Making Type-Class Dictionaries Explicit

Atze Dijkstra S. Doaitse Swierstra

Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{atze,doaitse}@cs.uu.nl

Abstract

The Haskell class system provides a mechanism for implicitly passing extra arguments: 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 it possible to override the choices made by the implicit mechanisms. In this paper we describe, in the context of Haskell, a mechanism that allows a programmer to explicitly construct implicit arguments. This extension blends well with existing resolution mechanisms, since it only overrides the default behavior. We show how, by a careful set of design decisions, we manage to combine type inferencing with a type system that deals with implicit arguments.

0.1 Introduction

The Haskell class system, originally introduced by both Wadler and Blott [32] and Kaes [20], offers a powerful abstraction mechanism for dealing with overloading (ad-hoc polymorphism). The basic idea is to restrict a polymorphic parameter by specifying that some predicates have to be satisfied when the function is called:

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

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

class *Eq a* **where**
  $(==) :: a \to a \to Bool$

The presence of such a class predicate in a type requires the availability of a collection of functions and values (here a collection with just one element) which can only be used on a type a for which the class predicate holds. 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*. One may look at this as if the compiler turns such instance declarations into records (called dictionaries) containing the functions as fields, and thus an explicit version of the internal machinery reads:

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

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

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.

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 $f\ eqDInt\ 3\ 4$.

Traditionally, the name "implicit parameter" is used to refer to dynamically scoped variables [24]. However, passing a dictionary for a predicate also falls under implicit parameterisation. In general, we would prefer the name "implicit parameter" to be used to refer to any mechanism which implicitly passes parameters. To avoid confusion, we use the name *implicit dictionary* as a special case of "implicit parameter" to refer to the dictionary passing associated with class predicates as described above.

Motivation The translation from $f\ 3\ 4$ to $f\ eqDInt\ 3\ 4$ is done implicitly, without any intervention from the programmer. This becomes problematic as soon as a programmer desires to express something which the language definition cannot infer automatically. For example, we may 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 such an extra instance declaration introduces an ambiguity, and is thus forbidden by the language definition; the instances are said to overlap. However, a programmer might be able to resolve the issue if it were possible to explicitly specify which of these two possible instances should be passed to f.

Submitted to Principles of Programming Languages 2007 (POPL2007), Nice, France, Jan 17-19

In this paper we propose a mechanism which allows explicit passing of values for implicit dictionaries; here we demonstrate how this can be accomplished in Haskell and point out the drawbacks and limitations of this approach. Haskell does not treat instances as first class values to be passed to functions at the programmers will, but Haskell[1] allows the programmer to tag instances indirectly with a type. Because instances are defined for a particular type, we can use a value of such a type as an index into available instances:

```
class Eq' l a where eq' :: l → a → a → Bool
data Dflt
instance Eq a ⇒ Eq' Dflt a where
  eq' _ = (==)    -- use equality of class Eq
data Mod2
instance Eq' Mod2 Int where
  eq' _ x y = (x `mod` 2) == (y `mod` 2)
newtype W l a = W a
f :: Eq' l a ⇒ W l a → W l a → Int
f = λ(W x :: W l a) (W y) → if eq' (⊥ :: l) x y then 3 else 4
v1 = f (W 2 :: W Dflt    Int) (W 4)
v2 = f (W 2 :: W Mod2 Int) (W 4)
```

By explicitly assigning the type *W Mod2 Int* to the first parameter of f, we also choose the instance *Eq' Mod2* of class *Eq'*. This approach, which is used to solve similar problems [12, 22], has the following drawbacks and limitations:

- It is verbose, indirect, and clutters the namespace of values, types, and classes.
- We only can pass a value explicitly if it is bound to a type via an instance declaration. Consequently, this is a closed world (of values) because all explicitly passed values must be known at compile time.

Instead, we propose to solve this problem by passing the dictionary of an instance directly, using fantasy syntax which combines Haskell with the notation introduced later in this paper:

```
f :: Eq a ⇒ a → a → Int
f = λ          x     y → if x == y then 3 else 4
v1 = f 2 4
v2 = f {!((==) = λx y → (x `mod` 2) == (y `mod` 2)) ⇜⇝ Eq Int!}
       2 4
```

A dictionary (encoded as a record) d of an instance is constructed and passed directly as part of the language construct {!d ⇜⇝ ...!} (to be explained later).

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 [12, 22].

Our motivation to deal with the interaction between implicit dictionaries and its explicit use is based on the following observations:

- Explicit and implicit mechanisms are both useful. Explicit mechanisms allow a programmer to fully specify all intricacies of a program, whereas implicit mechanisms allow a language to automate the simple (and boring) parts.
- Allowing a programmer to explicitly interact with implicit mechanisms avoids type class wizardry and makes programs simpler.
- Some of the problems are now solved using compiler directives.

[1] That is, GHC 6.4.1

Haskell's point of view Haskell's class system has turned out to be theoretically sound and complete [14], although some language constructs prevent Haskell from having principal types [6]. The class system is flexible enough to incorporate many useful extensions [13, 17]. Its role in Haskell has been described in terms of an implementation [16] as well as its semantics [5, 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 dictionaries.

The Haskell language definition completely determines 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 values for these record types once the type parameters of the records are known.
- The type of a function mentions the predicates for which dictionaries have to be passed.
- 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.

  The language definition precisely determines how to compute the proper dictionaries [5, 18].
- 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 in resolving ambiguities, which are forbidden to occur. 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 [10], but in the end it is only the programmer who can fully express his intentions.

The issue central to this paper is that Haskell requires 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 feasible.

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 extension and the Haskell98 approach clear in the remainder of this paper, we refer to our explicit language as Explicit Haskell (EH).

- 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 best to infer the missing information.

Our approach allows the programmer and the EH system to construct the completely explicit version of the program together, 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, or infers something that differs from the intentions of the programmer then the programmer can always provide extra information. In this sense we get the best of two worlds: the simplicity and expressiveness of systems like system F [8, 27] 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 disambiguated.
- 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 above list: the explicit passing of values for implicit dictionaries. Although explicit typing forms the foundation on which we build [3, 4], 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 dictionary passing.

Outline of this paper In this paper we focus on the exploration of explicitly specified implicit dictionaries, to be presented in the context of EH, a Haskell variant [2–4] in which all features described in this paper have been implemented. In Section 0.2 we start with preliminaries required for understanding the remainder of this paper. In Section 0.3 we present examples of what we can express in EH. In Section 0.4 we give some insight in our design, highlighting the distinguishing aspects as compared to traditional approaches. In Section 0.5 we discuss some remaining design issues and related work. We conclude in Section 0.6.

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 [31] as it offers a starting point for practical experience. We limit ourselves in the following way:

- We present examples and parts of the overall design, so the reader gets an impression of what can be done and how it ties in with other parts of the type system [2].
- We do *not* present all the context required to make our examples work. This context can be found elsewhere [3, 4].
- Here we focus on prototypical description before proving properties of EH. We come back to this in Section 0.5, where we also discuss open issues and loose ends.
- Here we give algorithmic type rules. A corresponding implementation based on attribute grammars can be found in [2].

0.2 Preliminaries

Intended as a platform for both education and research, EH offers a combination of advanced concepts: like higher ranked types, existential types, partial type signatures and records. Syntactic sugar has been kept to a minimum in order to ease experimentation.

Figure 1 and Figure 2 show the terms and types featured in EH. Throughout this paper all language constructs are gradually introduced. 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 [7, 19]. Record extension and updates are useful for re-using values from a record.

The type language as used in this paper is shown in Figure 2. A programmer can specify types using the same syntax. We mention this because often types are stratified based on the presence of (universal) quantifiers and predicates [11]. 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:

$$(\forall a.a \to a) \to (\forall b.Eq\ b \Rightarrow b \to b)$$

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 type formers, 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 any higher ranked type; instead we propagate explicitly specified types to wherever this information may be needed. Our strategies here are elaborated elsewhere [3].

0.3 Implicit parameters

In this section we give EH example programs, demonstrating most of the features related to implicit dictionaries. After pointing out these features we elaborate on some of the finer details.

Basic explicit implicit dictionaries 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 examples.

```
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 ...
```

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

```
nub          :: (Eq a) ⇒ [a] → [a]
nub [ ]      = [ ]
nub (h : t) = h : nub (filter (not.(h ==)) t)

nubBy        :: (a → a → Bool) → [a] → [a]
nubBy eq  = nub {!((==) = eq) ⇜ Eq Int!}
```

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

1. The notation ⇜ 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 ⇜ 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*, and for n_2 a new record is created: a dictionary can also be created by updating an already existing one like *dEqInt*.

This example demonstrates our view on implicit dictionaries:

- 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.

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*. One may see this as the specification of a function that maps an instance for *Eq a* to and instance 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 :: (∀ a.Eq a ⇒ Eq (List a)) ⇒ Int → List Int → Bool
f = λp q → eq (Cons p Nil) q
```

The effect is that the dictionary for *Eq* (*List Int*) will be implicitly constructed 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 such higher order predicates really becomes apparent when genericity is implemented using the class system [12]. The elaboration of this aspect can be found in the extended version of this paper [3].

Argument ordering and mixing implicit and explicit parameters A subtle issue to be addressed is to define where implicit parameters are placed in the inferred types, and how to specify to which implicit dictionaries explicitly passed arguments bind. Our design aims at providing maximum flexibility, while keeping upwards compatibility with Haskell.

Dictionaries are passed according to their corresponding predicate position. The following example is used to discuss the subtleties of this aspect:

```
let f = λp q r s → (eq p q, eq r s)
in f 3 4 5 6
```

Haskell infers the following type for *f*:

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

On the other hand, EH infers:

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

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 to pass an implicit dictionary corresponding to a predicate as late as possible, where later is defined as the order in which arguments are passed. We call this the *natural position* of a quantifier and a predicate. The *natural universal quantifier position* is defined to be in front of the argument which contains the first occurrence of the type variable quantified over. Similarly, the *natural predicate position* is defined to be in front of the argument which contains the first occurrence of a type variable of the predicate such that first occurrences of remaining type variables of the predicate are "at the left". Quantifiers go in front of predicates.

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 dictionary 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 we may pass a dictionary, and finally we must pass two normal arguments:

```
let f :: ∀a.Eq a ⇒ a → a → ∀b.Eq b ⇒ b → b → (Bool, Bool)
    f = λp q r s → (eq p q, eq r s)
in f                                3 4
     {!(eq = eqMod2) ⇜ Eq Int!} 5 6
```

Both for *Eq a* and *Eq b* a dictionary value corresponding to *Eq Int* has to be passed. For the first one the implicit *Eq Int* that is currently in scope is passed, whereas the second one is explicitly

constructed and passed by means of {! !}. Inside these delimiters we specify both the value and the predicate for which it is a witness. The notation $\{!e \leftsquigarrow p!\}$ 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 as the proof of the existence of the implicit argument to the function f.

Explicitly passing a value for an implicit dictionary is optional. However, if we explicitly pass a value, all preceding implicit dictionaries in a consecutive sequence of implicit dictionaries must be passed as well. In a type expression, a consecutive sequence of implicit dictionaries 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 giving an explicit type signature for f, 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. In general, when a dictionary is explicitly passed for a predicate which is part of a group of consecutive predicates, all preceding predicates must also be explicitly passed a dictionary. This can be circumvented be providing wrapper functions which explicitly reorder predicates in their signature.

Overlapping instances By explicitly providing a dictionary the default decision made by EH is overruled. This is useful in situations where ambiguities arise, as in the presence of overlapping instances:

```
let instance dEqInt1 ⇜ Eq Int where
      eq = primEqInt
    instance dEqInt2 ⇜ Eq Int where
      eq = eqMod2
    f = λp q r s → ...
in f {!dEqInt1 ⇜ Eq Int!} 3 4
     {!dEqInt2 ⇜ Eq Int!} 5 6
```

The two instances for *Eq Int* overlap, but we still can refer to each associated dictionary individually, using 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 from participating in the predicate proving machinery by:

```
instance dEqInt2 :: Eq Int where
  eq = \__ → False
```

The naming of a dictionary by means of ⇜ thus achieves two things. It binds the name to the dictionary and it specifies to use this dictionary as the current 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 be done by specifying:

```
instance dEqInt2 ⇜ 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 ⇜ Eq Int where ...
    instance dEqInt2 :: Eq Int where ...
    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.

In our discussion we will come back to local instances.

0.4 Implementation

We focus on the distinguishing characteristics of our implementation in the EH compiler [2–4]. Although we use records to represent dictionaries, we omit their treatment.

The type system is given in Figure 3. 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 ($\sigma \to \sigma$), implicit abstraction ($\pi \Rightarrow \sigma$), (higher ranked) universal quantification ($\forall\alpha.\sigma$), predicates (π) and their transformations ($\pi \Rightarrow \pi$). Following Faxen [5], translations ϑ represent code resulting from the transformation from implicit dictionary passing to explicit parameter passing. An environment Γ binds value identifiers to types ($i \mapsto \sigma$). 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 ($\pi \rightsquigarrow \sigma$):

$$bind = i \mapsto \sigma \mid \pi \rightsquigarrow \vartheta : \sigma \mid \pi \rightsquigarrow \sigma$$
$$\Gamma \quad = \overline{bind}$$

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 ','.

Basic typing rules Type rules in Figure 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. In this paper versions *Ev*, *EvK* and *I* are used, respectively addressing evidence translation, use of expected types and the handling of implicit dictionaries. 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 [5].

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 [25]. This occurs because the last rule E-PRED is not associated with a syntactic construct in 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 type rule variables a computation order becomes apparent.

Once we have established these two properties, all we need is a parser and an attribute grammar where type rule variables are mapped to attributes. Our situation is complicated by 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 dictionary may spontaneously pop up in any expression.
- In the presence of type inference, nothing may be known yet about e at all, let alone which implicit dictionaries 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, rank-1 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. In this paper, we focus on the predicate part.

Implicitness made explicit So, the bad news is that we do not know where implicit dictionaries need to be passed; the good news is that if we represent this lack of knowledge explicitly in the type language we can still figure out if and where implicit dictionaries 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: we represent an indirection in time by introducting a free variable. In a later stage of the 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 an as 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.

Figure 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 rule for predicates shown in Figure 6, which are adapted from their non-algorithmic versions in Figure 3. This rule makes two things explicit:

- The context provides the expected (or known) type σ^k of e. All our rules maintain the invariant that e will get assigned a type σ which is a subtype of σ^k, denoted by $\sigma \leqslant \sigma^k$ (σ is said to be subsumed by σ^k), and enforced by a *fit* judgement (see Figure 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 dictionary can be passed anywhere; this is made explicit by stating that the known type of e may start with a sequence of implicit dictionaries. 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 arguments and, at the same time, makes it possible to refer 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, constrained by $\sigma \leqslant \sigma^k$. If a fully specified expected type σ^k is passed downwards, σ will turn out to be equal to this type. If a partially known type is passed downwards the unspecified parts may be refined in by the type inferencer.

The adapted typing rule E-PRED in Figure 6 still is not of much a help in 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 Figure 8 and will be explained soon. The rules in Figure 7 (explicit implicit abstraction/application) and Figure 8 (normal abstraction/application with possible implicit arguments) look complex, because they bring together all the different aspects of an algorithmic inferencer. The reader should realize that the implementation is described using an attribute grammar system [1, 4] which allows the independent specification of all aspects which now appear together in a condensed form in Figure 7 and Figure 8. The tradeoff is between compact but complex type rules and more lengthy but more understandable attribute grammar notation.

Notation The typing rules in Figure 7 and Figure 8 are directed towards an implementation; additional information flows through the rules to provide extra contextual information. Also, these rules are 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 Figure 7 and Figure 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 [5, 14, 25]. Figure 5 provides a summary of the judgement forms we use.

The judgement **pred** (Figure 5) for proving predicates is standard with respect to context reduction and the discharge of predicates [5, 14, 18], 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.

Explicit parameter passing The rules in Figure 7 specify the typing for the explicit parameter passing where an implicit dictionary 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 dictionary. The *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.

Implicit parameter passing: application From bottom to top, rule E-APP in Figure 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 $\varpi^k \Rightarrow v^k$. 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 dictionaries 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 $\varpi \Rightarrow v \rightarrow \sigma_r^k$ 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 $\overline{\pi_a} \Rightarrow \sigma_a \rightarrow \sigma$, 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 dictionaries $\overline{\pi_a}$ are used to compute a translation for the application, which accepts the implicit dictionaries 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 dictionaries passed before the argument.

Even though the rule for implicitly passing an implicit dictionary 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 [16], 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 Figure 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 dictionaries. 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.

0.5 Discussion and related work

Soundness, completeness and principal types Although we do not prove nor claim any of the usual type system properties we believe we have constructed both type system and its implementation in such a way that the following holds (stated more formally elsewhere [3]):

- (Completeness with respect to HM (Hindley-Milner), or, conservative extension) All HM typeable expressions also type in EH.
- (Soundness with respect to HM) In absence of higher-rank types and predicates, all EH typeable expressions also type in HM.
- (Soundness with respect to System F) The type annotated (dictionary-)translation of EH typeable expressions type according to System F.

Furthermore, principality is lost because of the existence of incompatible (but isomorphic) types for types which allow freedom in the placement of quantifiers and predicates. Our algorithmic approach also lacks a declarative version describing the characteristics of our system. It is unclear what can be achieved in this area, considering the increase in complexity of the combination of language feature. For Haskell98 (without recent extensions), Faxen's work [5] comes closest to a formal algorithmic description of Haskell's static semantics.

Local instances Haskell only allows global instances because the presence of local instances results in the loss of principal types for HM type inference [32]:

```
let class Eq a where eq :: a → a → Bool
    f = λx → let instance Eq Int where
                 instance Eq Char where
             in eq x x
in f 3
```

Usually, local instances are avoided because, in the given example, for f the following types can be derived:

$f :: Int \rightarrow Bool$
$f :: Char \rightarrow Bool$
$f :: Eq\ a \Rightarrow a \rightarrow Bool$

Normally, no principal type can be derived because these types are incomparable (when both instances are out of scope). However, within the scope of both instances the most general type is $Eq\ a \Rightarrow$

$a \to Bool$; this is the type our system uses. We restrict the use of local instances to the resolution of predicates. Thus, in the example none of the instances are used unless we use explicit mechanisms, for example by providing a type signature or passing a dictionary explicitly.

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 Figure 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.
- 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.

Named instances Scheffczyk has explored named instances as well [21, 29]. 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 being able to state at each point in the program which instances participate in the proof process. In other words, the policy of how to use the implicit dictionary passing mechanism is made by the programmer, on a case by case basis.
- 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 [23] and as described by Jones [15] 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 [3, 4] and its implementation is publicly available [2], where it is part of a work in progress. Similar strategies for coping with the combination of inferencing and checking are described by Pierce [26] and Peyton Jones [31].

Future work Future work needs to address the following (unmentioned) issues:

- In relation to local instances: multiparameter type classes, functional dependencies, coherence, subject reduction, more liberal scoping, interaction with module import/export.
- Principality of the type system.
- Declarative version of the type system.

Furthermore, we intend to shift towards a constraint-based approach as many problems can be described elegantly using such an approach. (Prototypical) implementations based on constraint solving are becoming mature [10, 28, 30].

0.6 Conclusion

In general, programming languages aim at maximising the amount of work done for a programmer, while minimising the effort required by the programmer and language implementation. Within this area of tension a good cooperation between explicit (defined by a programmer) and implicit (automatically inferred by the system) program fragments plays an important role:

- Explicitness allows the programmer to specify what he wants.
- Implicitness relieves the programmer from the obligation to specify the obvious; the language can take care of that.
- A good cooperation between these extremes is important because both extremes on their own do not provide a usable language or implementation: either the programmer would be required to be explicit in too many details, or the language would have to be unimplementably smart in guessing the intention of the programmer.

Hence, both explicitness and implicitness are important for a programming language. For Haskell (and ML) the trend already is to grow towards System F while preserving the pleasant properties of type inference. In this paper we propose a mechanism which offers explicit interaction with the implicitness of Haskell's class system. The need for such an interaction is evident from the (ab)use of the class system as a solution for a variety of unintended problems, the intended problem being overloading.

References

[1] A. Baars, S. D. Swierstra, and A. Löh. Attribute Grammar System. `http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem`, 2004.

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

[3] A. Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

[4] A. Dijkstra and S. D. Swierstra. Typing Haskell with an Attribute Grammar. In *Advanced Functional Programming Summerschool*, number 3622 in LNCS. Springer-Verlag, 2004.

[5] K.-F. Faxén. A Static Semantics for Haskell. *Journal of Functional Programming*, 12(4):295, 2002.

[6] K.-F. Faxén. Haskell and Principal Types. In *Haskell Workshop*, pages 88–97, 2003.

[7] B. R. Gaster and M. 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.

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

[9] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, March 1996.

[10] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, Institute of Information and Computing Sciences, 2005.

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

[12] R. Hinze and S. Peyton Jones. Derivable Type Classes. In *Haskell Workshop*, 2000.

[13] M. 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.

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

[15] M. 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.

[16] M. P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

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

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

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

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

[21] W. Kahl and J. Scheffczyk. Named Instances for Haskell Type Classes. In *Haskell Workshop*, 2001.

[22] O. Kiselyov and C.-c. Shan. Implict configuration - or, type classes reflect the value of types. In *Haskell Workshop*, 2004.

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

[24] J. R. Lewis, M. B. Shields, E. Meijer, and J. 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.

[25] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

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

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

[28] A. Rossberg. The Alice Project. `http://www.ps.uni-sb.de/alice/`, 2005.

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

[30] M. Sulzmann. An Overview of the Chameleon System, 2003.

[31] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy type inference for higher-rank types and impredicativity (submitted to ICFP2005), 2005.

[32] P. Wadler and S. 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.

Value expressions:

$e ::= int \mid char$	literals
$\mid i$	program variable
$\mid e\ e$	application
$\mid$ **let** $\overline{d}$ **in** e	local definitions
$\mid \lambda i \rightarrow e$	abstraction
$\mid (l = e, ...)$	record
$\mid e.l$	record selection
$\mid (e \mid l := e, ...)$	record update
$\mid e\ \{!e \leftsquigarrow \pi!\}$	*explicit implicit application*
$\mid \lambda\{!i \leftsquigarrow \pi!\} \rightarrow e$	*explicit implicit abstraction*

Declarations of bindings:

$d ::= i :: \sigma$	value type signature
$\mid i = e$	value binding
$\mid$ **data** $\overline{\sigma} = \overline{I\ \overline{\sigma}}$	data type
$\mid$ **class** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	class
$\mid$ **instance** $\overline{\pi} \Rightarrow \pi$ **where** $\overline{d}$	class 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:

$i ::= i$	lowercase: (type) variables
$\mid I$	uppercase: (type) constructors
$\mid l$	field labels

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

Types:

$\sigma ::= Int \mid Char$	literals
$\mid v$	variable
$\mid \sigma \rightarrow \sigma$	abstraction
$\mid \forall v.\sigma$	universally quantified type
$\mid \sigma\ \sigma$	type application
$\mid \pi \Rightarrow \sigma$	predicate abstraction
$\mid (l :: \sigma, ...)$	record

Figure 2. EH types

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

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

$$\frac{(i \mapsto \sigma_i) \in \Gamma}{\Gamma \overset{expr}{\vdash} i : \rightsquigarrow i}\ \text{E-ID}_{Ev}$$

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

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

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

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

Figure 3. Type rules for expressions

Notation	Meaning
σ	type
σ^k	expected/known type
$\square$	any type
v	type variable
i	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

Version	Judgement	Read as
I	$\Gamma; \sigma^k \overset{expr}{\vdash} e : \sigma \rightsquigarrow C; \vartheta$	With environment 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	$\Gamma \overset{\pi}{\vdash} \pi \rightsquigarrow \vartheta : \sigma$	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

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

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

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

Figure 6. Implicit parameter passing with expected type

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

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

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

Figure 7. Type rules for explicit implicit parameters

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

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

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

Figure 8. Implicit parameter type rules