Generic Programming with Dependent Types

Stephanie Weirich
University of Pennsylvania

Work in progress: Extending GHC to Agda

Material in this talk based on discussions with Simon Peyton Jones, Conor McBride, Dimitrios Vytiniotis and Steve Zdancewic

Outline of talk

- What generic programming is
- Why generic programming matters to dependently-typed programming languages
- Problems
- Extensions to improve Haskell

Generic Programming

- A truly Generic term? But what does it mean?
- To "lift algorithms and data structures from concrete examples to their most general and abstract form" (Stroustrup)
- Ok, how do we make algorithms and data-structures more abstract (in typed, functional programming languages)?

Generalize over values

- Add a new parameter to a function

```
onex   f x = f x
twox   f x = f (f x)
threex f x = f (f (f x))

nx   0 f x = x
nx   n f x = f (nx (n-1) f x)
```

Generalize over types

- Add a type parameter to a function

```
appInt :: (Int -> Int) -> Int -> Int
appInt f x = f x


appBool :: (Bool -> Bool) -> Bool -> Bool
appBool f x = f x


app :: (a -> b) -> a -> b
app f x = f x
```

Type-parametric function

Generalize over types

```
eqBool :: Bool -> Bool -> Bool
eqBool x y = …

eqNat :: Nat -> Nat -> Bool
eqNat x y = …

eq :: a -> a -> Bool
eq x y = if bool? x then eqBool x y
   else if nat? x then eqNat else error
```

Behavior of function depends on the type of the argument

Type-indexed function

Generalize over values

```
oneApp    f x  = f x
twoApp    f x  = f x x
threeAapp f x  = f x x x
```

Value-indexed type

```
nApp 0 f x = f
nApp n f x = nApp (n-1) (f x) x
```

The behavior of the function depends of the new argument

Type of the function depends on the new argument

Generic programming is a 'killer app' for dependently-typed languages

- **All** generalization patterns available in dependently-typed languages
  - Type-dependent types -> functions
  - Value-dependent types -> Strong elimination
  - Type-dependent programming -> Universe elimination
- Enabling technology: no distinction between compile-time (types) and runtime (terms)

Strong eliminators

- A function from values to types

```
NAPP : Nat -> * -> *
NAPP 0 a = a
NAPP (suc n) a = a -> NAPP n a

nApp : {a:*} -> (n:Nat) ->
       NAPP n a -> a -> a
nApp 0 f x = f
nApp (suc n) f x = nApp n (f x) x
```

Universe elimination

```
data Type = CNat | CBool
i : Type -> *
i CNat = Nat
i CBool = CBool

eq : (x:Type) -> i x -> i x -> Bool
eq CNat x y = …
eq CBool x y = …
```

A function from values to types

Sounds great, what is the problem?

Universes & type inference

- Type-dependent functions can be expressed but not conveniently used.

```
eq : (x : Type) -> i x -> i x -> Bool
eq Bool True False
```

- Implicit arguments don't help

```
eq : { x : Type } -> i x -> i x -> Bool
eq True False
```

Type checker does not know that i is injective

Type classes & type inference

- Type classes support type-directed functions in Haskell

```
class Eq a where
  eq :: a -> a -> Bool
```

- Only one instance per type

```
instance Eq Bool where
  eq x y = if x then y else not y
```

- Allows type checker to determine appropriate instance at use site

```
eq True False
```

No explicit compile-time specialization/parametricity

- Sometimes computation can be resolved completely at compile-time
  - Example: `nApp 2 (+) x y`
- Sometimes arguments are not needed at runtime
  - Type parametricity
- Lack of staging makes dependently-typed languages difficult to compile efficiently

Logical Soundness

- Insistence on total correctness influences and complicates the language
- Agda restricted to predicative language, where everything can be shown terminating
- Workarounds exist, but discouraged:

  --set-in-set --no-termination-check
- Standard library designed for programming without these flags

Two ways to make progress

- Improve Agda (partial evaluator?)
- Improve Haskell
  - Agda: No distinction between compile-time runtime
  - Haskell: Strong distinction that interferes with generic programming

Of course, the answer is to do both, but in this talk, I'll concentrate on the second idea.

GHC today: Type-dependent types

```
data Z
data S n

type family NAPP (n :: *) (a :: *)
type instance NAPP Z a = a
type instance NAPP (S n) a = a -> (NAPP n a)
```

Natural numbers implemented with empty data declarations

A function from types to types

GHC today: Type-dependent values

Singleton GADT reflecting type-level Nats to computation

```
data SNat n where
  SZ :: SNat Z
  SS :: SNat n -> SNat (S n)
```

Type inference aid: explicit type argument

```
data Proxy a

napp :: Proxy a -> SNat n -> NAPP n a -> a -> a
napp a n f x = case n of
               SZ     -> f
               (SS m) -> napp a m (f x) x
```

Problems with example

- Type-level programming is weakly-typed

```
Z :: *
S :: * -> *
NAPP :: * -> * -> *
```

- Duplication! Nats at term level (not shown), Nats at type level, Singleton Nats
- Ambiguity in type inference
  - All compile-time arguments must be inferred
  - If a type variable does not appear outside a type function application, it cannot be inferred

```
{-# LANGUAGE IDEAL #-}

data Nat = Z | S Nat

type family NAPP (n :: Nat) (a :: *)
type instance NAPP Z a = a
type instance NAPP (S n) a = a -> (NAPP n a)

napp :: forall a n. RT Nat n => NAPP n a -> a -> a
napp f x = case %n of
             Z     -> f
             (S m) -> napp @a @m (f x) x
```

Can appear in expressions and types

Informative kind

Class constraint ensures parametricity

Analysis of type variable

Explicit type application

New Haskell Extensions: Summary

- Datatype lifting
  - Allow datatype constructors to appear in types
  - And datatypes to appear in kinds
- Case analysis of lifted datatype
  - Informative dependent case analysis
  - Compiler automatically replaces with case analysis of singleton
- Explicit type application
  - Tame ambiguity with type family usage

Datatype lifting

- Allow data constructors to appear in types
- Allow data types to appear in kinds
- Coalesce types & kinds together

New type language

```
t, s ::= a             Variables
     |  T              Constants (List, Int)
     |  K              Data constr. (Cons, Z)
     |  s t            Application
     |  F t1 .. tn     Indexed type (i.e. NAPP)
     |  *              Kind 'type'
     |  s -> t         Arrow type/kind
     |  all a. t       Polymorphism
     |  C => t         Constrained type
```

Type formation: `G |- t : s`

Advantage of coalesced types

- Simple kind polymorphism (for terms & types)
  `Cons :: all a. a -> List a -> List a`
- Data-structures available for type level programming
  `Cons Int (Cons Bool Nil) :: List *`
- Type families indexed by kinds
  `F :: all k. k -> *`

Typecase

- Idea: allow case analysis of 'types'
  - `case %t of ….`
  - Constrained by type class RT
- Implemented by desugaring to case analysis of singleton type
  - RT type class is just a carrier for singleton type!
- Singleton type automatically defined by compiler

Questions and Difficulties

- What datatypes can be lifted to types?
  - Only simple, regular datatypes? (List)
  - Existentials?
  - GADTs?
  - Those using type families?
  - Class constraints?
- What kinds have singleton types?
  - Only lifted datatypes?
  - Also kind *?
  - Other kinds (k1 => k2, all a. k) ?

Run-time Nats

- Can we coerce a runtime Nat type into an expression?

```
f :: all n. RT Nat n => Nat
f = case %n of
    Z -> 0
    S m -> %m
```

- What about an indexed type function?

```
%(PLUS m (S Z))
```

Do we need singletons?

- Given a type t, do programmers ever need to explicitly use the singleton type?
  - CSP covers non-dependent use
  - RT class constraint implicitly covers any singleton used as an argument
  - What about singletons returned from functions?

    forall a. RT Nat a => Singleton (FACT a)

    Where it is eventually used, replace with %FACT ?

Observations

- Singletons key to dependent case analysis
- Dependency mostly independent of staging
  - compile-time, dependent arg: `all n:Nat. t`
  - runtime, dependent arg:
    `all n:Nat. RT Nat n => t`
  - runtime, nondependent arg: `Nat -> t`
  - compile-time, nondependent arg?
    - doesn't make sense?

What about compile time specialization?

- Haskell Type class resolution is a form of compile-time programming
- How does this mechanism interact with new vision?

Compile time specialization

```
class Napp n a where
  snapp :: NAPP n a -> a -> a

instance Napp Z a where
  snapp f x = f

instance Napp n a => Napp (S n) a where
   snapp f x = snapp @n @a (f x) x

x :: Int
x = snapp @(S (S Z)) (+) 1 2
```

- Explicit type application
- Scoped variables in instances

Misgivings about type classes

- Certainly useful, but do they fit into the programming model?
- Should they?
  - Non-uniformity: Logic programming instead of FP
  - Duplication of mechanism: "Eq t" is an implicit runtime argument
- Is there a more orthogonal language feature? Default implicit arguments, irrelevant arguments, injectivity?

Current progress & future work

- Integrate dependency into FC
  - Intermediate language for type functions, GADTs with explicit type coercions
  - Current struggle between complexity and expressiveness
- Formalize singleton type translation
  - New coercion in FC from singleton to regular type?
- Integrate with source language & type inference
  - Dependent case analysis relies on singleton translation

Conclusion

- This slide intentionally left blank.