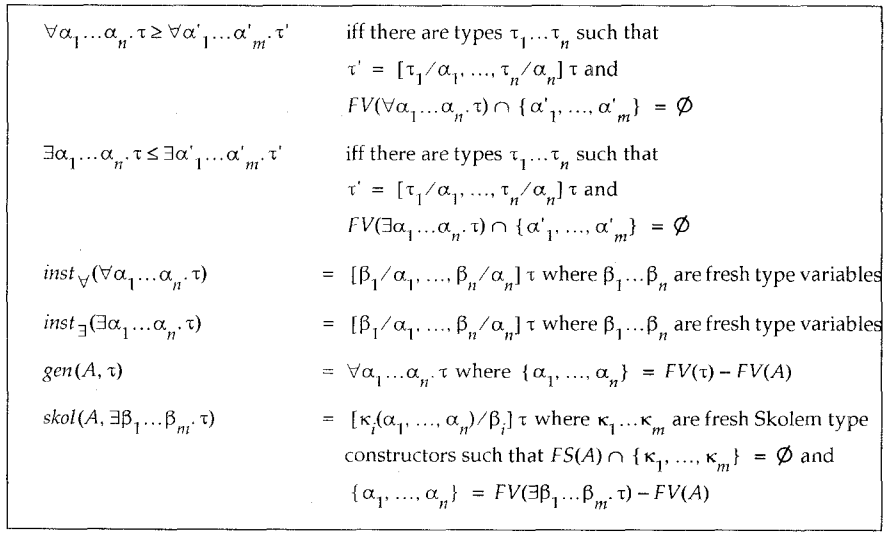
Fig. 1. Syntax of ExML types and expressions.

6. THE TYPE SYSTEM OF ExML

In this section, we present the ExML type system. Our system is deterministic and syntax directed, so there is exactly one type rule for each syntactic construct. A (*type*) *environment* is a finite mapping $A = [z_1 : \sigma_1, \ldots, z_n : \sigma_n]$ from names to type schemes. Value constructors are mapped to the recursive type schemes to which they belong. $A(K)$ is the type scheme σ such that $\sigma = \forall\alpha_1 \ldots \alpha_n \ldots + K\eta + \ldots$. The *domain* of A is $Dom(A) = \{z_1, \ldots, z_n\}$. The extension $A[z : \sigma]$ is a new environment that maps z to σ and every z' in $Dom(A)$ to $A(z')$. The free type variables in A are given by $FV(A) = FV(A(z_1)) \cup \ldots \cup FV(A(z_n))$. The free Skolem types in a type τ are given by $FS(\tau)$; FS generalizes to environments analogously to FV.

The auxiliary predicates and functions in Figure 2 are used in the type inference rules. The predicates $\geq$ and $\leq$ describe instantiation of type schemes and generalization of existential types, respectively. The corresponding functions $inst_\forall$ and $inst_\exists$ replace the bound type variables in type schemes and existential types with fresh type variables, and they are used in the type inference algorithm (see Section 7). The function *gen* universally quantifies all free variables in a type that are not free in the environment. Finally, the function *skol* replaces all bound type variables in an existential type by fresh Skolem type constructors parameterized by the free type variables in the environment.

The four typing rules shown in Figure 3 are the same as in the Mini-ML system. They are used in the typing of variables, abstractions, applications, and let expressions. Four new rules are given in Figure 4; they are used to type data type declarations, value constructors, **is** expressions, and pattern-matching **let** expressions. We explain each of the new rules in turn. The



$\forall\alpha_1 \ldots \alpha_n.\tau \geq \forall\alpha'_1 \ldots \alpha'_m.\tau'$ iff there are types $\tau_1 \ldots \tau_n$ such that
$\tau' = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau$ and
$FV(\forall\alpha_1 \ldots \alpha_n.\tau) \cap \{\alpha'_1, \ldots, \alpha'_m\} = \emptyset$

$\exists\alpha_1 \ldots \alpha_n.\tau \leq \exists\alpha'_1 \ldots \alpha'_m.\tau'$ iff there are types $\tau_1 \ldots \tau_n$ such that
$\tau' = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau$ and
$FV(\exists\alpha_1 \ldots \alpha_n.\tau) \cap \{\alpha'_1, \ldots, \alpha'_m\} = \emptyset$

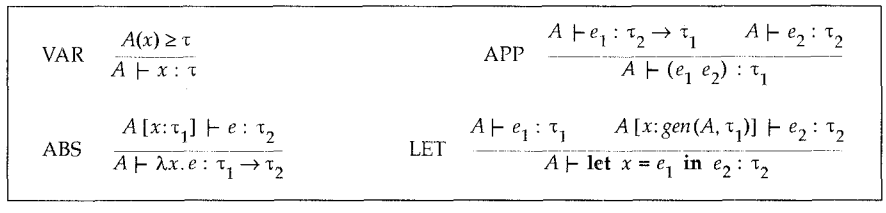
$inst_\forall(\forall\alpha_1 \ldots \alpha_n.\tau) = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\tau$ where $\beta_1 \ldots \beta_n$ are fresh type variables

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

$gen(A, \tau) = \forall\alpha_1 \ldots \alpha_n.\tau$ where $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau) - FV(A)$

$skol(A, \exists\beta_1 \ldots \beta_{ni}.\tau) = [\kappa_i(\alpha_1, \ldots, \alpha_n)/\beta_i]\tau$ where $\kappa_1 \ldots \kappa_m$ are fresh Skolem type constructors such that $FS(A) \cap \{\kappa_1, \ldots, \kappa_m\} = \emptyset$ and $\{\alpha_1, \ldots, \alpha_n\} = FV(\exists\beta_1 \ldots \beta_m.\tau) - FV(A)$

Fig. 2. Auxiliary predicates and functions for type inference.



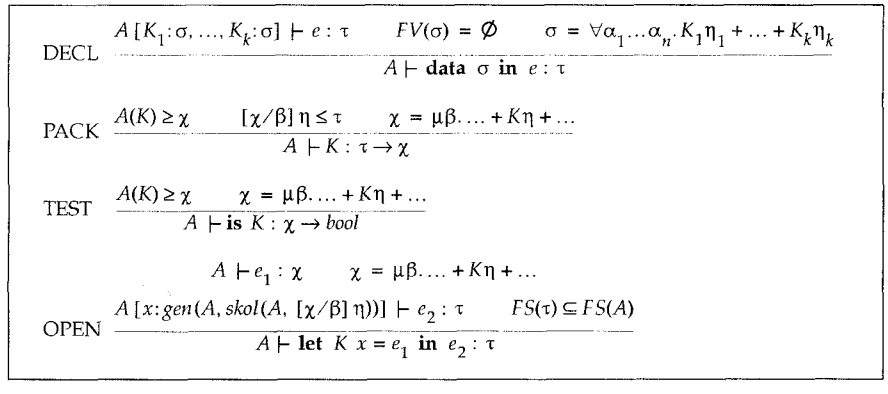
VAR $\dfrac{A(x) \geq \tau}{A \vdash x : \tau}$

APP $\dfrac{A \vdash e_1 : \tau_2 \to \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash (e_1\ e_2) : \tau_1}$

ABS $\dfrac{A[x{:}\tau_1] \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \to \tau_2}$

LET $\dfrac{A \vdash e_1 : \tau_1 \qquad A[x{:}gen(A, \tau_1)] \vdash e_2 : \tau_2}{A \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$

Fig. 3. Type inference rules for mini-ML expressions.



DECL $\dfrac{A[K_1{:}\sigma, \ldots, K_k{:}\sigma] \vdash e : \tau \qquad FV(\sigma) = \emptyset \qquad \sigma = \forall\alpha_1 \ldots \alpha_n.K_1\eta_1 + \ldots + K_k\eta_k}{A \vdash \textbf{data}\ \sigma\ \textbf{in}\ e : \tau}$

PACK $\dfrac{A(K) \geq \chi \qquad [\chi/\beta]\eta \leq \tau \qquad \chi = \mu\beta.\ldots + K\eta + \ldots}{A \vdash K : \tau \to \chi}$

TEST $\dfrac{A(K) \geq \chi \qquad \chi = \mu\beta.\ldots + K\eta + \ldots}{A \vdash \textbf{is}\ K : \chi \to bool}$

OPEN $\dfrac{A \vdash e_1 : \chi \qquad \chi = \mu\beta.\ldots + K\eta + \ldots \qquad A[x{:}gen(A, skol(A, [\chi/\beta]\eta))] \vdash e_2 : \tau \qquad FS(\tau) \subseteq FS(A)}{A \vdash \textbf{let}\ K\ x = e_1\ \textbf{in}\ e_2 : \tau}$

Fig. 4. Type inference rules for ExML expressions involving existential types.

DECL rule elaborates a recursive data type declaration **data** σ **in** e. This adds the new outermost constructors $K_1, \ldots, K_k$ to the environment as belonging to type σ. It also guarantees that σ does not contain any free type variables. The PACK rule assigns a type to a constructor K by looking up in the environment the recursive type to which K belongs. By generalizing the component type $[\chi/\beta]\eta$ of the constructor to the type τ of the prospective argument, the rule observes that existential quantification in argument position means universal quantification over the whole function or constructor type. The TEST rule ensures that the predicate **is** K is applied only to arguments whose type χ is an instance of the result type of the constructor K. Finally, the OPEN rule governs the typing of pattern-matching **let** expressions. It requires that the expression e_1 be an instance of the result type χ of the constructor K. Then it types the body e_2 under the environment extended with a typing for the bound identifier x, whose type is a Skolemized, generalized version of the component type of K. The new Skolem types $\kappa_1, \ldots, \kappa_m$ must not appear in A; this ensures that they do not appear in the type of any identifier free in e_2 other than x. The rule also guarantees that the Skolem types do not appear in the result type τ.

The DECL rule does not prohibit nesting undeclared recursive types within the data type being declared. As a consequence of the PACK rule, however, values of the nested data type can only be constructed if its outermost value constructors have already been put in the environment by a preceding application of the DECL rule. Furthermore, if the same value constructor is redeclared in a subsequent data type declaration, then that declaration hides the first one. Therefore, a recursive data type comes into existence only by the presence of its outermost value constructors in the environment. This mechanism corresponds to generativity in ML. The following theorem states that ExML is a conservative extension of Mini-ML:

THEOREM 6.1. *For any Mini-ML expression* e, $A \vdash e : \tau$ *iff* $A \vdash_{Mini\text{-}ML} e : \tau$.

PROOF. By structural induction on e. □

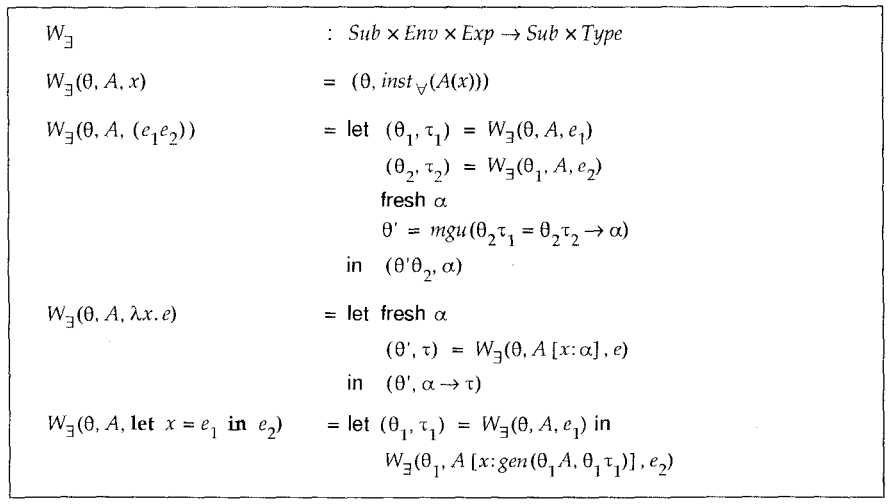
The theorem still holds if we extend Mini-ML to include recursive data types and pattern-matching **let** expressions without existential quantification.

7. COMPUTING PRINCIPAL TYPES FOR ExML

In this section, we present the type inference algorithm $W_\exists$ for ExML and show its correctness. We start out with some definitions. For an environment A and a substitution θ, we define $\theta A = [x : \theta(A(x)) | x \in Dom(A)]$. We call A a *closed* environment if $FV(A) = \emptyset$. The free variables of a substitution θ are given by

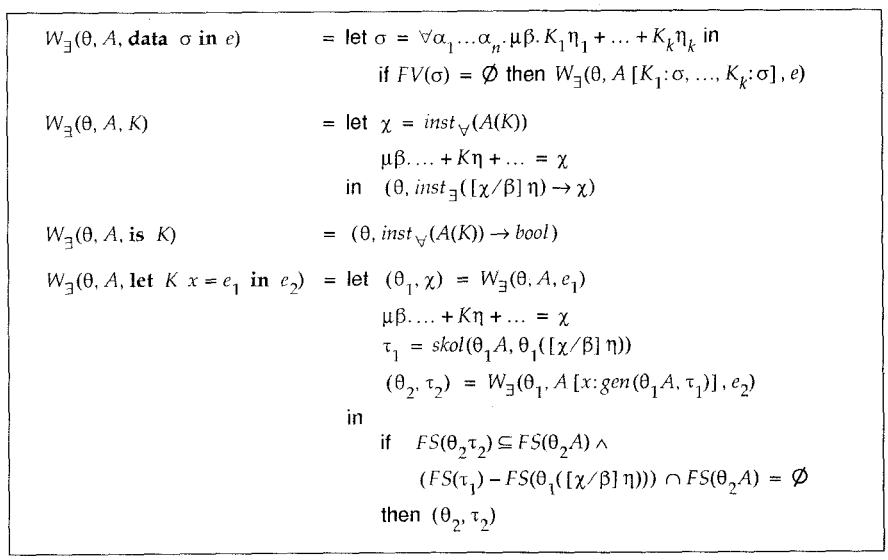
$$FV(\theta) = Dom(\theta) \cup \bigcup_{\alpha \in Dom(\theta)} FV(\theta\alpha).$$

Because the algorithm $W_\exists$ follows the syntax-directed type inference rules, there is one case for each rule. $W_\exists$ takes as arguments the current substitu-



$$
\begin{array}{ll}
W_\exists & : \mathit{Sub} \times \mathit{Env} \times \mathit{Exp} \to \mathit{Sub} \times \mathit{Type} \\
W_\exists(\theta, A, x) & = (\theta, \mathit{inst}_\forall(A(x))) \\
W_\exists(\theta, A, (e_1 e_2)) & = \text{let } (\theta_1, \tau_1) = W_\exists(\theta, A, e_1) \\
 & \qquad (\theta_2, \tau_2) = W_\exists(\theta_1, A, e_2) \\
 & \qquad \text{fresh } \alpha \\
 & \qquad \theta' = \mathit{mgu}(\theta_2\tau_1 = \theta_2\tau_2 \to \alpha) \\
 & \quad \text{in } (\theta'\theta_2, \alpha) \\
W_\exists(\theta, A, \lambda x.e) & = \text{let fresh } \alpha \\
 & \qquad (\theta', \tau) = W_\exists(\theta, A[x{:}\alpha], e) \\
 & \quad \text{in } (\theta', \alpha \to \tau) \\
W_\exists(\theta, A, \textbf{let } x = e_1 \textbf{ in } e_2) & = \text{let } (\theta_1, \tau_1) = W_\exists(\theta, A, e_1) \text{ in} \\
 & \qquad W_\exists(\theta_1, A[x{:}\mathit{gen}(\theta_1 A, \theta_1\tau_1)], e_2)
\end{array}
$$

Fig. 5. Type inference algorithm for mini-ML expressions.



$$
\begin{array}{ll}
W_\exists(\theta, A, \textbf{data } \sigma \textbf{ in } e) & = \text{let } \sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_k\eta_k \text{ in} \\
 & \qquad \text{if } FV(\sigma) = \emptyset \text{ then } W_\exists(\theta, A[K_1{:}\sigma, \ldots, K_k{:}\sigma], e) \\
W_\exists(\theta, A, K) & = \text{let } \chi = \mathit{inst}_\forall(A(K)) \\
 & \qquad \mu\beta.\ldots + K\eta + \ldots = \chi \\
 & \quad \text{in } (\theta, \mathit{inst}_\exists([\chi/\beta]\eta) \to \chi) \\
W_\exists(\theta, A, \textbf{is } K) & = (\theta, \mathit{inst}_\forall(A(K)) \to \mathit{bool}) \\
W_\exists(\theta, A, \textbf{let } K\ x = e_1 \textbf{ in } e_2) & = \text{let } (\theta_1, \chi) = W_\exists(\theta, A, e_1) \\
 & \qquad \mu\beta.\ldots + K\eta + \ldots = \chi \\
 & \qquad \tau_1 = \mathit{skol}(\theta_1 A, \theta_1([\chi/\beta]\eta)) \\
 & \qquad (\theta_2, \tau_2) = W_\exists(\theta_1, A[x{:}\mathit{gen}(\theta_1 A, \tau_1)], e_2) \\
 & \quad \text{in} \\
 & \qquad \text{if } FS(\theta_2\tau_2) \subseteq FS(\theta_2 A) \wedge \\
 & \qquad\quad (FS(\tau_1) - FS(\theta_1([\chi/\beta]\eta))) \cap FS(\theta_2 A) = \emptyset \\
 & \qquad \text{then } (\theta_2, \tau_2)
\end{array}
$$

Fig. 6. Type inference algorithm for ExML expressions involving existential types.

tion, the current environment, and the expression to be typed; and returns the new substitution and the inferred type of the expression. The four cases in Figure 5 are identical to algorithm W [Damas 1985]. The four additional cases given in Figure 6 deal with data type declarations, value constructors, **is** expressions, and pattern-matching **let** expressions:

The **"data"** case adds the new outermost constructors $K_1, \ldots, K_k$ in a recursive data type declaration to the environment and checks that the new

type does not contain any free type variables. The "K" case first looks up the data type $A(K)$ to which the constructor K belongs. Then, it generalizes the component type of K and instantiates the result type $A(K)$ with fresh type variables. The assigned type guarantees that the constructor K is applied only to arguments whose type is a generalization of the component type of K. Similarly, the "**is** K" case looks up the data type $A(K)$ and instantiates this type with fresh type variables. The assigned type guarantees that the predicate **is** K is applied only to arguments whose type is an instance of the result type of K. Finally, the "**let** K" case assigns a type to a pattern-matching **let** expression. It requires that the expression e_1 be an instance of the result type χ of the constructor K. It then types the body e_2 under an extended environment, where the type of the bound identifier x is a Skolemized version of the argument type of K. This case also checks that the new Skolem types $\kappa_1, \ldots, \kappa_m$ do not appear in A or in the result type τ of e_2.

The remainder of this section presents the lemmas and theorems needed to establish the soundness and completeness of the algorithm. (See Läufer [1992] for proofs.)

LEMMA 7.1. *If* $A \vdash e : \tau$, *then* $\theta A \vdash e : \theta\tau$.

PROOF. By induction on the structure of e. □

THEOREM 7.2. (Syntactic Soundness of Algorithm $W_\exists$). *If* $W_\exists(\theta, A, e) = (\theta', \tau)$, *then* $\theta' A \vdash e : \theta'\tau$.

PROOF. By induction on the structure of e, using Lemma 7.1. □

Definition 7.3. Let A be a closed environment. The type τ is a *principal type* of an expression e if $A \vdash e : \tau$ and if $A \vdash e : \tau'$ implies $gen(A, \tau) \geq \tau'$.

LEMMA 7.4. *If* $A' \vdash e : \tau'$ *where* $A' = \delta'\theta A$, *then* $W_\exists(\theta, A, e) = (\theta_1, \tau_1)$ *and there exists a* δ_1 *such that* $A' = \delta_1\theta_1 A$ *and* $\tau' = \delta_1\theta_1\tau_1$.

PROOF. By induction on the structure of e. □

THEOREM 7.5. (Syntactic Completeness and Principal Types). *If* $A \vdash e : \tau'$ *and* A *is closed, then* $W_\exists(\varnothing, A, e) = (\theta, \tau)$ *and* $\theta\tau$ *is a principal type of* e.

PROOF. Follows from Lemma 7.4. □

8. A FORMAL SEMANTICS FOR ExML

In this section we present a standard denotational semantics of ExML and show that our type system is sound with respect to this semantics. The evaluation function E maps an expression e to some semantic value v, in the context of an evaluation environment ρ. An evaluation environment is a partial mapping from identifiers to semantic values. Tagged values are used to capture the semantics of algebraic data types. We distinguish between three error situations: run-time type errors (wrong), nontermination ($\perp$), and a mismatch (fail) when an attempt is made to decompose a tagged value whose tag does not match the tag of the destructor. Our type inference system is sound with respect to the evaluation function: a well-typed program

Unit domain	$U = \{\mathsf{unit}\}_{\perp,\,\mathsf{fail}}$
Boolean domain	$B = \{\mathsf{false}, \mathsf{true}\}_{\perp,\,\mathsf{fail}}$
Constructor tags	$T = \{K_1, K_2, \ldots\}_{\perp,\,\mathsf{fail}}$
Semantic domain	$V \cong U + B + T + (V \to V) + (V \times V) + \{\mathsf{wrong}\}_{\perp,\,\mathsf{fail}}$

Fig. 7. Semantic domains.

$$
\begin{array}{ll}
E & : Exp \to EEnv \to V \\
E[\![\, x \,]\!]\rho & = \rho(x) \\
E[\![\, e_1 e_2 \,]\!]\rho & = \mathsf{if}\ E[\![\, e_1 \,]\!]\rho \in V \to V\ \mathsf{then} \\
 & \quad (E[\![\, e_1 \,]\!]\rho)(E[\![\, e_2 \,]\!]\rho) \\
 & \quad \mathsf{else\ wrong} \\
E[\![\, \lambda x.e \,]\!]\rho & = \lambda v \in V.E[\![\, e \,]\!]\rho[x{:}v] \\
E[\![\, \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \,]\!]\rho & = E[\![\, e_2 \,]\!]\rho[x{:}E[\![\, e_1 \,]\!]\rho] \\
E[\![\, \mathbf{data}\ \sigma\ \mathbf{in}\ e \,]\!]\rho & = E[\![\, e \,]\!]\rho \\
E[\![\, K \,]\!]\rho & = \lambda v \in V.\langle K, v\rangle \\
E[\![\, \mathbf{is}\ K \,]\!]\rho & = \lambda v \in V.v \in \{K\} \times V \\
E[\![\, \mathbf{let}\ K\ x = e_1\ \mathbf{in}\ e_2 \,]\!]\rho & = E[\![\, e_2 \,]\!]\rho[x{:}\ \mathsf{if}\ E[\![\, e_1 \,]\!]\rho \in \{K\} \times V\ \mathsf{then} \\
 & \quad \mathsf{snd}(E[\![\, e_1 \,]\!]\rho) \\
 & \quad \mathsf{else\ fail}]
\end{array}
$$

Fig. 8. Semantic functions for expressions.

never evaluates to wrong. The semantic domains of ExML are shown in Figure 7. In the definition of V, $+$ stands for the coalesced sum, so all types over V share the same $\perp$ and fail values. The semantic function E for ExML expressions is given in Figure 8. Although E is strict in both $\perp$ and fail to model the semantics of the ML language, our soundness considerations are orthogonal to the issue of strictness.

We identify types with *weak ideals* [MacQueen et al. 1986] over the semantic domain V. A type environment ψ is a partial mapping from type variables to ideals and from Skolem type constructors to functions between ideals. The semantic interpretation of types is defined in Figure 9. The universal and existential quantifications range over the set $\mathfrak{R} \subseteq \mathfrak{I}(V)$ of all ideals that do not contain wrong.

It should be noted that our interpretation handles only μ-recursive data types, which can always be expressed in the form $\forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_k\eta_k$. *Nonregular* data types, such as

```
type 'a NONREG = Leaf | Node of 'a * ('a NONREG) NONREG
```

$$
\begin{array}{ll}
T & : TExp \to TEnv \to \Im(V) \\
T[\![\ unit\ ]\!]\psi & = U \\
T[\![\ bool\ ]\!]\psi & = B \\
T[\![\ \alpha\ ]\!]\psi & = \psi(\alpha) \\
T[\![\ \sigma_1 \times \sigma_2\ ]\!]\psi & = T[\![\ \sigma_1\ ]\!]\psi \times T[\![\ \sigma_2\ ]\!]\psi \\
T[\![\ \sigma_1 \to \sigma_2\ ]\!]\psi & = T[\![\ \sigma_1\ ]\!]\psi \to T[\![\ \sigma_2\ ]\!]\psi \\
T[\![\ \kappa(\sigma_1, \ldots, \sigma_n)\ ]\!]\psi & = (\psi(\kappa))(T[\![\ \sigma_1\ ]\!]\psi, \ldots, T[\![\ \sigma_n\ ]\!]\psi) \\
T[\![\ K_1\eta_1 + \ldots + K_k\eta_k\ ]\!]\psi & = \{K_1\} \times T[\![\ \eta_1\ ]\!]\psi \cup \ldots \cup \{K_k\} \times T[\![\ \eta_k\ ]\!]\psi \\
T[\![\ \mu\alpha.\sigma\ ]\!]\psi & = \mu(\lambda I \in \Im(V).T[\![\ \sigma\ ]\!]\psi[\alpha{:}I]) \\
T[\![\ \forall\alpha.\sigma\ ]\!]\psi & = \bigsqcap_{I \in \Re} \lambda I \in \Im(V).T[\![\ \sigma\ ]\!]\psi[\alpha{:}I] \\
T[\![\ \exists\alpha.\sigma\ ]\!]\psi & = \bigsqcup_{I \in \Re} \lambda I \in \Im(V).T[\![\ \sigma\ ]\!]\psi[\alpha{:}I]
\end{array}
$$

Fig. 9. Semantic functions for types.

would require recursion over type constructors. An adequate semantics for nonregular types can be given by extending the weak ideal model (private communication with M. Abadi, June 1992). The machinery for this extension is given by Plotkin [1983].

THEOREM 8.1. *The semantic function for types is well defined.*

PROOF. All type expressions are formally contractive [MacQueen et al. 1986], so the fixed points exist. □

Definition 8.2. (*Semantic Type Judgment*). Let A be an assumption set, e an expression, and σ a type scheme.

(i) $\models_{\rho,\psi} A$ means that for every $x \in Dom(A)$, $x \in Dom(\rho)$ and $\rho(x) \in T[\![A(x)]\!]\psi$;
(ii) $A \models_{\rho,\psi} e : \sigma$ means that $\models_{\rho,\psi} A$ implies $E[\![e]\!]\rho \in T[\![\sigma]\!]\psi$; and
(iii) $A \models e : \tau$ means that $A \models_{\rho,\psi} e : \sigma$ for all $\rho \in EEnv$ and $\psi \in TEnv$.

THEOREM 8.3. (SEMANTIC SOUNDNESS). *If* $A \vdash e : \tau$, *then* $A \models e : \tau$.

PROOF. By induction on the structure of the proof tree for $A \vdash e : \tau$. □

COROLLARY 8.4. *If* $A \vdash e : \tau$ *and* $\models_{\rho,\psi} A$, *then* $E[\![e]\!]\rho \neq$ wrong.

9. CONCLUSIONS

We demonstrated how light-weight abstract data types with first-class implementations regardless of strictness considerations can be integrated into any functional language with a static, polymorphic type system, explicit type variables, and algebraic data type declarations. We showed how abstract data

types add significant flexibility and expressiveness to a language without even changing its syntax. Then, we presented a type system that extends the Damas-Milner system with existentially quantified component types of data and record types, provided a type inference algorithm, and proved that the type system is semantically sound.

The work on first-class modules by Harper and Lillibridge [1994] and independently Leroy [1994] can provide alternative solutions to the problems that motivated our proposal. However, modules are rather heavy semantic machinery for expressing first-class abstract data types. By contrast, our data and record types with existentially quantified component types have quite a different flavor: they provide light-weight abstract types with easily understandable semantics functioning directly in the functional core of a language. Thus, it might be desirable to include both proposals in the same language and even to allow some redundancy between core and module languages.

ACKNOWLEDGMENTS

We would like to express thanks to Martin Abadi, Ben Goldberg, Fritz Henglein, Radha Jagadeesan, Stefan Kaes, Xavier Leroy, Michel Mauny, Tobias Nipkow, Ross Paterson, Nigel Perry, Benjamin Pierce, and Phil Wadler for helpful feedback and stimulating discussions. We are also grateful to the three anonymous referees for their detailed comments.

REFERENCES

BURSTALL, R., MACQUEEN, D., AND SANNELLA, D. 1980. Hope: An experimental applicative language. In *Stanford LISP Conference 1980*, 136–143.

CARDELLI, L. AND LEROY, X. 1990. Abstract types and the dot notation. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods* (Sea of Galilee, Israel). IFIP, 466–491.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec.), 471–522.

CLEMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. 1986. A simple applicative language: Mini-ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. ACM, New York, 13–27.

DAMAS, L. 1985. Type assignment in programming languages. Ph.D. thesis, Univ. of Edinburgh, Edinburgh, Scotland.

DAMAS, L. AND MILNER, R. 1982. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, (Jan.). ACM, New York, 207–212.

HARPER, R. 1990. Introduction to Standard ML. Tech. Rep., School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

HARPER, R. AND LILLIBRIDGE, M. A. 1994. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21th ACM Symposium on Principles of Programming Languages*. ACM, New York, 123–137.

HUDAK, P., PEYTON-JONES, S., WADLER, P., EDS. 1992. Report on the programming language Haskell: A non-strict, purely functional language Version 1.2. *ACM SIGPLAN Not. 27*, 5 (May).

LÄUFER, K. 1992. Polymorphic type inference and abstract data types. Ph.D. thesis, New York Univ., New York. Available as Tech. Rep. 622, Dec. 1992, from New York Univ., Dept. of Computer Science.

LÄUFER, K. AND ODERSKY, M. 1991. Type classes are signatures of abstract types. In *Proceedings of the Phoenix Seminar and Workshop on Declarative Programming* (Nov.).

LEROY, X. 1994. Manifest types, modules, and separate compilation. In *Proceedings of the 21th ACM Symposium on Principles of Programming Languages*. (Jan.). ACM, New York, 109–123.

LEROY, X. AND MAUNY, M. 1992. *The Caml Light System, Release 0.5, Documentation and User's Manual*. Sept. Distributed with the Caml Light system.

LEROY, X. AND MAUNY, M. 1991. Dynamics in ML. In *Proceedings ACM Functional Programming Languages and Computer Architecture*. ACM, New York, 406–426.

MACQUEEN, D. 1986. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM, New York, 277–286.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. 1986. An ideal model for recursive polymorphic types. *Inf. Control 71*, 95–130.

MEYER, A. AND REINHOLD, M. 1986. Type is not a type. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. (Jan.). ACM, New York, 287–295.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Mass.

MITCHELL, J. AND HARPER, R. 1988. The essence of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (Jan.). ACM, New York.

MITCHELL, J. AND PLOTKIN, G. 1988. Abstract types have existential type. *ACM Trans. Program. Lang. Syst. 10*, 3, 470–502.

MITCHELL, J., MELDAL, S., AND MADHAV, N. 1991. An extension of Standard ML modules with subtyping and inheritance. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages* (Jan.). ACM, New York.

ODERSKY, M. 1991. Objects and subtyping in a functional perspective. Tech. Rep. RC 16423, IBM, T. J. Watson Research Center, Yorktown Heights, N.Y.

PERRY, N. 1990. The implementation of practical functional programming languages. Ph.D. thesis, Imperial College, London, U.K.

PIERCE, B. AND TURNER, D. 1993. Simple type-theoretic foundations for object-oriented programming. *J. Functional Program.* (April). To be published.

PLOTKIN, G. 1983. *Domains. Course notes*. TeX-ed edition.

TURNER, D. 1985. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of Functional Programming Languages and Computer Architecture* (Nancy, France). Lecture Notes in Computer Science, vol. 201, Springer-Verlag, New York, 1–16.

WADLER, P. AND BLOTT, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages* (Jan.). ACM, New York, 60–76.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Mass.

Received October 1993; revised March 1994; accepted May 1994