Polymorphic Type Inference and Abstract Data Types

Konstantin Läufer
Loyola University of Chicago
laufer@math.luc.edu

Martin Odersky
Universität Karlsruhe
odersky@ira.uka.de

June 5, 1994

Technical Report LUC-001
Abridged version to appear in Transactions of Programming Languages and Systems

Abstract

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 a 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 60660, laufer@math.luc.edu; M. Odersky, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany, 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 [10], Hope [2], Miranda [27], and ML [19], 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 several distinct constructs that can be used to express abstract data types:

- Tuples or records of closures can be used to model abstract data types [23]. 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 [11].
- 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 [18, 20]. Therefore structures are not treated as first-class values. However, this causes considerable difficulties in a number of practical programming situations [11]. (Some recent proposals that do treat structures as first-class values are discussed toward the end of this section.)
- 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; thus it 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 [22] and subsequently Cardelli and Wegner [4] 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 paper 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. For the sake of concreteness, our proposal is presented as an extension to ML. It equally applies 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

- 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 [7] 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 [11] to the first extension.

Our proposal has been implemented by Leroy and Mauny [15] in the Caml Light compiler for ML. All examples from this paper 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 [24] 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, here given 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 [28]. We extend Haskell's `data` declaration similarly to the ML `datatype` declaration [12, 11]. 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 [14] 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 since 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 point.

Pierce and Turner [25] 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 [22] and MacQueen [16], 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 [19]. Since modules are not first-class values for type-theoretic reasons, their use as abstract data types is limited.

Mitchell, Meldal, and Madhav [21] 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 [9] and independently Leroy [13] 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 mechanism of the Standard ML module system and supports true separate compilation.

In the remainder of this paper, 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. Section 6 and Section 7 discuss a type system and a type inference algorithm for ExML. Section 8 presents a denotational semantics for 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 [15], and all examples are given in Caml syntax. (See [8] for an introduction to ML.)

An algebraic data type declaration is of the form

$$\texttt{type}\ [\mathit{args}]\ T\ \texttt{=}\ K_1\ \texttt{of}\ \tau_1\ \mid\ \ldots\ \mid\ K_k\ \texttt{of}\ \tau_k$$

where the K's 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's.

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 is what we can say about the existentially quantified type variable `'a`. The answer is, nothing, except that it ensures that the type of the 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 `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 since 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 [4, 22] 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 since 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 following 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 [16] 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. 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 [3] propose 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 Appendix B 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 since we cannot determine statically when two abstract types have compatible representations. This point is further discussed by Leroy [13].

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 since the existential type variable in the type of `f` escapes the scope of `z`. So is

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

because different identifiers are given different private types. As 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 since 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 [15]. (See [11] 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.

```
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 `KEY`s 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 each have only one constructor. 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, for example:

```
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 since 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`, and we 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))
```

Evaluation of the expression `apply fl 2` then 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.

```
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 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; we use it here to express *component-wise, non-destructive* 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 [5] 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.

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	$e \;=\; x \mid (e_1 e_2) \mid \lambda x.e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$
	$\quad\mid \textbf{data}\ \sigma\ \textbf{in}\ e \mid K \mid \textbf{is}\ K \mid \textbf{let}\ K\ x = e_1\ \textbf{in}\ e_2$

Figure 1: Syntax of ExML types and expressions

The syntax of ExML types includes recursive algebraic data types χ and Skolem types κ; the latter are used to type identifiers that are bound 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 syntax of ExML types is included in Figure 1.

The `match`-expression in source-level Caml syntax corresponds to nested **if**-expressions with **is**-expressions as conditions and pattern-matching **let**-expressions for the 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:

$$\begin{array}{ll} \textbf{data}\ \forall\alpha.\,\mu\beta.\,K\alpha + L\ \ (int \times \beta)\ + M\ \ unit & \\ \textbf{in}\ \ \ldots & \\ \quad \textbf{if is}\ K\,x\ \textbf{then} & 1 \\ \quad \textbf{else if is}\ L\,x\ \textbf{then} & \textbf{let}\ L\,z = x\ \textbf{in}\ fst\ z \\ \quad \textbf{else} & 0 \end{array}$$

ExML lacks special syntax for mutually recursive type declarations since 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 [29] 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:

$$\textbf{data}\ \ \mu\beta_S.\,SNil\ \ unit + SCon\ \ \beta_S + SMut\ \ (\mu\beta_T.\,TNil\ \ unit + TCon\ \ \beta_T + TMut\ \ \beta_S)\ \ \textbf{in}$$

$$\textbf{data}\ \ \mu\beta_T.\,TNil\ \ unit + TCon\ \ \beta_T + TMut\ \ (\mu\beta_S.\,SNil\ \ unit + SCon\ \ \beta_S + SMut\ \ \beta_T)\ \ \textbf{in}\ldots$$

6 The Type System of ExML

In this section, we present the type system of ExML. Our system is deterministic and syntax-directed, thus 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 all 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 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 that are 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:

$\forall\alpha_1\ldots\alpha_n.\tau \geq \forall\alpha'_1\ldots\alpha'_m.\tau'$		iff there are types $\tau_1\ldots\tau_n$ such that $\tau' = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\,\tau$ and $FV(\forall\alpha_1\ldots\alpha_n.\tau) \cap \{\alpha'_1, \ldots, \alpha'_m\} = \varnothing$
$\exists\alpha_1\ldots\alpha_n.\tau \leq \exists\alpha'_1\ldots\alpha'_m.\tau'$		iff there are types $\tau_1\ldots\tau_n$ such that $\tau' = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\,\tau$ and $FV(\exists\alpha_1\ldots\alpha_n.\tau) \cap \{\alpha'_1, \ldots, \alpha'_m\} = \varnothing$
$inst_\forall(\forall\alpha_1\ldots\alpha_n.\tau)$	$=$	$[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\,\tau$ where $\beta_1\ldots\beta_n$ are fresh type vars
$inst_\exists(\exists\alpha_1\ldots\alpha_n.\tau)$	$=$	$[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\,\tau$ where $\beta_1\ldots\beta_n$ are fresh type vars
$gen(A, \tau)$	$=$	$\forall\alpha_1\ldots\alpha_n.\tau$ where $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau) - FV(A)$
$skol(A, \exists\beta_1\ldots\beta_m.\tau)$	$=$	$[\kappa_i(\alpha_1, \ldots, \alpha_n)/\beta_i]\,\tau$ where $\kappa_1\ldots\kappa_m$ are fresh Skolem type constructors such that $FS(A) \cap \{\kappa_1, \ldots, \kappa_m\} = \varnothing$ and $\{\alpha_1, \ldots, \alpha_n\} = FV(\exists\beta_1\ldots\beta_m.\tau) - FV(A)$

Figure 2: Auxiliary predicates and functions for type inference

$$\text{VAR}\quad \frac{A(x) \geq \tau}{A \vdash x : \tau} \qquad\qquad \text{APP}\quad \frac{A \vdash e_1 : \tau_2 \to \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash (e_1\ e_2) : \tau_1}$$

$$\text{ABS}\quad \frac{A[x{:}\tau_1] \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad\qquad \text{LET}\quad \frac{A \vdash e_1 : \tau_1 \qquad A[x{:}gen(A, \tau_1)] \vdash e_2 : \tau_2}{A \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

Figure 3: Type inference rules for Mini-ML expressions

The DECL rule elaborates a recursive data type declaration **data** σ **in** e. It 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. It then types the body

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

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

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

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

Figure 4: Type inference rules for ExML expressions involving existential types

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

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 $\theta A = [x{:}\theta\,(A(x)) \mid x \in Dom(A)]$. We call A a *closed* environment if $FV(A) = \varnothing$. The free variables of a substitution θ are given by

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

The algorithm $W_\exists$ follows the syntax-directed type inference rules, hence there is one case for each rule. $W_\exists$ takes as arguments the current substitution, the current environment, and the expression to be typed; it returns the new substitution and the inferred type of the expression. The four cases in Figure 5 are identical to algorithm W [6]. The four additional cases given in Figure 6 deal with data type declarations, value constructors, **is**-expressions, and pattern-matching **let**-expressions:

$$
\begin{array}{lll}
W_\exists & : & Sub \times Env \times Exp \to Sub \times Type \\[1ex]
W_\exists(\theta, A, x) & = & (\theta, inst_\forall(A(x))) \\[1ex]
W_\exists(\theta, A, (e_1 e_2)) & = & \text{let } \; (\theta_1, \tau_1) = W_\exists(\theta, A, e_1) \\
& & \qquad (\theta_2, \tau_2) = W_\exists(\theta_1, A, e_2) \\
& & \qquad \text{fresh } \alpha \\
& & \qquad \theta' = mgu(\theta_2\tau_1 = \theta_2\tau_2 \to \alpha) \\
& & \text{in } \; (\theta'\theta_2, \alpha) \\[1ex]
W_\exists(\theta, A, \lambda x. e) & = & \text{let fresh } \alpha \\
& & \qquad (\theta', \tau) = W_\exists(\theta, A[x : \alpha], e) \\
& & \text{in } \; (\theta', \alpha \to \tau) \\[1ex]
W_\exists(\theta, A, \mathbf{let} \; x = e_1 \; \mathbf{in} \; e_2) & = & \text{let } (\theta_1, \tau_1) = W_\exists(\theta, A, e_1) \text{ in} \\
& & \qquad W_\exists(\theta_1, A[x : gen(\theta_1 A, \theta_1\tau_1)], e_2)
\end{array}
$$

Figure 5: Type inference algorithm for Mini-ML 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. It then 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 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

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

Figure 6: Type inference algorithm for ExML expressions involving existential types

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 [11] 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 never evaluates to wrong. The formal proof of semantic soundness is given below.

The semantic domains of ExML are shown in Figure 7. In the definition of V, + stands for the coalesced sum so that all types over V share the same $\perp$ and fail values. The semantic function E

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	$\cong$	$U + B + T + (V \to V) + (V \times V) + \{\mathsf{wrong}\}_{\perp, \mathsf{fail}}$

Figure 7: Semantic domains

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* [17] 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 $\Re \subseteq \Im(V)$ 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
```

would require recursion over type constructors. An adequate semantics for nonregular types can be given by extending the weak ideal model [1]; the machinery for this extension is given by Plotkin [26].

Theorem 8.1 The semantic function for types is well-defined.

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

Figure 8: Semantic function for expressions

Proof: All type expressions are formally contractive [17], hence 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 $A \vdash e : \tau$. See Appendix A for a proof sketch. □

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

9 Conclusion

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

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

Figure 9: Semantic function for types

out even changing its syntax. We have presented a type system that extends the Damas-Milner system with existentially quantified component types of data and record types, have given a type inference algorithm, and have proved that the type system is semantically sound.

The work on first-class modules by Harper and Lillibridge [9] and independently Leroy [13] 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 an easily understandable semantics directly in the functional core of a language. It might thus be desirable to include both proposals in the same language and even to allow some redundancy between the core and module languages.

Acknowledgments

We would like to express our 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 LOPLAS referees for their detailed comments.

References

1. Abadi, M. Private communication, June 1992.
2. Burstall, R., MacQueen, D., and Sannella, D. Hope: An experimental applicative language. In *Stanford LISP Conference 1980*, pages 136–143, 1980.

3. Cardelli, L., and Leroy, X. 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.
4. Cardelli, L., and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
5. Clement, D., Despeyroux, J., Despeyroux, T., and Kahn, G. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.
6. Damas, L. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.
7. Damas, L., and Milner, R. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, pages 207–212, January 1982.
8. Harper, R. Introduction to Standard ML. Technical report, School of Computer Science, Carnegie Mellon University, September 1990.
9. Harper, R., and Lillibridge, M. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21th ACM Symp. on Principles of Programming Languages*, pages 123–137, January 1994.
10. Hudak, P., Peyton-Jones, S., Wadler, P., et al. Report on the programming language Haskell A non-strict, purely functional language Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
11. Läufer, K. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, July 1992. Available as Technical Report 622, December 1992, from New York University, Department of Computer Science.
12. Läufer, K., and Odersky, M. Type classes are signatures of abstract types. In *Proc. Phoenix Seminar and Workshop on Declarative Programming*, November 1991.
13. Leroy, X. Manifest types, modules, and separate compilation. In *Proc. 21th ACM Symp. on Principles of Programming Languages*, pages 109–123, January 1994.
14. Leroy, X., and Mauny, M. Dynamics in ML. In *Proc. Functional Programming Languages and Computer Architecture*, pages 406–426. ACM, 1991.
15. Leroy, X., and Mauny, M. The Caml Light system, release 0.5, documentation and user's manual, September 1992. Distributed with the Caml Light system.
16. MacQueen, D. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, January 1986.
17. MacQueen, D., Plotkin, G., and Sethi, R. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.
18. Meyer, A., and Reinhold, M. Type is not a type. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 287–295, January 1986.
19. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, 1990.
20. Mitchell, J., and Harper, R. The essence of ML. In *Proc. Symp. on Principles of Programming*

Languages. ACM, January 1988.

21. Mitchell, J., Meldal, S., and Madhav, N. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, January 1991.
22. Mitchell, J., and Plotkin, G. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.
23. Odersky, M. Objects and subtyping in a functional perspective. Technical Report RC 16423, IBM, 1991.
24. Perry, N. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.
25. Pierce, B., and Turner, D. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, April 1993.
26. Plotkin, G. Domains. Course notes, 1983. TeX-ed edition.
27. Turner, D. Miranda: A non-strict functional language with polymorphic types. In *Proc. Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985. Springer. Lecture Notes in Computer Science, Vol. 201.
28. Wadler, P., and Blott, S. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, January 1989.
29. Winskel, G. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

A Proof of Semantic Soundness

We will use the following two lemmas in our proof:

Lemma A.1 Let ψ be a type environment such that for every $\alpha \in Dom(\psi)$, $\mathsf{wrong} \notin \psi(\alpha)$. Then for every type scheme σ, $\mathsf{wrong} \notin T[\![\ \sigma\ ]\!]\psi$.

Proof: By structural induction on σ. □

Lemma A.2 (Substitution) $T[\![\ [\sigma'/\alpha]\sigma\ ]\!]\psi = T[\![\ \sigma\ ]\!]\psi[\alpha : T[\![\ \sigma'\ ]\!]\psi]$

Proof: By structural induction on σ. □

Proof: To prove Theorem 8.3, we 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 four new cases. Lemma A.2 will be used with frequency.

$A \vdash$ **data** σ **in** $e : \tau$

The premises in the type derivation are $A[K_1 : \sigma, \ldots, K_k : \sigma] \vdash e : \tau$ and $\sigma = \forall \alpha_1 \ldots \alpha_n . \mu\beta . K_1\eta_1 + \ldots + K_k\eta_k$. By definition, $\models_{\rho,\psi} A[K_1 : \sigma, \ldots, K_k : \sigma]$, and by the inductive assumption, $E[\![\ \textbf{data}\ \sigma\ \textbf{in}\ e\ ]\!]\rho = E[\![\ e\ ]\!]\rho \in T[\![\ \tau\ ]\!]\psi$.

$A \vdash K : \tau \to \chi$

The premises are $A(K) \geq \chi$ and $[\chi/\beta]\,\eta \leq \tau$, where $\chi = \mu\beta.\ldots + K\eta + \ldots$ and $\eta = \exists\beta_1\ldots\beta_m.\tilde{\tau}$. By definition, $\tau = [\tau_i/\beta_i, \chi/\beta]\,\tilde{\tau}$ for some types $\tau_1, \ldots, \tau_m$. First, choose an arbitrary $v \in T[\![\ \tau\ ]\!]\,\psi$ and a finite $a \leq v$. Then

$$\begin{array}{rcl} a & \in & (T[\![\ [\tau_i/\beta_i, \chi/\beta]\,\tilde{\tau}\ ]\!]\,\psi)^\circ \\ & = & (T[\![\ [\chi/\beta]\,\tilde{\tau}\ ]\!]\,\psi\,[\beta_i{:}\,T[\![\ \tau_i\ ]\!]\,\psi])^\circ \\ & \subseteq & \bigcup_{J_1, \ldots, J_m \in \Re} (T[\![\ [\chi/\beta]\,\tilde{\tau}\ ]\!]\,\psi\,[\beta_i{:}\,J_i])^\circ \\ & = & (\bigsqcup_{J_1, \ldots, J_m \in \Re} T[\![\ [\chi/\beta]\,\tilde{\tau}\ ]\!]\,\psi\,[\beta_i{:}\,J_i])^\circ \\ & = & (T[\![\ [\chi/\beta]\,\eta\ ]\!]\,\psi)^\circ \end{array}$$

Hence $v = \bigsqcup \{a \mid a \text{ finite} \wedge a \leq v\} \in T[\![\ [\chi/\beta]\,\eta\ ]\!]\,\psi$ by the closure of ideals under limits. Thus

$$\begin{array}{rcl} \langle K, v\rangle & \in & \{K\} \times T[\![\ [\chi/\beta]\,\eta\ ]\!]\,\psi \\ & \subseteq & \ldots \cup \{K\} \times T[\![\ [\chi/\beta]\,\eta\ ]\!]\,\psi \cup \ldots \\ & = & T[\![\ \ldots + K\eta + \ldots\ ]\!]\,\psi\,[\beta{:}\,T[\![\ \chi\ ]\!]\,\psi] \\ & = & T[\![\ \chi\ ]\!]\,\psi \end{array}$$

and $E[\![\ K\ ]\!]\,\rho \in T[\![\ \tau \to \chi\ ]\!]\,\psi$.

$A \vdash \textbf{is}\ K : \chi \to bool$

Since $(E[\![\ \textbf{is}\ K\ ]\!]\,\rho)\,v \in B$ for any $v \in T[\![\ \chi\ ]\!]\,\psi$, we have $E[\![\ \textbf{is}\ K\ ]\!]\,\rho \in T[\![\ \chi \to bool\ ]\!]\,\psi$.

$A \vdash \textbf{let}\ K\ x = e_1\ \textbf{in}\ e_2 : \tau$

We follow the proof by MacQueen, Plotkin, and Sethi [17]. We know that $A \vdash e_1 : \chi$, where $\chi = \mu\beta.\ldots + K\eta + \ldots$ and $\eta = \exists\beta_1\ldots\beta_m.\tilde{\tau}$. Let $\{\alpha_1, \ldots, \alpha_n\} = FV([\chi/\beta]\,\eta) - FV(A)$. Then $\models_{\rho, \psi[\alpha_i : I_i]} A$ holds for every $I_1, \ldots, I_n \in \Im(V)$ since none of the α_i's is free in A.

Let $v = E[\![\ e_1\ ]\!]\,\rho$, thus by the inductive assumption, $v \in T[\![\ \chi\ ]\!]\,\psi\,[\alpha_i{:}\,I_i]$. Since $I \sqcap J = I \cap J$ for any weak ideals I and J, we have

$$\begin{array}{rcl} v & \in & \bigsqcap_{I_1, \ldots, I_n \in \Re} T[\![\ \chi\ ]\!]\,\psi\,[\alpha_i{:}\,I_i] \\ & = & \ldots \cup \{K\} \times \bigsqcap_{I_1, \ldots, I_n \in \Re} T[\![\ \eta\ ]\!]\,\psi\,[\alpha_i{:}\,I_i, \beta{:}\,T[\![\ \chi\ ]\!]\,\psi\,[\alpha_i{:}\,I_i]] \cup \ldots \end{array}$$

In the case $\mathsf{fst}(v) \neq K$, $E[\![\ \textbf{let}\ K\ x = e_1\ \textbf{in}\ e_2\ ]\!]\,\rho = \mathsf{fail} \in T[\![\ \tau\ ]\!]\,\psi$ and we are done.

In the more interesting case $v = \langle K, \tilde{v}\rangle$, where

$$\tilde{v} \in \bigsqcup_{I_1, \ldots, I_n \in \Re} \quad \bigsqcup_{J_1, \ldots, J_m \in \Re} T [\![\tilde{\tau}]\!] \psi [\alpha_i : I_i, \beta_j : J_j, \beta : T [\![\chi]\!] \psi [\alpha_i : I_i]]$$

Let $\alpha_1, \ldots, \alpha_h$ be those variables among $\alpha_1 \ldots \alpha_n$ that are free in $[\chi/\beta] \tilde{\tau}$, where $h \le n$. We choose a finite $a \le \tilde{v}$. Thus

$$a \quad \in \quad \bigcap_{I_1, \ldots, I_h \in \Re} \bigcup_{J_1, \ldots, J_m \in \Re} (T [\![[\chi/\beta] \tilde{\tau}]\!] \psi [\alpha_i : I_i, \beta_j : J_j])^{\circ}$$

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

$$\begin{aligned}
a \quad &\in \quad \bigsqcup_{I_1, \ldots, I_n \in \Re} (T [\![[\chi/\beta] \tilde{\tau}]\!] \psi [\alpha_i : I_i, \beta_j : f_j(I_1, \ldots, I_h)])^{\circ} \\
&\subseteq \quad \bigsqcup_{I_1, \ldots, I_n \in \Re} T [\![[\chi/\beta] \tilde{\tau}]\!] \psi [\alpha_i : I_i, \beta_j : f_j(I_1, \ldots, I_h)] \\
&= \quad \bigsqcup_{I_1, \ldots, I_n \in \Re} T [\![[\kappa_j(\alpha_1, \ldots, \alpha_h)/\beta_j, \chi/\beta] \tilde{\tau}]\!] \psi [\alpha_i : I_i, \kappa_j : f_j] \\
&= \quad T [\![\forall \alpha_1 \ldots \alpha_n . [\kappa_j(\alpha_1, \ldots, \alpha_h)/\beta_j, \chi/\beta] \tilde{\tau}]\!] \psi [\kappa_j : f_j] \\
&= \quad T [\![gen(A, skol(A, [\chi/\beta] \eta))]\!] \psi [\kappa_j : f_j]
\end{aligned}$$

assuming that the κ_j's are the Skolem type constructors generated by $skol(A, [\chi/\beta] \eta)$.

Since none of the κ_j's are free in A, we have $\models_{\rho, \psi[\kappa_j : f_j]} A$ and we can extend A and ρ to obtain $\models_{\rho[a/x], \psi[\kappa_j : f_j]} A [x : gen(A, skol(A, [\chi/\beta] \eta))]$. By applying the inductive assumption to the premise $A [x : gen(A, skol(A, [\chi/\beta] \eta))] \vdash e_2 : \tau$ and using $FS(\tau) \subseteq FS(A)$, we obtain $E [\![e_2]\!] \rho [x : a] \in T [\![\tau]\!] \psi [\kappa_j : f_j] = T [\![\tau]\!] \psi$. Finally, by the continuity of E,

$$\begin{aligned}
E [\![\textbf{let } K\ x = e_1 \textbf{ in } e_2]\!] \rho \quad &= \quad E [\![e_2]\!] \rho [x : \tilde{v}] \\
&= \quad \bigsqcup \{ E [\![e_2]\!] \rho [x : a] \mid a \text{ finite} \wedge a \le \tilde{v} \}
\end{aligned}$$

The latter expression is in $T [\![\tau]\!] \psi$ by the closure of ideals under limits. □

B A "Dotless" Dot Notation in ExML

We formalize the dot notation presented in Section 3 in an extension of ExML called ExML°. As records are merely named tuples and field selection is syntactic sugar for tuple component selection, abstract types are again modeled by data types with existentially quantified component types. Values of abstract type are created by applying a constructor to a value and decomposed by pattern-matching in a **let**-expression. However, we allow existentially quantified type variables to escape the scope of the identifier in whose type they appear as long as the expression decomposed is an identifier and the existentially quantified type variables do not escape the scope of that identifier. Each decomposition of an identifier, using the same constructor, produces identical Skolem type constructors. We call our notation a "dotless" dot notation since it uses decomposition by pattern-matching instead of record field selection.

The type inference rules of ExML° are given in Figure 10. The VAR and APP rules are the same as in the original system. The ABS and LET rules are modified to prevent Skolem type constructors associated with a bound identifier to escape its scope. The DATA, PACK, and TEST rules remain unchanged. Finally, the new OPEN rule prevents Skolem type constructors associated with the bound identifier y from escaping their scope, but imposes no restrictions on Skolem type constructors associated with x, the identifier being decomposed. A type inference algorithm to compute principal types can be obtained by straightforward modification of the algorithm discussed in Section 7.

$$\text{VAR}\quad \frac{A(x) \geq \tau}{A \vdash^{\circ} x : \tau} \qquad\qquad \text{APP}\quad \frac{A \vdash^{\circ} e_1 : \tau_2 \to \tau_1 \qquad A \vdash^{\circ} e_2 : \tau_2}{A \vdash^{\circ} (e_1\ e_2) : \tau_1}$$

$$\text{ABS}\quad \frac{A\,[x{:}\tau_1] \vdash^{\circ} e : \tau_2 \qquad FS_x(A) \cup FS_x(\tau_2) = \varnothing}{A \vdash^{\circ} \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{LET}\quad \frac{A \vdash^{\circ} e_1 : \tau_1 \qquad A\,[x{:}gen(A, \tau_1)] \vdash^{\circ} e_2 : \tau_2 \qquad FS_x(A) \cup FS_x(\tau_2) = \varnothing}{A \vdash^{\circ} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\text{DECL}\quad \frac{A\,[K_1{:}\sigma, \ldots, K_k{:}\sigma] \vdash^{\circ} e : \tau \qquad FV(\sigma) = \varnothing \qquad \sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_k\eta_k}{A \vdash^{\circ} \mathbf{data}\ \sigma\ \mathbf{in}\ e : \tau}$$

$$\text{PACK}\quad \frac{A(K) \geq \chi \qquad [\chi/\beta]\,\eta \leq \tau \qquad \chi = \mu\beta.\ldots + K\eta + \ldots}{A \vdash^{\circ} K : \tau \to \chi}$$

$$\text{TEST}\quad \frac{A(K) \geq \chi \qquad \chi = \mu\beta.\ldots + K\eta + \ldots}{A \vdash^{\circ} \mathbf{is}\ K : \chi \to bool}$$

$$\text{OPEN}\quad \frac{\begin{array}{c} A(x) \geq \chi \qquad \chi = \mu\beta.\ldots + K\eta + \ldots \\ A\,[y{:}gen(A, skol_{x,K}(A, [\chi/\beta]\,\eta))] \vdash^{\circ} e : \tau \qquad FS_y(A) \cup FS_y(\tau) = \varnothing \end{array}}{A \vdash^{\circ} \mathbf{let}\ K\ y = x\ \mathbf{in}\ e : \tau}$$

Figure 10: Type inference rules for the dotless dot notation

We retain our original semantic function E from Section 8. Following Cardelli and Leroy [3], we prove semantic soundness by giving a type- and semantics-preserving translation to ExML. The idea is that we can enclose any expression e with subexpressions of the form **let** $K\ y = x$ **in** e' within an surrounding expression that defines y and replace each subexpression of the form **let** $K\ y = x$ **in** e' by e'. That is, we replace e by

$$\mathbf{let}\ K\ y = x\ \mathbf{in}\ [e'/\mathbf{let}\ K\ y = x\ \mathbf{in}\ e']\,e$$

We chose the enclosing **let**-expression defining y large enough that no existentially quantified type variables arising through the inner **let**-expressions escape this surrounding definition. Since the ABS, LET, and OPEN rules guarantee that no existentially quantified variables emerging from the decomposition of x escape the scope of x, it is safe to enclose the whole body of a λ or **let**-expression. However, we must be careful, since the surrounding decomposition in the translation might fail, while the enclosed decomposition in the original expression might not necessarily have been reached; this is possible if the value held by x does not have the constructor tag K. Therefore, we need to replace e by an **if**-expression with branches for each possible constructor in the type of x. This is reflected in the definition of the auxiliary translation function *branch*.

We modify the OPEN rule of ExML to use the function $skol_{x,K}$. This facilitates the translation since the Skolem type constructors can be left unchanged in the translation of a type environment. The translation and auxiliary functions are shown in Figure 11. In the remainder of this section, we establish semantic soundness of ExML°.

Lemma B.1 If $trenv(A\,[x{:}\sigma]) \vdash e : \tau$ and $FS_x(A) \cup FS_x(\tau) = \emptyset$, then $trenv(A)\,[x{:}\sigma] \vdash branch(e, x) : \tau$.

Proof: Follows immediately using suitable typings for **if** and **fail**. □

Lemma B.2 $E\,[\![\, e \,]\!]\, trrun(\rho\,[x{:}v], e) = E\,[\![\, branch(e, x) \,]\!]\, trrun(\rho, branch(e, x))\,[x{:}v]$.

Proof: By definition of *branch* and *trrun*. □

Theorem B.3 (Preservation of types) If $A \vdash^{\circ} e : \tau$, then $trenv(A) \vdash trexp(e) : \tau$.

Proof: By induction on the structure of e. We consider the following cases:

$A \vdash^{\circ} \lambda x.e : \tau_1 \to \tau_2$

The premises in the derivation are $A\,[x{:}\tau_1] \vdash^{\circ} e : \tau_2$ and $FS_x(A) \cup FS_x(\tau_2) = \emptyset$. By the inductive assumption, $trenv(A\,[x{:}\tau_1]) \vdash trexp(e) : \tau_2$, and by Lemma B.1, $trenv(A)\,[x{:}\tau_1] \vdash branch(trexp(e), x) : \tau_2$. The claim follows by applying the original ABS rule.

$A \vdash^{\circ}$ **let** $x = e_1$ **in** $e_2 : \tau_2$

The premises in the derivation are $A \vdash^{\circ} e_1 : \tau_1$, $A\,[x{:}gen(A, \tau_1)] \vdash^{\circ} e_2 : \tau_2$, and $FS_x(A) \cup FS_x(\tau_2) = \emptyset$. By the inductive assumption, $trenv(A) \vdash trexp(e) : \tau_1$ and $trenv(A\,[x{:}gen(A, \tau_1)]) \vdash trexp(e) : \tau_2$. Thus by Lemma B.1, $trenv(A)\,[x{:}gen(A, \tau_1)] \vdash branch(trexp(e_2), x) : \tau_2$, and we can apply the original LET rule.

$A \vdash^{\circ}$ **let** $K\ y = x$ **in** $e : \tau$

The premises in the derivation are $A(x) \geq \chi$ and $A\,[y{:}gen(A, skol_{x,K}(A, [\chi/\beta]\,\eta))] \vdash^{\circ} e : \tau$, where $\chi = \mu\beta.\ldots + K\eta + \ldots$. By the inductive assumption, $trenv(A\,[y{:}gen(A, skol_{x,K}(A, [\chi/\beta]\,\eta))]) \vdash trexp(e) : \tau$. Thus by Lemma B.1,

$$
\begin{array}{lll}
trexp(e_1\ e_2) & = & trexp(e_1)\ trexp(e_2) \\
trexp(\lambda x.e) & = & \lambda x.\,branch(trexp(e), x) \\
trexp(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathbf{let}\ x = trexp(e_1)\ \mathbf{in}\ branch(trexp(e_2), x) \\
trexp(\mathbf{data}\ \sigma\ \mathbf{in}\ e) & = & \mathbf{data}\ \sigma\ \mathbf{in}\ trexp(e) \\
trexp(\mathbf{let}\ K\ y = x\ \mathbf{in}\ e) & = & \mathbf{let}\ y = x_K\ \mathbf{in}\ branch(trexp(e), y) \\
trexp(e) & = & e \\
branch(e, x) & = & \mathbf{if\ is}\ K_1 x\ \mathbf{then} \\
 & & \quad \mathbf{let}\ K_1 x_{K_1} = x\ \mathbf{in}\ \left[\mathbf{fail}/x_{K_2}, \ldots, \mathbf{fail}/x_{K_m}\right] e \\
 & & \mathbf{else\ if\ is}\ K_2 x\ \mathbf{then} \\
 & & \quad \mathbf{let}\ K_2 x_{K_2} = x\ \mathbf{in}\ \left[\mathbf{fail}/x_{K_1}, \mathbf{fail}/x_{K_3}, \ldots, \mathbf{fail}/x_{K_m}\right] e \\
 & & \ldots \\
 & & \mathbf{else\ if\ is}\ K_m x\ \mathbf{then} \\
 & & \quad \mathbf{let}\ K_m x_{K_m} = x\ \mathbf{in}\ \left[\mathbf{fail}/x_{K_1}, \ldots, \mathbf{fail}/x_{K_{m-1}}\right] e \\
 & & \mathbf{else\ fail} \\
 & & \text{where } \{K_1, \ldots, K_m\} = \{K \mid x_K \in FV(e)\} \\
branch(e, x) & = & e \\
trenv(A\,[x{:}\sigma]) & = & trenv(A)\left[x{:}\sigma, x_{K_1}{:}\sigma_1, \ldots, x_{K_k}{:}\sigma_k\right] \text{ where} \\
 & & \chi = \mu\beta.\,K_1\eta_1 + \ldots + K_k\eta_k,\ \sigma = \forall\alpha_1\ldots\alpha_n.\chi, \\
 & & \sigma_i = gen(A', skol_{x, K_i}(A', [\chi/\beta]\,\eta_i)), \text{ and } A' = A\,[x{:}\sigma] \\
trenv(A\,[x{:}\sigma]) & = & trenv(A)\,[x{:}\sigma] \\
trenv([]) & = & [] \\
trrun(\rho\,[x{:}v], e) & = & trrun(\rho, e)\left[x{:}v, x_{K_1}{:}v_1, \ldots, x_{K_k}{:}v_k\right] \text{ where } v_i = \mathsf{snd}(x) \text{ if} \\
 & & x \in \{K_i\} \times V \text{ and } v_i = \mathsf{fail} \text{ otherwise and} \\
 & & \{K_1, \ldots, K_m\} = \{K \mid x_K \in FV(e)\} \\
trrun([], e) & = & [] \\
skol_{x, K}(A, \exists\beta_1\ldots\beta_m.\tau) & = & [\kappa_{x, K, i}(\alpha_1, \ldots, \alpha_n)/\beta_i]\,\tau \text{ where} \\
 & & \{\alpha_1, \ldots, \alpha_n\} = FV(\exists\beta_1\ldots\beta_m.\tau) - FV(A)
\end{array}
$$

Figure 11: Auxiliary translation functions

$trenv(A)\,[y{:}\,gen(A, skol_{x,K}(A, [\chi/\beta]\,\eta))] \vdash branch(trexp(e), y) : \tau$. Since $x \in Dom(A)$, $x_K \in Dom(trenv(A))$ and $A(x_K) = gen(A, skol_{x,K}(A, [\chi/\beta]\,\eta))$. The claim follows from the original LET rule. □

Theorem B.4 (Preservation of semantics) $E[\![\ e\ ]\!]\rho = E[\![\ trexp(e)\ ]\!]\,trrun(\rho, trexp(e))$.

Proof: By induction on the structure of e. We abbreviate $trexp(e)$ as $\bar{e}$. The interesting cases are as follows:

$\lambda x.e$:

$$\begin{aligned}
& E[\![\ \lambda x.e\ ]\!]\rho \\
= \;& \lambda v \in V.\ (E[\![\ e\ ]\!]\rho\,[x{:}v]) \\
= \;& \lambda v \in V.\ (E[\![\ \bar{e}\ ]\!]\,trrun(\rho\,[x{:}v], \bar{e})) \\
= \;& \lambda v \in V. E[\![\ branch(\bar{e}, x)\ ]\!]\,trrun(\rho, branch(\bar{e}, x))\,[x{:}v] \\
= \;& E[\![\ \overline{\lambda x.e}\ ]\!]\,trrun(\rho, \overline{\lambda x.e})
\end{aligned}$$

let $x = e_1$ **in** e_2:

$$\begin{aligned}
& E[\![\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ ]\!]\rho \\
= \;& E[\![\ e_2\ ]\!]\rho\,[x{:}E[\![\ e_1\ ]\!]\rho] \\
= \;& E[\![\ \overline{e_2}\ ]\!]\,trrun(\rho\,[x{:}E[\![\ \overline{e_1}\ ]\!]\,trrun(\rho, \overline{e_1})], \overline{e_2}) \\
= \;& E[\![\ branch(\overline{e_2}, x)\ ]\!]\,trrun(\rho, branch(\overline{e_2}, x))\,[x{:}E[\![\ \overline{e_1}\ ]\!]\,trrun(\rho, \overline{e_1})] \\
= \;& E[\![\ \mathbf{let}\ x = \overline{e_1}\ \mathbf{in}\ branch(\overline{e_2}, x)\ ]\!]\,trrun(\rho, \mathbf{let}\ x = \overline{e_1}\ \mathbf{in}\ branch(\overline{e_2}, x))
\end{aligned}$$

let $K\ y = x$ **in** e:

Let $\tilde{e} = \mathbf{let}\ y = x_K\ \mathbf{in}\ branch(\bar{e}, y)$. First, if $\rho(x) \notin \{K\} \times V$, then $E[\![\ \mathbf{let}\ K\ y = x\ \mathbf{in}\ e\ ]\!]\rho = \mathsf{fail} = E[\![\ \tilde{e}\ ]\!]\,trrun(\rho, \tilde{e})$. Otherwise,

$$\begin{aligned}
& E[\![\ \mathbf{let}\ K\ y = x\ \mathbf{in}\ e\ ]\!]\rho \\
= \;& E[\![\ e\ ]\!]\rho\,[y{:}\mathsf{snd}(\rho(x))] \\
= \;& E[\![\ \bar{e}\ ]\!]\,trrun(\rho\,[y{:}\mathsf{snd}(\rho(x))], \bar{e}) \\
= \;& E[\![\ branch(\bar{e}, y)\ ]\!]\,trrun(\rho, \bar{e})\,[y{:}\mathsf{snd}(\rho(x))] \\
= \;& E[\![\ branch(\bar{e}, y)\ ]\!]\,trrun(\rho, \tilde{e})\,[y{:}E[\![\ x_K\ ]\!]\,trrun(\rho, \tilde{e})] \\
= \;& E[\![\ \tilde{e}\ ]\!]\,trrun(\rho, \tilde{e})
\end{aligned}$$

Proof: Since all other cases are trivial, our claim is proved. □

Corollary B.5 (Semantic soundness) If $A \vdash^{\circ} e : \tau$ and $\models_{\rho,\psi} A$, then $E[\![\ e\ ]\!]\rho \neq \mathsf{wrong}$.

Proof: Follows from the previous theorem and Theorem 8.3. □