Explicit implicit parameters

Atze Dijkstra
S. Doaitse Swierstra

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-059

www.cs.uu.nl

Explicit implicit parameters

Atze Dijkstra and S. Doaitse Swierstra

November 26, 2004

Abstract

In almost all languages arguments to functions are to be given explicitly in the program text. There are however a few interesting exceptions to this rule. In Haskell functions take arguments which are either passed explicitly or implicitly. An instance of the latter is the class system in Haskell where dictionaries are passed as evidence for class predicates. However, the construction as well as the passing of these dictionaries is invisible to the programmer. Unfortunately the approach taken here is that the language is a bit autistic in the sense that the programmer cannot provide any help if the built-in proof mechanism fails. In this paper we propose, in the context of Haskell, a mechanism that allows the programmer to explicitly pass implicit parameters. This extension blends well with existing resolution mechanisms for determining which implicit parameters have to be passed, since it only overrides default the behavior of such mechanisms. We also describe how this extension can be implemented for Haskell. The implementation also gives us the additional bonus of partial type signatures, liberating the programmer from the obligation to specify either full signatures or not to specify a signature at all.

1 Introduction

The Haskell class system originally introduced by both Wadler [24] and Kaes [16] offers a powerful abstraction mechanism for dealing with overloading (or ad-hoc polymorphism). The basic idea is to restrict the polymorphism of a parameter by specifying that some predicates are to be satisfied when the function is called:

```
f :: Eq a ⇒ a → a → Int
f = λ        x    y → if x ≡ y then 3 else 4
```

In this example the type signature for f specifies that any type a can be passed as an argument as long as it satisfies predicate *Eq a*. Such predicates are introduced by *class declaration*s:

```
class Eq a where
  (≡) :: a → a → Bool
  ...
```

A class declaration also introduces functions (and values) which can only be used on a type a for which the predicate *Eq a* is satisfied. A class declaration alone is not sufficient, *instance declarations* are needed to specify for which types the predicate holds. Because a class declaration also specifies the existence of its associated functions, an instance declaration is also required to provide an implementation for those functions, and hence prove their existence:

```
instance Eq Int where
  x ≡ y = ...
  ...
instance Eq Bool where
  x ≡ y = ...
  ...
```

The compiler turns these declarations into records (dictionaries) containing the functions as fields. An explicit version of this internal machinery reads:

$$\begin{aligned}&\textbf{data}\ \textit{EqDict a} = \textit{EqDict}\{\textit{eqDictEq} :: a \rightarrow a \rightarrow \textit{Bool}\}\\ &\textit{eqDictInt} \quad\ \ = \textit{EqDict} \ldots\\ &\textit{eqDictBool} \quad = \textit{EqDict} \ldots\end{aligned}$$

Inside a function the elements of the predicate's dictionaries are available, as if they were defined as global variables. A dictionary for the actual type is required as an additional argument in order to be able to invoke the type specific variant of the class functions. So the actual implementation of *f* is:

$$f = \lambda\ \textit{dEq}\ x\ y \rightarrow \textbf{if}\ (\textit{eqDictEq dEq})\ x\ y\ \textbf{then}\ 3\ \textbf{else}\ 4$$

At the call site of the function *f* the dictionary that corresponds to the actual type of the polymorphic argument must be passed. Thus in our case the expression *f True False* can be seen as an abbreviation for the semantically more complete *f eqDictBool True False*.

Haskell's class system has turned out to be theoretically sound [11] as well as flexibile enough to incorporate extensions [10, 14]. Its role in Haskell has been described in terms of an implementation [13] as well as its semantics [7, 5]. Nevertheless, some observations with respect to its design and implementation can be made.

First, to start with the issue we will deal with in this paper, the compiler determines which dictionary to pass for a predicate. This is both a blessing and a curse. A blessing because it silently solves a problem (i.e. overloading). And a curse because as a programmer we cannot easily override the choices made by the compiler.

Second, class predicates are considered somewhat special in the sense that in a type signature predicates may only occur at the beginning, preceding the rest of the type in which no predicates may occur, forming what usually is called a type scheme. In this way predicates, as seen from a programmers point of view, loose some of their first-classness. This is the price to be paid for keeping the type inferencing process decidable.

We intend to remedy these limitations in this paper by allowing

- predicates to occur anywhere in a type signature,
- manipulation of dictionaries as normal (record) values, and
- explicit specification of (parts of a) type.

Some of these problems also have been addressed by Scheffczyk [23, 17] by giving names to dictionaries for later use. In section 4 we will discuss the differences between the two approaches.

In the remainder of this paper we will first explore the use of implicit parameters further in section 2. We will then look at the implementation in section 3. Finally, we relate our work to others in section 4 and conclude in section 5.

2 Implicit parameters

The exploration of explicit implicit parameters will be presented in the context of a Haskell variant named EH (Essential Haskell) [2, 4, 3]. Meant as a platform for education and research, EH already offers advanced features like higher ranked types, existential types, partial type signatures and records[1]. Syntactic sugar has been kept to a minimum in order to ease experimentation with and understanding of the implementation.

In this section we will first give an example of an EH program using most of the features related to implicit parameters. After pointing out these features and other differences with Haskell we continue with exploring

[1]In the current implementation not yet extensible

the relevant aspects in greater depth. The following is an EH program including the standard Haskell function *nub* which removes duplicate elements from a list. Notice that a separate *nubBy* is no longer needed:

```
let data List a = Nil | Cons a (List a)                       -- (1)
    data Bool = False | True
    ¬ :: Bool → Bool
    ¬ = ...
    class Eq a where
       eq :: a → a → Bool
       ne :: a → a → Bool
    instance dEqInt ↜ Eq Int where                            -- (2)
       eq = λ_ _ → True
       ne = λx y → ¬ (eq x y)
in  let filter :: (a → Bool) → List a → List a
        filter = ...
        nub    :: Eq a ⇒ List a → List a
        nub    = λxx → case xx of
                          Nil       → Nil
                          Cons x xs → Cons x (nub (filter (ne x) xs))
    in  nub (#(dEqInt | eq:=λ_ _ → False) ↜ Eq Int#)     -- (3)
             (Cons 3 (Cons 3 (Cons 4 Nil)))
```

In general, we designed EH to be as upwards compatible as possible with Haskell and as simple as possible. By definition these design constraints are contradictory. We point out some of the differences; each item corresponds to the commented number in the example.

1. We shortly mention the absence of features like a module system, a Prelude and infix operators.
2. The notation ↜ binds an identifier, here *dEqInt1*, to the dictionary representing the instance. The record *dEqInt1* is available as a normal value.
3. An explicitly passed parameter is syntactically denoted by an expression inside (# and #). The predicate after the ↜ explicitly states the predicate for which the expression is an instance dictionary (or *evidence*). The expression in the example itself is formed by updating a field of an already existing record. Record notation is based on Jones (et. al.) proposal [15] with the simplification that tuples and records are merged into one parenthesis delimited notation. In the example $(r \mid l{:=}e)$ means update record r at field with label l with value e.

Much simpler is the following example which we take as the starting point of our discussion:

```
let data Bool = False | True
in  let class Eq a where
          eq :: a → a → Bool
        instance Eq Int where
          eq = λ_ _ → True
    in  let f = λp q r s → (eq p q, eq r s)
        in  let v = f 3 4 5 6
            in  v
```

A Haskell compiler like Hugs [1] would infer 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 would infer for f:

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

EH places predicates and quantifiers in a type signature for a function as much to the right as possible. If quantification over a type variable can take place, the quantifier is placed just before the first occurrence of the type variable in the type signature. This is also done for a predicate referring to a type variable. The idea is to instantiate a quantified type variable or pass an implicit parameter corresponding to a predicate as late as possible, where later is defined as the order in which arguments are passed. In this way polymorphism can be retained as long as possible by the type inferencer.

The type inferred for f by Hugs also shows that the set of required predicates for f is an unordered set for which Haskell does not prescribe an order. However, if implicit parameter are to be passed explicitly, the order of the predicates is important as it tells us on which argument position a value for a predicate is expected. In EH, the order of the predicates in a type signatures also specifies the order in which the corresponding implicit parameters need to be passed. If f needs to be passed implicit parameters explicitly, a type signature is required for f. The type signature explicitly states the order of implicit parameters. For example, if the dictionary corresponding to the predicate over the type of the third and fourth parameter of f needs to be passed first, as suggested by the type inferred by Hugs, its type signature has to be specfied accordingly:

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

If, for example, the dictionary for the first and second argument needs to be passed first, this is specified by swapping the two predicates:

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

Partial type signatures. Explicitly specifying type signatures can be a burden for 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. For example, if we only want to specify the aforementioned restriction of the third and fourth parameter, the following signature is sufficient:

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

Or, if instead the predicate associated with the first and second parameter needs to have a fixed position:

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

The dots "..." in the type signature specify a part of the signature for which the programmer leaves the task of finding what it should be to the type inferencer. For the remainder of this paper we will call "..." a *type wildcard* or an *implicits wildcard* if placed on a predicate position in the type. Although the given example suggests that a wildcard may be used anywhere in a type, there are some restrictions.

- A type wildcard can only occur between $\rightarrow$'s, if any, that is on argument or result positions. A type wildcard is equivalent to a type variable without an identifier, so it cannot be referred to. 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 result position. For example, the type wildcard, that is the last "..." in

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

  is bound to

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

  after further type inferencing. The implicits wildcard, that is the first "..." stands for $\geqslant 0$ predicates. In this example no predicate remains to be filled in on this implicits wildcard position.

- If the non type wildcard part of the type signature refers to a type variable, all uses of this type variable must be specified. This is necessary because the type signature will be quantified over explicitly introduced type variables in order to allow its polymorphic use.
- An implicits wildcard stands for $\geqslant$ 0 predicates, in an unspecified order. For a sequence of explicit predicates and implicits wildcards only one implicits wildcard is allowed, at the end of this sequence. Multiple occurrences of an implicits wildcard or in between explicit predicates would defeat the purpose of being partially explicit. For example, the type signature $(Eq\ b, ..., Eq\ c) \Rightarrow ...$ would make the position of *Eq c* unknown.
- The absence of an implicits wildcard in front of a type means *no* predicates are allowed. The only exception to this rule is if it concerns a single type variable since the type variable may be bound to a type which itself contains predicates.

We make use of the fact that predicate instances also stand for actual values in the implementation when explicitly passing an implicit parameter. A class declaration introduces a record type for the dictionary corresponding to the predicate introduced by the class declaration. For example, the class declaration for *Eq* introduces the record type $(eq :: a \rightarrow a \rightarrow Bool)$ (record with one field with label *eq*) as the type of the dictionary to be passed when an implicit parameter for predicate *Eq a* is required. Now, instead of letting the compiler determine which implicit parameter to pass we construct a dictionary ourselves:

```
let data Bool = False | True
in  let class Eq a where
          eq :: a → a → Bool
    in  let f :: Eq a ⇒ a → a → Eq b ⇒ b → b → (Bool, Bool)
            f = λp q r s → (eq p q, eq r s)
        in  let v = f  (#(eq = λ_ _ → False) ⇜ Eq Int#) 3 4
                       (#(eq = λ_ _ → True ) ⇜ Eq Int#) 5 6
            in  v
```

The constructed dictionary must be of the expected dictionary type. This condition is made explicit by means of ⇜ (appearing in the source text as `<:`). The notation $(\#e ⇜ p\#)$ 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 instead of the proof evidence for an implicit parameter of a function f. By default, this value is computed by the predicate proving machinery of the compiler [11].

By explicitly providing a dictionary the default choice made by the compiler is overridden. This can also be used in situations where the compiler fails to make a choice, for example in the presence of overlapping instances:

```
let data Bool = False | True
in  let class Eq a where
          eq :: a → a → Bool
        instance dEqInt1 ⇜ Eq Int where
          eq = λ_ _ → True
        instance dEqInt2 ⇜ Eq Int where
          eq = λ_ _ → False
    in  let f :: Eq a ⇒ a → a → Eq b ⇒ b → b → (Bool, Bool)
            f = λp q r s → (eq p q, eq r s)
        in  let v = f  (#dEqInt1 ⇜ Eq Int#) 3 4
                       (#dEqInt2 ⇜ Eq Int#) 5 6
            in  v
```

The dictionaries computed as a result of an instance declaration can be given a name (denoted by ⇜) for later use. In the example the overlapping instance error is avoided by letting the programmer instead of the compiler make the choice by specifying explicitly which dictionaries to pass to the call f 3 4 5 6.

Overlapping instances of course can also be avoided by not introducing those overlapping instances 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 letting only one instance participate in the predicate proving machinery of the compiler and inhibit participation for the remaining instances:

```
instance dEqInt2 :: Eq Int where
   eq = λ_ _ → False
```

Instead of introducing the named dictionary via $\leftsquigarrow$, :: is used. The naming of a dictionary by means of $\leftsquigarrow$ actually does two things. It binds the name to the dictionary and it tells the compiler to use this dictionary for instances of *Eq Int* for its proof process. The notation :: tells the compiler to only bind the name and not use it for proving predicates. However, if one at a later point wants to introduce the dictionary for use by the proving machinery of the compiler this can be done by specifying:

```
instance dEqInt2 ⇜ Eq Int
```

In combination with a scoping mechanism for instances this mechanism also allows the programmer to influence which instances are actually used by the compiler:

```
let class Eq a where ...
    instance dEqInt1 ⇜ Eq Int where ...
    instance dEqInt2 :: Eq Int where ...
in  let g = λx y → eq x y
    in  let v1 = g 3 4
            v2 = let instance dEqInt2 ⇜ 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. Instances for use by the compiler are introduced in a scoped fashion: the instances introduced in an inner enclosing scope taking precedence over the ones introduced in an outer scope.

3 Implementation

Explicit passing of implicit parameters as described in the preceding section has been implemented in the EH compiler (EHC) [2, 4, 3]. Because of space limitations we only provide a sketch of its implementation and related design issues.

Combining type inference and type checking. Of all the typing rules normally used to describe the semantics of Haskell and qualified types [5, 11], the rule shown in figure 1 is the one where the difference between our work and others is to be found. This rule for the elimination of a predicate introduction is the place where a function is applied to an implicit parameter. The typing rule e-pred9A states that if an expression e accepts an implicit parameter corresponding to predicate π which we can prove to be true, we can apply the computed value ϑ_e of e to the evidence ϑ_π for π to obtain the result of the compilation.

The typing judgement for expressions itself has a 'type' too, shown in the top box of figure 1, in the sense that it is a structure with elements of a certain type. Its usual reading goes like this: given contextual information Γ it can be proven ($\vdash$) that term e has (:) type σ and some additional ($\rightsquigarrow$) results, which in our case is the code ϑ in which passing of implicit parameters is made explicit.

The rule uses types described by the type language consisting of basic types, type variables, records, functions (taking normal and implicit parameters), universally quantified types and predicates respectively:

$$\sigma = Int \mid Char \mid v \mid (l_1 :: \sigma_1, ..., l_n :: \sigma_n) \mid \sigma \to \sigma \mid \pi \to \sigma \mid \forall\, \alpha.\sigma$$

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

$$\frac{\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta_\pi : \_ \qquad \Gamma \overset{expr}{\vdash} e : \pi \to \sigma \rightsquigarrow \vartheta_e}{\Gamma \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \vartheta_\pi} \quad \text{(e-pred9A)}$$

Figure 1: Basic implicit parameter passing

$$\pi = I\ \overline{\sigma}$$

The context, or environment Γ is an ordered list of bindings which either bind value identifiers to types or predicates to a translation (a dictionary) and the type of the translation:

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

For convenience, the environment Γ holds bindings for value identifiers as well as predicates. Concatenation is denoted with a comma ’,’. Identifiers ξ can be lowercase i for values as well as uppercase I of type, class names and data constructors, thereby following the Haskell convention. If the σ in $\pi \rightsquigarrow \vartheta : \sigma$ is irrelevant in the context of a rule it is omitted.

Furthermore, we use a term language based on Haskell, that is, lambda calculus with EH specific extensions which in turn are similar to those in Haskell unless explicitly stated otherwise. A translation ϑ itself is also a term, be it of a restricted form. We will not make this more explicit.

$$e = e\ e \mid \lambda i \to e \mid i \mid ...$$

Though the conciseness of the given rule suggests that its implementation should not pose much of a problem, the opposite is true. In general, typing rules give us equations which should hold but do not tell us how to find out if and under what conditions those rules hold. Algorithmic variants of typing rules usually are closely connected to the syntactic structure of a source language. It is then at least clear which rule applies for a particular language construct. Algorithmic variants of typing rules usually also incorporate additional information which is passed from and to the premises and conclusions of a rule. This additional information corresponds to information being passed up and down a syntax tree, or in terms of an attribute grammar, this information is encoded as synthesized and inherited attributes. Finding a suitable algorithm for explicit implicit parameters is even further complicated due to a combination of several factors:

- The structure of the source language cannot be used to determine if the rule should be applied: the term e in the premise and the conclusion is the same. As a consequence, the structure of the corresponding abstract syntax tree cannot drive the decision to use the rule or not. Furthermore, the predicate π is not mentioned in the conclusion so the structure of a syntax tree will not help here too. In other words, the necessity to pass an implicit parameter may spontaneously pop up at any expression.

- In the presence of type inferencing nothing may be known about e at all, let alone which implicit parameters it may take. This information usually only becomes available after generalization of the inferred type of e.

- These problems are usually circumvented by limiting the type language for types used during inferencing to those types which do not contain predicates. By effectively stripping a type from both its predicates

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

$$\frac{\begin{array}{c}\Gamma \overset{pred}{\vdash} \pi \rightsquigarrow \vartheta_\pi : \_ \\ \Gamma; \varpi \to \sigma^k \overset{expr}{\vdash} e : \pi \to \sigma \rightsquigarrow \vartheta_e\end{array}}{\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow \vartheta_e \vartheta_\pi} \quad \text{(e-pred9B)}$$

Figure 2: Implicit parameter passing with expected type

> and quantifiers the standard Hindley-Milner type inference becomes is possible. However, we allow predicated as well as quantified types to participate in type inferencing. Consequently, a predicate as well as a quantifier can be present in any type encountered during type inferencing.

So, the bad news is that we do not know when an implicit parameter needs to be passed, but the good news is that if we make this lack of knowledge explicit 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 known. In a later stage of a type inferencing algorithm this type variable is replaced by more accurate knowledge, if any. In our approach we employ also the notion of variables, called *implicits variable*s, representing a yet unknown series ($\geqslant 0$) of implicit parameters, or, more accurately their corresponding predicates. These implicits variables are used in a type inferencing/checking algorithm which explicitly deals with expected (or known) types σ^k as well as inferred type information.

These key aspects are expressed in a slightly adapted typing rule shown in figure 2. This rule makes two things explicit:

- The context contains the expected type σ^k of e. The implementation of this rules maintains the invariant that e has a type σ which is a subtype of σ^k, σ is said to be subsumed by σ^k. This also involves coercions but in this paper we will not concern us with that additional aspect.
- An explicit parameter can be expected anywhere; this is made explicit by stating that the known type of e can have an additional sequence of implicit parameters in front. This is expressed by letting the expected type in the premise be $\varpi \to \sigma^k$.

The idea is that this implicits variable makes explicit that we can expect a (possibly empty) sequence of implicits parameters and at the same time gives an identity to this sequence. It requires the type language to be extended by an implicits variable ϖ (or 'pivar'), also corresponding to the dots "..." in the source language for predicates:

$$\pi = I\ \overline{\sigma} \mid \varpi$$

In terms of an algorithm, 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 be equal to this type.

This typing rule e-pred9B still is not much of a help as to 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 figure 3. These rules also use an additional parameter v influencing certain aspects of subsumption $\leqslant$. Also, the rule is more explicit in its handling of constraints computed by the rule labeled *fit* for the subsumption $\leqslant$:

$$C = v \mapsto \sigma \mid \varpi \mapsto \pi, \varpi \mid \varpi \mapsto \emptyset$$

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

$$\frac{\begin{array}{c}\overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma \overset{pred}{\vdash} C_3\overline{\pi^a} \leadsto \overline{\vartheta^a} : \_ \\ \nu_{inst-l}; \overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma; \sigma^a \overset{expr}{\vdash} e_2 : \_ \leadsto C_3; \vartheta_2 \\ \nu; \overline{\pi_i^k \rightsquigarrow \vartheta_i^k}, \Gamma; \varpi \to \nu \to \sigma_r^k \overset{expr}{\vdash} e_1 : \overline{\pi^a} \to \sigma^a \to \sigma \leadsto C_2; \vartheta_1 \\ \overline{\pi_i^k \rightsquigarrow \vartheta_i^k} \equiv inst_\pi(\overline{\pi_a^k}) \\ \nu; \Gamma \overset{fit}{\vdash} \varpi^k \to \nu^k \leqslant \sigma^k : \overline{\pi_a^k} \to \sigma_r^k \leadsto C_1; \_ \\ \varpi, \varpi^k, \nu, \nu^k \text{ fresh}\end{array}}{\nu; \Gamma; \sigma^k \overset{expr}{\vdash} e_1\ e_2 : C_3\sigma \leadsto C_3C_2C_1; \lambda\overline{\vartheta_i^k} \to \vartheta_1\ \overline{\vartheta^a}\ \vartheta_2} \quad \text{(e-app-impl9-impl)}$$

$$\frac{\begin{array}{c}\nu_{str}; \Gamma; \sigma^a \overset{expr}{\vdash} e_2 : \_ \leadsto C_2; \vartheta_2 \\ \nu_{pred}; \_ \overset{fit}{\vdash} \pi^\Gamma \to \sigma^\Gamma \leqslant \pi^a \to \nu : \_ \to \sigma^a \leadsto \_; \_ \\ \pi^\Gamma \leadsto \_ : \sigma^\Gamma \in \Gamma \\ \nu_{impl}; \Gamma; \pi_2 \to \sigma^k \overset{expr}{\vdash} e_1 : \pi^a \to \sigma \leadsto C_1; \vartheta_1 \\ \nu \text{ fresh}\end{array}}{\nu; \Gamma; \sigma^k \overset{expr}{\vdash} e_1\ (\#e_2 \leftsquigarrow \pi_2\#) : C_2\sigma \leadsto C_2C_1; \vartheta_1\ \vartheta_2} \quad \text{(e-app-impl9-expl)}$$

Figure 3: Type checking/inferencing for expression application with implicit parameters

The mapping from type variables ν constitutes the usual substitution for type variables. The second alternative maps an implicit variable to a list of predicates.

From bottom to top, the first rule in figure 3 reads as follows. To keep matters simple we ignore the handling of constraints C and the use of ν. The type for the application itself is expected to be σ^k, which in general will have the structure $\varpi^k \to \nu^k$. This structure is enforced and checked by the subsumption check described by the rule *fit*. We will not look into the subsumption rules; it performs unification, subsumption, predicate entailment and computation of necessary coercions. For this discussion it only is relevant to know that if an ϖ cannot be matched to a predicate it will be constrained to $\varpi \mapsto \emptyset$. In other words, we start with assuming that implicits may be everywhere and attempt to proof 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 $\varpi \to \nu \to \sigma_r^k$ for e_1. As it is the responsibility of the application $e_1\ e_2$ to return something which accepts $\overline{\pi_a^k}$, fresh names for those predicates are created by $inst_\pi$. Its binding with the predicates is used to extend the environment in which both e_1 and e_2 are type checked. The judgement for e_1 will give us a type $\overline{\pi^a} \to \sigma^a \to \sigma$, of which σ^a forms the expected type for e_2. The predicates $\overline{\pi^a}$ need to be proven and evidence computed. 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 of this lambda expression contains the actual application itself. The implicit parameters are passed before the argument itself.

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 [13], 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.

The second rule in figure 3 for explicitly passing an implicit parameter is simpler than the rule for normal application because all the required type information has been made explicit. We now only have to supply the judgement for e_2 with the type σ^a of the evidence for π^a as the expected type for e_2.

Binding time of instances. One other topic in particular deserves attention, especially since it deviates from the standard semantics of Haskell. In the example for *nub*, the invocation of *nub* is parameterized with a modified record:

$$\begin{array}{l} \mathit{nub}\ (\#(\mathit{dEqInt} \mid \mathit{eq}{:=}\lambda\_\ \_ \rightarrow \mathit{False}) \leftsquigarrow \mathit{Eq}\ \mathit{Int}\#) \\ \quad (\mathit{Cons}\ 3\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ \mathit{Nil}))) \end{array}$$

In our implementation *Eq*'s function *ne* invokes *eq*, in particular the one provided by means of the explicit parameterization. In essence, this means 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 would mean that *ne* still will use *dEqInt*'s *eq* instead of the *eq* provided in the updated $(\mathit{dEqInt} \mid \mathit{eq}{:=}\lambda\_\ \_ \rightarrow \mathit{False})$.
- A combination of these solutions, for example, default definitions use late binding, instances use Haskell's binding.

It is yet unclear which solutions is the best one, but we notice that whatever approach is taken, the programmer has all the means available to express his differing intentions.

4 Related work

Implicit parameters not only implement the passing of dictionaries as evidence for predicates. In Haskell, extensible records (if implemented) also use the predicate proving machinery available in Haskell compilers, integer offsets into records being the evidence for so called lacking predicates describing where a value for a labeled field should be inserted [11, 6, 15]. Plain values passed as implicit parameters [9, 20] is offered in (e.g.) GHC.

Scheffczyk in particular has explored named instances as well [17, 23]. 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 of the compiler. Instances are split likewise into named and unnamed instances. Named instances are used for explicit passing and do not participate in the predicate proving of the compiler. For unnamed instances this is the other way around. Our approach allows a programmer to make this partitioning by explicitly 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 [18, 19] and as described by Jones [12] 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.

The type inferencing/checking algorithm employed in this paper is described in greater detail in [4, 3] and its implementation is publicly available [2], where it is part of a work in progress. Similar strategies are described by Pierce [22] and Peyton-Jones [21] but to our knowledge ours is the first to also handle the combination of partially specified types, existentials and higher ranked polymorphic types.

5 Conclusion

Allowing explicit parameterization for implicit parameters gives a 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 decisions made by the compiler.

The approach taken in this paper still leaves much to be sorted out. In particular the relation with functional dependencies of multiparameter type classes, existentials and dictionary transformers participating in the proof process (as required by Hinze for generics [8])

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 as used for the implementation of EHC [2], are a better vehicle for expressing implementation aspects.

References

[1] Hugs 98. `http://www.haskell.org/hugs/`, 2003.

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

[3] Atze Dijkstra and Doaitse Swierstra. Typing Haskell with an Attribute Grammar (Part I). Technical report, Department of Computer Science, Utrecht University, 2004.

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

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

[6] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical report, Languages and Programming Group, Department of Computer Science, Nottingham, Nov 1996.

[7] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18:109–138, mar 1996.

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

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

[10] 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.

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

[12] 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.

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

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

[15] Mark P. Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *Haskell Workshop*. Utrecht University, Institute of Information and Computing Sciences, 1999.

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

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

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

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

[20] 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, jan 2000.

[21] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, 2003.

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

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

[24] 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.