

# Explicit implicit parameters

Atze Dijkstra and S. Doaitse Swierstra

June 2, 2005

- ▶ 'Explicit': corresponds to program text specified by programmer
- ▶ We all are familiar with it
- ▶ Haskell:  
 $square :: Int \rightarrow Int$   
 $square\ x = x * x$
- ▶ C:  

```
int square (int x){  
    return x * x;  
}
```
- ▶ ...

# Implicit parameters

- ▶ ‘Implicit’: not specified explicitly
- ▶ Also familiar?
  - ▶ Yes!
  - ▶ functions often are parameterized by data without the data being explicitly passed

- ▶ Haskell’s class system:

**class** *Num* *a* **where**

$(*) :: a \rightarrow a \rightarrow a$

**instance** *Num* *Int* **where**

$(*) = \text{primMullInt}$

*square* :: *Num* *a*  $\Rightarrow a \rightarrow a$

*square* *x* = *x* \* *x*

... *square* 2 ...

- ▶ Implicit parameter: how multiplication should be done
  - ▶ fully determined by language
  - ▶ type describes implicit behavior

# Implicit parameters

- ▶ C too, e.g. global values:

```
int factor = ...;
```

```
int square (int x){  
    return factor * x * x;  
}
```

- ▶ But the type does not include a description of this behavior
  - ▶ that's 'cheating'!

- ▶ Haskell views explicit and implicit parameter passing as separate
  - ▶ function requires implicit parameter?
  - ▶ programmer cannot directly pass a value for the implicit parameter
  - ▶ only indirectly via **instance** declarations, used by the language to automatically determine the proper implicit parameter
- ▶ Language defines what to pass implicitly
  - ▶ breaks when no automatic choice can be made
  - ▶ breaks when a wrong choice is made
- ▶ Our approach: provide the means to allow programmer and compiler jointly specify a program
  - ▶ language fills in the parts (types, implicit parameter passing) as far as it is capable of
  - ▶ programmer specifies the remaining parts
  - ▶ gradual shift between implicit and explicit

# Content of this talk

- ▶ Context
  - ▶ Haskell, EH
- ▶ Haskell's solution
- ▶ EH mechanism's for explicitly passing values for implicit parameters
- ▶ Partial type signatures

- ▶ Starting point: Haskell
  - ▶ already provides combination of strong type checking and class system
- ▶ Explicit implicit parameter passing situated in context of Explicit Haskell (EH)
  - ▶ 'as simple as possible' Haskell
  - ▶ while also providing extensions: higher ranked types, existentials, records
  - ▶ used for research and education

## ► Language constructs

- core ( $\lambda$ -calculus) of Haskell

$e ::= int \mid char$	literals
$i$	value variable
$e e$	application
$\lambda i \rightarrow e$	abstraction
<b>let</b> $\bar{d}$ <b>in</b> $e$	binding

- + extensions (records, higher rank polymorphism, existentials, ...)

$e ::= \dots$	
$(l = e, \dots)$	record
$(e \mid l := e, \dots)$	record update
$e.l$	record selection

# Haskell's class system

- ▶ Class defines a predicate over type(s)
  - ▶ together with values (class members)
  - ▶ which are available when predicate is satisfied

- ▶ Example: equality on values of type  $a$ :

**class**  $Eq\ a$  **where**

$(==) :: a \rightarrow a \rightarrow Bool$

- ▶ Predicate is part of type of value

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

$f = \lambda \quad x \quad y \rightarrow \mathbf{if}\ x == y\ \mathbf{then}\ 3\ \mathbf{else}\ 4$

# Haskell's class system

- ▶ Meaning (in practical terms):
  - ▶ for body of  $f$ :  $Eq\ a$  is satisfied hence  $==$  on values of type  $a$  can be used
  - ▶ for caller of  $f$ : has to prove  $Eq\ a$  if values of type  $a$  are passed
- ▶ Implementation via evidence for proof of satisfaction:
  - ▶ for body of  $f$ : is passed a dictionary (record) holding value for  $==$  (and other class members)
  - ▶ for caller of  $f$ : constructs and passes this dictionary
- ▶ Basic proofs (instances) given by programmer
  - instance**  $Eq\ Int$  **where**  
 $x == y = primEqInt\ x\ y$
  - instance**  $Eq\ Char$  **where**  
 $x == y = primEqChar\ x\ y$
- ▶ Used under the hood to construct dictionaries to be passed

## Haskell's class system

- ▶ Do it ourselves Haskell implementation

```
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 = λ dEq    x    y → if (eqEqD dEq) x y then 3 else 4
```

- ▶ Usual translation to internal machinery

# EH: explicit passing for implicit parameter

- ▶ Class and instance

```
let class Eq a where
```

```
  eq :: a → a → Bool
```

```
instance Eq Int where
```

```
  eq = primEqInt
```

...

- ▶ Dictionary is record, each field corresponds to member of class
  - ▶ class translates to record type
  - ▶ instance translates to record value

## Explicit passing for implicit parameter

- ▶ Parameter passing

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

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

$eqMod2 = \lambda x\ y \rightarrow eq\ (mod\ x\ 2)\ (mod\ y\ 2)$

**in**  $f$  3 4

$(!(eq = eqMod2) \llsim Eq\ Int!)$  5 6

- ▶ Predicate position in type determines parameter passing position

## Explicit passing for implicit parameter

- ▶ (! !) specifies value to be passed for an implicit parameter
  - ▶ implicitly passed: dictionary for  $Eq\ a$  for  $eq\ p\ q$
  - ▶ explicitly passed: dictionary for  $Eq\ b$  for  $eq\ r\ s$
- ▶  $(!(eq = eqMod2) \Leftarrow Eq\ Int!)$ 
  - ▶  $(eq = eqMod2)$  must be evidence for predicate  $Eq\ Int$
- ▶  $\Leftarrow$  appears as  $<:$  in program text
  - ▶ resembles  $::$  (explicit typing, type annotation)
  - ▶ resembles an arrow  $<-$  (leads to, proves)

# Overlapping instances

- ▶ Multiple instances for same predicate

**let instance** *Eq Int* **where**

*eq = primEqInt*

**instance** *Eq Int* **where**

*eq = eqMod2*

*f = λp q r s → ...*

**in** *f 3 4 5 6*

- ▶ Overlapping instances

- ▶ Which dictionary must be passed?
- ▶ Language definition does not specify a choice

- ▶ Solutions:

- ▶ give a name to the dictionary for each instance, use it to pass dictionary explicitly
- ▶ avoid multiple instances for use by implicit parameter mechanism
- ▶ allow scoped instances, a shadowing mechanism

- ▶ Bind dictionary for an instance to value identifier

```
let instance dEqInt1  $\Leftarrow$  Eq Int where
```

```
  eq = primEqInt
```

```
  instance dEqInt2  $\Leftarrow$  Eq Int where
```

```
    eq = eqMod2
```

```
    f =  $\lambda p q r s \rightarrow \dots$ 
```

```
in f (!dEqInt1  $\Leftarrow$  Eq Int!) 3 4
```

```
    (!dEqInt2  $\Leftarrow$  Eq Int!) 5 6
```

- ▶  $\Leftarrow$  binds to identifier + allows participation in underlying machinery
- ▶ At parameter passing location: override automatic decisions made by underlying machinery

# Selectively naming an instance

- ▶ Don't let instance participate in automatic choice for implicit parameter

```
let instance Eq Int where
```

```
  eq = primEqInt
```

```
  instance dEqInt2 :: Eq Int where
```

```
    eq = eqMod2
```

```
    f = λp q r s → ...
```

```
in f                                3 4
```

```
  (!dEqInt2 <~ Eq Int!) 5 6
```

- ▶ :: (only) binds to identifier

- ▶ Shadow previous instances

```
let instance dEqInt1  $\leftarrow$  Eq Int where ...
```

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

```
g =  $\lambda$ x y  $\rightarrow$  eq x y
```

```
in let v1 = g 3 4 -- (1)
```

```
    v2 = let instance dEqInt2  $\leftarrow$  Eq Int  
        in g 3 4 -- (2)
```

```
in ...
```

- ▶ **instance** dEqInt2  $\leftarrow$  Eq Int without **where**  
introduces dEqInt2 for use by internal machinery
  - ▶ shadows outer Eq Int instances
- ▶ Actual values used
  - ▶ (1): dEqInt1
  - ▶ (2): dEqInt2

# Instances which require other instances

- ▶ Equality on lists needs equality on elements

```
let data List a = Nil | Cons a (List a)
```

```
instance dEqInt  $\Leftarrow$  Eq Int where
```

```
  eq = primEqInt
```

```
instance dEqList  $\Leftarrow$  Eq a  $\Rightarrow$  Eq (List a) where
```

```
  eq =  $\lambda l_1 l_2 \rightarrow \dots$ 
```

```
  f ::  $\forall a. Eq a \Rightarrow a \rightarrow List a \rightarrow Bool$ 
```

```
  f =  $\lambda p q \rightarrow eq (Cons p Nil) q$ 
```

```
in f 3 (Cons 4 Nil)
```

## Instances which require other instances

- ▶ Dictionary for *List* instance needs dictionary for elements

- ▶ Translation:

```
let dEqInt  :: (eq :: Int → Int → Bool)
      dEqList :: ∀ a.(eq :: a → a → Bool)
                → (eq :: List a → List a → Bool)
      eq      = λdEq  x y → dEq.eq x y
      f      = λdEq_a p q → eq (dEqList dEq_a) (Cons p Nil) q
in f dEqInt 3 (Cons 4 Nil)
```

- ▶ *dEqList*: dictionary transformer

## Dictionary transformers

- ▶ Implicit variant

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

- ▶ Explicit variant

$$f :: \forall a. Eq\ a \Rightarrow a \rightarrow List\ a \rightarrow Bool$$
$$f = \lambda(!dEq\_a \Leftarrow Eq\ a!)$$
$$\rightarrow \lambda p\ q \rightarrow eq\ (!dEqList\ dEq\_a \Leftarrow Eq\ (List\ a)!) \\ (Cons\ p\ Nil)\ q$$

- ▶ Translated variant

$$f = \lambda dEq\_a\ p\ q \rightarrow eq\ (dEqList\ dEq\_a)\ (Cons\ p\ Nil)\ q$$

- ▶ We can do it ourselves, explicitly, if necessary!

## Dictionary transformers

- ▶ Are first class

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

- ▶ Here explicit passing may be omitted to achieve same effect
  - ▶ but done by underlying machinery

## Dictionary transformers

- ▶ Useful in class based implementation of generics

```
let data Bit          = Zero | One
data GRose f a = GBranch a (f (GRose f a))
concat ::  $\forall$  a. List a  $\rightarrow$  List a  $\rightarrow$  List a
in let class Binary a where
    showBin :: a  $\rightarrow$  List Bit
instance dBl  $\Leftarrow$  Binary Int where
    showBin = ...
instance dBl  $\Leftarrow$  Binary a  $\Rightarrow$  Binary (List a) where
    showBin = ...
instance (Binary a, ( $\forall$  b. Binary b  $\Rightarrow$  Binary (f b)))
     $\Rightarrow$  Binary (GRose f a) where
    showBin =  $\lambda$ (GBranch x ts)
               $\rightarrow$  concat (showBin x) (showBin ts)
in let v1 = showBin (GBranch 3 Nil)
in v1
```

## Partial type signatures

- ▶ Specifying full type signatures becomes cumbersome
- ▶ Idea:
  - ▶ programmer specifies explicitly what cannot be inferred
  - ▶ system infers the rest

- ▶ Fully explicit

$$f :: \forall a. Eq a \Rightarrow a \rightarrow a \rightarrow \forall b. Eq b \Rightarrow b \rightarrow b \rightarrow (Bool, Bool)$$
$$f = \lambda \quad \quad p \quad q \quad \quad r \quad s \rightarrow (eq p q, eq r s)$$

- ▶ If dictionary for  $Eq b$  needs to be passed before others

$$f :: \forall \quad b. (Eq b, \dots) \Rightarrow \dots \rightarrow \dots \rightarrow b \rightarrow b \rightarrow \dots$$

-- INFERRED:

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

- ▶ '...': explicit notation for missing type information to be inferred

## Partial type signatures

- ▶ Monomorphic as well as polymorphic

$$f :: \forall a. (Eq\ a, \dots) \Rightarrow a \rightarrow a \rightarrow \dots$$
$$f = \lambda \quad \quad \quad p \quad q \quad r \quad s \quad \rightarrow (eq\ p\ q, eq\ r\ 3)$$

-- INFERRED:

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

- ▶ Type variables for monomorphic types

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

-- INFERRED:

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

- ▶ Explicit mechanisms (for parameter passing and/or in general)
  - ▶ allow full control by programmer
  - ▶ but also burden the programmer
- ▶ Implicit mechanisms
  - ▶ allow the language to do 'boring' stuff for the programmer
  - ▶ but limit expressiveness if the programmer cannot intervene when the language fails
- ▶ Explicit and implicit
  - ▶ Haskell: black and white only
  - ▶ EH: also the grey in between
- ▶ The message: grey is good
  - ▶ coöperation with compiler instead of fighting against