Modalities and Polarities

Jesper Cockx

9 May 2017

What am I talking about?

A modality is an annotation on a function type giving some information beyond the domain and codomain of the function.

Examples currently in Agda:

- Hiding: $\{x : A\} \to B$
- Relevance: $.(x : A) \to B$

What is the goal?

We want explicit polarity annotations[1][2][3]:

$$\textbf{data}\ \texttt{Fix}\ (F : \texttt{Set} \xrightarrow{+} \texttt{Set}) : \texttt{Set}\ \textbf{where}$$
$$\quad \texttt{fix} : F\ (\texttt{Fix}\ F) \to \texttt{Fix}\ F$$

[1]Explicit polarity annotation (#570)

[2]Polarity checking not compositional for phantom arguments (#1291)

[3]Subtyping inductive sized types (#2411)

What is the goal?

We also want other annotations[4]:

- Linearity
- Parametricity
- Strictness
- Custom meta solvers
- ...

What is standing in the way of adding these?

[4]Ulf Norell, Extensible syntax for function space annotations (#2513)

Current situation

Problems

Challenges and tradeoffs

Current situation

Problems

Challenges and tradeoffs

Surface syntax

```
f₁ :    (x : A)   → B x
f₂ :    {x : A}   → B x
f₃ :    {{x : A}} → B x
f₄ :   .(x : A)   → B x
f₅ :   .{x : A}   → B x
f₆ :   .{{x : A}} → B x
f₇ :  ..(x : A)   → B x
f₈ :  ..{x : A}   → B x
f₉ :  ..{{x : A}} → B x
```

Hiding: internally

```
data Hiding  = Hidden | Instance | NotHidden
  deriving (Typeable, Show, Eq, Ord)
```

Relevance: internally

```
-- | A function argument can be relevant or irrelevant.
--   See "Agda.TypeChecking.Irrelevance".
data Relevance
  = Relevant    -- ^ The argument is (possibly) relevant at compile-time.
  | NonStrict   -- ^ The argument may never flow into evaluation position.
                --   Therefore, it is irrelevant at run-time.
                --   It is treated relevantly during equality checking.
  | Irrelevant  -- ^ The argument is irrelevant at compile- and runtime.
  | Forced Big  -- ^ The argument can be skipped during equality checking
                --   because its value is already determined by the type.
                --   If a constructor argument is big, it has to be regarded
                --   absent, otherwise we get into paradoxes.
  | UnusedArg   -- ^ The polarity checker has determined that this argument
                --   is unused in the definition.  It can be skipped during
                --   equality checking but should be mined for solutions
                --   of meta-variables with relevance 'UnusedArg'
    deriving (Typeable, Show, Eq)
```

Polarities: internally

```
-- | Polarity for equality and subtype checking.
data Polarity
  = Covariant      -- ^ monotone
  | Contravariant  -- ^ antitone
  | Invariant      -- ^ no information (mixed variance)
  | Nonvariant     -- ^ constant
  deriving (Typeable, Show, Eq)
```

```
-- | Subterm occurrences for positivity checking.
--   The constructors are listed in increasing information they provide:
--   @Mixed <= JustPos <= StrictPos <= GuardPos <= Unused@
--   @Mixed <= JustNeg <= Unused@.
data Occurrence
  = Mixed     -- ^ Arbitrary occurrence (positive and negative).
  | JustNeg   -- ^ Negative occurrence.
  | JustPos   -- ^ Positive occurrence, but not strictly positive.
  | StrictPos -- ^ Strictly positive occurrence.
  | GuardPos  -- ^ Guarded strictly positive occurrence (i.e., under ∞).
  | Unused    --  ^ No occurrence.
  deriving (Typeable, Show, Eq, Ord, Enum, Bounded)
```

Current situation

Problems

Challenges and tradeoffs

Problem 1

'Forced' and 'UnusedArg' are internally treated as relevances, but they are invisible to the user.

This leads to inconsistencies in Agda[5].

[5]Agda accepts incorrect (?) code, subject reduction broken (#2480)

Problem 1

‘Forced’ and ‘UnusedArg’ are internally treated as relevances, but they are invisible to the user.

This leads to inconsistencies in Agda[5].

It also prevents us from having e.g. a nonstrict, unused function argument.

[5] Agda accepts incorrect (?) code, subject reduction broken (#2480)

Problem 2

Fact: we can't keep adding .'s and $\{\}$'s.

But the number of reserved symbols is very limited!

Problem 3

Polarity is currently a property of a *definition*, not of a *function type*.

In particular, we can't express that a variable f : `Set` $\rightarrow$ `Set` must be positive.

Current situation

Problems

Challenges and tradeoffs

Extensible syntax for function annotations?

Proposal by Ulf: $(\mathsf{pos} \mid x : A) \to B$

Proposal by Andrea: $(x :\{\mathsf{pos}\}\ A) \to B$

Extensible syntax for function annotations?

Proposal by Ulf: $(\mathsf{pos} \mid x : A) \to B$

Proposal by Andrea: $(x :\{\mathsf{pos}\}\ A) \to B$

- Nondependent functions?
- Declarations / record fields?
- Module/datatype parameters?

Declared vs. inferred?

If we make all polarities declared, we lose backwards compatibility, so that is probably a bad idea.

Maybe we can store both a declared and a (more precise) inferred polarity.

Should we use declared or inferred polarity for imported definitions?

Other questions for the future

- Subtyping between modalities?
- Modality polymorphism?

Properties of definition vs properties of function type?

Annotations on function types are compositional and can also be used in types of variables.

However, they fragment the space of function types.

We can't have the cake and eat it too (see UnusedArg problems).

Where are annotations used?

For type checking?
For constraint solving?
For compilation?

Roadmap

- Get rid of UnusedArg relevance
- Make polarity a property of the type
- Add parser for extensible annotations
- ???
- Profit!

Roadmap

- Get rid of UnusedArg relevance
- Make polarity a property of the type
- Add parser for extensible annotations
- ???
- Profit!

Time for a discussion!