Making Implicit Parameters Explicit

Atze Dijkstra
S. Doaitse Swierstra

institute of information and computing sciences,
utrecht university

technical report UU-CS-2005-032

www.cs.uu.nl

Making Implicit Parameters Explicit

Atze Dijkstra and S. Doaitse Swierstra

August 8, 2005

Abstract

In almost all languages all arguments to functions are to be given explicitly. The Haskell class system however is an exception: functions can have class predicates as part of their type signature, and dictionaries are implicitly constructed and implicitly passed for such predicates, thus relieving the programmer from a lot of clerical work and removing clutter from the program text. Unfortunately Haskell maintains a very strict boundary between the implicit and the explicit world; if the implicit mechanisms fail to construct the hidden dictionaries there is no way the programmer can provide help, nor is he able to override the choices made by the implicit mechanisms. In this paper we describe, in the context of Haskell, a mechanism that allows the programmer to explicitly construct implicit arguments. This extension blends well with existing resolution mechanisms, since it only overrides the default behavior. We include a description of the use of partial type signatures, which liberates the programmer from having to choose between specifying a complete type signature or no type signature at all. Finally we show how the system can easily be extended to deal with higher-order predicates, thus enabling the elegant formulation of some forms of generic programming.

1 Introduction

The Haskell class system, originally introduced by both Wadler and Blott [43] and Kaes [27], offers a powerful abstraction mechanism for dealing with overloading (ad-hoc polymorphism). The basic idea is to restrict the polymorphism of a parameter by specifying that some predicates have to be satisfied when the function is called:

$$f :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Int$$
$$f = \lambda \quad x \quad y \rightarrow \textbf{if}\ x == y\ \textbf{then}\ 3\ \textbf{else}\ 4$$

In this example the type signature for f specifies that values of any type a can be passed as arguments, as long as the predicate $Eq\ a$ is satisfied. Such predicates are introduced by *class declaration*s, as in the following version of Haskell's *Eq* class declaration:

$$\textbf{class}\ Eq\ a\ \textbf{where}$$
$$(==) :: a \rightarrow a \rightarrow Bool$$

The presence of such a class predicate in a type requires the availability of a collection of functions and values which can only be used on a type a for which the class predicate holds. For brevity, the given definition for class *Eq* omits the declaration for /=. A class declaration alone is not sufficient: *instance declarations* specify for which types the predicate actually can be satisfied, simultaneously providing an implementation for the functions and values as a witness for this:

```
instance Eq Int where
  x == y = primEqInt x y
instance Eq Char where
  x == y = primEqChar x y
```

Here the equality functions for *Int* and *Char* are implemented by the primitives *primEqInt* and *primEqChar*. The compiler turns such instance declarations into records (dictionaries) containing the functions as fields, and thus an explicit version of this internal machinery reads:

```
data EqD a = EqD {eqEqD :: a → a → Bool}   -- class Eq
eqDInt     = EqD primEqInt                 -- Eq Int
eqDChar    = EqD primEqChar                -- Eq Char
```

Inside a function the elements of the predicate's dictionaries are available, as if they were defined as top-level variables. This is accomplished by implicitly passing a dictionary for each predicate occurring in the type of the function. So the actual implementation of f (apart from all kind of optimisations) is:

```
f ::   EqD a → a → a → Int
f = λ dEq       x     y → if (eqEqD dEq) x y then 3 else 4
```

At the call site of the function f the dictionary that corresponds to the actual type of the polymorphic argument must be passed. Thus the expression f 3 4 can be seen as an abbreviation for the semantically more complete f *eqDInt* 3 4.

Motivating examples The translation from f 3 4 to f *eqDInt* 3 4 is done implicitly; a programmer has little or no control over the passing of dictionaries. This becomes problematic as soon as a programmer desires to express something which the language definition cannot infer automatically. For example, we may we want to call f with an alternate instance for *Eq Int*, which implements a different equality on integers:

```
instance Eq Int where
  x == y = primEqInt (x `mod` 2) (y `mod` 2)
```

Unfortunately this extra instance declaration would introduce an ambiguity, and is thus forbidden by the language definition; the instances are said to overlap. However, a programmer could resolve the issue if he was only able to explicitly specify which of these two possible instances should be passed to f.

As a second example we briefly discuss the use Kiselyov and Chan [31] make of the type class system to configure programs. In their modular arithmetic example integer arithmetic is configured by a modulus: all integer arithmetic is done modulo this modulus. The modulus is implemented by a class function *modulus*:

```
class Modular s a | s → a where modulus :: s → a
newtype M s a = M a
normalize :: (Modular s a, Integral a) ⇒ a → M s a
normalize a :: M s a = M (mod a (modulus (⊥ :: s)))
instance (Modular s a, Integral a) ⇒ Num (M s a) where
  M a + M b = normalize (a + b)
  ...    -- remaining definitions omitted
```

The problem now is to create for a value m of type a an instance of *Modular s a* for which *modulus* returns this m. Some ingenious type hackery is involved where phantom type s (evidenced by $\perp$'s) uniquely represents the value m, and as such is used as an index into the available instances for *Modular s a*. This is packaged in the following function which constructs both the type s and the corresponding dictionary (for which *modulus* returns m) for use by k:

```
withModulus :: a → (∀s.Modular s a ⇒ s → w) →  w
withModulus   m    k                        =  ...
```

They point out that this could have been done more directly if local type class instances would have been available:

```
data Label
withModulus :: a → (∀s.Modular s a ⇒ s → w) → w
withModulus   m    k
  = let instance Modular Label a where modulus _ = m
    in  k (⊥ :: Label)
```

The use of explicit parameter passing for an implicit argument proposed by us in this paper would have even further simplified the example, as we can avoid the phantom type *Label* and related type hackery altogether and instead create and pass the instance directly.

As we may infer from the above the Haskell class system, which was originally only introduced to describe simple overloading, has become almost a programming language of its own, used (and abused as some may claim) for unforeseen purposes.

Haskell's point of view Haskell's class system has turned out to be theoretically sound and complete [21], although some language constructs prevent Haskell from having principal types [10]. The class system is flexible enough to incorporate many useful extensions [20, 24]. Its role in Haskell has been described in terms of an implementation [23] as well as its semantics [13, 9]. Many language constructs do their work

automatically and implicitly, to the point of excluding the programmer from exercising influence. Here we feel there is room for improvement, in particular in dealing with implicit parameters.

The compiler is fully in control of which dictionary to pass for a predicate, determined as part of the resolution of overloading. This behavior is the result of the combination of the following list of design choices:

- A class definition introduces a record type (for the dictionary) associated with a predicate over type variables.
- Instance definitions describe how to construct a value for the record type for the class predicate specialized for a specific type (or combination of types in the case of multiparameter type classes).
- The type of a function specifies the predicates for which dictionaries have to be passed at the call site of the function.
- Which dictionary is to be passed at the call site of a function is determined by:
  - required dictionaries at the call site of a function; this is determined by the predicates in the instantiated type of the called function.
  - the available dictionaries introduced by instance definitions.

  Internally the compiler uses a predicate proving machinery and heuristics [25, 35, 9] to compute the proper dictionaries.
- Which dictionaries are to be passed is fully fixed by the language definition.
- The language definition uses a statically determined set of dictionaries introduced by instance definitions and a fixed algorithm for determining which dictionaries are to be passed.

The result of this is both a blessing and a curse. A blessing because it silently solves a problem (i.e. overloading), a curse because as a programmer we cannot easily override the choices made in the design of the language (i.e. via Haskell's default mechanism), and worse, we can in no way assist the compiler if no unique solution according to the language semantics exists. For example, overlapping instances occur when more than one choice for a dictionary can be made. Smarter, more elaborate versions of the decision making algorithms can and do help [14], but in the end it is only the programmer who can fully express his intentions. The system at best can only make a guess.

The issue central to this paper is that Haskell demands from a program that all choices about which dictionaries to pass can be made automatically and uniquely, whereas we also want to be able to specify this ourselves explicitly. If the choice made (by Haskell) does not correspond to the intention of the programmer, the only solution is to convert all involved implicit arguments into explicit ones, thus necessitating changes all over the program. Especially for (shared) libraries this may not always be feasable.

Our contribution Our approach takes explicitness as a design starting point, as opposed to the described implicitness featured by the Haskell language definition. To make the distinction between our and Haskell's approach clear in the remainder of this paper, we call our explicit language and its implementation Explicit Haskell (EH) whereas we refer to Haskell language and its implementations by just Haskell.

- In principle, all aspects of an EH program can be explicitly specified, in particular the types of functions, types of other values, and the manipulation of dictionaries, without making use of or referring to the class system.
- The programmer is allowed to omit explicit specification of some program aspects; EH then does its utmost to infer the missing information.

Our approach allows the programmer and the EH system to jointly construct the completely explicit version of a program, whereas an implicit approach inhibits all explicit programs which the type inferencer cannot infer but would otherwise be valid. If the type inferencer cannot infer what a programmer expects it to infer, then the programmer can provide the required information. In this sense we get the best of two worlds: the simplicity of systems like system F [12, 40] and Haskell's ease of programming.

In this paper explicitness takes the following form:

- Dictionaries introduced by instance definitions can be named; the dictionary can be accessed by name as a record value.
- The set of class instances and associated dictionaries to be used by the proof machinery can be used as normal values, and normal (record) values can be used as dictionaries for predicates as well.
- The automatic choice for a dictionary at the call site of a function can be overruled.
- Types can be partially specified, thus having the benefit of explictness as well as inference, but avoiding the obligation of the "all or nothing" explicitness usually enforced upon the programmer. Although this feature is independent of explicit parameter passing, it blends nicely with it.
- Types can be composed of the usual base types, predicates and quantifiers (both universal and existential) in arbitrary combinations.

We will focus on all but the last items of the preceding list: the explicit passing of values for implicit parameters. Although explicit typing forms the foundation on which we build [6, 5], we discuss it only as much as is required.

Related to programming languages in general, our contribution, though inspired by and executed in the context of Haskell, offers language designers a mechanism for more sophisticated control over parameter passing, by allowing a mixture of explicit and implicit parameter passing.

Outline of this paper In this paper we focus on the exploration of explicitly specified implicit parameters, to be presented in the context of EH, a Haskell variant [4, 7, 6, 5] in which all features described in this paper have been implemented. In Section 2 we start with preliminaries required for understanding the remainder of this paper. In Section 3 we present examples of what we can express in EH. The use of partial type signatures and their interaction with predicates is demonstrated in Section 4. In Section 5 we give some insight in our implementation, highlighting the distinguishing aspects as compared to traditional implementations. In Section 6 we discuss some remaining design issues and related work. We conclude in Section 7.

Limitations of this paper Our work is made possible by using some of the features already available in EH, for example higher ranked types and the combination of type checking and inferencing. We feel that our realistic setting contributes to a discussion surrounding the issues of combining explicitly specified and inferred program aspects [42] as it offers a starting point for practical experience. For reasons of space we have made the following choices:

- We present examples and part of our implementation, so the reader gets an impression of what can be done and how it ties in with other parts of the implementation [4].
- We do *not* present all the context required to make our examples work. This context can be found elsewhere [6, 7, 5].
- We focus on prototypical implementation before considering formally proving properties of EH.
- We do not prove properties like soundness, completeness and principality. In Section 6 we will address the reasons why have chosen not to deal with those issues here.
- Our type rules therefore describe an algorithm which has been implemented using an attribute grammer system [18, 2]. An attribute grammar provides better separation of implementation aspects whereas type rules are more concise in their presentation; we therefore have chosen to incorporate typing rules in this paper. We describe the similarities between typing rules and their attribute grammar counterpart in a companion paper [8].

2 Preliminaries

Intended as a platform for both education and research, EH offers advanced features like higher ranked types, existential types, partial type signatures and records. Syntactic sugar has been kept to a minimum in order to ease experimentation with and understanding of the implementation; other mechanisms like syntax macro's [3] provide the means for including additional syntax into the language without having to change the compiler.

Values (expressions, terms):		
e	::= $int \mid char$	literals
	$\mid i$	program variable
	$\mid e\ e$	application
	$\mid \lambda i \rightarrow e$	abstraction
	$\mid$ **let** $\overline{d}$ **in** e	local definitions
	$\mid (l = e, ...)$	record
	$\mid (e \mid l := e, ...)$	record update
	$\mid e.l$	record selection
	$\mid e\ (!e \leftsquigarrow \pi!)$	*explicit implicit application*
	$\mid \lambda(!i \leftsquigarrow \pi!) \rightarrow e$	*explicit implicit abstraction*
Declarations of bindings:		
d	::= $i = e$	value binding
	$\mid i :: \sigma$	value type signature
	$\mid$ **data** $\sigma = \overline{I\ \overline{\sigma}}$	data type
	$\mid$ **class** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	class
	$\mid$ **instance** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	introduced instance
	$\mid$ **instance** $i \leftsquigarrow \overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	*named introduced instance*
	$\mid$ **instance** $i :: \overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	*named instance*
	$\mid$ **instance** $e \leftsquigarrow \pi$	*value introduced instance*
Identifiers:		
ι	::= i	lowercase: (type) variables
	$\mid I$	uppercase: (type) constructors
	$\mid l$	field labels

Figure 1: EH terms (emphasized ones explained throughout the text)

Fig. 1 and Fig. 2 show the terms and types featured in EH. Throughout this paper all language constructs will be gradually introduced and explained. In general, we designed EH to be as upwards compatible as possible with Haskell. We point out some aspects required for understanding the discussion in the next section:

- An EH program is single stand alone term. All types required in subsequent examples are either silently assumed to be similar to Haskell or will be introduced explicitly.
- All bindings in a **let** expression are analysed together; in Haskell this constitutes a binding group.
- We represent dictionaries by records. Records are denoted as parenthesized comma separated sequences of field definitions. Extensions and updates to a record e are denoted as $(e \mid ...)$, with e in front of the vertical bar '|'. The notation and semantics is based on existing work on extensible records [11, 26]. Record extension and updates are useful for re-using values from a record.

The universe of types as used in this paper is shown in Fig. 2. A programmer can specify types using the same syntax. We mention this because often types are categorized based on the presence of (universal) quantifiers and predicates [15, 37]. We however allow quantifiers at higher ranked positions in our types and predicates as well. For example, the following is a valid type expression in EH:

$$(\forall a.a \rightarrow a) \rightarrow (\forall b.b \rightarrow b)$$

Existential types are part of EH, but are omitted here because we will not use them in this paper. Quantification has lower priority than the other composite types, so in a type expression without parentheses the scope of the quantifier extends to the far right of the type expression.

We make no attempt to infer higher ranked types [29, 30, 17]; instead we propagate explicitly specified types as good as possible to wherever this information is needed. Our strategies here are elaborated in a forthcoming publication [5].

3 Implicit parameters

In this section we give EH example programs, demonstrating most of the features related to implicit parameters. After pointing out these features we continue with exploring the finer details.

Basic explicit implicit parameters Our first demonstration EH program contains the definition of the standard Haskell function *nub* which removes duplicate elements from a list. A definition for *List* has been included; definitions for *Bool*, *filter* and *not* are omitted. In this example the class *Eq* also contains *ne* which we will omit in later

Types:		
σ ::=	$Int \mid Char$	literals
|	v	variable
|	$\sigma \to \sigma$	abstraction
|	$\pi \Rightarrow \sigma$	implicit abstraction
|	$\sigma\ \sigma$	type application
|	$\forall v.\sigma$	universally quantified type
|	$(l :: \sigma, ...)$	record
Types for impredicativity inferencing:		
σ ::=	...	
|	$v\ [\overline{\varphi}]$	type alternatives
$\mathfrak{o}$ ::=	σ	distinguishing notation for σ with $v\ [\overline{\varphi}]$

Figure 2: EH types

examples. Notice that a separate *nubBy*, which is in the Haskell libraries enabling the parameterisation of *nub* with an equality test, is no longer needed:

```
let data List a = Nil | Cons a (List a)
    class Eq a where
      eq :: a → a → Bool
      ne :: a → a → Bool
    instance dEqInt ⇜ Eq Int where     -- (1)
      eq = primEqInt
      ne = λx y → not (eq x y)
    nub :: ∀ a.Eq a ⇒ List a → List a
    nub = λxx → case xx of
                  Nil         → Nil
                  Cons x xs → Cons x (nub (filter (ne x) xs))
    eqMod2 :: Int → Int → Bool
    eqMod2 = λx y → eq (mod x 2) (mod y 2)
    n1 = nub (!dEqInt ⇜ Eq Int!)       -- (2)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
    n2 = nub (!(eq = eqMod2            -- (2)
                ,ne = λx y → not (eqMod2 x y)
                ) ⇜ Eq Int
              !)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
in  ...
```

This example demonstrates the use of the two basic ingredients required for being explicit in the use of implicit parameters (the list items correspond to the commented number in the example):

1. The notation $\leftsquigarrow$ binds an identifier, here *dEqInt*, to the dictionary representing the instance. The record *dEqInt* from now on is available as a normal value.

2. Explicitly passing a parameter is syntactically denoted by an expression between (! and !). The predicate after the $\leftsquigarrow$ explicitly states the predicate for which the expression is an instance dictionary (or *evidence*). The dictionary expression for n_1 is formed by using *dEqInt*, for n_2 a new record is created: a dictionary can also be created by updating an already existing one like *dEqInt*; in our discussion (Section 6) we will come back to this.

This example demonstrates our view on implicit parameters:

- Program values live in two, possibly overlapping, worlds, *explicit* and *implicit*.
- Parameters are either passed explicitly, by the juxtapositioning of explicit function and argument expressions, or passed implicitly (invisible in the program text) to an explicit function value. In the implicit case the language definition determines which value to take from the implicit world.
- Switching between the explicit and implicit world is accomplished by means of additional notation. We go from implicit to explicit by instance definitions with the naming extension, and in the reverse direction by means of the (! !) construct.

The *Modular* motivating example now can be simplified to (merging our notation into Haskell):

```
class Modular a where modulus :: a
newtype M a = M a
normalize :: (Modular a, Integral a) ⇒ a → M a
normalize a = M (mod a modulus)
instance (Modular a, Integral a) ⇒ Num (M a) where
  M a + M b = normalize (a + b)
  ...    -- remaining definitions omitted
withModulus :: a →     (Modular a ⇒ w) → w
withModulus    (m :: a) k
   = k (!(modulus = m) ⇜ Modular a!)
```

Higher order predicates We also allow the use of higher order predicates. Higher order predicates are already available in the form of instance declarations. For example, the following program fragment defines the instance for *Eq* (*List a*) (the code for the body of *eq* has been omitted):

```
instance dEqList ⇜ Eq a ⇒ Eq (List a) where
  eq = λx y → ...
```

The important observation is that in order to be able to construct the dictionary for *Eq* (*List a*) we need a dictionary for *Eq a*. This corresponds to interpreting *Eq a* ⇒

Eq (*List a*) as stating that *Eq* (*List a*) can be proven from *Eq a*. The implementation for this instance is a function taking the dictionary for *Eq a* and constructing the dictionary for *Eq* (*List a*). Such a function is called a *dictionary transformer*.

We allow higher order predicates to be passed as implicit arguments, provided the need for this is specified explicitly. For example, in f we can abstract from the dictionary transformer for *Eq* (*List a*), which can then be passed either implicitly or explicitly:

$$f :: (\forall\ a.Eq\ a \Rightarrow Eq\ (List\ a)) \Rightarrow Int \rightarrow List\ Int \rightarrow Bool$$
$$f = \lambda p\ q \rightarrow eq\ (Cons\ p\ Nil)\ q$$

The effect is that the dictionary for *Eq* (*List Int*) will be computed inside f as part of its body, using the passed dictionary transformer and a more globally available dictionary for *Eq Int*. Without the use of this construct the dictionary would be computed only once globally by:

let *dEqListInt* = *dEqList dEqInt*

The need for higher order predicates really becomes apparent when genericity is implemented using the class system. The following example is taken from Hinze [16]:

```
let data Bit          = Zero | One
    data GRose f a = GBranch a (f (GRose f a))
in  let class Binary a where
          showBin :: a → List Bit
        instance dBI ↚ Binary Int where
          showBin  = ...
        instance dBL ↚ Binary a ⇒ Binary (List a) where
          showBin  = ...
        instance dBG ↚ (Binary a, (∀ b.Binary b ⇒ Binary (f b)))
                              ⇒ Binary (GRose f a) where
          showBin  = λ(GBranch x ts)
                              → showBin x ++ showBin ts
in  let v1 = showBin (GBranch 3 Nil)
    in  v1
```

The explicit variant of the computation for v_1 using the explicit parameter passing mechanism reads:

$$v_1 = showBin\ (!dBG\ dBI\ dBL \leftsquigarrow Binary\ (GRose\ List\ Int)!)$$
$$(GBranch\ 3\ Nil)$$

The value for *dBG* is defined by the following translation to an explicit variant using records; the identifier *showBin* has been replaced by *sb*, *List* by *L* and *Bit* by *B* in order to keep the programfragment compact:

$$sb = \lambda d \rightarrow d.sb$$

$$
\begin{aligned}
&dBG :: \quad (sb :: a \to L\ B) \\
&\qquad \to (\forall\ b.(sb :: b \to L\ B) \to (sb :: f\ b \to L\ B)) \\
&\qquad \to (sb :: GRose\ f\ a \to L\ B) \\
&dBG = \lambda dBa\ dBf \to d \\
&\quad \textbf{where}\ d = (sb = \lambda(GBranch\ x\ ts) \\
&\qquad\qquad\qquad\qquad \to sb\ dBa\ x +\!\!+ sb\ (dBf\ d)\ ts \\
&\qquad\qquad)
\end{aligned}
$$

Hinze's solution essentially relies on the use of the higher order predicate $Binary\ b \Rightarrow Binary\ (f\ b)$ in the context of $Binary\ (GRose\ f\ a)$. The rationale for this particular code fragment falls outside the scope of this paper, but the essence of its necessity lies in the definition of the *GRose* data type which uses a type constructor f to construct the type $(f\ (GRose\ f\ a))$ of the second member of *GBranch*. When constructing an instance for $Binary\ (GRose\ f\ a)$ an instance for this type is required. Type (variable) f is not fixed, so we cannot provide an instance for $Binary\ (f\ (GRose\ f\ a))$ in the context of the instance. However, given dictionary transformer $dBf \leftsquigarrow Binary\ b \Rightarrow Binary\ (f\ b)$ and the instance $d \leftsquigarrow Binary\ (GRose\ f\ a)$ under construction, we can construct the required instance: $dBf\ d$. The type of v_1 in the example instantiates to *GRose List Int*; the required dictionary for the instance $Binary\ (GRose\ List\ Int)$ can be computed from *dBI* and *dBL*.

The finer details For our discussion we take the following fragment as our starting point:

$$
\begin{aligned}
&\textbf{let}\ f = \lambda p\ q\ r\ s \to (eq\ p\ q, eq\ r\ s) \\
&\textbf{in}\ f\ 3\ 4\ 5\ 6
\end{aligned}
$$

Haskell infers the following type for f:

$$f :: \forall\ a\ b.(Eq\ b, Eq\ a) \Rightarrow a \to a \to b \to b \to (Bool, Bool)$$

On the other hand, EH infers:

$$f :: \forall\ a.Eq\ a \Rightarrow a \to a \to \forall\ b.Eq\ b \Rightarrow b \to b \to (Bool, Bool)$$

EH not only inserts quantifiers as close as possible to the place where the quantified type variables occur, but does this for the placement of predicates in a type as well. The idea is to instantiate a quantified type variable or pass an implicit parameter corresponding to a predicate as lately as possible, where later is defined as the order in which arguments are passed.

The position of a predicate in a type determines the position in a function application (of a function with that type) where a value for the corresponding implicit parameter may be passed explicitly. For example, for f in the following fragment first we may pass a dictionary for $Eq\ a$, then we must pass two normal arguments, then (again) we may pass a dictionary, and finally (again) we must pass two normal arguments:

$$
\begin{array}{l}
\textbf{let}\ f :: \forall a.Eq\ a \Rightarrow a \to a \to \forall b.Eq\ b \Rightarrow b \to b \to (Bool, Bool) \\
\quad f = \lambda p\ q\ r\ s \to (eq\ p\ q, eq\ r\ s) \\
\textbf{in}\ f \qquad\qquad\qquad\qquad\qquad\qquad 3\ 4 \\
\qquad (!(eq = eqMod2) \leftsquigarrow Eq\ Int!)\ 5\ 6
\end{array}
$$

The value for the first implicit parameter (*Eq a*) is computed automatically, the value (an explicitly constructed dictionary record) for the second (*Eq b*) is explicitly passed by means of (! !). Inside these delimiters we specify both value and the predicate for which it is a witness. The notation ($!e \leftsquigarrow p!$) ($\leftsquigarrow$ appears in the source text as `<:`) suggests a combination of "is of type" and "is evidence for". Here "is of type" means that the dictionary *e* must be of the record type introduced by the class declaration for the predicate *p*. The phrase "is evidence for" means that the dictionary *e* is used as the proof of the existence of the implicit argument to the function *f*.

Explicitly passing a value for an implicit parameter is optional. However, if we explicitly pass a value, all preceding implicit parameters in a consecutive sequence of implicit parameters must be passed as well. In a type expression, a consecutive sequence of implicit parameters corresponds to sequence of predicate arguments delimited by other arguments. For example, if we were to pass a value to *f* for *Eq b* with the following type, we need to pass a value for *Eq a* as well:

$$f :: \forall\ a\ b.(Eq\ a, Eq\ b) \Rightarrow a \to a \to b \to b \to (Bool, Bool)$$

We can avoid this by swapping the predicates, as in:

$$f :: \forall\ a\ b.(Eq\ b, Eq\ a) \Rightarrow a \to a \to b \to b \to (Bool, Bool)$$

For this type we can pass a value explicitly for *Eq b*. We may omit a parameter for *Eq a* because dictionaries for the remaining predicates (if any) are automatically passed, just like Haskell.

The above types for *f* have to be specified explicitly. All types signatures for *f* are isomorphic, so we always can write wrapper functions for the different varieties.

Overlapping instances By explicitly providing a dictionary the default decision making by EH is overruled. This is useful in situations where no unique choice is possible, for example in the presence of overlapping instances:

$$
\begin{array}{l}
\textbf{let instance}\ dEqInt1 \leftsquigarrow Eq\ Int\ \textbf{where} \\
\qquad eq = primEqInt \\
\quad \textbf{instance}\ dEqInt2 \leftsquigarrow Eq\ Int\ \textbf{where} \\
\qquad eq = eqMod2 \\
\quad f = ... \\
\textbf{in}\ f\ (!dEqInt1 \leftsquigarrow Eq\ Int!)\ 3\ 4 \\
\qquad (!dEqInt2 \leftsquigarrow Eq\ Int!)\ 5\ 6
\end{array}
$$

The two instances for *Eq Int* overlap, but we still can refer to each associated dictionary individually, because of the names *dEqInt1* and *dEqInt2* that were given to the dictionaries. Thus overlapping instances can be avoided by letting the programmer decide which dictionaries to pass to the call *f* 3 4 5 6.

Overlapping instances can also be avoided by not introducing them in the first place. However, this conflicts with our goal of allowing the programmer to use different instances at different places in a program. This problem can be overcome by excluding instances participating in the predicate proving machinery by:

instance *dEqInt2* :: *Eq Int* **where**
  $eq = \lambda\_\ \_ \rightarrow False$

The naming of a dictionary by means of $\leftsquigarrow$ actually does two things. It binds the name to the dictionary and it specifies to use this dictionary as the default instance for *Eq Int* for use in its proof process. The notation :: only binds the name but does not introduce it into proving predicates. If one at a later point wants to introduce the dictionary nevertheless, possibly overriding an earlier choice, this may done by specifying:

instance *dEqInt2* $\leftsquigarrow$ *Eq Int*

Local instances We allow instances to be declared locally, within the scope of other program variables. A local instance declaration shadows an instance declaration introduced at an outer level:

- If their names are equal, the innermost shadows the outermost.
- In case of having overlapping instances available during the proof of predicates arising inside the **let** expression, the innermost instance takes precedence over the outermost.

This mechanism allows the programmer to fully specify which instances are active at any point in the program text:

let instance *dEqInt1* $\leftsquigarrow$ *Eq Int* **where** ...
  instance *dEqInt2* :: *Eq Int* **where** ...
  $g = \lambda x\ y \rightarrow eq\ x\ y$
in **let** $v_1 = g\ 3\ 4$
    v_2 = **let instance** *dEqInt2* $\leftsquigarrow$ *Eq Int*
      in $g\ 3\ 4$
in ...

The value for v_1 is computed with *dEqInt1* as evidence for *Eq Int*, whereas v_2 is computed with *dEqInt2* as evidence.

In our discussion we will come back to local instances.

Higher order predicates revisited As we mentioned earlier, the declaration of an instance with a context actually introduces a function taking dictionaries as arguments:

let instance *dEqInt* $\leftsquigarrow$ *Eq Int* **where**
    $eq = primEqInt$

```
    instance dEqList ⇜ Eq a ⇒ Eq (List a) where
      eq = ...
    f :: ∀ a.Eq a ⇒ a → List a → Bool
    f = λp q → eq (Cons p Nil) q
in f 3 (Cons 4 Nil)
```

In terms of predicates the instance declaration states that given a proof for the context *Eq a*, the predicate *Eq* (*List a*) can be proven. In terms of values this translates to a function which takes the evidence of the proof of *Eq a*, a dictionary record ($eq :: a \rightarrow a \rightarrow Bool$), to evidence for the proof of *Eq* (*List a*) [21]:

```
dEqInt  :: (eq :: Int → Int → Bool)
dEqList :: ∀ a.(eq :: a → a → Bool)
                → (eq :: List a → List a → Bool)
eq      = λdEq x y → dEq.eq x y
```

With these values, the body of f is mapped to:

```
f = λdEq_a p q → eq (dEqList dEq_a) (Cons p Nil) q
```

This translation can now be expressed explicitly as well; a dictionary for *Eq* (*List a*) is explicitly constructed and passed to *eq*:

```
f :: ∀ a.Eq a ⇒ a → List a → Bool
f = λ(!dEq_a ⇜ Eq a!)
          → λp q → eq (!dEqList dEq_a ⇜ Eq (List a)!)
                      (Cons p Nil) q
```

The type variable a is introduced as a lexically scoped type variable [36], available for further use in the body of f.

The notation *Eq a* ⇒ *Eq* (*List a*) in the instance declaration for *Eq* (*List a*) introduces both a predicate transformation for a predicate (from *Eq a* to *Eq* (*List a*)), to be used for proving predicates, as well as a corresponding dictionary transformer function. Such transformers can also be made explicit in the following variant:

```
f :: (∀a.Eq a ⇒ Eq (List a)) ⇒ Int → List Int → Bool
f = λ(!dEq_La ⇜ ∀a.Eq a ⇒ Eq (List a)!)
      → λp q → eq (!dEq_La dEqInt ⇜ Eq (List Int)!)
                  (Cons p Nil) q
```

Instead of using *dEqList* by default, an explicitly specified implicit predicate transformer, bound to *dEq_La* is used in the body of f to supply *eq* with a dictionary for *Eq* (*List Int*). This dictionary is explicitly constructed and passed to *eq*; both the construction and binding to *dEq_La* may be omitted. We must either pass a dictionary for *Eq a* ⇒ *Eq* (*List a*) to f ourselves explicitly or let it happen automatically; here in both cases *dEqList* is the only choice possible.

4 Partial type signatures

Explicitly specifying complete type signatures can be a burden to the programmer, especially when types become large and only a specific part of the type needs to be specified explicitly. EH therefore allows partial type signatures. We will show its use based on the function:

$$f = \lambda p\ q\ r\ s \rightarrow (eq\ p\ q, eq\ r\ s)$$

for which we infer the following type if no specification of its type is given:

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

Variation 1: Now, if we want to make clear that the dictionary for b should be passed before any of the a's we write:

$$\begin{array}{l} f :: \forall\quad b.(Eq\ b,\ ...\quad) \Rightarrow ... \rightarrow ... \rightarrow b \rightarrow b \rightarrow ... \\ \quad \text{-- INFERRED:} \\ f :: \forall\ a\ b.(Eq\ b, Eq\ a) \Rightarrow a\ \rightarrow a\ \rightarrow b \rightarrow b \rightarrow (Bool, Bool) \end{array}$$

The parts indicated by '...' are inferred.

Variation 2: The dots '...' in the type signature specify parts of the signature to be filled by the type inferencer. The inferred type may be polymorphic if no restrictions on its type are found by the type inferencer, or it may be monomorphic as for $r :: Int$ in:

$$\begin{array}{l} f :: \forall\ a.(Eq\ a,\ ...\ ) \Rightarrow a \rightarrow a \rightarrow\ ... \\ f = \lambda\qquad\qquad\quad p\quad q\quad r\quad\ s\quad\ \rightarrow (eq\ p\ q, eq\ r\ 3) \\ \quad \text{-- INFERRED:} \\ f :: \forall\ a.\ Eq\ a\qquad \Rightarrow a \rightarrow a \rightarrow Int \rightarrow \forall\ b.b \rightarrow (Bool\ \ , Bool\ \ ) \end{array}$$

Variation 3: If instead we still want s to have the same type as r we can use a more general variant of '...' in which we can refer to the inferred type using a type variable prefixed with a percent symbol '%', called a *named wildcard*:

$$\begin{array}{l} f :: \forall\ a.(Eq\ a,\ ...\ ) \Rightarrow a \rightarrow a \rightarrow\ \%\ b \rightarrow\ \%\ b \rightarrow\ ... \\ f = \lambda\qquad\qquad\quad p\quad q\quad r\quad\ s\quad\ \rightarrow (eq\ p\ q, eq\ r\ 3) \\ \quad \text{-- INFERRED:} \\ f :: \forall\ a.\ Eq\ a\qquad \Rightarrow a \rightarrow a \rightarrow Int\ \ \rightarrow Int\ \ \rightarrow (Bool\ \ , Bool\ \ ) \end{array}$$

For the remainder of this paper we mainly use '...', called a *type wildcard*, or *predicate wildcard* in predicate positions. Although the given example suggests that a wildcard may be used anywhere in a type, there are some restrictions:

- A named wildcard *%a* cannot be used as a predicate wildcard, because *%a* then would refer to a set of predicates; it does not make much sense to pass this set twice.

- A type wildcard can occur at an argument or result position of a function type. A type wildcard itself may bind to a polymorphic type with predicates. In other words, impredicativeness is allowed. This is particularly convenient for type wildcards on a function's result position. For example, the type wildcard $\%b$ in

  $$f :: \forall\ a.Eq\ a \Rightarrow a \rightarrow a \rightarrow \%b$$

  is bound to

  $$\forall\ b.Eq\ b \Rightarrow b \rightarrow b \rightarrow (Bool, Bool)$$

  after further type inferencing.

- For the non wildcard part of a type signature all occurrences of a type variable in the final type must be given. This is necessary because the type signature will be quantified over explicitly introduced type variables.

- A sequence of explicit predicates may end with a predicate wildcard, standing for an optional collection of additional predicates. Multiple occurrences of a predicate wildcard or between explicit predicates would defeat the purpose of being partially explicit. For example, for the type signature $(Eq\ b, ..., Eq\ c) \Rightarrow ...$ the argument position of $Eq\ c$'s dictionary cannot be predicted by the programmer.

- The absence of a predicate wildcard in front of a type means *no* predicates are allowed. The only exception to this rule is a single type variable or a type wildcard, since these may be bound to a type which itself contains predicates.

5 Implementation

Because of space limitations we focus on the distinguishing characteristics of our implementation in the EH compiler [4, 7, 6].

The type system is given in Fig. 3 which describes the relationship between types in the type language in Fig. 2. Our σ types allow for the specification of the usual base types ($Int, Char$) and type variables (v) as well aggregrate types like normal abstraction ($\sigma \rightarrow \sigma$), implicit abstraction ($\pi \Rightarrow \sigma$), (higher ranked) universal quantification ($\forall\alpha.\sigma$), predicates (π) and their transformations ($\pi \Rightarrow \pi$). Translations ϑ represent code resulting from the transformation from implicit parameter passing to explicit parameter passing. An environment Γ binds value identifiers to types ($\iota \mapsto \sigma$). Instance declarations result in bindings of predicates to translations (dictionary evidence) paired with their type ($\pi \rightsquigarrow \vartheta : \sigma$) whereas class declarations bind a predicate to its dictionary type ($\pi \rightsquigarrow \sigma$):

$$\begin{aligned} bind &= \iota \mapsto \sigma \mid \pi \rightsquigarrow \vartheta : \sigma \mid \pi \rightsquigarrow \sigma \\ \Gamma &= \overline{bind} \end{aligned}$$

We use vector notation for any ordered collection, denoted with a horizontal bar on top. Concatenation of vectors and pattern matching on a vector is denoted by a comma ',.

$$\boxed{\Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta}$$

$$\frac{}{\Gamma \overset{expr}{\vdash} int : Int \rightsquigarrow int}\ \textsc{E-INT}_{Ev}$$

$$\frac{(\iota \mapsto \sigma_\iota) \in \Gamma}{\Gamma \overset{expr}{\vdash} \iota : \rightsquigarrow \iota}\ \textsc{E-ID}_{Ev}$$

$$\frac{\Gamma \overset{expr}{\vdash} e_2 : \sigma_a \rightsquigarrow \vartheta_2 \qquad \Gamma \overset{expr}{\vdash} e_1 : \sigma_a \to \sigma \rightsquigarrow \vartheta_1}{\Gamma \overset{expr}{\vdash} e_1\ e_2 : \sigma \rightsquigarrow \vartheta_1\ \vartheta_2}\ \textsc{E-APP}_{Ev}$$

$$\frac{i \mapsto \sigma_i, \Gamma \overset{expr}{\vdash} e : \sigma_e \rightsquigarrow \vartheta_e}{\Gamma \overset{expr}{\vdash} \lambda i \to e : \sigma_i \to \sigma_e \rightsquigarrow \lambda i \to \vartheta_e}\ \textsc{E-LAM}_{Ev}$$

$$\frac{i \mapsto \sigma_i, \Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \qquad i \mapsto \sigma_i, \Gamma \overset{expr}{\vdash} e_i : \rightsquigarrow \vartheta_i}{\Gamma \overset{expr}{\vdash} \textbf{let}\ i :: \sigma_i; i = e_i\ \textbf{in}\ e : \sigma \rightsquigarrow \textbf{let}\ i = \vartheta_i\ \textbf{in}\ \vartheta_e}\ \textsc{E-LET-TYSIG}_{Ev}$$

$$\frac{\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta_\pi : \sigma_\pi \qquad \Gamma \overset{expr}{\vdash} e : \pi \Rightarrow \sigma \rightsquigarrow \vartheta_e}{\Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \vartheta_\pi}\ \textsc{E-PRED}_{Ev}$$

Figure 3: Type rules for expressions

Basic typing rules Type rules in Fig. 3 read like this: given contextual information Γ it can be proven ($\vdash$) that term e has (:) type σ and some additional ($\leadsto$) results, which in our case is the code ϑ in which passing of all parameters has been made explicit. Later type rules will incorporate more properties; all separated by a semicolon ';'. If some property does not matter or is not used, an underscore '_' is used to indicate this. Rules are labeled with names of the form $x - variant_{version}$ in which x is a single character indicating the syntactic element, *variant* its variant and *version* a particular version of the type rule which also corresponds to a compiler version in the implementation. In this paper only versions *Ev*, *EvK* and *I* are used, respectively addressing evidence translation, use of expected types and the handling of implicit parameters. We have only included the most relevant type rules and have omitted those dealing with the introduction of classes and instances; these are all standard [9].

The conciseness of the rules suggests that its implementation should not pose much of a problem, but the opposite is true. Unfortunately, in their current form the rules do not fully specify how to combine them in order to build a complete proof tree, and hence are not algorithmic [38]. This is especially true for the last rule E-PRED, since its use is not associated with a syntactic construct of the source language. Algorithmic variants of the rules have two pleasant properties:

- The syntax tree determines how to combine the rules.
- By distributing data over a larger set of variables an order in which to compute them becomes apparent.

The first property is taken care of by the parser, and based on the second property we can implement rules straightforwardly using an attribute grammar, mapping variables in rules to attributes. Our situation is complicated due to a combination of several factors:

- The structure of the source language cannot be used to determine where rule E-PRED should be applied: the term e in the premise and the conclusion are the same. Furthermore, the predicate π is not mentioned in the conclusion so discovering whether this rule should be applied depends completely on the typing rule. Thus the necessity to pass an implicit parameter may spontaneously pop up in any expression.
- In the presence of type inferencing nothing may be known yet about e at all, let alone which implicit parameters it may take. This information usually only becomes available after the generalization of the inferred types.
- These problems are usually circumvented by limiting the type language for types that are used during inferencing to predicate-free types. By effectively stripping a type from both its predicates and quantifiers standard Hindley-Milner (HM) type inferencing becomes possible. However, we allow predicated as well as quantified types to participate in type inferencing. As a consequence, predicates as well as quantifiers can be present in any type encountered during type inferencing.

Implicitness made explicit So, the bad news is that we do not know where implicit parameters need to be passed; the good news is that if we represent this lack of knowledge explicitly we can still figure out if and where implicit parameters need to be passed. This is not a new idea, because type variables are usually used to refer to a particular type about which nothing is yet known. The general strategy is to represent an indirection in time by the introduction of a free variable. In a later stage of a type inferencing algorithm such type variables are then replaced by more accurate knowledge, if any. Throughout the remainder of this section we work towards algorithmic versions of the type rules in which the solution to equations between types are computed by means of

- the use of variables representing unkown information
- the use of constraints on type variables representing found information

In our approach we also employ the notion of variables for sets of predicates, called *predicate wildcard variable*s, representing a yet unknown collection of implicit parameters, or, more accurately their corresponding predicates. These predicate wildcard variables are used in a type inferencing/checking algorithm which explicitly deals with expected (or known) types σ^k, as well as extra inferred type information.

Notation	Meaning
σ	type
σ^k	expected/known type
v	type variable
ι	identifier
i	value identifier
I	(type) constructor identifier, type constant
Γ	assumptions, environment, context
C	constraints, substitution
$C_{k..l}$	constraint composition of $C_k ... C_l$
$\leqslant$	subsumption, "fits in" relation
ϑ	translated code
π	predicate
ϖ	predicate wildcard (collection of predicates)

Figure 4: Legenda of type related notation

Fig. 5 provides a summary of the judgement forms we use. The presence of properties in judgements varies with the version of typing rules. Both the most complex and its simpler versions are included.

These key aspects are expressed in the adapted rule for predicates shown in Fig. 6. This rule makes two things explicit:

- The context provides the expected (or known) type σ^k of e. Jointly operating, all our rules maintain the invariant that e will get assigned a type σ which is a

Version	Judgement	Read as
I	$\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow C; \vartheta$	With assumptions Γ, expected type σ^k, expression e has type σ and translation ϑ (with dictionary passing made explicit), requiring additional constraints C.
EvK	$\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta$	version for evidence + expected type only
Ev	$\Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta$	version for evidence only
I	$\Gamma \overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma \rightsquigarrow C; \delta$	σ^l is subsumed by σ^r, requiring additional constraints C. C is applied to σ^r returned as σ. Proving predicates (using Γ) may be required resulting in coercion δ.
EvK	$\overset{fit}{\vdash} \sigma^l \leqslant \sigma^r : \sigma$	version for evidence + expected type only
I	$\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta : \sigma$	Prove π, yielding evidence ϑ and evidence type σ.
I	$\sigma^k \overset{pat}{\vdash} p : \sigma; \Gamma_p \rightsquigarrow C$	Pattern has type σ and variable bindings Γ_p.

Figure 5: Legenda of judgement forms for each version

$$\boxed{\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta}$$

$$\frac{\overset{fit}{\vdash} \sigma_\iota \leqslant \sigma^k : \sigma \qquad (\iota \mapsto \sigma_\iota) \in \Gamma}{\Gamma; \sigma^k \overset{expr}{\vdash} \iota : \sigma \rightsquigarrow \iota}\ \text{E-ID}_{EvK}$$

$$\frac{\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta_\pi : \sigma_\pi \qquad \Gamma; \varpi \Rightarrow \sigma^k \overset{expr}{\vdash} e : \pi \Rightarrow \sigma \rightsquigarrow \vartheta_e}{\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \vartheta_\pi}\ \text{E-PRED}_{EvK}$$

Figure 6: Implicit parameter passing with expected type

subtype of σ^k, denoted by $\sigma \leqslant \sigma^k$ (σ is said to be subsumed by σ^k), enforced by a *fit* judgement (see Fig. 5 for the form of the more complex variant used later in this paper). The *fit* judgement also yields a type σ, the result of the subsumption. This type is required because the known type σ^k may only be partially known, and additional type information is to be found in σ.

- An implicit parameter can be passed anywhere; this is made explicit by stating that the known type of e may start with a sequence of implicit parameters. This is expressed by letting the expected type in the premise be $\varpi \to \sigma^k$. In this way we require the type of e to have the form $\varpi \to \sigma^k$ and also assign an identifier ϖ to the implicit part.

A predicate wildcard variable makes explicit that we can expect a (possibly empty) sequence of implicit parameters and at the same time gives an identity to this sequence. The type language for predicates thus is extended with a predicate wildcard variable ϖ, corresponding to the dots '...' in the source language for predicates:

$$\begin{aligned} \pi ::= & \; I\ \overline{\sigma} \\ \mid & \; \pi \Rightarrow \pi \\ \mid & \; \varpi \end{aligned}$$

In algorithmic terms, the expected type σ^k travels top-to-bottom in the abstract syntax tree and is used for type checking, whereas σ travels bottom-to-top and holds the inferred type. If a fully specified expected type σ^k is passed downwards, σ will turn out to be equal to this type. If a partially specified type is passed downwards the unspecified parts may be filled in by the type inferencer.

The adapted typing rule E-PRED in Fig. 6 still is not much of a help as to deciding when it should be applied. However, as we only have to deal with a limited number of language constructs, we can use case analysis on the source language constructs. In this paper we only deal with function application, for which the relevant rules are shown in their full glory in Fig. 8 and will be explained soon. The rules in Fig. 8 look complex. The reader should realize that the implementation is described using an attribute grammar system [7, 2] which allows the independent specification of all aspects which now appear together in a condensed form in Fig. 8. The tradeoff is between compact but complex type rules and more lengthy but more understandable attribute grammar notation.

Notation The typing rules in Fig. 7 and Fig. 8 are directed towards an implementation; additional information flows through the rules to provide extra contextual information. Also, the rule is more explicit in its handling of constraints computed by the rule labeled *fit* for the subsumption $\leqslant$; a standard substitution mechanism constraining the different variable variants is used for this purpose:

$$\begin{aligned} bindv &= v \mapsto \sigma \mid \varpi \mapsto \pi, \varpi \mid \varpi \mapsto \emptyset \\ C &= \overline{bindv} \end{aligned}$$

The mapping from type variables to types $v \mapsto \sigma$ constitutes the usual substitution for type variables. The remaining alternatives map a predicate wildcard variable to a possibly empty list of predicates.

Not all judgement forms used in Fig. 7 and Fig. 8 are included in this paper; in the introduction we indicated we focus here on that part of the implementation in which explicit parameter passing makes a difference relative to the standard [9, 38, 21]. Fig. 5 provides a summary of the judgement forms we use.

The judgement **pred** (Fig. 5) for proving predicates is standard with respect to context reduction and the discharge of predicates [9, 21, 25], except for the scoping mechanism introduced. We only note that the proof machinery must now take the scoped availability of instances into account and can no longer assume their global existence.

$$\boxed{\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow C; \vartheta}$$

$$\frac{\begin{array}{c}\Gamma; \sigma_a \overset{expr}{\vdash} e_2 : \_ \rightsquigarrow C_2; \vartheta_2 \\ \_ \overset{fit}{\vdash} \pi_d \Rightarrow \sigma_d \leqslant \pi_a \Rightarrow v : \_ \Rightarrow \sigma_a \rightsquigarrow \_; \_ \\ \pi_d \rightsquigarrow \sigma_d \in \Gamma \\ \Gamma; \pi_2 \Rightarrow \sigma^k \overset{expr}{\vdash} e_1 : \pi_a \Rightarrow \sigma \rightsquigarrow C_1; \vartheta_1 \\ v \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} e_1\ (!e_2 \leftsquigarrow \pi_2!) : C_2\sigma \rightsquigarrow C_{2..1}; \vartheta_1\ \vartheta_2}\ \text{E-IAPP}_I$$

$$\frac{\begin{array}{c}[\pi_a \rightsquigarrow p : \sigma_a], \Gamma^p, \Gamma; \sigma_r \overset{expr}{\vdash} e : \sigma_e \rightsquigarrow C_3; \vartheta_e \\ \sigma_a \overset{pat}{\vdash} p : \_; \Gamma^p \rightsquigarrow C_2 \\ \_ \overset{fit}{\vdash} \pi_d \Rightarrow \sigma_d \leqslant \pi_a \Rightarrow v_2 : \_ \Rightarrow \sigma_a \rightsquigarrow \_; \_ \\ \pi_d \rightsquigarrow \sigma_d \in \Gamma \\ \Gamma \overset{fit}{\vdash} \pi \Rightarrow v_1 \leqslant \sigma^k : \pi_a \Rightarrow \sigma_r \rightsquigarrow C_1; \_ \\ v_1, v_2 \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} \lambda(!p \leftsquigarrow \pi!) \rightarrow e : C_{3..2}\pi_a \Rightarrow \sigma_e \rightsquigarrow C_{3..1}; \lambda p \rightarrow \vartheta_e}\ \text{E-ILAM}_I$$

Figure 7: Type rules for explicit implicit parameters

Explicit parameter passing The rules in Fig. 7 specify the typing for the explicit parameter passing where an implicit parameter is expected. The rules are similar to those for normal parameter passing; the difference lies in the use of the predicate. For example, when reading through the premises of rule E-IAPP, the function e_1 is typed in a context where it is expected to have type $\pi_2 \rightarrow \sigma^k$. We then require a class definition for

the actual predicate π_a of the function type to exist, which we allow to be instantiated using the *fit* judgement which matches the class predicate π_d with π_a and returns the dictionary type in σ_a. This dictionary type σ_a is the expected type of the argument.

Because we are explicit in the predicate for which we provide a dictionary value, we need not use any proving machinery. We only need the predicate to be defined so we can use its corresponding dictionary type for further type checking.

The rule E-ILAM for λ-abstractions follows a similar strategy. The type of the λ-expression is required to have the form of a function taking an implicit parameter. This *fit* judgement states this, yielding a predicate π_a which via the corresponding class definition gives the dictionary type σ_a. The pattern is expected to have this type σ_a. Furthermore, the body e of the λ-expression may use the dictionary (as an instance) for proving other predicates so the environment Γ for e is extended with a binding for the predicate and its dictionary p.

$$\boxed{\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow C; \vartheta}$$

$$\frac{\begin{array}{c}\overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma \overset{pred}{\vdash} C_3\overline{\pi_a} \rightsquigarrow \overline{\vartheta_a} : \_ \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma; \sigma_a \overset{expr}{\vdash} e_2 : \_ \rightsquigarrow C_3; \vartheta_2 \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma; \varpi \Rightarrow v \rightarrow \sigma_r^k \overset{expr}{\vdash} e_1 : \overline{\pi_a} \Rightarrow \sigma_a \rightarrow \sigma \rightsquigarrow C_2; \vartheta_1 \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k} \equiv inst_\pi(\overline{\pi_a^k}) \\ \Gamma \overset{fit}{\vdash} \varpi^k \Rightarrow v^k \leqslant \sigma^k : \overline{\pi_a^k} \Rightarrow \sigma_r^k \rightsquigarrow C_1; \_ \\ \varpi, \varpi^k, v^k, v \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} e_1\ e_2 : C_3\sigma \rightsquigarrow C_{3..1}; \lambda\overline{\vartheta_i^k} \rightarrow \vartheta_1\ \overline{\vartheta_a}\ \vartheta_2}\ \text{E-APP}_I$$

$$\frac{\begin{array}{c}\overline{\pi_i^p \rightsquigarrow \vartheta_i^p}, \Gamma_p, \Gamma; \sigma_r \overset{expr}{\vdash} e : \sigma_e \rightsquigarrow C_3; \vartheta_e \\ \overline{\pi_i^p \rightsquigarrow \vartheta_i^p} \equiv inst_\pi(\overline{\pi_a}) \\ \sigma_p \overset{pat}{\vdash} p : \_; \Gamma_p \rightsquigarrow C_2 \\ \Gamma \overset{fit}{\vdash} \varpi \Rightarrow v_1 \rightarrow v_2 \leqslant \sigma^k : \overline{\pi_a} \Rightarrow \sigma_p \rightarrow \sigma_r \rightsquigarrow C_1; \_ \\ \varpi, v_i \text{ fresh}\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} \lambda p \rightarrow e : C_{3..2}\overline{\pi_a} \Rightarrow C_3\sigma_p \rightarrow \sigma_e \rightsquigarrow C_3; \lambda\overline{\vartheta_i^p} \rightarrow \lambda p \rightarrow \vartheta_e}\ \text{E-LAM}_I$$

Figure 8: Implicit parameter type rules

Implicit parameter passing: application From bottom to top, rule E-APP in Fig. 8 reads as follows (to keep matters simple we do not mention the handling of constraints C). The result of the application is expected to be of type σ^k, which in general will

have the structure $\varpi^k \to v^k$. This structure is enforced and checked by the subsumption check described by the rule *fit*; the rule binds ϖ^k and v^k to the matching parts of σ^k similar to pattern matching. We will not look into the *fit* rules for $\leqslant$; for this discussion it is only relevant to know that if a ϖ cannot be matched to a predicate it will be constrained to $\varpi \mapsto \emptyset$. In other words, we start with assuming that implicit parameters may occur everywhere and subsequently we try to prove the contrary. The subsumption check $\leqslant$ gives a possible empty sequence of predicates $\overline{\pi_a^k}$ and the result type σ_r^k. The result type is used to construct the expected type $\varpi \to v \to \sigma_r^k$ for e_1. The application $e_1\ e_2$ is expected to return a function which can be passed evidence for $\overline{\pi_a^k}$. We create fresh identifiers $\overline{\vartheta_i^k}$ and bind them to these predicates. Function $inst_\pi$ provides these names bound to the instantiated variants $\overline{\pi_i^k}$ of $\overline{\pi_a^k}$. The names $\overline{\vartheta_i^k}$ are used in the translation, which is a lambda expression accepting $\overline{\pi_a^k}$. The binding $\overline{\pi_i^k \rightsquigarrow \vartheta_i^k}$ is used to extend the type checking environment Γ for e_1 and e_2 which both are allowed to use these predicates in any predicate proving taking place in these expressions. The judgement for e_1 will give us a type $\overline{\pi_a} \to \sigma_a \to \sigma$, of which σ_a is used as the expected type for e_2. The predicates $\overline{\pi_a}$ need to be proven and evidence to be computed; the top judgement **pred** takes care of this. Finally, all the translations together with the computed evidence forming the actual implicit parameters $\overline{\pi_a}$ are used to compute a translation for the application, which accepts the implicit parameters it is supposed to accept. The body $\vartheta_1 \overline{\vartheta_a} \vartheta_2$ of this lambda expression contains the actual application itself, with the implicit parameters are passed before the argument.

Even though the rule for implicitly passing an implicit parameter already provides a fair amount of detail, some issues remain hidden. For example, the typing judgement for e_1 gives a set of predicates π_a for which the corresponding evidence is passed by implicit arguments. The rule suggests that this information is readily available in an actual implementation of the rule. However, assuming e_1 is a **let** bound function for which the type is currently being inferred, this information will only become available when the bindings in a **let** expression are generalized [23], higher in the corresponding abstract syntax tree. Only then the presence and positioning of predicates in the type of e_1 can be determined. This complicates the implementation because this information has to be redistributed over the abstract syntax tree.

Implicit parameter passing: λ-abstraction Rule E-LAM for lambda expressions from Fig. 8 follows a similar strategy. At the bottom of the list of premises we start with an expected type σ^k which by definition has to accept a normal parameter and a sequence of implicit parameters. This is enforced by the judgement *fit* which gives us back predicates $\overline{\pi_a}$ used in a similar fashion as in rule E-APP.

6 Discussion and related work

Soundness, completeness and principal types EH allows type expressions where quantifiers and predicates may be positioned anywhere in a type, and all terms can be explicitly typed with a type annotation. Thus we obtain the same expressiveness as

system-F, making the issue of soundness and completeness of our type system irrelevant. What remains relevant are the following questions:

- For a completely explicitly typed program, is our algorithm and implementation sound and complete?
- For a partially explicitly typed program, what is the characterisation of the types that can be inferred for the terms for which no type has been given?

We have not investigated these questions in the sense of proving their truth or falsehood. However, we have taken the following as our starting point:

- Stick to HM type inferencing, except for the following:
- Combine type checking and inferencing. In order to be able to do this, impredicative types are allowed to participate in HM type inferencing. This is a separate issue we deal with elsewhere [5].

By design we avoid 'breaking' HM type inferencing. However, Faxen [10] demonstrates the lack of principal types for Haskell due to a combination of language features. EH's quantified class constraints solve one of the problems mentioned by Faxen.

Our choice to allow quantifiers and predicates at any position in a type expression provides the programmer with the means to specify the type signature that is needed, but also breaks principality because the type inferencer will infer only a specific one (with quantifiers and predicates as much as possible to the right) of a set of isomorphic types. We have not investigated this further.

In general it also is an open question what can be said about principal types and other desirable properties when multiple language features are combined into a complete language. In this light we take a pragmatic approach and design starting point: if the system guesses wrong, the programmer can repair it by adding extra (type) information.

Local instances Haskell only allows global instances because the presence of local instances results in the loss of principal types for HM type inference [43]:

```
let class Eq a where eq :: a → a → Bool
    instance Eq Int where
    instance Eq Char where
in  eq
```

With HM the problem arises because *eq* is instantiated without being applied to an argument, hence no choice can be made at which type *Eq a* (arising from *eq*) should be instantiated at. In EH, we circumvent this problem by delaying the instantiation of *eq*'s type until it is necessary, for example when the value is used as part of an application to an argument [6, 5].

Coherence is not a problem either because we do not allow overlapping instances. Although local instances may overlap with global instances, their use in the proving machinerey is dictated by their nesting structure, which is static: local instances take priority over global instances.

How much explicitness is needed Being explicit by means of the (! ... !) language construct very soon becomes cumbersome because our current implementation requires full specification of all predicates involved inside (! ... !). Can we do with less?

- Rule E-IAPP from Fig. 7 uses the predicate π_2 in ($!e_2 \leftsquigarrow \pi_2!$) directly, that is, without any predicate proving, to obtain π_d and its corresponding dictionary type σ_d. Alternatively we could interpret ($!e_2 \leftsquigarrow \pi_2!$) as an addition of π_2 to the set of predicates used by the predicate proving machinery for finding a predicate whose dictionary matches the type of e_2. However, if insufficient type information is known about e_2 more than one solution may be found. Even if the type of e_2 would be fully known, its type could be coerced in dropping record fields so as to match different dictionary types.
- We could drop the requirement to specify a predicate and write just ($!e_2!$) instead of ($!e_2 \leftsquigarrow \pi_2!$). In this case we need a mechanism to find a predicate for the type of the evidence provided by e_2. This is most likely to succeed in the case of a class system as the functions introduced by a class need to have globally unique names. For other types of predicates like those for dynamically scoped values this is less clear. By dropping the predicate in ($!e_2!$) we also loose our advocated advantage of explicitness because we can no longer specify type related information.
- The syntax rule E-ILAM requires a predicate π in its implicit argument ($!p \leftsquigarrow \pi!$). It is sufficient to either specify a predicate for this form of a lambda expression or to specify a predicate in a corresponding type annotation.

Whichever of these routes leads to the most useful solution for the programmer, if the need arises our solution always gives the programmer the full power of being explicit in what is required.

Binding time of instances One other topic deserves attention, especially since it deviates from the standard semantics of Haskell. We allow the re-use of dictionaries by means of record extension. Is the other way around allowed as well: can previously defined functions of a dictionary use newly added values? In a variation of the example for *nub*, the following invocation of *nub* is parameterized with an updated record; a new definition for *eq* is provided:

```
nub (!(dEqInt | eq := eqMod2) ⇜ Eq Int!)
    (Cons 3 (Cons 3 (Cons 4 Nil)))
```

In our implementation *Eq*'s function *ne* invokes *eq*, the one provided by means of the explicit parameterization, thus allowing open recursion. This corresponds to a late binding, much in the style employed by object oriented languages. This is a choice out of (at least) three equally expressive alternatives:

- Our current solution, late binding as described. The consequence is that all class functions now take an additional (implicit) parameter, namely the dictionary where this dictionary function has been retrieved from.

- Haskell's solution, where we bind all functions at instance creation time. In our *nub* example this means that *ne* still uses *dEqInt*'s *eq* instead of the *eq* provided in the updated (*dEqInt* | *eq* := ...).
- A combination of these solutions, such as using late binding for default definitions, and Haskell's binding for instances.

Again, whichever of the solutions is preferred as the default case, especially in the light of the absence of open recursion in Haskell, we notice that the programmer has all the means available to express his differing intentions.

Dynamically scoped variables GHC [1] enables the passing of plain values as dynamically scoped variables (also known as implicit parameters). It is possible to model this effect [19, 34, 1] with the concepts described thus far. For example, the following program uses dynamically scoped variable ?*x*:

```
let f  :: (?x :: Int) ⇒ ...
    f  = λ              ... → ... ?x + 2 ...
    ?x = 3
in f ...
```

The signature of *f* specifies a predicate ?*x* :: *Int*, meaning that *f* can refer to the dynamically scoped variable *x* with type *Int*. Its value is introduced as a binding in a **let** expression and is used in the body of *f* by means of ?*x*. This can be encoded using the class system:

```
let class Has_x a where
      value_x :: a
    f :: (Has_x Int) ⇒ ...
    f = λ              ... → ...value_x + 2 ...
    instance Has_x Int where
      value_x = 3
in f ...
```

We only mention briefly some issues with this approach:

- The type for which an instance without context is defined usually is specified explicitly. This is no longer the case for ? predicates if an explicit type signature for e.g. **let** ?*x* = 3 is omitted.
- GHC [1] inhibits dynamically scoped variable predicates in the context of instance declarations because it is unclear which scoped variable instance is to be taken. Scoping for instances as available in EHC may well obviate this restriction.
- Use of records for dictionaries can be optimized away because each class contains a single field only.

Our approach has the additional benefit that we are not obliged to rely on the proving machinery by providing a dictionary directly:

let class *Has_x a ...*
  f :: (*Has_x Int*) $\Rightarrow$ ...
  $f = \lambda$ ... $\rightarrow$ ...*value_x* + 2 ...
in f (!(*value_x* = 3) $\leftsquigarrow$ *Has_x Int*!) ...

Named instances Scheffczyk has explored named instances as well [28, 41]. Our work differs in several aspects:

- Scheffczyk partitions predicates in a type signature into ordered and unordered ones. For ordered predicates one needs to pass an explicit dictionary, unordered ones are those participating in the normal predicate proving by the system. Instances are split likewise into named and unnamed instances. Named instances are used for explicit passing and do not participate in the predicate proving. For unnamed instances this is the other way around. Our approach allows a programmer to make this partitioning explicitly, by stating which instances should participate in the proof process. In other words, the policy of how to use the implicit parameter passing mechanism is made by the programmer.
- Named instances and modules populate the same name space, separate from the name space occupied by normal values. This is used to implement functors as available in ML [32, 33] and as described by Jones [22] for Haskell. Our approach is solely based on normal values already available.
- Our syntax is less concise than the syntax used by Scheffczyk. This is probably difficult to repair because of the additional notation required to lift normal values to the evidence domain.

Implementation The type inferencing/checking algorithm employed in this paper is described in greater detail in [7, 6] and its implementation is publicly available [4], where it is part of a work in progress. Similar strategies for coping with the combination of inferencing and checking are described by Pierce [39] and Peyton Jones [37].

7 Conclusion

Allowing explicit parameterization for implicit parameters gives the programmer an additional mechanism for reusing existing functions. It also makes explicit what otherwise remains hidden inside the bowels of a compiler. We feel that this a 'good thing': it should be possible to override automatically made decisions.

We have implemented all features described in this paper in the context of a compiler for EH [6, 7, 5]; in this paper we have presented the relevant part concerning explicit implicit parameters in an as compact form as possible. To our knowledge our

implementation is the first combining language features like higher ranked types, existentials, class system, explicit implicit parameters and extensible records into one package together with a description of the implementation. We feel that this has only been possible thanks to the use of an attribute grammar system which allows us to independently describe all the separate aspects.

On a metalevel one can observe that the typing rules incorporate many details, up to a point where their simplicity may easily get lost. A typing rule serves well as a specification of the semantics of a language construct, but as soon as a typing rule evolves towards an algorithmic variant it may well turn out that other ways of describing, in particular attribute grammars, are a better vehicle for expressing implementation aspects.

References

[1] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.

[2] Arthur Baars. Attribute Grammar System. `http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem`, 2004.

[3] Arthur Baars and S. Doaitse Swierstra. Syntax Macros (Unfinished draft). `http://www.cs.uu.nl/people/arthurb/macros.html`, 2002.

[4] Atze Dijkstra. EHC Web. `http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

[5] Atze Dijkstra. *Haskell, Step by Step (to be published)*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

[6] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar (Part I). Technical Report UU-CS-2004-037, Department of Computer Science, Utrecht University, 2004.

[7] Atze Dijkstra and S. Doaitse Swierstra. Typing Haskell with an Attribute Grammar (to be published). In *Advanced Functional Programming Summerschool*, LNCS. Springer-Verlag, 2004.

[8] Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming Type Rules (submitted to POPL06), 2005.

[9] Karl-Filip Faxen. A Static Semantics for Haskell. *Journal of Functional Programming*, 12(4):295, 2002.

[10] Karl-Filip Faxen. Haskell and Principal Types. In *Haskell Workshop*, pages 88–97, 2003.

[11] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Languages and Programming Group, Department of Computer Science, Nottingham, November 1996.

[12] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[13] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, March 1996.

[14] Bastiaan Heeren and Jurriaan Hage. Type Class Directives. In *Seventh International Symposium on Practical Aspects of Declarative Languages*, pages 253 – 267. Springer-Verlag, 2005.

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

[16] Ralf Hinze and Simon Peyton Jones. Derivable Type Classes. In *Haskell Workshop*, 2000.

[17] Trevor Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS TM-531, MIT, 1995.

[18] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

[19] Mark Jones. Exploring the design space for typebased implicit parameterization. Technical report, Oregon Graduate Institute, 1999.

[20] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, 1993.

[21] Mark P. Jones. *Qualified Types, Theory and Practice*. Cambridge Univ. Press, 1994.

[22] Mark P. Jones. Using Parameterized Signatures to Express Modular Structure. In *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[23] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[24] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000,*, March 2000.

[25] Mark P. Jones. Typing Haskell in Haskell. `http://www.cse.ogi.edu/ mpj/thih/`, 2000.

[26] Mark P. Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *Haskell Workshop*, number UU-CS-1999-28. Utrecht University, Institute of Information and Computing Sciences, 1999.

[27] Stefan Kaes. Parametric overloading in polymorphic programming languages . In *Proc. 2nd European Symposium on Programming*, 1988.

[28] Wolfram Kahl and Jan Scheffczyk. Named Instances for Haskell Type Classes. In *Haskell Workshop*, 2001.

[29] A.J. Kfoury and J.B. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of Second-Order lambda-Calculus. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, 1994.

[30] A.J. Kfoury and J.B. Wells. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *Principles of Programming Languages*, pages 161–174, 1999.

[31] Oleg Kiselyov and Chung-chieh Shan. Implict configuration - or, type classes reflect the value of types. In *Haskell Workshop*, 2004.

[32] Xavier Leroy. Manifest types, modules, and separate compilation. In *Principles of Programming Languages*, pages 109–122, 1994.

[33] Xavier Leroy. Applicative Functors and Fully Transparent Higher-Order Modules. In *Principles of Programming Languages*, pages 142–153, 1995.

[34] Jeffrey R. Lewis, Mark B. Shields, Erik Meijer, and John Launchbury. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 108–118, January 2000.

[35] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[36] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. In *ICFP*, 2003.

[37] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. `http://research.microsoft.com/-Users/simonpj/papers/putting/index.htm`, 2004.

[38] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[39] Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM TOPLAS*, 22(1):1–44, January 2000.

[40] J.C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, number 19 in LNCS, pages 408–425, 1974.

[41] Jan Scheffzcyk. Named Instances for Haskell Type Classes. Master's thesis, Universitat der Bundeswehr München, 2001.

[42] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity (submitted to ICFP2005), 2005.

[43] Phil Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, 1988.