On the Bright Side of Type Classes: Instance Arguments in Agda

Dominique Devriese Frank Piessens
KULeuven
{dominique.devriese,frank.piessens}@cs.kuleuven.be

Abstract

We present instance arguments: an alternative to type classes and related features in the dependently typed, purely functional programming language/proof assistant Agda. They are a new, general type of function arguments, resolved from call-site scope in a type-directed way. The mechanism is inspired by both Scala's implicits and Agda's existing implicit arguments, but differs from both in important ways. Our mechanism is designed and implemented for Agda, but our design choices can be applied to other programming languages as well.

Like Scala's implicits, we do not provide a separate structure for type classes and their instances, but instead rely on Agda's standard dependently typed records, so that standard language mechanisms provide features that are missing or expensive in other proposals. Like Scala, we support the equivalent of local instances. Unlike Scala, functions taking our new arguments are first-class citizens and can be abstracted over and manipulated in standard ways. Compared to other proposals, we avoid the pitfall of introducing a separate type-level computational model through the instance search mechanism. All values in scope are automatically candidates for instance resolution. A final novelty of our approach is that existing Agda libraries using records gain the benefits of type classes without any modification.

We discuss our implementation in Agda (to be part of Agda 2.2.12) and we use monads as an example to show how it allows existing concepts in the Agda standard library to be used in a similar way as corresponding Haskell code using type classes. We also demonstrate and discuss equivalents and alternatives to some advanced type class-related patterns from the literature and some new patterns specific to our system.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Agda, instance arguments, type classes, ad hoc polymorphism

1. Introduction

1.1 Type Classes

In 1998, a group of scholars on the Haskell Committee were facing the problem of fixing the types of the numeric and equality operators in the emerging Haskell programming language [8]. These operators feature a natural requirement for overloading or "ad hoc" polymorphism. For example, the == operator, of type t → t → Bool, should only be defined for certain types t (e.g. Bool, Integer) and not for others (e.g. function types). Additionally, different implementations are required for different types t.

The committee at the time recognized the issue as an instance of a more general problem in need of a general solution and adopted Wadler's proposal for what is now known as the Haskell type class system. For the == operator, the solution is based on a type class Eq t, with instances for appropriate types t. To avoid troubling this section with notations for infix operators, we write equal for ==.

```
class Eq t where equal :: t → t → Bool
instance Eq Bool where equal = primEqBool
instance Eq Integer where equal = primEqInteger
neq :: Eq t ⇒ t → t → Bool
neq a b = not (equal a b)
test :: Bool
test = equal (5 :: Integer) 5
```

Subclasses can also be defined:

```
data Ordering = LT | EQ | GT
class Eq t ⇒ Ord t where
  compare :: t → t → Ordering
```

An essential requirement for type classes to work is that functions like neq which use the equal operator for an abstract type t declare this in their type. The compiler can then check that the required instances are defined when t is instantiated to a concrete type: when equal is called on two Integer values in the definition of test, it looks for an Integer instance of the type class in scope and uses that instance's implementation of the equal operator.

Before we continue, we want to make it clear that when talking about Haskell, we will amalgamate the type class concept in Haskell proper with common and uncontroversial extensions like FlexibleContexts, FlexibleInstances, MultiParamTypeClasses, TypeFamilies and RankNTypes.

Note also that when we mention ad hoc polymorphism, we mean *open* ad hoc polymorphism. This means that additional instances of abstract concepts can be added independently by users of functions that require the concept. If openness is not required, Agda supports other solutions based on the definition of a universe representing the complete set of types that satisfy the concept.

1.2 The downsides of an extra structuring concept

A disadvantage of Haskell's type class system is that classes and instances form a separate, special-purpose structuring concept, in addition to the more standard algebraic data types (ADTs). Because of this duplication of functionality, many of the features that have in the past been introduced as extensions of type classes duplicate features that already existed for ADTs. Constraint families [18] (allowing classes to have abstract constraints on type class parameters) and associated type families [23] (allowing classes to specify abstract types) both roughly correspond to how generalized algebraic data type [21] values can carry types or type functors that are not parameters of the data type. In the area of generic programming, succesful techniques existed for ADTs [19], but these have had to be adapted for type classes [12, 27].

Another example is how higher-rank types [22] have long allowed ADTs to be abstracted over. However, in a paper about a datatype-generic programming technique [12], Lämmel and Peyton Jones note that this is not possible for type classes. In the following pseudo-code, they wanted to abstract over a type class cxt (the meaning of these type classes is not important here):

```
  -- Pseudo-Haskell
class (Typeable a, cxt a) ⇒ Data cxt a where
  gmapQ :: (forall b.Data cxt b ⇒ b → r) → a → [r]
instance Data Size t ⇒ Size t where
  gsize t = 1 + sum (gmapQ gsize t)
```

This pseudo-code is not legal Haskell so Lämmel and Peyton Jones provide a solution based on a "generic" type class Sat parameterised by the type of a dictionary record that it should carry:

```
class Sat a where dict :: a
```

We find this a clever solution, but it amounts to replacing a type class with an ADT for which the desired feature (abstracting over one) is available.

1.3 Dictionary Translation

A well-known model of type classes using standard ADTs is known as *dictionary translation* [26]. This translation is often used as an implementation strategy, but also gives an accurate semantic model of the type class concept. A type class is modelled as a dictionary record type, with the type class operations as record fields. Instances become record values with as fields the definitions in the instance. The above code translates to the following:

```
data Eq t = EqDict {equal :: t → t → Bool}
data Ord t = OrdDict {eqDict :: Eq t
                     , compare :: t → t → Ordering}

boolEq :: Eq Bool
boolEq = EqDict primEqBool

intEq :: Eq Integer
intEq = EqDict primEqInteger

neq :: Eq t → t → t → Bool
neq dict a b = not (equal dict a b)

test :: Bool
test = equal intEq 5 5
```

Striking about this translation is that the resulting code is not actually that far from the original. Apart from the additional naming of instances (which has also been proposed for Haskell [10]), the translation only produces extra verbiage in the implementation of functions that use the type class's operations. In the neq function, the dictionary of type Eq t is now passed around explicitly where this was implicitly done for us before. Additionally, in the definition of test, we need to explicitly specify the intEq dictionary as an extra parameter whereas it was inferred by the compiler before.

Apart from the automatic inference of instances, the dictionary model has many advantages over the standard type class system. All the power of normal language record mechanisms is available, and they can be defined, manipulated and abstracted over in standard ways.

1.4 Instance search

An aspect of type classes we have not yet touched upon is instance search. Haskell allows instance definitions like the following:

```
instance Eq a ⇒ Eq [a] where
```

```
equal [] [] = True
equal (a : as) (b : bs) = equal a b ∧ equal as bs
equal _ _ = False
```

With this instance, Haskell will resolve constraints of the form Eq [a] by recursively resolving the constraint Eq a and then using the above definition of equal. This mechanism makes the instance search algorithm a lot more powerful and complex and a set of restrictions is enforced on the structure of the types involved in instance contexts to ensure that the instance search remains decidable. Two widely used Haskell extensions (associated type families [23] and functional dependencies [9]) introduce further complexity by adding what are essentially decidable type-level computation primitives, which can be triggered during the instance search process. As a reviewer notes, these extensions effectively expose an interpreter for a simple logic programming language (no backtracking) which can reason about Haskell types.

In a non-dependently typed language like Haskell, type-level computation is not directly available in the base language. Therefore, the type-level computation primitives provided by these primitives fill a certain gap in the language and various people have demonstrated the surprising amount of power that these extensions offer [11, 14]. However, the computational model for these primitives differs strongly from the standard Haskell model: a form of structural recursion is used instead of non-structural (although many compilers provide an option to change this), pattern matching is open instead of closed and the syntactic order of pattern matching and recursive calls is reversed.

1.5 Scala Implicits

The Scala programming language features an alternative feature to type classes called implicits that avoids the introduction of a new structuring mechanism [2, 17]. Powerful mechanisms like existential types [17, §3.2.10] and abstract type declarations [17, §4.3] can be used to model features that would have to be specifically defined and implemented for type classes. Our running equal example can be encoded in Scala as follows:

```
trait Eq [A] {def equal (x: A, y: A) : Boolean}
def equal [A] (x: A, y: A) (implicit eqA : Eq [A]) =
  eqA.equal (x, y)
implicit object boolEq extends Eq [Boolean] {
  def equal (x: Boolean, y: Boolean) : Boolean =
    x == y}
implicit object intEq extends Eq [Int] {
  def equal (x: Int, y: Int) : Boolean = x == y}
def neq [A] (x:A, y: A) (implicit eqA : Eq [A]) =
  ! equal (x, y)
val test = equal (1, 3) ∨ equal (true, false)
```

The type class Eq is modelled as a dictionary trait Eq [A]. Traits are a general object-oriented structuring concept provided by Scala, similar (for our purposes) to records. Two dictionary objects intEq and boolEq are introduced and annotated with the **implicit** modifier. The equal function is defined to accept an argument eqA of type Eq A that is marked as **implicit**, with the effect that when the function is called, and the implicit arguments are not explicitly provided, values are inferred. Candidate values are searched from a precisely defined set of candidates, which includes all definitions that were annotated with the **implicit** modifier and are either accessible at the call-site without a prefix, or were defined in companion modules of the implicit argument's type or its components. A function can accept several implicit arguments, but they have to come after all other function arguments.

Unfortunately, functions with implicit arguments are not first-class citizens in Scala, mostly due to syntax-technical problems. Some important features of Scala's standard functions (currying, partial application, lambda expressions) either require non-trivial encodings or are not available for implicits.

More concretely, Scala implicits have the following restrictions. In the first place, implicit arguments are restricted to occur after all the other arguments. Abstracting over functions taking implicit arguments is not syntactically possible but requires such functions to be encoded as objects with an apply method taking an implicit argument or first converted by the caller to normal functions. There is no user syntax for the type of a function accepting implicit arguments. Anonymous functions cannot accept implicit arguments[1]. Full and tight control on the insertion of implicit arguments does not seem to be available and it does not seem possible to partially apply a given function with implicit arguments to any chosen subset of its (implicit and ordinary) arguments (while keeping the implicit arguments implicit).

Scala also uses a certain search algorithm to infer a value for implicit arguments that are not explicitly provided. This algorithm is similar to Haskell's instance search. To resolve an implicit argument of type T, Scala will consider candidate values that have been marked "implicit" and which are *accessible* (can be named without a prefix) or in the *implicit scope* of type T (defined in a module related in a specific way [17, §7.2 pp. 104–107] to type T). It will consider values of type T, but also functions that themselves take only implicit arguments and return a value of type T. This makes the instance search recursive, like Haskell's instance search. To ensure decidability of the search, Scala keeps track of the "call stack" of instance searches and detects infinite loops using a certain criterion [17, §7.2 p. 106].

Like Haskell's instance search, Scala's implicit search can also be exploited as a type-level computation primitive. Oliveira et al. demonstrate [2] an encoding of session types, an arity-polymorphic zipWith function and a form of generalized constraints. However, this computation primitive presents a computational model like Haskell's instance search, different from Scala's standard model.

1.6 Agda implicit function arguments

A final language feature we want to present before introducing our proposal, can be found in our target language itself: Agda's *implicit* or *hidden arguments* [15]. Agda allows function arguments to be marked as "implicit", indicating that they do not need to be provided explicitly at the call site. For example, an Agda polymorphic identity function can be defined as follows.

```
id : {A : Set} → A → A
id v = v
```

[1] The Scala syntax (**implicit** x ⇒ x) defines an anonymous function that takes a *normal* argument x, but makes x eligible for implicit resolution in the function body [17, §6.23].

When type checking the expression id true, Agda silently inserts a meta-variable (as if the expression was id {_} true), and Agda's unifier will instantiate this meta-variable to Bool. The argument can optionally be specified explicitly, by writing id {Bool} true. Implicit arguments are pervasive in most Agda code, and Agda would arguably be nearly unusable without it.

Unfortunately, Agda's implicit arguments are of no help for passing around and inferring type class dictionaries in the background. This is because Agda will only infer implicit arguments if it can statically decide that only a single value[2] exists of the required type. This makes the feature unsuitable for passing our type class dictionaries, because for a type `t`, many values of type `Eq t` can typically be defined. For example, even for a simple type like `Bool`, we can define a trivial equality operator `equal _ _ = true` in addition to the standard one.

However, unlike for Scala's implicit arguments, functions taking an implicit argument in Agda are first-class citizens. They can be abstracted over, their types can be spelled out, anonymous functions with hidden arguments are no problem and syntax is available to keep a tight control over whether or not implicit arguments are inferred or not. In some cases this requires writing for example `λ {A} → id {A}` instead of `id` to make sure that Agda does not try to infer the hidden argument.

1.7 Contributions

In this paper, we propose and study *instance arguments*: a new language feature that is an alternative to type classes, with an implementation in Agda. Our proposal is inspired by Scala's and Agda's implicit argument mechanisms. It does not introduce a separate structuring concept, but our ad hoc polymorphic functions are first-class citizens. Our proposal can work with less or more powerful types of instance resolution, but we choose a simple one that avoids the introduction of a separate computational model.

To the best of our knowledge, no other proposal in the literature offers equivalents to *all* of the following features: associated type families and constraint families, multi-parameter type classes, local instances, abstraction over type classes and first-class ad hoc polymorphic functions (although Coq, Haskell and Scala each have almost all of them). No other proposal has explored an alternative to type classes without introducing a separate computational model in the language. No other proposal did not require “instances” to be somehow marked eligible for implicit resolution. Finally, no other proposal has an equivalent to the way that we automatically bring the benefits of type classes to unmodified records.

We formally define the workings of our feature, and discuss our design choices. We demonstrate the use of monads and present (often simpler) encodings of some type class based patterns from the literature. We also present some novel patterns of our own.

2. Instance arguments

The feature we propose in this text is inspired by both Agda's implicit arguments and Scala's implicits. It is a new kind of function arguments, which we call *instance arguments*. These arguments can be inferred by the compiler even if multiple possible values exist with the expected type, provided only a single definition of such a value is in scope at the call site (more details in section A.3). We do not require values to be marked in a specific way to be eligible for this resolution. Like for Agda's existing implicit arguments, functions with the new arguments are first class citizens and there are no limitations on the location of the implicit arguments. We take care to limit the computational power of our instance search algorithm so that we do not unwantedly introduce an alternative computational model.

[2] In a dependently typed language like Agda, types are values too.

2.1 By example

Let us consider our running example equal, and define a standard Agda record called Eq representing the Eq type class dictionary, and "instances" for the ℕ and Bool type from the Agda standard library [3]. These two types correspond (for our purposes) to Haskell's Integer and Bool.

```
record Eq (t : Set) : Set where
  field equal : t → t → Bool

eqBool : Eq Bool
eqBool = record {equal = primEqBool}

eqNat : Eq ℕ
eqNat = record {equal = primEqNat}
```

All of this is standard Agda code. Our modified version of Agda allows us to define the following:

```
equal : {t : Set} → {{eqT : Eq t}} → t → t → Bool
```

This type signature says that the function equal takes a Set (type) as its first (standard) implicit argument t. The double braces mark the function's second argument eqT of type Eq t as an instance argument. Next, the function takes two standard arguments of type t and returns a Bool. In equal's definition, we simply take the implicitly passed dictionary and return the equal function contained in it:

```
equal {{eqT}} = Eq.equal eqT
```

With this type signature, we can now use equal as if it were defined in a Haskell type class. The following definition for example, normalises to the expected value false (assuming standard definitions of primEqBool and primEqNat).

```
test = equal 5 3 ∨ equal true false
```

What happens underneath, for example for the application equal 5 3, is that the Agda type-checker notices that in order to pass the non-hidden argument 5 to function equal, it first needs to infer the two implicit arguments (t and eqT). It will assign a new meta-variable [16] to both, but for the second argument, a constraint will additionally be registered indicating that that meta-variable needs to be resolved as an instance argument. The argument 5 will then be passed to equal as the third argument, and Agda will unify the first meta-variable with value ℕ. Agda will now notice that there is *only one value of type Eq ℕ in scope* (eqNat) and assign it to the second metavariable.

Like for implicit arguments, it is also possible to provide the instance arguments explicitly, should this be necessary:

```
test2 = equal {{eqNat}} 5 3 ∨ equal {{eqBool}} true false
```

Our version of neq looks nice: like equal, it accepts a dictionary of type Eq t as an instance argument:

```
neq : ∀ {t} → {{eqT : Eq t}} → t → t → Bool
neq {{eqT}} a b = ¬ (equal {{eqT}} a b)
```

In the definition, we explicitly accept the dictionary argument and pass it to the equal function, but in fact this is not necessary. If we leave out the dictionaries in the definition, Agda will silently insert an unnamed instance argument in the left-hand side and will silently infer equal's instance argument to that unnamed value:

```
neq a b = ¬ (equal a b)
```

Notice how the mechanism is strikingly similar to Agda's existing implicit arguments in many respects. Only the resolution of the instance value is different.

2.2 Native support code for records

An important and novel feature of our proposed system is that we can automatically bring its benefits to unmodified libraries that use standard dependently-typed records. In the above example, it is the function equal of type

```
equal : ∀ {t} → {{eqT : Eq t}} → t → t → Bool
```

which allows us to use the Eq record in a more convenient, type-class-like way. The similarity to the standard projection operator Eq.equal which Agda generates in the background [15, 4.3 pp.82–83] is striking:

```
Eq.equal : ∀ {t} → (eqT : Eq t) → t → t → Bool
```

From this observation, it is not a big leap to automatically generate new versions of the projection functions which (like our equal) take the record as an instance argument instead of as a standard one. In fact, it is even easier and more powerful to generate an Agda module application like the following:

```
module EqInst {t} {{eqT : Eq t}} = Eq eqT
```

Module applications are part of Agda's module system. They allow to manipulate in certain ways the contents of a module. This module application will create a new module EqInst containing all the definitions in the right-hand-side module Eq, abstracted on the left-hand-side arguments {t} and {{eqT : Eq t}} and applied to the right-hand-side argument eqT. This makes not only the projection functions available in the new module, but also other functions defined in the scope of the record (more details in section 2.4). For the above example, we could have omitted the definition of function equal, instead just importing equal from a new type of module application, written Eq {{...}}:

```
open Eq {{...}} using (equal)
```

The function equal in these modules is identical to our custom definition above. Our monads case study in section 3 demonstrate that these definitions expose an interface that is very similar to the equivalent with type classes.

2.3 Subclasses

In dictionary models of type classes, a subclass dictionary typically carries a superclass dictionary as one of its fields. The Agda standard library for example uses such a model. In the context of a dependently typed language, there is actually another possible model for subclasses, known as Pebble-style structuring, which is recommended by Sozeau et al. [24, §4.1]. In this style, subclass dictionaries carry superclass dictionaries as parameters instead of fields.

Both models can be expressed with our system. Each has some specific advantages, e.g. a requirement to explicitly bring in scope superclass dictionaries or the need for an extra implicit superclass dictionary parameter in functions with a subclass constraint. In this section, we demonstrate a Pebble-style model of an Ord subclass of our previously defined Eq:

```
record Ord {A : Set} (eqA : Eq A) : Set where
  field _<_ : A → A → Bool
```

Let's now suppose that we have values of type Eq ℕ, Ord ℕ and Eq Bool in scope, but no instance of type Ord Bool. We can then still open the Eq {{...}} and Ord {{...}} modules and use the appropriate methods on ℕ and Bools, with the correct dictionaries being resolved in the background.

```
open Ord {{...}}
open Eq {{...}}
test₁ = 5 < 3
test₂ = eq 5 3
test₃ = eq true false
```

An ad-hoc polymorphic function _⩽_ now looks as follows:

```
_⩽_ : {A : Set} → {eqA : Eq A} →
            {{ordA : Ord eqA}} → A → A → Bool
a ⩽ b = a < b ∨ eq a b
```

Note how the Pebble-style subclass model requires us to explicitly mention a superclass constraint in the type signature of _⩽_. This argument eqA : Eq A is accepted as an implicit argument, not an instance, because it can typically be inferred from the parameters of the chosen ordA parameter. Because we require the superclass dictionary as an argument, it is automatically in scope for resolution inside the method.

The above shows that our mechanism does not impose a choice as to how subclasses are to be modelled by the programmer. We think this demonstrates that our instance arguments are a fundamental mechanism, giving the programmer the freedom to make his or her own design choices.

2.4 Native support code for records

A novel feature of our instance arguments is the fact that we can automatically bring its benefits to existing record based code, as demonstrated in the introduction. Let us assume we have a dependently-typed record definition of the following form:

```
record R Δ : Set where
  field
    x₁ : A₁
    x₂ : A₂ [x₁]
    ...
    xₙ : Aₙ [x₁ ⋯ xₙ₋₁]
  y₁ : B₁ [x₁ ⋯ xₙ]
  y₁ = v₁ [x₁ ⋯ xₙ]
  ...
```

$$\begin{array}{lll}
\mathsf{y_n} & : & \mathsf{B_1}\,[\mathsf{x_1} \cdots \mathsf{x_n}, \mathsf{y_1} \cdots \mathsf{y}_{n-1}] \\
\mathsf{y_n} & = & \mathsf{v_n}\,[\mathsf{x_1} \cdots \mathsf{x_n}, \mathsf{y_1} \cdots \mathsf{y}_{n-1}]
\end{array}$$

Agda will automatically generate a module of the following form:

$$\begin{array}{l}
\textbf{module}\ \mathsf{R}\ \{\Delta\}\ (\mathsf{r} : \mathsf{R}\ \Delta)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{x_1} & : & \mathsf{A_1} \\
\mathsf{x_1} & = & \cdots \\
\cdots & & \\
\mathsf{x_n} & : & \mathsf{A_n}\,[\mathsf{x_1} \cdots \mathsf{x}_{n-1}] \\
\mathsf{x_n} & = & \cdots \\
\mathsf{y_1} & : & \mathsf{B_1}\,[\mathsf{x_1} \cdots \mathsf{x_n}] \\
\mathsf{y_1} & = & \mathsf{v_1}\,[\mathsf{x_1} \cdots \mathsf{x_n}] \\
\cdots & & \\
\mathsf{y_n} & : & \mathsf{B}\,[\mathsf{x_1} \cdots \mathsf{x_n}, \mathsf{y_1} \cdots \mathsf{y}_{n-1}] \\
\mathsf{y_n} & = & \mathsf{v_n}\,[\mathsf{x_1} \cdots \mathsf{x_n}, \mathsf{y_1} \cdots \mathsf{y}_{n-1}]
\end{array}
\end{array}$$

This module is parameterised on a record value r, and the functions in the module are implemented using its field values. We now allow a new form of module application:

$$\textbf{open}\ \mathsf{R}\ \{\!\{\ldots\}\!\}$$

Like for Agda's standard module applications, the modifiers **public**, **using**, **renaming** and **hiding** can be used to control precisely what is imported. This new module application is equivalent to the following older-style notation (except that it doesn't name the RInst module):

$$\textbf{open module}\ \mathsf{RInst}\ \{\Delta\}\ \{\!\{\mathsf{r} : \mathsf{R}\ \Delta\}\!\} = \mathsf{R}\ \{\Delta\}\ \mathsf{r}$$

In this new module, all the definitions from module R, including the field projections $\mathsf{x_1} \cdots \mathsf{x_n}$ and additional declarations $\mathsf{y_1} \cdots \mathsf{y_n}$, are available in a form that accepts the record value r as a instance argument. As we have demonstrated in the introduction, these definitions allow them to be used in a *type-class-like* way.

Technically, this new type of section applications can in fact be applied to any module taking at least one argument, turning the last (normal or hidden) argument into an instance argument. This allows the mechanism to also be used for situations like in section 3, where the Agda standard library's monad concept is not defined as a record directly, but as a special case of an indexed monad.

2.5 Considerations for instance arguments in other languages

An important question about our proposed instance arguments is how Agda-specific they are. We believe that the mechanism is widely applicable, and that many variations on our design choices are possible.

Let us consider the different modifications that we have made. A first step is the introduction of a new, specially annotated type of arguments to functions, which is likely unproblematic in many programming languages. Clearly, in non-dependently typed languages, the arguments' type must be restricted to not depend on earlier non-type arguments' values, but this reflects the rules for normal arguments in those languages. However, care must be taken that functions with the new type of arguments are fully first-class on the one hand and that the programmer can tightly control the introduction of values for implicit arguments on the other hand.

To the best of our knowledge, Agda was the first language to demonstrate that these two requirements can be combined with a natural syntax through a careful balancing in the type checking rules which govern function applications, lambda expressions, and the implicit insertion of implicit lambda's. The rules in section A.1 and A.2 for our instance arguments are simply adaptations of the corresponding rules for Agda's existing implicit arguments [15]. We think that a similar type of function arguments and similar rules can be introduced for any language which has some form of partial function application and lambda expressions.

The choice of the algorithm by which not explicitly provided instance arguments are inferred is in fact orthogonal to the rest of our design. We clearly choose a relatively low-power one (more explanation in section 4), but we think that other choices can also be combined with the rest of our design, ranging from our relatively low-power inference (see section 4) to a full-power automated proof-search like Coq's [24]. An advantage of our algorithm is that we do not require values in scope to be specially annotated to be eligible for this inference, but an annotation similar to Scala implicit annotations can be used to limit the complexity of a more powerful inference. Another advantage of our low-power inference algorithm is that it does not introduce a separate type-level computational model in the language.

2.6 Formal Developments

In appendix A, we formally develop instance arguments, based on the formalism that Norell uses to present the Agda language [15]. We formally define functions with instance arguments, how values for them are type-checked, when values for instance arguments are inferred and the rules for this resolution. We discuss various technical points and present a soundness result.

2.7 Implementation

We have implemented the above proposal in Agda. Our implementation is surprisingly cheap, with a non-context-diff of about 750 lines. It's hard to compare line-counts between different programming languages (Agda is implemented in Haskell, Coq in OCaml), but for what it's worth: the initial diff of Sozeau's Coq type classes [24] was ~2k lines long. Our implementation has been incorporated in the development version of Agda[3], and will be part of Agda 2.2.12 when it is released.

3. Monads case study

Our instance arguments provide an alternative for type classes. They lift some of the limitations of type classes but our inference algorithm is less powerful than Haskell's. To demonstrate that our mechanism is at least powerful enough for many common use cases of type classes, we take a look at a typical type class example: monads. In this section, we demonstrate that with our instance arguments, we can use the Agda standard library's RawMonad's in similar ways to Haskell's monads.

The closest equivalent of Haskell's Monad type class in Agda's standard library is the RawMonad concept in the Category.Monad module. Unlike its Haskell relative, it is defined as a special kind of indexed monad, a concept that is defined as RawIMonad in module Category.Monad.Indexed[4]. We copy the most important parts of the definitions here:

```
RawMonad  : ∀ {f} → (Set f → Set f) → Set _
RawMonad M  =  RawIMonad {I = ⊤} (λ _ _ → M)
record RawIMonad {i f} {I : Set i} (M : IFun I f) :
  Set (i ⊔ suc f) where
  infixl 1 _>>=_ _>>_
  field
    return  : forall {i A} → A → M i i A
    _>>=_  : forall {i j k A B} → M i j A →
      (A → M j k B) → M i k B
  _>>_ : ∀ {i j k A B} →
      M i j A → M j k B → M i k B
  m₁ ≫ m₂ = m₁ ≫= λ _ → m₂
```

We see that RawMonad is defined as an indexed monad which just ignores its indices. An indexed monad contains the essential monadic operators return and _≫=_ as fields and provides the _≫_ operation. In order to highlight correspondence with Haskell's monads, we slightly modify the RawIMonad record module to additionally include a syntax definition (a form of restricted macro) for a form of do-notation. This addition is orthogonal to the use of instance arguments.

```
bind : ∀ {i j k A B} →
  M i j A → (A → M j k B) → M i k B
bind {i} {j} {k} {A} {B} =
  _≫=_ {i} {j} {k} {A} {B}
syntax bind m (λ x → c) = do x ← m then c
```

We can bring in scope some monad "instances" from the Agda standard library. We bring in scope a state monad with mutable state variable of fixed type ℕ, a partiality monad (defining a form of partial or possibly non-terminating computations) and the list monad (defining monadic operations over the List type constructor):

```
open import Category.Monad.State using (StateMonad)
stateMonad = StateMonad ℕ
open import Category.Monad.Partiality using ()
  renaming (monad to partialityMonad)
pmonad = partialityMonad
open import Data.List using (List; _∷_; [])
```

---

[3] Use the command `darcs get http://code.haskell.org/Agda` to download the latest source code.

[4] The definitions are called *raw* because they do not contain proofs of the monad laws.

```
    renaming (monad to listMonad)
  lmonad  =  listMonad
```

There is a technical reason, related to the universe polymorphism of these monad instances, why we need to provide the apparently trivial definitions of pmonad and lmonad. We will come back to this in section 4.

In current Agda, the most convenient way to use these monad "instances", is to apply the RawMonad module to the correct monad instance at the location where it is used.

```
  test1 : ℕ → ℕ ⊥
  test1 k  =  let open RawMonad partialityMonad in
              do x ← return k then
              if (equal x 4) then return 10 else never
```

This code does not look too bad actually. Opening a monad instance's module brings in scope just the definitions of the monadic operations we need. However, it becomes more difficult if we decide that we need to use for example the monadic bind operator on a list, requiring monadic operations from two different monad "instances". In this case, current Agda requires us to rename one:

```
  postulate nToList : ℕ → List ℕ

  test₂ : ℕ → (List ℕ) ⊥
  test₂ k  =
    let open RawMonad partialityMonad
        open RawMonad lmonad using ()
          renaming (_≫=_ to _≫=ₗ_) in
    do x ← return [k] then
    if (equal k 4) then return (x ≫=ₗ nToList) else never
```

We can improve upon this situation using instance arguments. First, we bring the definitions from the RawMonad {{ ... }} module application in scope. We have to define it ourselves because RawMonad is not directly defined as a record, but it is general and could be added to Agda's standard library. We can then define our examples in a simpler way and let Agda infer the correct values for the instance arguments.

```
open RawMonad {{...}}
test₁ : ℕ → ℕ ⊥
test₁ k  =  do x ← return k then
              if (equal x 4) then return 10 else never
test₂ : ℕ → (List ℕ) ⊥
test₂ k  =
  do x ← return [k] then
  if (equal k 4) then return (x ≫= nToList) else never
```

In the case of test₁, one could argue that we don't actually gain all that much. Agda now automatically chooses the correct monad "instance" from the values in scope instead of requiring the programmer to make this choice. However, the second example shows that in a case where we use monadic operations from different monad "instances", instance arguments effectively spare us some uninteresting bookkeeping, by inferring appropriate "instances" in the background.

4. No automated proof search

Our resolution algorithm is only a restricted analog to Haskell's instance search. The mechanism is designed so that only one type-directed scope-based resolution will be done per not explicitly provided instance argument in the program (see section A.3). This limitation is a deliberate choice, intended to avoid introducing a separate computational model through the instance search mechanism, as for Scala implicits or Haskell and Coq type classes. However, this decision does unavoidably limit the functionality of our mechanism. For example, for the Eq type introduced in section 2, we could have a definition like the following:

```
listEq : {A : Set} → Eq A → Eq (List A)
listEq {A} eqA = record {equal = eq'} where
  eq' : List A → List A → Bool
  eq' [] [] = true
  eq' (a :: as) (b :: bs) = and (equal eqA a b) (eq' as bs)
  eq' _ _ = false
```

Now, with the eqBool value from section 2 in scope, you might expect an instance of Eq (List Bool) to be automatically inferred as listEq eqBool. However, in our system, this is not the case: in such a situation, we require the user to explicitly construct a value of the correct type himself. It suffices to bring this value in scope at the call site, for example by placing it in a local **where** block.

```
test = equal (true :: false :: true :: []) (true :: false :: [])
  where listBoolEq = listEq eqBool
```

We encountered another interesting case of this problem in the previous section, where we had to provide seemingly “trivial” definitions for values pmonad and lmonad. The reason that these were needed is that the definitions were not actually trivial. The types of the values involved are as follows:

```
partialityMonad : {l : Level} → RawMonad (_⊥ {l})
pmonad : RawMonad (_⊥ {_})
listMonad : {l : Level} → RawMonad (List {l})
```

```
  lmonad : RawMonad (List {_})
```

This is an example of Agda's universe polymorphism. The value _⊥ is not a functor of type Set → Set, but instead, for any *level* l, _⊥{l} is a functor of type Set l → Set l. This means that partial computations can be defined producing values (Set 0), types (Set 1), kinds (Set 2), and for each of these types of partial computations, a monad "instance" is provided as partialityMonad {l}.

The monadic computations and lists we used before all produced or contained values, so our pmonad and lmonad are defined as monad instances for resp. partial computations and lists working with values. Note that we did not need to specify level 0 explicitly for this. Because we just omit the level argument, Agda inserts an underscore implicitly and infers its value when it resolves pmonad and lmonad as values for the instance arguments.

4.1 But why not?

So, in both of the above examples, our resolution search was not smart enough to automatically infer certain instance arguments that one might expect it to. In both cases, help from the programmer is required to make it find the correct value. The extra information is limited (placing the required value in scope) and does not require explicitly passing the instance arguments everywhere they are used.

We actually believe that a smarter resolution algorithm can be defined for our mechanism. Extensions can be imagined where functions like listEq are annotated somehow to make the resolution consider them. Such an extension would be largely orthogonal to the rest of our design. However, introducing such an extension makes the instance search recursive. Even if it can still be kept decidable with restrictions on the functions considered, it would inevitably expose an additional computational model, similar to Haskell's, Scala's or Coq's instance search.

Because our implementation is in Agda, we are extra sensitive to this point. Any instance search that can automatically infer for example the value of listBoolEq above, must somehow perform a reasonably powerful automated proof search. This is an area where up to now, the Agda language designers have taken a very principled approach. Agda does provide such a mechanism (dubbed auto), but only in the interactive proving/programming environment, not in the language. Agda has also refrained from introducing an equivalent to Coq's untyped, imperative meta-programming facility (Coq "tactics"), instead developing a more principled mechanism through the quoteGoal construct. This construct is intended to allow Agda to function in a sense as "its own tactic language", although it is currently still limited because no access to the context or scope is available to the meta-programs. We believe that Agda's approach in this area is very promising, and the limited power of our inference algorithm avoids compromising Agda's principled design choices in these areas by introducing a parallel computational model which could be exploited as a meta-programming construct, as has happened in other languages.

4.2 Advantages

Note also that our simple resolution scheme has some advantages of its own. We have used it for all of the examples in section 3 and 5 and the resolution has proven practical, predictable, intuitive and sufficient. Also, we do not need to limit resolution complexity by requiring candidate values to be annotated specially, instead considering all values in scope. This lowers the impact of our feature on users' code and makes for example the ellipsis in section 5 more widely usable. Its intuitive meaning also changes from "Fill in this value from an *annotated* value in scope" to "Fill in this value from the scope", which feels more natural to us.

Note that because the entire scope is considered for resolution, it is up to the programmer to make sure that only a single value of a correct type is in scope. Instance arguments should only be used on types which are informative enough so that they typically identify values uniquely. If there still is a conflict, existing features in Agda's module system (e.g. **hiding** and **using** modifiers) can be used to control the scope. Note also that values that are not directly in scope but via a named module are not considered for instance resolution, but they can still be accessed explicitly. In our experiments, we find that instance arguments provide a solution (ad hoc overloading) for many of the name conflicts that arise in typical use of Agda's standard library (e.g. $\_\overset{?}{=}\_$ in Data.Nat, Data.Bool etc.) and that type conflicts for reasonably typed instance arguments occur seldom.

5. Some advanced patterns

It turns out that our relatively simple extension of Agda can support analogs or variants of many features which have required important implementation efforts in Haskell, as well as some new patterns of its own. In this section, we discuss a selection of such topics.

5.1 Standing on the shoulders of records

We discussed in the introduction how modelling type classes using an existing powerful record mechanism such as Agda's dependently typed records makes certain features available "for free" that require separate extensions for Haskell type classes. Sozeau et al. and Oliveira et al. have previously demonstrated this observation for Coq type classes (which are Coq dependently typed records underneath [24]) and Scala implicits (where type classes are typically modelled as Scala traits [2]).

One such feature is the equivalent of Haskell's associated type families [23]. An associated type family is essentially a type class member that is a type or type functor. Using a dictionary model of a type class in a dependently typed language, there is nothing special about records with members that are not just values and we essentially get associated type families for free.

Another feature which we get "for free" is known as constraint families for Haskell [18]. A constraint (synonym) family in Haskell is a member of a type class that represents a class constraint on a type class's parameters and/or other types. Using a dictionary model of type classes, this concept actually reduces to type families. Orchard and Schrijvers' example of constrained functors (positive type functors whose fmap function is restricted to types in a certain type class) can be modelled as follows:

```
record ConstrainedFunctor (F : Set → Set) : Set where
  field Constraint : Set → Set
        fmap : ∀ {A B : Set} → Constraint A →
          Constraint B → (A → B) → F A → F B
listConstrainedFunctor : ConstrainedFunctor List
listConstrainedFunctor = record {
  Constraint = λ _ → ⊤
 , fmap = λ _ _ → List.map}
setConstrainedFunctor : ConstrainedFunctor Set
setConstraintedFunctor = record {
  Constraint = Ord, fmap = Set.map}
```

5.2 Multi-parameter type classes and functional dependencies

A multi-parameter type class in Haskell is (obviously) a type class with more than one parameter. The equivalent in our approach would be instance arguments of a record type with more than one parameter, and this is clearly allowed in our system. Functional dependencies in a multi-parameter type class are annotations which indicate that certain parameters of a type class can be deduced from (a subset of) the others [9]. Such an annotation cannot directly be provided in our framework. However, in this section, we highlight certain behaviour of our system, which is reminiscent of functional dependencies, even though it works differently under the hood.

Consider the following code, which uses the IsDecEquivalence record from module Relation.Binary in the Agda standard library. We use explicit **using** declarations to avoid certain name clashes, but also to make it more clear what is happening under the hood.

```
open import Relation.Binary using (module DecSetoid;
  module IsDecEquivalence)
open import Data.Bool using (false; true; decSetoid)
open DecSetoid decSetoid using (isDecEquivalence)

open IsDecEquivalence {{...}} using (_≟_)

test = false ≟ true
```

The IsDecEquivalence t _≈_ record is semantically a more developed version of the record Eq from the introduction, containing essentially an equality decision procedure _≟_ for a binary predicate _≈_ on type t (as well as proof that _≈_ is an equivalence relation). The field _≟_ has the following type:

```
_≟_ : (a : t) → (b : t) → Dec (a ≈ b)
```

A value of type Dec (a ≈ b) contains either a proof that a ≈ b or a proof that a ≉ b. We can bring a value of type IsDecEquivalence Bool _≡_ in scope by importing Data.Bool and opening the decSetoid record (this would be more convenient if isEquivalence were exported directly by the Data.Bool module). Finally, we bring the new record field projection operator (taking the record as a instance argument) into scope by importing it from the IsDecEquivalence {{...}} module application (see section 2.4). From that point on, we can transparently use the function $\_\overset{?}{=}\_$ on Bools, as demonstrated in the definition of test.

A first thing to note is that the IsDecEquivalence record takes two arguments, making it the equivalent of a multi-parameter type class. It is interesting to consider what happens when type-checking the definition of test. The function $\_\overset{?}{=}\_$ has the following type (ignoring universe polymorphism):

$$\begin{aligned}\_\overset{?}{=}\_ \;:\; &\{\mathsf{A} : \mathsf{Set}\} \to \{\_\approx\_ : \mathsf{A} \to \mathsf{A} \to \mathsf{Set}\} \to \\ &\{\{\mathsf{isDE} : \mathsf{IsDecEquivalence}\ \mathsf{A}\ \_\approx\_\}\} \to \\ &(\mathsf{a} : \mathsf{A}) \to (\mathsf{b} : \mathsf{B}) \to \mathsf{Dec}\ (\mathsf{a} \approx \mathsf{b})\end{aligned}$$

When false $\overset{?}{=}$ true is type checked, Agda infers that A = Bool from the arguments of $\_\overset{?}{=}\_$. It then infers the instance argument isDE from the local scope. The only candidate value in scope is isEquivalence, typed IsDecEquivalence Bool $\_\equiv\_$. From unifying the type of this value with the expected type of isDE, Agda infers that the implicit argument $\_\approx\_$ must be the binary predicate $\_\equiv\_$.

In this case, we see that one argument of the IsDecEquivalence type constructor already uniquely determines the value to be used from the scope. Its other arguments can then be inferred from this value, producing an effect similar to a hypothetical situation where IsDecEquivalence were a multi-parameter type class with a functional dependency from type A to binary predicate $\_\approx\_$.

Nevertheless, our mechanism works very differently from

Haskell type classes with functional dependencies. First of all, nowhere have we declared any functional dependencies between arguments of the IsDecEquivalence record type, and these dependencies were not checked when we brought values of type IsDecEquivalence into scope. Only when we actually needed to infer an instance argument, it was checked that only a single suitably-typed value was in scope.

Semantically, declaring the equivalent of a functional dependency on the arguments of the IsDecEquivalence record type corresponds to an assertion that only one decidable equality predicate can exist for any given type A. This assertion is semantically wrong here and can cause problems in scenarios where multiple such predicates are used together. Our system manages to infer the value of the $\_\approx\_$ predicate without such a dependency, because only one value of type IsDecEquivalence Bool $\_\approx\_$ is *in scope* at the call site of $\_\overset{?}{=}\_$, which is semantically a much weaker requirement.

Note finally that it is a value, not a type, that is being inferred in a functional dependencies-like way. In fact, our mechanism does not make any fundamental distinction between types or values, as one might expect in a dependently-typed language like Agda. The mechanism will even happily infer types from values, which is not possible in Haskell.

5.3 Implicit Configurations

One pattern implemented in the context of type classes which is rendered almost trivial in the context of our proposal is Kiselyov and Shan's implicit configurations [11]. The authors discuss a solution to what they call the *configurations problem*: propagating run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation. Their main example concerns modular arithmetic: they want to be able to build expressions in modular arithmetic which are parameterised over a concrete modulus but without the need to pass the modulus around explicitly. They also want static assurance that the same modulus is used for all operations in such an expression.

Kiselyov and Shan's solution is based on a mix of phantom types, type classes and type-level computation. We demonstrate that a simpler encoding is possible in our system, and that we can even fully avoid one of the main difficulties in their work: the reflection at type-level of run-time values. Let us suppose that we have a signature like the following: we assume an Integral dictionary record and add, mul and obj operations taking such a dictionary as a instance argument. We also assume we have a type N containing values zero, one, two and three and a dictionary nInt of type Integral N.

```
postulate
  Integral : Set → Set
  add  : ∀ {A} {{intA : Integral A}} → A → A → A
  mul  : ∀ {A} {{intA : Integral A}} → A → A → A
  mod  : ∀ {A} {{intA : Integral A}} → A → A → A
  N : Set
  zero one two three : N
  nInt : Integral N
```

Like Kiselyov and Shan, we define a wrapper data type M s A parameterised by phantom token s (in our case not a type but a value of opaque type Token) and type A. This wrapper represents a value of type A that is being considered under an unspecified modulus. We also define a dictionary record Modulus s A (also parameterised by a token s and type A).

```
private postulate Token : Set
```

```
record Modulus (s : Token) (A : Set) : Set where
  field modulus : A

data M (s : Token) (A : Set) : Set where
  MkM : A → M s A

unMkM : ∀ {s A} → M s A → A
unMkM (MkM a) = a
```

Our withModulus function is simpler than Kiselyov and Shan's because we don't have to bother with constructing a type for which the Modulus instance returns a certain value, but instead just pass the desired dictionary explicitly.

```
private postulate theOnlyToken : Token

withModulus :
  ∀ {A} → {{intA : Integral A}} → (modulus : A) →
  (∀ {s} → {{mod : Modulus s A}} → M s A) → A
withModulus modulus f = unMkM $
  f {theOnlyToken} {{record {modulus = modulus}}}
```

Our addition and multiplication become pretty similar to Kiselyov and Shan's version:

```
normalize : ∀ {s A} {{intA : Integral A}}
                {{mod : Modulus s A}} → A → M s A
normalize a = MkM $ mod modulus a

_+_ : ∀ {s A} {{intA : Integral A}}
  {{mod : Modulus s A}} → M s A → M s A → M s A
(MkM a) + (MkM b) = normalize $ add a b

_*_ : ∀ {s A} {{intA : Integral A}}
  {{mod : Modulus s A}} → M s A → M s A → M s A
(MkM a) * (MkM b) = normalize $ mul a b
```

These operators are used similarly to Kiselyov and Shan's:

```
test₁  : N
test₁  = withModulus two $
           let o = MkM one in (o + o) * (o + o)
testExpr : ∀ {s} → {{mod : Modulus s N}} → M s N
testExpr = let o = MkM one; t = MkM two
             in (o + t) * t

test₂  : N
test₂  = withModulus three testExpr
```

With this, our encoding of Kiselyov and Shan's implicit configurations is done. We believe that we achieve the same goals as Kiselyov and Shan, but we avoid their threading of values into types (through an involved type-level reflection of values) and back again (through a form of type-level computation), which seems unneeded, very complex and possibly inefficient (depending on what optimisations the compiler can perform). Interestingly, the fact that we don't need to reflect values at the type level is not (as one might expect) a consequence of Agda's dependently typed nature. Instead, it is the value-level representation of dictionaries which allows this greater simplicity. More concretely, in the definition of withModulus above, we can construct the dictionary as a value and pass it explicitly to the computation, whereas Kiselyov and Shan need to jump through a lot of hoops to construct a type for which the correct instance will be inferred. Kiselyov and Shan instead proposed adding a form of local instances to Haskell, of which we also support an equivalent (see section 5.5).

5.4 Implicit Proof Obligations

In the context of Agda, we believe that instance arguments are useful for a pattern which is (to the best of our knowledge) novel: implicit proof obligations. Consider the integer division operator in module Data.Nat.DivMod in Agda's standard library:

```
_div_ : (dividend divisor : ℕ)
            {≢0 : False (divisor ≟ 0)} → ℕ
```

This division operator requires a guarantee that the provided divisor is non-zero. However, instead of requiring a normal argument of type divisor ≢ 0, the _div_ operator cleverly accepts a value of type False (divisor ≟ 0). This type contains a single value if and only if divisor is non-zero, but additionally, this value can be automatically inferred if Agda knows that divisor is of the form suc n for some n. For example, if we write 5 div 3, then Agda will infer the non-zeroness proof obligation. This pattern has been described by Norell [15, 3.7.1 p.71], and critically depends on the fact that the property in question (non-zeroness) can be decided. Proof obligations modelled using this pattern are not passed on implicitly to other methods that require it. Finally, the _div_ operator becomes somewhat clumsy to use in a situation where only a normal proof divisor ≢ 0 is available.

We propose an additional operator _div′_ which takes the proof obligation as an instance argument (we omit the definition in terms of the above _div_). This operator does not have the limitations of the _div_ operator discussed above, but does have some limitations of its own: for example in the call 5 div′ 3, Agda can only infer the implicit argument of our operator if a value of type 3 ≢ 0 is in scope.

```
_divMod′_ : (dividend divisor : ℕ)
               {{≢0 : divisor ≢ 0}} → ℕ × ℕ
_divMod′_ =   -- omitted
```

```
_div′_ : (dividend divisor : ℕ) {{≢0 : divisor ≢ 0}} → ℕ
a div′ b with a divMod′ b
a div′ b | (q, _) = q

postulate
  d : ℕ
  d≢0 : d ≢ 0

test : ℕ
test = 5 div′ d
```

Note how in the definition of _div′_, the proof obligation is implicitly passed on to the _divMod′_ function, which also requires it. We believe that this example shows that our proposed instance arguments have uses that go beyond those of type classes. Not only dictionary records can be usefully passed around implicitly but also other values which are uniquely identified by their type in call-site scopes. In a dependently typed language like Agda, implicit proof obligations are a clear example of such values.

5.5 Local instances

A feature that is not supported by Haskell type classes are local type class instances. Consider the following two equality functions on Strings: the first represents standard equality and the second only compares the strings' lengths. The first definition uses the standard string equality decision procedure and the second applies the EqInst.eq operator after first applying a string length function to its two arguments. Note that we assume a single, standard value of type Eq ℕ in scope.

```
eqString₁ : String → String → Bool
eqString₁ s₁ s₂ = ⌊ s₁ ≟ s₂ ⌋
eqString₂ : String → String → Bool
```

```
eqString₂ = eq on length
```

Now suppose that we have a function whose behaviour depends on a configuration argument, determining which type of equality it should use throughout a series of calculations. We can support this by defining the equivalent of a *local instance* eqLocal of the Eq *type class*, which uses the correct string equality operator, depending on the configuration parameter.

```
test : Bool → Bool
test lengthEq = if eq "abcd" "dcba" then ... else ...
  where eqLocal = record {eq =
    if lengthEq then eqString₂ else eqString₁ }
```

The value eqLocal functions as a local type class instance, something which is also supported by Scala implicits, but not by Haskell or Coq type classes, where type class instances are always global.

5.6 Two final examples

As a small encore in this section, we can't resist discussing two code snippets using instance arguments. The first is an example of a function abstracting over functions with implicit arguments. It demonstrates the first-class nature of our new type of arguments: functions with instance arguments can be abstracted over, their types can be written out etc.

```
explicitize : ∀ {A : Set} {B : A → Set} →
  ({{x : A}} → B x) → (x : A) → B x
explicitize f x = f {{x}}
```

Our final example is small, but very useful: it is an analog of Agda's standard underscore construct for instance arguments, similar to Scala's implicitly or ?. Like in Scala, we don't need to introduce special syntax for this: the following definition suffices. This ellipsis can be used as a shorthand in any situation where only a single type-correct value is in scope. Because our resolution algorithm does not require candidates to be specially annotated to be eligible, our ellipsis is more generally useful than Scala's implicitly.

```
··· : {A : Set} → {{a : A}} → A
··· {{a}} = a
```

6. Related Work

There exists a lot of literature about type classes, extensions of them and alternatives to them [2, 5, 6, 9, 10, 18, 20, 23, 24, 26, 27]. We have already discussed Haskell type classes and Scala implicits in the introduction and we do not repeat this here.

We do not further discuss implicit parameters as implemented in Hugs and GHC [13], as these use a name-based resolution, instead of the type-based resolution of our design and are thus not suited for our use cases.

6.1 Coq Type Classes

Sozeau and Oury have recently presented Coq type classes [24]. Coq is a dependently typed, purely functional programming language/proof assistant like Agda, with a longer history and a larger user base. Unlike Agda, it has a very principled core language and associated type-checker. On top of that, there is a variety of arguably less principled language features and meta-programming/proof automation facilities.

The authors introduce type classes as essentially a new way to define dependently typed record types. If a function has an implicit argument of such a record type, and its value cannot be inferred through Coq's standard unification, then Coq will try to infer a value through an instance search. This instance search is implemented as an automated proof search using a special-purpose port of the `eauto` tactic. It performs a bounded breadth- or depth-first search using the type class's instances as lemma's. This can include both direct instances (objects of the record type) and parameterized instances (functions which take certain arguments and return such an object).

Sozeau and Oury go on to discuss some superficial syntax extensions and relatively straightforward models of superclasses and substructures and then provide a discussion of various aspects of their system, most importantly their instance search tactic. They think their current instance search tactic is not sufficient in the context of multi-parameter type classes and arbitrary instances (which their system currently allows). They state the algorithm's non-determinism and impredictability as problems which they hope to address in the future by restricting the shape of allowed instances and using a more predictable algorithm.

In addition to these problems, we believe that Sozeau et al.'s instance search procedure is currently at least as powerful as Haskell's or Scala's instance/implicit search and exposes the same kind of separate computational model (see section 4). Also, if we understand Sozeau and Oury's text correctly, a given implicit function argument can sometimes behave as a type class constraint and sometimes as a normal implicit argument, if its type depends on the value of previous arguments. Our resolution algorithm is less powerful than Sozeau et al.'s, but it is predictable, deterministic and does not expose an alternate computational model.

Sozeau and Oury's mechanism is limited to *record types* that were *defined as a type class*, so existing libraries need to be adapted to benefit from it. Type class instances can be defined locally (see section 5.5), but it seems that the local instance will not be considered for automatic resolution.

6.2 Coq Canonical Structures

Coq features another type system concept which can be exploited as an alternative to type classes: *canonical structures* [1, 25]. This feature allows certain values of a record type to be marked as *canonical structures*. Such values are then automatically considered when the Coq type inferencer tries to infer a value of the record type from the value of one of its fields. Canonical structures have existed for some time in Coq, but have recently attracted the attention of authors looking to provide easy to use libraries of complex concepts, exploiting canonical structures as a powerful meta-programming feature that can be implicitly triggered to resolve values in the background. There are some similarities in the design to ours, as it does not introduce a separate type of structure and does a form of implicit resolution from call-site scope. However, because the resolution is keyed on values instead of types, the idea is not suitable for non-dependently typed languages, where we can only reason about types at compile time.

However, the entire design of the feature seems very pragmatic. We have not been able to find a detailed (formal?) description of the exact workings of the system. From what we understand, it is deeply coupled to Coq's type inference engine and uses certain syntactic criteria, behaving differently for semantically equivalent values. There seems to be a certain interaction with a form of backtracking in Coq's type inference engine, which can be exploited for encoding backtracking in the specification of how values should be inferred. All of this leads to a meta-programming model that seems even more powerful than Haskell's instance resolution, hard to understand and strongly different from Coq's standard computational model(s).

6.3 Explicit Haskell

In an unpublished technical report, Dijkstra and Swierstra describe an implicit arguments system which they have implemented in a Haskell variant called *Explicit Haskell* [4]. Their main motivation is that Haskell does not provide a way to override the automatic resolution of instances (dictionaries) for functions with a type class constraint. They extend Haskell with named instances, local instances, and a way to explicitly provide an instance to a function with a type class constraint, either by naming the instance or by lifting a value of a record type corresponding to the type class. They also allow type class constraints to appear anywhere in a type, not just at the beginning. For resolving type class constraints, they use a resolution close to Haskell's. The only difference is that instances can be annotated to not take part in this resolution (in which case they can only be used by name). In the same text, Dijkstra and Swierstra discuss a system for partial type signatures, which seems to have independent value. The system allows the user to partially specify types for values and leave the rest to be inferred.

This design has many similarities to our system. Their extensions to the concept of type class constraints effectively transforms them into a special form of function arguments similar to our instance arguments. Their design offers some of the same benefits as ours (e.g. local instances, named instances), and they discuss some of the same examples as we do in section 5.

However, they make some different choices than we do: their constraints remain limited to arguments whose type was defined as a type class instance and their resolution is similar to Haskell's. They do not fully unify type classes with their associated record types, so that some of the advantages we can offer are not available (e.g. abstracting over a type class).

6.4 Modular Type Classes

Dreyer et al. discuss an alternative to type classes in the context of ML [5]. They share our view that Haskell type classes duplicate functionality by introducing a separate structuring concept, and they argue that ML modules already provide functionality akin to associated type families and type class inheritance (like we do for ADT's). They propose to model single-parameter type classes as *class signatures*: module signatures with a single abstract type named `t`. Instances become modules and functors. Their primitive **overload** `fun` **from** `sig` returns a version of function `fun` from class signature `sig` that will resolve the appropriate module implementing `sig` from call-site scope. Another primitive `canon (sig)` resolves and returns this module.

Resolution of such a module takes into account modules and functors that have been annotated in the current scope with a **using** declaration. Since functors are considered, the instance search is recursive, and can likely be exploited as a type-level programming primitive similar to Haskell's instance search, even though Dreyer et al do not discuss this. Their proposal does not support the equivalent of multi-parameter type classes. It is not clear to us if and how their type class modules can be abstracted over.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven. Dominique Devriese holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO). The authors thank Adriaan Moors (EPFL) for providing valuable feedback on a draft of this text, as well as clarifying various points about Scala implicits. We thank the anonymous reviewers for many valuable comments and PC Chair Olivier Danvy for suggesting the final title.

References

[1] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, 2008.

[2] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.

[3] N. Danielsson, U. Norell, S. Mu, S. Bronson, D. Doel, P. Jansson, and L. Chen. The Agda standard library, 2009.

[4] A. Dijkstra and S. D. Swierstra. Making implicit parameters explicit. Technical Report UU-CS-2005-032, Dept. ICS, Utrecht University, 2005.

[5] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *POPL*, 2007.

[6] D. Duggan and J. Ophel. Type-checking multi-parameter type classes. *Journal of Functional Programming*, 12(02):133–158, 2002.

[7] B. Heeren, J. Hage, and S. Swierstra. Scripting the type inference process. In *ACM SIGPLAN Notices*, volume 38, 2003.

[8] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *History of Programming Languages*, 2007.

[9] M. Jones. Type classes with functional dependencies. *PLAS*, pages 230–244, 2003.

[10] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In *HASKELL*, 2001.

[11] O. Kiselyov and C.-c. Shan. Functional pearl: Implicit configurations. In *HASKELL*, 2004.

[12] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP*, pages 204–215, 2005.

[13] J. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.

[14] C. McBride. Faking it: Simulating dependent types in haskell. *Journal of Functional Programming*, 12(4&5), 2002.

[15] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[16] U. Norell and C. Coquand. Type checking in the presence of meta-variables. Unpublished draft., 2007. URL `http://www.cse.chalmers.se/~ulfn/papers/meta-variables.html`.

[17] M. Odersky. The Scala language specification, version 2.8. online, 2010. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[18] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *FLOPS*, 2010.

[19] S. Peyton Jones and R. Lämmel. Scrap your boilerplate. In *APLAS*, page 357, 2003.

[20] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *HASKELL*, 1997.

[21] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, 2006.

[22] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1), 2007.

[23] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP*, 2008.

[24] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, 2008.

[25] The Coq development team. The Coq reference manual. online, 2010. URL `http://coq.inria.fr/refman/`. v8.3.

[26] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad

hoc. In *POPL*, 1989.

[27] S. Weirich. Replib: a library for derivable type classes. In *HASKELL*, 2006.

A. Under the hood

In this appendix, we discuss the precise changes we make in more detail. The definitions in this section are extensions and adaptations of Norell's rules for current Agda type-checking and standard implicit arguments [15, 3.5 pp. 69–70, 5.1.5 pp. 99–100]. They should be read in the context of Norell's developments and may not be fully clear without them.

A.1 Implicit lambdas

We add another implicit function space $\{\!\{ \mathsf{x} : \mathsf{A} \}\!\} \to \mathsf{B}$ in addition to the existing $\{\mathsf{x} : \mathsf{A}\} \to \mathsf{B}$ and $(\mathsf{x} : \mathsf{A}) \to \mathsf{B}$. Like the existing implicit functions, the new functions are semantically equivalent to the corresponding ordinary functions. Values of type $\{\!\{\mathsf{x} : \mathsf{A}\}\!\} \to \mathsf{B}$ can be introduced as (typed or untyped) lambda values $\lambda\ \{\!\{\mathsf{x}\}\!\} \to \mathsf{e}$ or $\lambda\ \{\!\{\mathsf{x} : \mathsf{A}\}\!\} \to \mathsf{e}$ or they can be defined as constants (at the top-level or in where-clauses).

For type-checking values of this type, we extend the rules for Agda's standard implicit arguments [15, 3.5 pp. 69–70] as follows. If a value does not explicitly mention an instance argument from the type it is checked against, rule (2) infers implicit lambda's, like for normal implicit arguments.

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s}{\Gamma \vdash \lambda\{\!\{x\}\!\}.e \uparrow \{\!\{x : A\}\!\} \to B \rightsquigarrow \lambda\{\!\{x\}\!\}.s} \tag{1}$$

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s \qquad e \neq \lambda\{\!\{x\}\!\}.e'}{\Gamma \vdash e \uparrow \{\!\{x : A\}\!\} \to B \rightsquigarrow \lambda\{\!\{x\}\!\}.s} \tag{2}$$

A.2 Instance arguments

Next, we need to determine when instance arguments of a function are not provided explicitly and should be inferred. This mechanism is governed by the inference rules for argument checking judgements of the form $\Gamma \vdash A @ \bar{e} \downarrow B \rightsquigarrow \bar{s}$. Such a judgement means that the values $\bar{e}$ can be passed as arguments to a value of type A, producing a value of type B. The full list of arguments to be applied to the function (including implicitly inserted ones) is "returned" in $\bar{s}$.

We extend the corresponding rules for implicit arguments [15, 3.5 p. 70] as follows. For a non-provided instance argument, we do not just insert a meta-variable α, but we additionally add a constraint $\text{FindInScope}\,\alpha$. This is a special kind of constraint that indicates that α should be resolved as an instance argument. To do this, we need to extend the form of argument checking judgements to additionally return a set of constraints $\mathcal{C}$: $\Gamma \vdash A @ \bar{e} \downarrow B \rightsquigarrow \bar{s}, \mathcal{C}$. This adapted form actually corresponds more closely to Agda's existing implementation of the rules. Existing rules in the old form of the judgement should now be read as simply producing no constraints or simply pass generated constraints through if they recurse.

$$\frac{\Gamma \vdash e \uparrow A \rightsquigarrow s \qquad \Gamma \vdash B[x := s] @ \bar{e} \downarrow B' \rightsquigarrow \bar{s}, \mathcal{C}}{\Gamma \vdash \{\!\{x : A\}\!\} \to B @ \{\!\{e\}\!\}; \bar{e} \downarrow B' \rightsquigarrow s; \bar{s}, \mathcal{C}} \tag{3}$$

$$\begin{array}{c} \text{AddMeta}(\alpha : \Gamma \to A) \qquad\qquad \bar{e} \neq \{\!\{e\}\!\}; \bar{e}' \\ \Gamma \vdash \{\!\{x : A\}\!\} \to B @ \{\!\{\alpha\}\!\}; \bar{e} \downarrow B' \rightsquigarrow \bar{s}, \mathcal{C} \end{array}$$

$$\frac{}{\Gamma \vdash \{\!\{x : A\}\!\} \to B \,@\, \bar{e} \downarrow B' \rightsquigarrow \bar{s}, \mathcal{C} \cup \{\text{FindInScope}\,\alpha\}} \tag{4}$$

We change the last rule on [15, p. 70] to the following:

$$\frac{A \neq \{\!\{x : A_1\}\!\} \to A_2 \qquad A \neq \{x : A_1\} \to A_2}{\Gamma \vdash A \,@\, \epsilon \downarrow A \rightsquigarrow \epsilon, \{\}} \tag{5}$$

A somewhat technical point here is that at the moment, we do not allow meta-variables introduced for instance arguments to be η-expanded, as this is done for Agda's normal implicit arguments. We take a conservative approach because we currently do not have a good understanding of possible interactions between η-expansion and instance resolution. During our experiments, we have established that all of them (see section 3 and 5) have worked well without η-expansion. It is future work to get a better understanding of the issues involved.

A.3 Resolution algorithm

The resolution of a constraint FindInScope α in context Γ and scope S with $\Gamma \vdash \alpha : A$ tries to infer a value from either the values in the current context Γ or the constants in scope S. If only one candidate is found in both sets, it is selected. If there is more than one candidate, resolution of the constraint is postponed in the hope that more type information will become available further on, reducing the set of candidates further. If the constraint is not resolved when type checking finishes, this is reported to the user. If there are no candidates, then the constraint cannot be solved and this is also reported.

To formalise these rules, we need some extra information about meta variables introduced through inference rule (4) above: the context and scopes at the point where they were defined. We do not make this change explicit because the context is actually already implicitly being maintained throughout Norell's development [15], and because both the scope and the context are already being kept in the Agda implementation. For a meta variable α, we write $\mathrm{MScp}\,\alpha$ and $\mathrm{MCtx}\,\alpha$ for the scope resp. the context in which a meta variable α was introduced.

With these nuances, we can formally define how we solve constraints $\mathrm{FindInScope}\,\alpha$ as follows:

$$\frac{\begin{array}{c}\mathrm{Lookup}(\alpha : A) \\ \mathrm{Candidates}(\mathrm{MCtx}(\alpha), \mathrm{MScp}(\alpha), \alpha, A) = \{(n, A_n)\} \\ \Gamma \vdash A \simeq A_n \rightsquigarrow \mathcal{C} \qquad\qquad \alpha := n\end{array}}{\Gamma \vdash \mathrm{FindInScope}\,\alpha \rightsquigarrow \mathcal{C}} \tag{6}$$

This definition says that if we have a meta-variable α typed A, that is to be inferred as an instance argument, then it is resolved if there is a unique solution. In this case, we require convertibility of the types and assign the value to the meta-variable. The set of candidates in context Γ and scope S, for meta-variable α, of type A is defined by predicate Candidates:

$$\begin{array}{l}\mathrm{Candidates}(\Gamma, S, \alpha, A) = \\ \quad \{(n, A_n) \mid \mathrm{Cand}(\Gamma, S, n, A_n) \text{ and } \mathrm{ValidCand}(\alpha, A, n, A_n)\}\end{array} \tag{7}$$

The candidates are those terms n, of type A_n, that are potential candidates in the current context and scope (predicate Cand below) and are valid with respect to the current meta-variable and its type. This last property is defined by the ValidCand predicate.

$$\frac{\langle\Sigma\rangle\ \mathrm{CheckCand}(\alpha, A, n', A') \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle}{\langle\Sigma\rangle\ \mathrm{ValidCand}(\alpha, A, n', A') \Longrightarrow \langle\Sigma\rangle} \tag{8}$$

In the definition of this predicate, we perform a check, but if this check makes changes to the current signature, we do not yet carry them through here. This is formalised using the explicit notation of the signatures in the judgements [15, 3.3.1 p. 54].

$$\Gamma \vdash A \simeq A' \rightsquigarrow \mathcal{C} \qquad \alpha := n \qquad \mathrm{CurrentConstraints}(\mathcal{C})$$

$$\frac{\forall C \in \mathcal{C} : C \neq \mathrm{FindInScope}\,\alpha' \Rightarrow \neg(\nvdash C \rightsquigarrow \mathcal{C}')}{\mathrm{CheckCand}(\alpha, A, n', A') \rightsquigarrow \mathcal{C}} \tag{9}$$

The check that a certain term is valid for a certain meta-variable consists of two parts. First, the term's type must be convertible to the required type. Second, if we assign the value to the meta-variable, no other constraint must be immediately invalidated. For this last check, we do not recursively consider other FindInScope constraints, since this would introduce recursion in the instance search. This check is necessarily incomplete: in Norell's words, the type checker will give one of three answers [15, Note p. 65]: "yes it is type correct", "no it is not correct" or "it might be correct if the metavariables are instantiated properly". Only if we get the second answer, we reject the candidate under scrutiny.

Rule (9) strikes a fine balance. On the one hand, the resolution algorithm needs to be powerful enough to be usable, but we avoid making it too powerful (see discussion in section 4). The intuition behind the criterion above is that we consider any value that is type-correct in the sense that it has the correct type, but also in the sense that it does not immediately invalidate constraints. The criterion has proven sufficient for all use cases discussed in this text, but also necessary: without the check for invalidated constraints, monad instances for example are often not resolved. Note that we have used a new CurrentConstraints operation, which works on the signature that is implicit in the typing judgements:

$$\langle \Sigma \rangle \ \mathrm{CurrentConstraints}(\mathcal{C}) \Longrightarrow \langle \Sigma \rangle \text{ where } \mathcal{C} = \{C \mid C \in \Sigma\} \tag{10}$$

We still need to define the potential candidates in a given context and scope. The Cand property formalises this:

$$\frac{\Gamma = \Gamma_1; n : A; \Gamma_2}{\mathrm{Cand}(\Gamma, S, n, A)} \tag{11}$$

$$\frac{\mathrm{Visible}_{\mathrm{pri}}(n, S) \qquad \mathrm{Lookup}(n : A)}{\mathrm{Cand}(\Gamma, S, n, A)} \tag{12}$$

The somewhat technical predicate $\mathrm{Visible}_p(n, S)$ asserts that name n is publicly or privately (defined by p) in scope S:

$$\frac{\mathrm{Visible}_{\mathrm{pub}}(n, S)}{\mathrm{Visible}_{\alpha}(n, S \blacktriangleright \sigma)} \tag{13}$$

$$\frac{\mathrm{VisibleNS}_{\alpha}(n, \sigma) \vee \mathrm{VisibleNS}_{\mathrm{pub}}(n, \sigma)}{\mathrm{Visible}_{\alpha}(n, S \blacktriangleright \sigma)} \tag{14}$$

$$\frac{n \in ns_{\alpha}(cn)}{\mathrm{VisibleNS}_{\alpha}(n, \langle M, ns_{\mathrm{pub}}, ns_{\mathrm{pri}} \rangle)} \tag{15}$$

From rules (6) and (9) above, it is clear that resolution of constraints FindInScope α only compares types that have already been type checked, and does not trigger extra type checking. Therefore, only one constraint FindInScope α will be resolved per occurence in the user's code of a function taking an instance argument without a value being provided explicitly. This means that, contrary to other proposals, the computational power of our resolution algorithm is fundamentally limited, in the sense that it does not allow any form of recursion, looping or backtracking. It therefore does not introduce a separate computational model in the language (see section 4).

A.4 Soundness

Intuitively, soundness of the rules above is easily guaranteed, because all we do is assign terms of the correct type to meta-variables.

The following lemma reflects this intuition, supplementing Norell's Lemma 3.5.13:

Lemma 1 (Instance resolution preserves consistency). *If* $\Gamma \vdash_{|\Sigma|}$ **valid**, Σ *is consistent and*

$$\langle \Sigma \rangle\ \Gamma \vdash \mathrm{FindInScope}\, \alpha \Longrightarrow \langle \Sigma' \rangle$$

then Σ' *is consistent.*

Proof. A consequence of Norell's Lemma 3.5.12 (Refinement preserves consistent signatures), together with the observation that rule (6) will only ever perform a type correct assignment of a meta-variable (a signature refinement). □

This lemma suffices to establish that Norell's Lemma 3.5.14 (Constraint solving is sound) stays valid in the context of our new kind of constraints, as well as the main result, Theorem 3.5.18 (Soundness of type checking).

Like Norell for normal implicit arguments, we provide formal rules for the insertion of instance arguments and the insertion of instance lambda's, but do not prove any formal results about them.

Some of the rules above may give the impression that this resolution algorithm is sensitive to the order in which type-checking is interleaved with constraint solving. However, this sensitivity actually only exists for error reporting. Remember that during type-checking, constraints will only be added and solved (after a correct meta-variable assignment), but they cannot otherwise be removed. As a consequence of this, the candidates set for a given instance argument meta-variable α, defined by rule (7) above, form a descending series during type-checking: later sets are subsets of previous ones. Furthermore, if a value in scope can be assigned to α such that the entire Agda expression succesfully type checks, then this value will be contained in all of these candidate sets. All non-valid candidates will eventually be removed. Therefore, if no other valid candidates are available, the valid value will inevitably be chosen.

For erroneous programs, the order of constraint solving may determine the kind of error message that is generated. Depending on whether a constraint C is registered after a certain instance argument is already resolved, or before, an error will be reported for the $\mathrm{FindInScope}$ constraint or the constraint C. This influence of type-checking on error reporting also exists for standard Hindley-Milner type inferencing [7], so we consider it acceptable.

One extension of the current resolution scheme that we have considered in detail is based on a prioritisation of candidates, e.g. by giving precedence to values defined closer to the call site. However, contrary to our current resolution algorithm, such a prioritisation does make the result of instance resolution depend on the order of constraint resolution. Suppose there is a value in the highest priority set which is valid except for a constraint produced late during type checking and suppose this is the only candidate at the highest priority, but a lower priority candidate is also valid, and does not invalidate the late constraint. Since we don't know upfront which constraints will be produced during the rest of type-checking, we have to decide at some point which value to use. If the late constraint has then not yet been produced, the highest priority candidate will be selected and a type error will be reported when the late constraint is finally encountered. However, if the resolution occurs after the production of the late constraint, the valid low-priority candidate is chosen instead of the invalid high-priority one, and all goes well.

We currently do not see a solution for this problem, so we keep the introduction of a prioritised resolution algorithm as future work. Our experiments (see section 3 and 5) show that the current non-prioritised resolution scheme suffices for real use.