# COMBINING TYPE CLASSES AND EXISTENTIAL TYPES

Konstantin Läufer
laufer@math.luc.edu

Department of Mathematical Sciences
Loyola University of Chicago
6525 North Sheridan Road
Chicago, IL 60660, USA

*In Proc. Latin American Informatics Conference (PANEL)*
*ITESM-CEM, Mexico, September 1994*

## Abstract

This paper demonstrates that the novel combination of type classes and existential types adds significant expressive power to a language, requiring only a minor syntactic change. We explore this combination in the context of higher-order functional languages with static typing, parametric polymorphism, algebraic data types, and Hindley-Milner type inference. Since we have examined the underlying type-theoretic issues already, this paper focuses on the practical aspects of our extension.

We first examine limitations of existing functional and object-oriented languages. We then give examples to demonstrate how our first-class abstract types with user-defined interfaces address those limitations. Finally, we give an informal description of the translation from our language to a target language without type classes.

Our extension equally applies to other languages with similar type systems and is independent of strictness considerations. It has been implemented in the Chalmers Haskell B. system, and all examples from this paper have been developed using this system.

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

General Terms: Languages, Theory

Additional Key Words and Phrases: Object-oriented programming, dynamic dispatching, poly-

morphism, type inference, existentially quantified types

## 1 Introduction

The intent of this paper is to demonstrate that the combination of two independent programming language constructs, *type classes* and *existential types*, adds significant expressive power to a language, requiring only a minor syntactic change. We explore this combination in the context of higher-order functional languages with static typing, parametric polymorphism, algebraic data types, and Hindley-Milner type inference.

While this paper focuses on the practical consequences of combining type classes with existential types, we have already treated the underlying type-theoretic issues formally [10, 11]. We have developed a type system and a type inference algorithm for the resulting language. Furthermore, we have given a formal semantics by translation to an implicitly-typed second-order λ-calculus and have shown that the type system is semantically sound.

For the sake of concreteness, our language is presented here as an extension to Haskell [6]. Other languages with similar type systems can be extended analogously. Furthermore, our extension is independent of strictness considerations. Our proposal has been implemented in the Chalmers Haskell B. system [1]. Thus all examples from this paper have been developed and tested using the `hbi` interpreter and are given in Haskell syntax.

We first examine limitations of existing functional and object-oriented languages. We then demonstrate how type classes with existential types address those limitations; in particular, we give examples illustrating how we express

- first-class abstract types with user-defined interfaces,
- heterogeneous aggregates of different implementations of the same abstract type,
- dynamic dispatching of operations with respect to the representation type, and
- separate interface and implementation hierarchies.

Finally, we give an informal description of the translation from our language to a target language without type classes.

In the remainder of this paper, Section 2 motivates our extension by discussing limitations of existing functional and object-oriented languages. Section 3 describes how algebraic data types can be extended with existential quantification over type classes. Section 4 contains a collection of examples. Section 5 illustrates the implementation of our language. Section 6 concludes with an outlook on related and future work.

## 2 Limitations of Functional and Object-Oriented Languages

This section motivates our combination of type classes and existential types by examining shortcomings of existing languages.

### Functional languages

A common approach to dealing with heterogeneous lists in statically-typed languages is to introduce an algebraic data type with a separate constructor for each type allowed in the list:

```
data KEY = IntKey Int | BoolKey Bool | ListKey [Int]
```

We can now declare values of this type by applying any of the three constructors. An example of heterogenous list is the following:

```
hetList = [IntKey 5, ListKey [1,2,3], BoolKey True, IntKey 9]
```

If we now want to convert each value of the list to an integer, we can do so by first declaring the following function, using a pattern matching notation:

```
toInt (IntKey i)  = i
toInt (BoolKey b) = if b then 1 else 0
toInt (ListKey l) = length l
```

The expression `map toInt hetList` applies `toInt` to each element in the list and results in `[5,3,1,9]`.

There are several disadvantages to this approach:

- One has to remember and use a number of different constructors.
- The component types of the algebraic data type are not abstract; consequently, any operation can be applied to the components as long as it is type-correct.
- Most importantly, algebraic types of this form do not support extension. When a new case is added to an algebraic type, for example `FloatKey Float` to the type `KEY`, every function that operates on the type `KEY` has to be changed to include the new case.

We show below how algebraic data types with existential component types simultaneously cure all three of these drawbacks.

### Object-oriented languages

Object-oriented languages have been successful at expressing heterogeneous aggregates via extensible subtype hierarchies. However, as opposed to statically-typed functional languages, they either lack type inference or are dynamically-typed. Furthermore, most statically-typed object-oriented languages suffer from the contravariance rule for record subtyping, which makes it hard for these languages to model hierarchies of abstract algebraic structures.

To compare our view with the object-oriented approach, we identify several conceptual relations between classes and their implementations. We contrast the ways these relations are manifest in C++ [21] and in Haskell [6].

- The *is-a* relation: a class can be derived from one or more superclasses. In C++, this relation corresponds to public inheritance. In Haskell, it is expressed as a hierarchy of type classes.

- The *implements* relation: a class interface can have zero or more implementations. In C++, each class has exactly one implementation. In Haskell, each type class may have arbitrarily many instance types, and each type may belong to a number of type classes, as long as it also implements all of their superclasses.
- The *reuses* relation: an new implementation can be derived from an existing one for the purpose of code reuse. In C++, this relation is expressed by private inheritance. In Haskell, conditional instance declarations can express a limited form of reuse at the implementation level.

Although C++ offers two different kinds of inheritance, public and private, both are expressed via the same mechanism. Furthermore, multiple implementations in C++ are often modeled by deriving several classes from an abstract superclass. Other object-oriented languages suffer from similar problems. A recently proposed solution comes in the form of separate *signatures* [2]. On the other hand, Haskell and our language clearly separate the three relationships.

## 3 Algebraic Data Types with Existential Quantification over Type Classes

This section describes how abstract data types with user-defined interfaces can be provided by extending the syntax of algebraic data type definitions. While our extension can be applied to any language based on a polymorphic type system with algebraic data types, explicit type variables, and type classes, it has been implemented in the Chalmers Haskell B. system [1] and is presented as an extension to Haskell.

*Type classes* [9, 22] provide a systematic approach to ad-hoc operator overloading. Each *class declaration* introduces a new class name $C$ and one or more new overloaded functions $f_1, \ldots, f_n$. Each type that supports a group of overloaded functions is declared as an *instance* of the corresponding class. Algorithmic type inference of principal types in the style of Hindley and Milner is possible for ML-like languages extended with type classes, such as Haskell [17].

*Existential types* [16, 3] are a formalization of the concept of an abstract data type, such as the package in Ada, the cluster in CLU, and the module in Modula2. 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 $\tilde{\tau}$. Without giving up type inference, abstract data types can be incorporated into statically-typed functional languages by allowing the component types of algebraic data types to be existential(ly quantified) [13]. Such abstract data types are first-class in the sense that their instances are treated like ordinary values.

We combine type classes and existential types as follows: The existentially quantified type variables in the component types of algebraic data types stand for the hidden representation types of the abstract data type. By constraining the existentially quantified type variables to belong to certain type classes, we can require that the representation types support certain operations. It is in this sense that type classes serve as *interfaces* of abstract data types [12].

Syntactically, we extend algebraic data type definitions by dropping the restriction that only

type variables that are *bound* as formal type parameters may appear in the component types of the data type. Any *free* type variable in a data type declaration is considered to be local to and existentially quantified in the component types of the value constructor in which it appears, and universally quantified in the type of the value constructor itself. Just as universally quantified type variables can be constrained by a *context c* of the form $(C_1\ u_1, \ldots, C_n\ u_n)$, stating that type variable $u_i$ belongs to type class $C_i$, existentially quantified type variables can be constrained by *local contexts* $c_1, \ldots, c_n$ for the value constructors in which they appear. Thus the general form of a data type declaration is as follows:

$$\texttt{data}\ \ [c \Rightarrow]\ \ T\ \ u_1 \ldots u_k\ \ =\ \ [c_1 \Rightarrow] K_1\ t_{11} \ldots t_{1k_1}\ \ \mid\ \ \ldots\ \ \mid\ \ [c_n \Rightarrow] K_n\ t_{n1} \ldots t_{1k_n}$$

The types of the value constructors $K_1, \ldots, K_n$ are given as

$$K_i :: \forall u_1 \ldots u_k v_1 \ldots v_l .\ \tilde{c}_i \Rightarrow t_{i1} \rightarrow \ldots \rightarrow t_{ik_i} \rightarrow (T\ u_1 \ldots u_k)$$

where $v_1, \ldots, v_l$ are all type variables free in the types $t_{i1}, \ldots, t_{ik_i}$ except $u_1, \ldots, u_{k_i}$, and $\tilde{c}_i$ is the largest subset of $c \cup c_i$ that constrains only those type variables free in the types $t_{i1}, \ldots, t_{ik_i}$. When a value constructor $K$ appears in a pattern $K\ x_1 \ldots x_m$, each existentially quantified type variable in the component types of $K$ is replaced by a fresh anonymous type in the types of the bound variables $x_1, \ldots, x_m$. These anonymous types satisfy the local context for the constructor $K$; furthermore, they must not escape the scope of the bound variables $x_1, \ldots, x_m$.

# 4 Examples

The following examples illustrate various applications of existential quantification combined with type classes. All examples were developed and tested using the Chalmers Haskell B. interpreter `hbi` [1]. We assume that all numerals have type `Int`, except in the last example, where they have type `Float`.

### Minimum over a heterogeneous list

Consider the following algebraic data type declaration:

```
data KEY a = Key a (a -> Int)
```

This declaration introduces a new type constructor `KEY` with formal type parameter `a`. The type of the value constructor `Key` is universally quantified over `a`:

```
Key :: a -> (a -> Int) -> KEY a
```

By contrast, the declaration

```
data KEY = Key a (a -> Int)
```

introduces a new (parameterless) type KEY and a value constructor Key of type

```
Key :: a -> (a -> Int) -> KEY
```

Since all applications of Key have the same result type, KEY, we can construct the following list:

```
hetList = [Key 5       id,
           Key [1,2,3] length,
           Key True    (\x -> if x then 1 else 0),
           Key 9       (+ 1)]
```

Although hetList has type [KEY], it is actually heterogeneous in the sense that its elements have different representation types.

We use pattern matching to extract the two components of a value of type KEY. The type variable a in the component type of the constructor Key is existentially quantified. Thus the argument type of f and the type of x in the following function definition are identical, giving the function the type KEY -> Int:

```
whatkey (Key x f) = f x
```

Since existentially quantified type variables must not escape the scope in which they are introduced, the following function definition is ill-typed:

```
value (Key x f) = x
```

The function hetMin finds the minimum of a list of KEYs with respect to the integer value obtained by applying the function whatkey:

```
hetMin [x]    = x
hetMin (x:xs) = let y = hetMin xs in
                    if whatkey x < whatkey y then x else y
```

and the expression whatkey (hetMin hetList) evaluates to 1.

We observe that KEY is an abstract data type whose implementations *explicitly* bundle together a value of some type and a method on that type returning an Int. Each element of hetList may be viewed as a different implementation of the same abstract type. Two implementations of KEY differ either in representation types, for example the second and third elements, or in methods, such as the first and last elements of hetList. Given a value of type KEY, we do not know its representation type, but it is guaranteed that we can safely apply the second component (the method) to the first component (the value).

### Minimum over a heterogeneous list using type classes

Type classes provide a way of associating methods with a type in such a way that the methods are *implicitly* available for any value of this type. For example, the following type class Key specifies that its instances must implement a method key returning an integer:

```
class Key a where
  key :: a -> Int
```

Each instance type of `Key` declares how it implements the method `key`, for example:

```
instance Key Int  where key = id
instance Key Bool where key = \x -> if x then 1 else 0
instance Key [a]  where key = length
```

We can use the type class `Key` to define the *interface* of the abstract data type `KEY` by constraining the existentially quantified type variable `a` to be an instance of the type class `Key`. This interface is expressed by the constructor context `Key a` in the following type declaration:

```
data KEY = (Key a) => Key a
```

We can still define heterogeneous lists of `KEY`s:

```
hetList = [Key 5, Key [1,2,3], Key True, Key 9]
```

However, any two implementations of `KEY` with the same representation type now share the method implemented in the instance declaration corresponding to this type. Unlike in the explicit case, the method dictionaries are packed implicitly with the component values. This is reflected in the translation scheme presented in Section 5 and corresponds to *dynamic dispatching* in object-oriented programming language, where a method associated with an object is selected and applied to the object at run time. Unlike in the example in Section 2, additional implementations of the type `KEY` may be added simply by declaring additional instances of the class `Key`.

A straightforward way to compare different values of type `KEY` is by mapping them to integer values using the following function `whatkey`:

```
whatkey (Key x) = key x
```

Instead, we choose to take a much more general approach. We simply declare `KEY` as an instance of the equality and ordered classes, thus making a whole collection of predefined functions available. Only the methods `(==)` and `(<=)` need to be implemented here since the class declarations for `Eq` and `Ord` contain default implementations for all their other methods:

```
instance Eq KEY where
  (Key x) == (Key y) = key x == key y

instance Ord KEY where
  (Key x) <= (Key y) = key x <= key y
```

Using the predefined polymorphic `minimum` function on lists of ordered values, the expression `minimum hetList` evaluates to `Key True`.

## Composition of a list of functions

The algebraic data types in the preceding examples 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. For notational convenience, we declare the first constructor as right-associative:

```
infixr ‘FunCons‘

data FunList a b = (a -> c) ‘FunCons‘ (FunList c b) | FunOne(a -> b)
```

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:

```
funOne  = FunOne id

funList = (\x -> x * x)  ‘FunCons‘
          ((==) 9)       ‘FunCons‘
          (\x -> (x, x)) ‘FunCons‘
          funOne
```

We use a recursive function to apply the function resulting from the successive composition to an argument. Since the recursive call to `apply` has a different type from the one defined, we enable polymorphic recursion by giving an explicit type signature [7].

```
apply :: FunList a b -> a -> b

apply (FunOne f)       x = f x
apply (f ‘FunCons‘ fl) x = apply fl (f x)
```

Evaluation of the expression `apply funList 3` then results in `(True, True)`. We can even count the number of functions in the composition:

```
numberComposites :: Num c => (FunList a b) -> c

numberComposites (FunOne f) = 1
numberComposites (f ‘FunCons‘ fl) = 1 + numberComposites fl
```

The expression `numberComposites funList` evaluates to `4`.

## Points and color points

The following example demonstrates how object-oriented concepts can be modeled using type classes and existential quantification. We start out with a two-level class hierarchy of points and colored points. In object-oriented terminology, this is inheritance at the *interface level*, used to establish relationships between the abstract properties of classes.

```
data Color = Red | Green | Blue deriving Eq

type Pair = (Float, Float)

class Point p where
```

```
  move :: p -> Pair -> p
  pos  :: p -> Pair

class (Point p) => ColorPoint p where
  color  :: p -> Color
  paint  :: p -> Color -> p
```

Next, we define two implementations of class `Point`, one based on cartesian coordinates:

```
data CartPoint = C Pair

instance Point CartPoint where
  move (C(x,y)) (dx,dy) = C(x+dx,y+dy)
  pos (C p) = p
```

and one on polar coordinates:

```
data PolarPoint = P Pair

instance Point PolarPoint where
  move p (dx,dy) = moveTo p (x + dx, y + dy)
       where (x,y) = pos p
             moveTo p (x,y) = P(sqrt(x*x + y*y), atan2 y x)
  pos (P(r,a)) = (r * cos a, r * sin a)
```

As in the previous examples, we define an abstract data type with type class `Point` as its interface:

```
data POINT = (Point p) => Point p
```

By making type `POINT` an instance of class `Point`, we provide dynamic dispatching of the methods in the interface classes:

```
instance Point POINT where
  move (Point p) q = Point (move p q)
  pos  (Point p) = pos p
```

The following instance declaration states that any instance type of type class `Point` may be extended by additional fields. Without this general declaration, we would need a specific `Point` instance declaration for each `ColorPoint` instance since Haskell requires that an instance of a type class must also be an instance of all superclasses of the type class.

```
instance (Point p) => Point (p,t) where
  move (p,z) d = (move p d, z)
  pos  (p,z) = pos p
```

The next instance declaration states that any instance of `Point` extended with a field of type `Color` results in an instance of `ColorPoint`. In object-oriented terminology, this and the previous instance declaration provide inheritance at the *implementation level*, for code reuse. Furthermore, we automatically get a colored version of both implementations of `Point`, `ColorCartPoint` and `ColorPolarPoint`:

```
instance (Point p) => ColorPoint (p, Color) where
  color (p,c) = c
  paint (p,c) d = (p,d)
```

The following list of points contains various combinations of uncolored, colored, cartesian, and polar points:

```
pointList = [Point(P(5, 0)), Point(C(3, 4), Blue),
             Point(C(2, 7)), Point(P(5, pi / 4), Red)]
```

We can now evaluate expressions such as

```
map (\p -> moveBy p (1, 2)) pointList
```

resulting in

```
[Point(P(6.32, 0.32)), Point(C(4.00, 6.00), Blue),
 Point(C(3.00, 9.00)), Point(P(7.16, 0.89), Red)]
```

## 5 Implementation

In this section, we describe how Exell programs are translated to a suitable target language. Our approach is based on the original compile-time translation scheme by Wadler and Blott [22], which was further developed by Nipkow and Snelting [18]. The basic idea is to eliminate classes in favor of run-time *method dictionaries* that contain instances for particular types of the overloaded functions associated with a class. An identifier given a polymorphic type scheme in the original environment is typed as a function in the translated environment; the translated type has dictionary arguments for each class that constrains a type variable in the original type, and the result type of the function is the same as the original type. The translation is type-preserving in the sense that a well-typed Exell program translates to a well-typed target program. For type-theoretic reasons [11], the target language must be an explicitly- or implicitly-typed second-order λ-calculus. For illustrative purposes, we translate into an implicitly-typed calculus with a Haskell-like syntax.

We explain the translation by stepping through the second example in Section 4. We declare a new type for each class declaration to represent the corresponding method dictionaries. In this case, we introduce the type constructor `KeyD` corresponding to the class `Key`. All dictionaries for this class are created using the value constructor `KeyDict`.

```
data KeyD a = KeyDict (a -> Int)
```

The function `ky` selects from a method dictionary of type `KeyD` the (only) method it contains:

```
ky (KeyDict k) = k
```

Each instance declaration of the class `Key` translates to the declaration of a method dictionary of type `KeyD`. Corresponding to the instance `Key Int`, we declare a dictionary of type `KeyD Int`, and so forth:

```
keyDInt     = KeyDict id
keyDBool    = KeyDict (\x -> if x then 1 else 0)
keyDIntList = KeyDict length
```

An application of the function `key` to a value translates to an expression selecting the only method from a dictionary of type `KeyD` and applying that method to the value. For example, the expression `key [1,2,3]` translates to `(ky KeyDIntList) [1,2,3]` and evaluates to 3.

Existentially quantified type variables can occur in the component types of algebraic data types. Furthermore, each existentially quantified type variable may be constrained by one or more type classes. The functions required by these type classes are implicitly bundled with the component values and are available when the component values are accessed. This bundling is made explicit in the translation: each type variable that is constrained by one or more type classes requires including one or more dictionaries in the component type of the algebraic data type. In our example, the type variable `a` is an instance of the class `Key`; hence the translated data type `KEY'` contains a dictionary of type `KeyD a` in addition to the value of type `a`:

```
data KEY' = Key' (KeyD a) a
```

Each application of the original value constructor `Key` is translated to an application of the new constructor `Key'`, where an appropriate method dictionary is supplied as an argument:

```
hetList' = [Key' keyDInt 5, Key' keyDIntList [1,2,3],
            Key' keyDBool True, Key' keyDInt 9]
```

When a value of type `KEY` is decomposed, two components are available: a value of some type `t` and a suitable dictionary of type `KeyD t` containing a function that can be selected and applied to the value. This can be seen in the translated `whatkey` function:

```
whatkey' (Key' keyDa x) = (ky keyDa) x
```

It is not surprising that our translation of the second example in Section 4 almost exactly results in the first example, where a method was provided explicitly as a component of the data type.

## 6 Conclusion and Related Work

We have demonstrated how type classes and existential types can be combined in any functional language with a static, polymorphic type system, explicit type variables, and algebraic data type declarations, regardless of strictness considerations. We have illustrated how first-class abstract data types with user-defined interfaces overcome various drawbacks of existing functional and object-oriented languages. Finally, we have shown how to translate our extended language into a suitable target language.

Perry [19] first addresses Hindley-Milner type inference for existential types in the Hope+C system. 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 Haskell syntax, is well-typed according to the typing

rules, but rejected by the compiler. (The type variable a is existentially quantified.)

```
data T = K a
f x = case x of K z -> z

f(K 1) == f(K true)
```

Läufer and Odersky [13] present an extension of ML with existentially quantified types; in their language, operations on abstract types must be included explicitly in the existentially quantified component types of recursive data types. This is avoided by using Haskell type classes as interfaces of abstract types [12]; however, no formal semantics is given there.

Mitchell, Meldal, and Madhav [15] 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 [5] and independently Leroy [14] 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. However, the module-based approaches are semantically complex and lack support for type inference.

Pierce and Turner [20] describe an object-oriented language based on existential quantification instead of recursive record types. Their language is based on an extension of $F_\omega$ to include subtyping and seems sufficiently powerful to model most features found in typical object-oriented languages, including reference to the methods of the superclass and private instance variables, which are not supported in Exell. However, their language is explicitly typed, and type inference is not considered.

Chen, Hudak, and Odersky [4] present an extension of Haskell with parameterized type classes. Jones [8] describes a more general extension of Haskell with type constructor classes. We plan to extend Exell analogously to obtain container classes with abstract element types.

**Acknowledgments**

I would like to thank Martin Odersky for sharing his insights with me through numerous discussions. This work has greatly benefited from conversations with Stefan Kaes, Tobias Nipkow, and Phil Wadler. Lennart Augustsson's extension of Haskell B. with existential quantification made it possible to develop and test the examples contained in this paper. I would like to thank the anonymous referees of FPCA '93 and JFP for their valuable feedback on an earlier version of this work and for suggesting the interesting example involving the composition of a list of functions.

## References

[1] L. Augustsson. Haskell B. user manual, May 1993. Distributed with the HBC compiler.

[2] G. Baumgartner and V. Russo. Signatures: A C++ extension for type abstraction and

subtype polymorphism. *Software: Practice & Experience*, 1994. To appear.

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

[4] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992.

[5] R. Harper and M. Lillibridge. 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.

[6] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language Haskell: A non-strict, purely functional language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[7] M. Jones. Polymorphic recursion in Haskell, July 1993. Posted to the Haskell mailing list.

[8] M. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. Functional Programming Languages and Computer Architecture*. ACM, June 1993.

[9] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symposium on Programming, Lecture Notes in Computer Science, Vol. 300*, pages 131–144, Nancy, France, March 1988. Springer-Verlag.

[10] K. Läufer. *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.

[11] K. Läufer. Type classes with existential types. Preliminary Draft, June 1994.

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

[13] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1994. To appear.

[14] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21th ACM Symp. on Principles of Programming Languages*, pages 109–123, January 1994.

[15] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, January 1991.

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

[17] T. Nipkow and C. Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, 1993.

[18] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. Functional Programming Languages and Computer Architecture*, pages 1–14. ACM, 1991.

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

[20] B. Pierce and D. Turner. Simple type-theoretic foundations for object-oriented

programming. *Journal of Functional Programming*, April 1993.

[21] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

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