# Exploring the Design Space for Type-based Implicit Parameterization

Technical Report

Mark P. Jones
Oregon Graduate Institute of Science and Technology
20000 N.W. Walker Road, Beaverton, OR 97006, USA

July 1999

**Abstract**

A common task in programming is to arrange for data to be passed from the point where it first becomes available to the places where it is needed, which may be much further down the call hierarchy. One approach is to store the data in a global variable. Another alternative is to add extra parameters at each level to thread it from the point of supply to the point of use. Neither of these is particularly attractive: The former is inflexible, while the latter requires modification of the program text, adding auxiliary parameters to each function definition and call.

This paper explores the design space for a third alternative, using types to direct the introduction and use of *implicit parameters* and so carry data from supply to use. This gives the flexibility of parameterization without the burden of adding parameters by hand. The most general point in the design space uses relations on types to support sophisticated forms of overloading, and includes several known systems as special cases, including Haskell type classes and a recent proposal for dynamic scoping in statically-typed languages. Our work provides new insights about the relationships between these systems, as well as some new ideas that are useful in their own right. A particular novelty of this paper is in the application of ideas from the theory of relational databases to the design of type systems.

## 1 Introduction

A common task in everyday programming projects is to arrange for data to be passed from the point where it first becomes available to the places where it is needed. For example, any command line settings that are passed in at the top level of a program must be available much further down the call hierarchy in the parts where they are actually used. One way to deal with this is to store the data in a global variable, or at least in a variable whose scope extends over some relevant portion of the call graph. Another alternative is to add auxiliary parameters at each intermediate level in the call graph, whose sole purpose is to thread data from the point of supply to the point of use. These two approaches are illustrated side by side in the following example:

```
main x  =  ... a ...                main x  =  ... a x ...
  where                             a x     =  ... b x ...
    a  =  ... b ...                 b x     =  ... c x ...
    b  =  ... c ...                 c x     =  ... x ...
    c  =  ... x ...
```

In both cases, the goal is to pass the value $x$ that is supplied to *main* through the functions $a$ and $b$ to the point where it is used in the computation of $c$. The code on the left achieves this by placing the definition of $c$ within the scope of $x$. Unfortunately, this also restricts the scope of $c$ and forces us to do the same with the definitions of $a$ and $b$. Moreover, because the variable $x$ is hard-wired into the definition of $c$, it is difficult for the *main* program to invoke $a$ more than once with a different value for $x$ each time. In practice, these problems are often avoided by storing the value for $x$ in a global variable, and using side-effects to change it if different values are required at different times during execution. While this is often an adequate solution, it can also be limiting, and a source of subtle bugs.

The code on the right illustrates an alternative approach, avoiding the problems of the version on the left by adding $x$ as a parameter to each of $a$, $b$, and $c$. Because the dependency on $x$ has been made explicit, this program would still work perfectly well if these definitions were split across multiple compilation units. However, this flexibility comes at a price, requiring modifications throughout the program text, which makes it harder to maintain, and harder to read, as well as opening up another potential source of bugs.

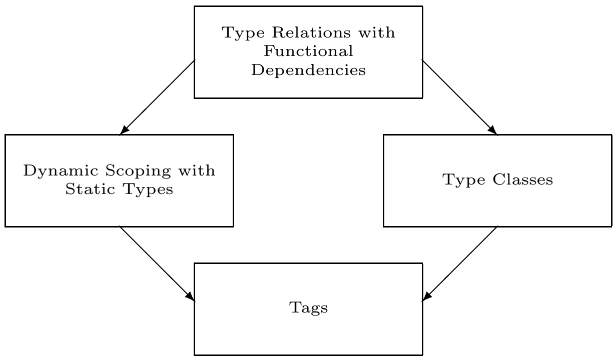
This paper explores the design space for a third alternative, using *implicit parameters* to carry data from supply to use, as in the following example:

$$\begin{array}{rcl}
\mathit{main}\ x & = & \ldots\ a\ \textbf{with}\ ?x = x\ \ldots \\
\textbf{where}\ a & = & \ldots\ b\ \ldots \\
b & = & \ldots\ c\ \ldots \\
c & = & \ldots\ ?x\ \ldots
\end{array}$$

In this example, which uses language features first proposed by Lewis *et al.* [12], the **with** construct in the definition of *main* binds $x$ to an implicit parameter $?x$, which is subsequently accessed at the point where it appears in the definition of $c$. Any such use of implicit parameters can be implemented by translating the source code into an equivalent program where the parameters are explicit: this translation is performed automatically by the compiler. Only the points at which data is supplied or used must be flagged in the source text using one of the constructs mentioned above. Most importantly, the added parameters do not clutter the definitions of values like $a$ and $b$ at intermediate levels in the call graph. Thus we obtain the flexibility of parameterization, without the burden of adding parameters by hand.

## 1.1 Contributions of This Paper

In this paper, we study a range of different systems that support implicit parameterization. In each case, types play a central role, both as the mechanism for guiding the translation from implicit to explicit parameterization, and as a way to ensure type safety. In particular, we will show how the mechanisms used in the above example, as well as other proposals, including Haskell overloading [19], and parametric type classes [3], can be described and compared as distinct points in the design space. In addition, we identify two new designs at opposite ends of the range, and show that each is useful in its own right. The most general of these systems is described in terms of programmer-defined relations on types together with sets of functional dependencies that these relations are required to satisfy. This can be seen as a form of multiple-parameter type classes [18], but avoiding the expressivity problems and ambiguities that occur with previous proposals. A particular novelty here is in the application of functional dependencies—from the theory of relational databases [13, 1]—to the design of programming language type systems. At the opposite end of the spectrum, the simplest system we consider allows types to be annotated with simple tags. Typechecking ensures that these tags are automatically propagated through the call graph, where they can be used for simple forms of program analysis. The following diagram summarizes the relationships between some of the systems mentioned here, and gives a simplified view of the design space.


Type Relations with Functional Dependencies
Dynamic Scoping with Static Types
Type Classes
Tags

For concreteness, the ideas in this paper are presented as extensions of the functional language Haskell [17], and have been prototyped as extensions to the Hugs interpreter [11]. In principle, however, they could be applied to a much wider range of typed languages.

## 1.2 Structure of This Paper

The remainder of this paper is as follows. In Section 2, we describe the key elements of type-based implicit parameterization. Two features distinguish the approach used here from more conventional type systems. First, we combine typing with translation by including both source terms and their translations in each typing judgment. Second, we work with assumption sets that associate each implicit parameter with the variable that will be used as its translation, and with its expected type.

In Section 3, we use our framework to complete the presentation of Lewis *et al.*'s proposal for dynamic scoping with static types. We add rules for the constructs that bind and access implicit parameters, and place a restriction on the form of assumption sets.

In Section 4, we show that overloading can be supported by allowing assumption sets to contain multiple translations for each implicit parameter. The key requirement is that each translation has a different type, which means that we can use typing information to choose between them in any given context.

In Section 5, we discuss *coherence*. This refers to the problems that occur when the semantics of a language is defined in terms of a (potentially non-deterministic) translation: If different translations of a source term have different meanings, then a semantic ambiguity results. We explain how such cases can be detected using a syntactic notion of ambiguity on inferred types.

In Section 6, we show that there is a range of language designs for overloading, varying in the level of detail that must be provided to declare and use an overloaded symbol. Haskell type classes and the system that we describe in Section 4 are at opposite ends of this spectrum, but use the same underlying machinery. Moreover, for each different approach to overloading, there is a corresponding version of the dynamic scoping proposal from Section 3.

In Section 7, we shift to a more general and less operational view of the systems in previous sections by thinking in terms of tuples from a programmer-defined relation on types. In the terminology of Haskell, these relations correspond to multiple parameter type classes [18], except that, critically, we also allow the programmer to specify functional dependencies between the parameters. For example, a dependency of the form $X \leadsto Y$ indicates that the $Y$ components of each tuple are uniquely determined by the $X$ components. This enables us to give a more accurate specification for each type relation, and to use more powerful rules for resolving overloading at the same time as a less restrictive characterization of ambiguity. In addition, we show that the treatment of dynamic scoping and of overloading in Sections 3 and 4 are just the special cases that occur when the only dependencies are of the form $\emptyset \leadsto Y$ and $X \leadsto \emptyset$, respectively. Firm theoretical foundations for these ideas are provided by earlier work on improvement for qualified types [9].

In Section 8, we highlight the further special case that occurs with nullary relations, corresponding to a form of implicit parameter whose type is fixed, once and for all, at the point where it is defined. The overall effect is to allow the types of certain operations to be tagged, and to propagate these tags throughout the call hierarchy, where they can be used for program analysis.

In Section 9, we draw our final conclusions.

## 2 Type-based Implicit Parameterization

The example in the introduction shows how implicit parameters might be used in a practical setting, but leaves several important questions unanswered. For instance, how can we determine where implicit parameters are required, and which values to pass or abstract over in each case? And how can we ensure that the type of any value bound to an implicit parameter is consistent with the type that is expected at the point of use?

An obvious answer to these questions is to reflect uses of implicit parameters in the types assigned to each expression or variable. More precisely, we will give each function a *qualified type* [7] of the form $P \Rightarrow \tau$. Here, $\tau$ gives the explicit portion the function's type, which includes the return type and the type of any explicit parameters. The *predicate* part, $P$, contains zero or more *predicates*, $\pi$, each of which acts as the specification for an implicit parameter. For example, a predicate of the form $(?x\ \tau)$ indicates that an implicit parameter $?x$ is required with a value of type $\tau$. In our earlier example, the use of $?x$ in the definition of $c$ would result in a predicate of the form $(?x\ \tau)$ in the type of $c$, and hence in the types of both $b$ and $a$. This will eventually be discharged by the **with** construct in the definition of *main*, at which point we can also check that $x$ has the type $\tau$ that it needs to be bound to $?x$. As a notational convenience, we will allow multiple predicates to be written together in a tuple. For example, $(\pi_1, \pi_2) \Rightarrow \tau$ and $\pi_1 \Rightarrow \pi_2 \Rightarrow \tau$ represent exactly the same qualified type, as do $() \Rightarrow \tau$ and $\tau$.

To make this work, typechecking and translation must be combined into a single process. This can be captured using judgments of the form: $I|A \vdash E \rightsquigarrow E' : \sigma$. Here, $E$ is a source term of type $\sigma$, and $E'$ is its translation. On the left of the $\vdash$ symbol, $A$ holds pairs of the form $(x\!:\!\sigma)$, each of which records an assumption about the type $\sigma$ of an explicit parameter or named constant $x$. The set $I$ plays a similar role for implicit parameters, using assumptions of the form $(i\!:\!\pi)$. In this case, $i$ is the variable that represents the implicit parameter specified by $\pi$ in the translation $E'$. We will assume that no $x$ (resp. $i$) appears more than once in any $A$ (resp. $I$). If a term has an inferred type of the form $\pi \Rightarrow \rho$, then its translation will include an extra parameter, as shown in the following rule:

$$\frac{I \cup \{i : \pi\} | A \vdash E \leadsto E' : \rho}{I | A \vdash E \leadsto (\lambda i.E') : \pi \Rightarrow \rho} \, (\Rightarrow I)$$

In a similar way, to use a value of type $\pi \Rightarrow \rho$, we must first supply an implicit parameter meeting the specification $\pi$ to obtain a result of type $\rho$:

$$\frac{I | A \vdash E \leadsto F : \pi \Rightarrow \rho \quad I \Vdash e : \pi}{I | A \vdash E \leadsto Fe : \rho} \, (\Rightarrow E)$$

For the time being, the *entailment* $I \Vdash e : \pi$ used here means simply that $(e : \pi)$ is one of the assumptions in $I$.

The $(\Rightarrow I)$ and $(\Rightarrow E)$ rules that we have just described are a fundamental, appearing in one form or another in all of the systems that we describe in this paper. A more complete list of such typing rules appears in Figure 1, most of which are fairly standard [4, 14, 6, 7]. Note the use of different symbols $\tau$ (monotypes), $\rho$ (qualified types), and $\sigma$ (polymorphic type schemes) to restrict the application of certain rules to specific sets of type expressions. In passing, we note that the $(\forall I)$ and $(\forall E)$ rules for dealing with polymorphism can be thought of as a form of implicit parameterization over types [8]. The judgment $I \Vdash S : I'$ in rule (*weak*) is a shorthand for a collection of entailments $I \Vdash e : \pi$, with one for each $(i : \pi) \in I'$. The results are collected in $S$, a substitution on terms, mapping each $i$ to the corresponding $e$. In the case described above, where entailment just means membership, this is equivalent to writing $I \supseteq I'$, and using the null substitution for $S$. Note also that we write $TV(A)$ in the statement of $(\forall I)$, and elsewhere in this paper, to denote the set of type variables that appear free in $A$.

## 3 Dynamic Scoping with Static Types

In this section, we return to our description of Lewis *et al.*'s [12] proposal for dynamic scoping with static types, hereafter referred to simply as DS. We

**Variables:** $(var)$

$$\frac{(x : \sigma) \in A}{I|A \vdash x \leadsto x : \sigma}$$

**Weakening:** $(weak)$

$$\frac{I'|A \vdash E \leadsto E' : \sigma \quad I \Vdash S : I'}{I|A \vdash E \leadsto SE' : \sigma}$$

**Explicit Parameters:** $(\to I)$

$$\frac{I|A_x, x : \tau' \vdash E \leadsto E' : \tau}{I|A \vdash (\lambda x.E) \leadsto (\lambda x.E') : \tau' \to \tau}$$

$(\to E)$

$$\frac{I|A \vdash E \leadsto E' : \tau' \to \tau \quad I|A \vdash F \leadsto F' : \tau'}{I|A \vdash EF \leadsto E'F' : \tau}$$

**Implicit Parameters:** $(\Rightarrow I)$

$$\frac{I \cup \{i : \pi\}|A \vdash E \leadsto E' : \rho}{I|A \vdash E \leadsto (\lambda i.E') : \pi \Rightarrow \rho}$$

$(\Rightarrow E)$

$$\frac{I|A \vdash E \leadsto E' : \pi \Rightarrow \rho \quad I \Vdash e : \pi}{I|A \vdash E \leadsto E'e : \rho}$$

**Polymorphism:** $(\forall I)$

$$\frac{I|A \vdash E \leadsto E' : \sigma \quad t \notin TV(A) \quad t \notin TV(I)}{I|A \vdash E \leadsto E' : \forall t.\sigma}$$

$(\forall E)$

$$\frac{I|A \vdash E \leadsto E' : \forall t.\sigma}{I|A \vdash E \leadsto E' : [\tau/t]\sigma}$$

Figure 1: Core type system for Implicit Parameterization

start by giving rules for the constructs that are used to bind and access implicit parameters. The first shows how application is used in the translation of **with**:

$$\frac{I|A \vdash E \leadsto E' : (?x\ \tau) \Rightarrow \rho \quad I|A \vdash F \leadsto F' : \tau}{I|A \vdash (E\ \textbf{with}\ ?x = F) \leadsto E'F' : \rho}$$

In practice, uses of this rule would normally be preceded by ($\Rightarrow I$) to obtain an appropriate type for $E$. This suggests a more direct rule for **with**:

$$\frac{I \cup \{i\!:\!(?x\ \tau)\}|A \vdash E \leadsto E' : \rho \quad I|A \vdash F \leadsto F' : \tau}{I|A \vdash (E\ \textbf{with}\ ?x = F) \leadsto (\textbf{let}\ i = F'\ \textbf{in}\ E') : \rho}$$

The remaining rule is an axiom, which shows that an implicit parameter $?x$ is treated as an (implicitly parameterized) identity function:

$$I|A \vdash ?x \leadsto (\lambda i.i) : \forall a.(?x\ a) \Rightarrow a$$

A more direct rule can be obtained by combining this with ($\Rightarrow E$)—and with a single $\beta$-reduction—to obtain:

$$\frac{I \Vdash e\!:\!(?x\ \tau)}{I|A \vdash ?x \leadsto e : \tau}$$

These two derived rules show very clearly how accesses and bindings of implicit parameters translate into the standard constructs for accessing and binding variables.

To complete our definition of DS, we need to add the restriction that no $?x$ can appear more than once in any given $I$. For example, this prevents us from using an $I$ that contains two distinct assumptions $i : (?x\ \tau)$ and $j\!:\!(?x\ \tau)$, with two different interpretations for $?x$ in the same context. The very fact that such parameters are implicit means that there would be no way for a programmer to specify which alternative should be used. As a result, the expression $(?x, ?x)$—a pair that uses the same implicit parameter in each component—will have a most general type of $\forall a.(?x\ a) \Rightarrow (a, a)$, with the same type for each component and with translation $(\lambda i.(i, i))$. On the other hand, the expression $(1+?x,\ \textit{not}\ ?x)$, which uses $?x$ as a number and as a boolean in the same scope, is not well-typed.

The typing rules in Figure 1 do not include a rule for local definitions. A suitable definition for DS is as follows:

$$\frac{\emptyset|A \vdash E \rightsquigarrow E' : \sigma \quad I|A_x, x\!:\!\sigma \vdash F \rightsquigarrow F' : \tau}{I|A \vdash (\textbf{let}\ x = E\ \textbf{in}\ F) \rightsquigarrow (\textbf{let}\ x = E'\ \textbf{in}\ F') : \tau}\ (\textit{let})$$

Requiring an empty set, $\emptyset$, of implicit parameter assumptions here in the typing of $E$ here is critical, both to ensure a coherent semantics (see Section 5) and to obtain the desired form of dynamic, rather than static scoping. Consider, for example, the following term:

$$(\textbf{let}\ y = ?x\ \textbf{in}\ (y\ \textbf{with}\ ?x = 0))\ \textbf{with}\ ?x = 1.$$

With static scoping, we would evaluate $y$ using the definition $?x = 1$ that is in scope at the point where $y$ is defined, and so the whole expression would evaluate to 1:

$$\begin{array}{l}\textbf{let}\ i = 1\ \textbf{in}\ (\textbf{let}\ y = i\ \textbf{in}\ (\textbf{let}\ i = 0\ \textbf{in}\ y)) \\ \quad \Longrightarrow 1.\end{array}$$

However, our (*let*) rule can only be applied if we have first moved any predicates that are required by $E$ into the inferred type $\sigma$, typically using $(\Rightarrow I)$. The overall effect is to defer the binding of implicit parameters to the point of use. Returning to the current example, this leads to a translation that simulates dynamic scoping, using the definition $?x = 0$ that is in scope at the point where $y$ is used, so that the whole expression evaluates to 0:

$$\begin{array}{l}\textbf{let}\ i = 1\ \textbf{in}\ (\textbf{let}\ y = (\lambda i.i)\ \textbf{in}\ (\textbf{let}\ i = 0\ \textbf{in}\ y\ i)) \\ \quad \Longrightarrow 0.\end{array}$$

Notice that we do not allow dynamic scoping of $\lambda$-bound variables: The $(\rightarrow I)$ and $(\rightarrow E)$ rules in Figure 1 allow only simple monotypes $\tau$ as function arguments, and not qualified types. This can be viewed as a distinct advantage of the approach described here because it helps to avoid some of the *context bugs* that occur with more traditional forms of dynamic scoping [5].

## 4 Overloading

In this section, we will show very briefly that a simple form of *overloading*—in which a single symbol can have multiple interpretations within the same scope—can be understood as a form of implicit parameterization. Several aspects of our presentation are new, but the observations on which it is based can be traced back at least as far as Wadler and Blott's original proposal for *type classes* [19], which was subsequently refined and adopted as part of the standard for Haskell [17].

Consider again the restriction that we made to complete our definition of DS in the previous section. By insisting that no $?x$ appear more than once in any given $I$, we rule out examples like $I_1 = \{i : (?x\ \tau), j : (?x\ \tau)\}$—which is entirely reasonable because there is no way to choose between the two translations. However, we also prohibit examples like $I_2 = \{i : (?x\ \tau), j : (?x\ \tau')\}$, where different types $\tau \neq \tau'$ could be used to guide such a choice.

To pursue this idea in a little more depth, we introduce a second class of implicit parameters, with names of the form $!x$, and with exactly the same typing rules as the $?x$ form that we saw in the previous section. The only difference will be to require that no $!x$ appears more than once *with the same type*[1] in any given $I$. With this more relaxed condition, the most general type of the expression $(!x, !x)$ will be: $\forall a, b.(!x\ \ a,\ \ !x\ \ b) \Rightarrow (a, b)$, with a translation $(\lambda i.\lambda j.(i, j))$, allowing each component of the resulting pair to have a different type. The following example illustrates more clearly how these constructs can be used to support a simple form of overloading, leaving the type system to match up each use of an implicit parameter with an appropriate binding:

$$\begin{array}{l} ((1+!x,\ \mathit{not}\ !x)\ \textbf{with}\ !x = 41)\ \textbf{with}\ !x = \mathit{True} \\ \quad \Longrightarrow (42,\ \mathit{False}). \end{array}$$

[1]A stronger version of this restriction is required for languages, where the type of an expression may not be uniquely determined. In a language with a Hindley/Milner type system, for example, we would need to prohibit any $I$ that associates two or more *unifiable* types with the same implicit parameter: if two types are unifiable, then there are substitution instances at which they would be equal and an ambiguity would occur.

## 5 Coherence and Ambiguity

Unfortunately, in some cases, it is not possible for the type system to determine which binding of an implicit parameter should be used. The following example uses a polymorphic function, *length*, to calculate the length of a list $!xs$, in the presence of two bindings for $!xs$, one of which is a list of integers of length 3, the other a list of booleans of length 2:

$$\begin{array}{ll} \mathit{length}\ !xs & \textbf{with}\ !xs = [1,\, 2,\, 3] \\ & \textbf{with}\ !xs = [\mathit{True},\, \mathit{False}] \end{array}$$

There is nothing in the way that $!xs$ is used here to indicate which of these two possibilities is intended. We can even see this in the principal type of $\mathit{length}\ !xs$, which is $\forall\, a\,.\,(!xs\ [a]) \Rightarrow \mathit{Int}$; notice that there is nothing in the portion of the type to the right of the $\Rightarrow$ symbol, reflecting the explicit context in which the term might be used, that would help to determine an appropriate type for $a$. Indeed, we can construct two translations for this term, one of which uses the first binding and evaluates to 3, while the other uses the second, and evaluates to 2. In other words, this is a semantically ambiguous expression because it does not have a well-defined meaning.

Practical experience suggests that semantic ambiguities do not occur too frequently in more realistic (and hence larger) applications; the context in which an overloaded symbol appears will usually give enough information to determine which interpretation is required. But we cannot expect to avoid ambiguity problems altogether. The best we can hope for is some simple test that a typechecker can use to detect such problems, and report them to the programmer.

Motivated by examples like the one above, we say that a type of the form $P \Rightarrow \tau$ is *ambiguous* if there is a variable in $TV(P)$ that does not appear in $TV(\tau)$. Now we can apply the results of previous work [2, 7] which guarantee that a term has a well-defined, or *coherent* semantics if its principal type is not ambiguous. This important result relates semantic ambiguity to a syntactic property of principal types, which is easy for a compiler to check.

In some cases, this characterization of ambiguity is too restrictive. There is no need to regard a type of the form $\forall a.(?x\ \ a) \Rightarrow \mathit{Int}$ as ambiguous, for example, because we can never have more than one binding for $?x$ in scope at any given time. In effect, the choice of $a$ will be determined, unambiguously, by the caller, rather than by the context in which $?x$ is used. For similar reasons, we do not need to treat $\forall a.(?x\ \ a, !x\ \ a) \Rightarrow \mathit{Int}$ as an ambiguous type, even though the type variable $a$ appears to the left of the $\Rightarrow$ in a $!x$ predicate; this use of $a$ is 'covered' by the simultaneous occurrence of $a$ in the $?x$ predicate, which means that it will be uniquely determined.

Later, in Section 7.3.3, we will give a more useful definition of syntactic ambiguity that reflects the intuitions presented here. For the time being, let us just summarize the key differences that we have observed between the $?x$ and $!x$ forms of implicit parameterization:

- If $(?x\ \ \tau)$ and $(?x\ \ \tau')$ both hold in the same context, then $\tau$ must be equal to $\tau'$. No such restriction is needed for the $!x$ form of predicates.
- If a type variable $a$ appears in a predicate $(!x\ \ \tau') \in P$ of an inferred type $P \Rightarrow \tau$, but not in the body $\tau$, then the type is ambiguous. No such test is needed for the $?x$ form of predicates.

In the remaining sections we show that these differences arise naturally by viewing the two forms of implicit parameterization as special cases of a more general system.

## 6 Declaring Implicit Parameters

In this section, we show that the system of overloading that we described in Section 4 is just one point in a spectrum of closely related designs, varying in the degree to which programmers are expected to declare overloaded operators before they are used. The $!x$ form of overloading is the most lightweight, requiring no explicit declaration. At the other extreme, where declarations are used to specify the name, type and predicate of each overloaded operator, we obtain a system more like Haskell type classes [19, 17]. More interestingly, for each variation on the theme of overloading, there is a corresponding language design for the dynamic scoping mechanisms that were presented in Section 3.

We start with the $!x$ form of overloading that was described in Section 4 in which there is no need—and indeed, no possibility—for programmers to declare a variable as an overloaded symbol; in effect, we have reserved a lexically distinct family of identifiers, specifically for that purpose. In addition, for each overloaded identifier, we have also provided a corresponding form of predicate, which is related to the identifier by the fact that it uses the same symbol. If, for example, we wanted to define an overloaded equality function that could be used to compare values of several different types, then we might naturally choose to use the symbol $!eq$ for that purpose, with type $\forall a.(!eq\ a) \Rightarrow a$.

An alternative to this is to have only one lexical form of identifier, and allow programmers to specify which will be used as overloaded operators:

**overload** $eq$.

Given this declaration, free occurrences of $eq$, would be treated as constants of type $\forall a.(eq\ a) \Rightarrow a$; without it, they would be flagged as errors. All that declarations like this really do is give programmers more freedom in choosing names for overloaded operators.

It is a small step to allow declarations of overloaded symbols to specify a type, constraining, but also documenting the way in which those symbols are expected to be used. The following illustrates one possible syntax:

**overload** $eq\ a :: a \rightarrow a \rightarrow Bool$.

The effect of this would be to treat $eq$ as a function of type $\forall a.(eq\ a) \Rightarrow a \rightarrow a \rightarrow Bool$. The declared type is treated as a template, with the expectation that any attempt to bind or use $eq$ at a type that does not match it will be flagged as an error. Note that we use a predicate of the form $eq\ a$; we do not need to include the full type of $eq$ here, just the part that can vary from one binding to another. With the particular choice of syntax here, the programmer writes the general form of the predicate to the left of the :: symbol; note that there may be more than one type argument if the type on the right has more than one free variable. The ability to declare types explicitly can result in a strictly more expressive system. For example, the following declaration associates a polymorphic type with the identifier *unit*:

**overload** $unit\ m :: \forall\, a\, .\, a \rightarrow m\ a$.

This takes us beyond the limits of Hindley-Milner typing because it means that a translation will need to pass around a polymorphic function as an implicit parameter.

The final variation that we consider provides a syntax that allows multiple overloaded symbols to be associated with a single form of predicate. The only difference here with the syntax for Haskell type classes is in the use of **overload** instead of **class**:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{overload}\ \mathit{Eq}\ a\ \textbf{where}}\\ \quad \mathit{eq} & :: & a \to a \to \mathit{Bool}\\ \quad \mathit{notEq} & :: & a \to a \to \mathit{Bool}\end{array}$$

Here, both *eq* and *notEq* would be treated as functions of type $\forall a.(\mathit{Eq}\ a) \Rightarrow a \to a \to \mathit{Bool}$. The implicit parameter in this case would be a tuple or record containing the implementations of these two functions for a particular choice of $a$, while *eq* and *notEq* would be implemented by the corresponding projections. This extension does not make the language any more expressive, but it can often be useful as a means of grouping related entities into a single package. In practice, it can also help to reduce the number of predicates in inferred types because multiple overloaded symbols can share the same predicate.

The four points in the design space for overloading that we have described here have corresponding points in the design space for DS-like system. Apart, perhaps, from using a different keyword to distinguish dynamic scoping mechanisms from overloading, the notation, and motivation in each case would be the same as in each of the examples above. The only difference would be in the rules for detecting ambiguity, and for preventing multiple interpretations of a symbol, with the same differences that we described for the $?x$ and $!x$ styles of implicit parameter at the end of Section 5.

## 7 Type Classes and Type Relations

In previous sections, we have assumed a fairly operational interpretation for qualified types, thinking in terms of the additional parameters that they will lead to in the translation of a given program. Another way to understand the predicates in a qualified type is as constraints on the way that a type can be instantiated—in fact that was the original motivation for the term 'predicate'. For example, given the following typing:

$$eq :: \forall a.(Eq\ a) \Rightarrow a \rightarrow a \rightarrow Bool,$$

we can think of the $(Eq\ a)$ predicate as restricting the choices for the (otherwise universally quantified) type variable $a$ to types for which an equality has been defined. In fact, we can identify $Eq$ with that particular set of types, and think of the predicate $(Eq\ a)$ as meaning $a \in Eq$, and of the type for $eq$ as representing all the types in the set $\{t \rightarrow t \rightarrow Bool \mid t \in Eq\}$. It is this set theoretic view of predicates representing membership within a set (or 'class') that motivates our use of the term 'type class'. In Haskell, a type class is populated by a collection of *instance* declarations that can be extended as necessary to include additional datatypes. These declarations, which correspond to a kind of top-level **with** construct, provide general rule schemes that can be used to construct bindings for overloaded symbols at many different types. This can be reflected in the definition of entailment [7], but we will not discuss these issues any further in this paper.

We can also think of the $(?x\ \tau)$ and $(!x\ \tau)$ forms of predicates as representing sets of types in a similar way. We can tell a little more about the kind of sets that can be represented by recalling the restrictions that we placed on the use of such predicates in an assumption set $I$. For example, the sets corresponding to predicates using $?x$ will have at most one element. Of course, different one element sets might be needed to interpret $?x$ at different points in a program, depending on the way that **with** bindings have been used.

We have already mentioned the possibility that a predicate may have multiple type arguments. In this case, we can think of the predicate as representing a *relation* on types. In the context of Haskell, these relations are referred to as *multiple parameter type classes*. In the ten years since type classes were first introduced, many interesting applications have been proposed for multiple parameter type classes [18], and several of the current Haskell implementations provide support for them. However, in our experience, the results can often be disappointing, leading to many ambiguities, and resulting in weak or inaccurate inferred types that delay the compiler's ability to detect type errors. This is probably why even the most recent revision of the Haskell standard [17] does not provide support for multiple parameter type classes.

## 7.1 Example: Collection Types

To illustrate the kinds of problem that can occur, consider the task of providing uniform interfaces to a wide range of different collection types. The following declaration, greatly simplified for the purposes of this presentation, introduces a two parameter class that could be used as the starting point for such a project:

```
class Collects e c where
   empty   :: c
   insert  :: e → c → c
   member  :: e → c → Bool
```

The intention here is that *Collects e c* holds if *c* represents a collection of values of type *e*, which would be demonstrated by providing appropriate definitions for the operations listed in the body of the class. To define the entries in this relation, we might use a collection of **instance** declarations, something like the following[2]:

```
instance Eq e ⇒ Collects e [e] where ...
instance Collects Char BitSet where ...
instance Eq e ⇒ Collects e (e → Bool) where ...
instance (Hashable a, Collects a c)
                 ⇒ Collects a (Array Int c) where ...
```

The first line declares lists of type [*e*] as collections of elements of type *e*, provided that *e* is an equality type. The second line indicates that bit sets can be used to represent sets of characters. The third line uses the representation of a collections as a characteristic functions. The last declaration is for hashtables, represented by arrays, with integer hash codes as indices, and using a second collection type to gather the elements in each bucket (i.e., elements with the same hash code).

[2]For brevity, we omit the definitions for *empty*, *insert*, and *member* in each case.

At first glance, this approach seems very attractive, but in practice, we quickly run into serious problems. For example, if we look more closely at the definition of *empty*, then we see that it has an ambiguous type:

$$\mathit{empty} \ :: \ \mathit{Collects}\ e\ c \Rightarrow c.$$

As a result, *any* program that uses *empty* will have an ambiguous type, and will be rejected by the typechecker. Different kinds of problem occurs with definitions like:

$$\mathit{findEither}\ x\ y\ c \ = \ \mathit{member}\ x\ c\ \mid\mid\ \mathit{member}\ y\ c.$$

For this example, we can infer a principal type:

$$\begin{aligned}\mathit{findEither} \ :: \ & (\mathit{Collects}\ e_1\ c,\ \mathit{Collects}\ e_2\ c) \\ & \quad \Rightarrow e_1 \to e_2 \to c \to \mathit{Bool}.\end{aligned}$$

However, this is actually more general than we would hoped for because it allows the arguments $x$ and $y$ to have different types. For the family of collection types that we are trying to model here, we expect any given collection to contain only one type of value, but the type system is not strong enough to capture this.

### 7.1.1 Using Constructor Classes

One way to avoid the problems with *Collects* is to use constructor classes [10], and replace the definition of *Collects* given previously with the following:

```
class Collects e t where
  empty  :: t e
  insert :: e → t e → t e
  member :: e → t e → Bool
```

This is the approach taken by Okasaki [15], and by Peyton Jones [16], in more realistic attempts to build this kind of library. Note that the type $c$ in the original definition has been replaced with a unary type constructor

$t$; a function of kind $* \to *$. This solves the immediate problems with the types for *empty* and *findEither* mentioned previously by separating out the container from the element type. However, it is considerably less flexible: Only the first of the four **instance** declarations given previously can be modified to work in this setting; the remaining instances define collections using types that do not unify with the form $t\ e$ that is required here.

### 7.1.2 Using Parametric Type Classes

Another alternative is to use *parametric type classes* [3], which allows predicates of the form $c \in \mathit{Collects}\ e$, meaning that $c$ is a member of the class $\mathit{Collects}\ e$. Intuitively, we think of having one type class $\mathit{Collects}\ e$ for each possible choice of the $e$ parameter. The definition of a parametric *Collects* class looks much like the original:

```
class c ∈ Collects e where
   empty   ::  c
   insert  ::  e → c → c
   member  ::  e → c → Bool
```

What makes it different is the implied assumption that the element type $e$ is uniquely determined by the collection type $c$. A compiler that supports parametric type classes must ensure that the declared instances of *Collects* do not violate this property. In return, it can often use the same information to avoid ambiguity and to infer more accurate types. For example, the type of *empty* is now $\forall e, c.(c \in \mathit{Collects}\ e) \Rightarrow c$, and we do not need to treat this as being ambiguous, because the unknown element type $e$ is uniquely determined by $c$.

### 7.1.3 Using Functional Dependencies

In this paper, we introduce a generalization of parametric type classes that allows programmer to declare explicit dependencies between the parameters of a predicate. For example, this allows us to annotate the original class definition with a dependency $c \rightsquigarrow e$, indicating that the element type $e$ is uniquely determined by the choice of $c$:

```
class Collects e c | c ⇝ e where
   empty   :: c
   insert  :: e → c → c
   member  :: e → c → Bool
```

We defer more detailed discussion about the interpretation of these dependency annotations to Section 7.3. We note, however, that this approach is strictly more general than parametric type classes because it allows us to express a larger class of dependencies, including mutual dependencies such as $\{a \rightsquigarrow b,\ b \rightsquigarrow a\}$. It is also easier to integrate with the existing syntax of Haskell because it does not require any changes to the syntax of predicates.

## 7.2 Relations and Functional Dependencies

Before we given any further examples, we pause for a brief primer on relations and functional dependencies, and for a summary of our notation. These ideas were originally developed as a foundation for the study of relational databases [1]. They are well-established, and detailed presentations of the theory, and of useful algorithms for working with them in practical settings, can be found in standard textbooks on the theory of databases [13]. The novelty of the current paper is in applying them to the design of a type system.

### 7.2.1 Relations

Following standard terminology, a *relation* $R$ over an indexed family of sets $\{D_i\}_{i\in I}$ is just a set of *tuples*, each of which is an indexed family of values $\{t_i\}_{i\in I}$ such that $t_i \in D_i$ for each $i \in I$. More formally, $R$ is just a subset of $\Pi i \in I.D_i$, where a tuple $t \in (\Pi i \in I.D_i)$ is a function that maps each index value $i \in I$ to a value $t_i \in D_i$ called the $i$th component of $t$. In the special case where $I = \{1, \ldots, n\}$, this reduces to the familiar special case where tuples are values $(t_1, \ldots, t_n) \in D_1 \times \ldots \times D_n$. If $X \subseteq I$, then we write $t_X$, pronounced "$t$ at $X$", for the restriction of a tuple $t$ to $X$. Intuitively, $t_X$ just picks out the values of $t$ for the indices appearing in $X$, and discards any remaining components.

### 7.2.2 Functional Dependencies

In the context of an index set $I$, a *functional dependency* is a term of the form $X \leadsto Y$, read as "$X$ determines $Y$," where $X$ and $Y$ are both subsets of $I$. If $X$ and $Y$ are known sets of elements, say $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, then we will often write the dependency $X \leadsto Y$ in the form $x_1 \ \ldots \ x_n \leadsto y_1 \ \ldots \ y_m$. If a relation satisfies a functional dependency $X \leadsto Y$, then the values of any tuple at $Y$ are uniquely determined by the values of that tuple at $X$. More formally, we write:

$$R \models X \leadsto Y \iff \forall t, s \in R. t_X = s_X \Rightarrow t_Y = s_Y.$$

This also extends to sets of dependencies:

$$R \models F \iff \forall (X \leadsto Y) \in F. R \models X \leadsto Y.$$

For example, if we take $I = \{1, 2\}$, then the relations satisfying $\{\{1\} \leadsto \{2\}\}$ are just the partial functions from $D_1$ to $D_2$, and the relations satisfying $\{\{1\} \leadsto \{2\}, \{2\} \leadsto \{1\}\}$ are the partial injective functions from $D_1$ to $D_2$.

If $F$ is a set of functional dependencies, and $J \subseteq I$ is a set of indices, then the *closure* of $J$ with respect to $F$, written $J^+_F$ is the smallest set such that:

- $J \subseteq J^+_F$; and
- If $(X \leadsto Y) \in F$, and $X \subseteq J^+_F$, then $Y \subseteq J^+_F$.

For example, if $I = \{1, 2\}$, and $F = \{\{1\} \leadsto \{2\}\}$, then $\{1\}^+_F = I$, and $\{2\}^+_F = \{2\}$. Intuitively, the closure $J^+_F$ is just the set of indices that are uniquely determined, either directly or indirectly, by the indices in $J$ and the dependencies in $F$. Closures like this are easy to compute; for practical purposes, a simple fixed point iteration will suffice.

## 7.3 Typing with Functional Dependencies

This section describes the steps that are needed to make use of information about functional dependencies during type checking. We will assume that dependencies are specified as part of whatever mechanism the language provides for introducing new forms of predicate, whether it be **class** declarations as in Section 7.1, **overload** declarations as in Section 6, or some other construct altogether. More specifically, we will assume that there is some collection of predicate symbols $p$, each of which has an associated set of indices $I_p$, and an associated set of functional dependencies $F_p$. We will also assume that all predicates are written in the form $p\ t$, where $t$ is a tuple of types indexed by $I_p$.

The techniques described in the following subsections have been used to extend the Hugs interpreter [11] to allow the declarations of classes to be annotated with sets of functional dependencies, as suggested in Section 7.1.3. The implementation works well in practice, and we expect that it will be included in the next public distribution. Pleasingly, some early users have already found new applications for this extension in their own work.

Formal justification for these techniques comes from the theory of improvement for qualified types [9]. The main idea is that we can apply certain forms of satisfiability-preserving substitutions during type checking, resulting in more accurate inferred types, without compromising on a useful notion of principal types.

### 7.3.1 Ensuring that Dependencies are Valid

Our first task is to ensure that all declared instances for a predicate symbol $p$ are consistent with the functional dependencies in $F_p$. For example, suppose that we have an instance for $p$ of the form:

**instance** $\ldots \Rightarrow p\ t$ **where** $\ldots$

Now, for each $(X \rightsquigarrow Y) \in F_p$, we must ensure that $TV(t_Y) \subseteq TV(t_X)$ or otherwise the elements of $t_Y$ might not be uniquely determined by the elements of $t_X$. A further restriction is needed to ensure pairwise compatibility between instances for $p$. For example, if we have a second instance:

**instance** $\ldots \Rightarrow p\ s$ **where** $\ldots$,

and a dependency $(X \rightsquigarrow Y) \in F_p$, then we must ensure that $t_Y = s_Y$ whenever $t_X = s_X$. In fact, on the assumption that the two instances will normally contain type variables—which could later be instantiated to more specific types—we will actually need to check that: if $t_X$ and $s_X$ have a most general unifier $U$, then $Ut_Y = Us_Y$. This is enough to guarantee that the declared dependencies are satisfied. For example, the instance declarations in Section 7.1 are consistent with the dependency $c \leadsto a$ that we are assuming for the predicate symbol *Collects*.

### 7.3.2 Improving Inferred Types

There are two ways that a dependency $(X \leadsto Y) \in F_p$ for a predicate symbol $p$ can be used to help infer more accurate types:

- Suppose that we have two predicates $p\ t$ and $p\ s$. If $t_X = s_X$, then $t_Y$ and $s_Y$ must be equal.
- Suppose that we have an inferred predicate $p\ t$, and an instance:

  **instance** $\ldots \Rightarrow p\ t'$ **where** $\ldots$

  If $t_X = St'_X$, for some substitution $S$ (which could be calculated by one-way matching), then $t_Y$ and $St'_Y$ must be equal.

In both cases, we can use unification to ensure that the equalities are satisfied: If unification fails, then we have detected a type error.

### 7.3.3 Detecting Ambiguity

As we described in Section 5, we must reject any program with an ambiguous principal type because of the potential for semantic ambiguity. With our previous definition, a type of the form $P \Rightarrow \tau$ is considered ambiguous if there is a type variable $a \in TV(P)$ that is not also in $TV(\tau)$; our intuition is that, if there is no reference to $a$ in the body of the type, then there will be no way to determine how it should be resolved. However, we have also seen that more relaxed notions of ambiguity can be used in situations where the potentially ambiguous variable $a$ might be resolved by some other means. So, in fact, we need not insist that every $a \in TV(P)$ is mentioned explicitly in $\tau$, so long as it is uniquely determined by the variables in $TV(\tau)$.

The first step to formalizing this idea is to note that every set of predicates $P$ induces a set of functional dependencies $F_P$ on the type variables in $TV(P)$:

$$\{\, TV(t_X) \leadsto TV(t_Y) \,|\, (p\ t) \in P,\ (X \leadsto Y) \in F_p \,\}.$$

This has a fairly straightforward reading: if all of the variables in $t_X$ are known, and if $X \leadsto Y$, then the components of $t$ at $X$ are also known, and hence so are the components, and thus the type variables, in $t$ at $Y$.

To determine if a particular type $P \Rightarrow \tau$ is ambiguous, we should first calculate the set of dependencies $F_P$, and then take the closure of $TV(\tau)$ with respect to $F_P$ to obtain the set of variables that are determined by $\tau$. The type is ambiguous only if there are variables in $P$ that are not included in this closure. More concisely, the type $P \Rightarrow \tau$ is *ambiguous* if, and only if: $TV(P) \not\subseteq (TV(\tau))^+_{F_P}$.

## 7.4 A Unified View of Implicit Parameters

We are now in a position to show that the mechanisms for dynamic scoping and for overloading in earlier sections are really just special cases of our more general system using type relations and functional dependencies:

- For dynamic scoping, we add a dependency of the form $\emptyset \leadsto a$ for each predicate argument $a$.
- For overloading, we add a dependency of the form $a \leadsto \emptyset$ for each predicate argument $a$.

The different treatments of multiple interpretations and of ambiguity for these two forms of implicit parameter that we observed in Section 5 follow directly from the more general definitions in Sections 7.3.2 and 7.3.3, respectively; it is as if we had declared a pair of classes for each identifier $x$:

$$\begin{array}{l}
\textbf{class}\ (?x\ \ a) \mid\ \leadsto a\ \textbf{where}\ ?x\ \ ::\ \ a \\
\textbf{class}\ (!x\ \ a) \mid a \leadsto\ \ \textbf{where}\ !x\ \ ::\ \ a.
\end{array}$$

In a similar way, each of the different designs described in Section 6, including Haskell type classes, can be understood as special cases of the more general system presented here.

## 8 Tags

In this penultimate section, we hint briefly at the potential of a further special case that occurs when we use predicates that have no parameters, and so corresponding to nullary relations. This provides a mechanism for attaching a 'tag' to the type of a function $f$, that will then be propagated through the call hierarchy to all functions that depend on $f$, either directly or indirectly. Thus we can use types to effect a dependency analysis on a program, and because it is type-based, this analysis can be made to work in a way that is compatible with separate compilation. We can be more selective in our analysis by tagging only some of the functions that might be used. One area where facilities like this would be particularly useful is in security. For example, if a language provides unsafe primitives, then they could be tagged as such using a definition like the following to ensure that the potential for unsafe behavior is clearly signaled in the type of any program that uses this primitive:

```
class Unsafe where
  unsafePerformIO  ::  IO a → a
```

Another possibility might be to define a security hierarchy, as in the following simple example:

```
class Public where ...
class Public ⇒ LowSecurity where ...
class LowSecurity ⇒ HighSecurity where ...
class HighSecurity ⇒ TopSecret where ...
```

The intention here would be to assign appropriate levels of security for potentially restricted operations, and then use type inference to determine which level of security any given program requires. The corresponding implicit parameters could be used to thread the required key or access code through the program to the point where it is required, without exposing it as an explicit parameter that could be inspected, modified, or misused.

## 9 Conclusions and Future Work

In this paper we have explored the design space for type-based implicit parameterization, investigating the similarities between different systems, and exposing some of the choices for language design. Our most general system makes essential use of relations on types, together with functional dependencies, and is useful in its own right, as well as providing a unified view of other systems.

In constructing this system, we have used ideas from the theory of relational databases. One interesting area for future work would be to see if other ideas developed there could also be exploited in the design of programming language type systems.

## Acknowledgments

I would like to thank my friends and colleagues in PacSoft at the Oregon Graduate Institute for a stimulating environment, and for their interest in this work. Particular thanks to Jeff Lewis for helping me to understand his implementation of implicit parameters in Hugs, and to Lois Delcambre of the Database and Object Technology Laboratory at OGI for explaining the role that functional dependencies play in database theory.

The research reported in this paper was supported by the USAF Air Materiel Command, contract # F19628-96-C-0161.

## References

[1] W. W. Armstrong. Dependency structures of data base relationships. In *IFIP Cong.*, Geneva, Switzerland, 1974.

[2] S. M. Blott. *An approach to overloading with polymorphism.* PhD thesis, Department of Computing Science, University of Glasgow, July 1991. (draft version).

[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (extended abstract). In *ACM conference on LISP and Functional Programming*, San Francisco, CA, June 1992.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, NM, January 1982.

[5] M. J. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall International, 1988.

[6] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[7] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[8] M. P. Jones. ML typing, explicit polymorphism and qualified types. In *TACS '94: Conference on theoretical aspects of computer software, Sendai, Japan*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 789.

[9] M. P. Jones. Simplifying and improving qualified types. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995.

[10] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

[11] M. P. Jones and J. C. Peterson. *Hugs 98 User Manual*, May 1999. Available on the web from `http://www.haskell.org/hugs/`.

[12] J. Lewis, M. Shields, J. Launchbury, and E. Meijer. Implicit parameters: Dynamic scoping with static types. manuscript, submitted for publication, July 1999.

[13] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[15] C. Okasaki. *Edison User's Guide*, May 1999.

[16] S. Peyton Jones. Bulk types with class. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.

[17] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, February 1999. Available on the web from `http://www.haskell.org/definition/`.

[18] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997. Available on the web from `http://www.cse.ogi.edu/~mpj/pubs/multi.html`.

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