1 Example: Queues

A Specification of Queues

- A queue is simply a list.

\[
\text{type } \text{Queue} \ a = [a] \\
\text{empty} \quad = [] \\
\text{add} \ x \ q \quad = q ++ [x] \\
\text{isEmpty} \ q \quad = \text{null} \ q \\
\text{front} \ (x:q) \quad = x \\
\text{remove} \ (x:q) \quad = q
\]

- Enqueueing has linear time complexity.
- Implementation should have amortized constant time operations.

An Implementation of Queues

- A queue consists of a front list and a reversed back list.

\[
\text{type } \text{QueueI} \ a = ([a],[a]) \\
\text{retrieve} :: \text{QueueI} \ a \to \text{Queue} \ a \\
\text{retrieve} \ (f,b) \quad = f ++ \text{reverse} \ b
\]

- An data invariant:

If the front list is empty, then so is the back list.
Implementation of Queue Operations

- Auxiliary operation \( \text{flipQ} \) restores the invariant.

\[
\begin{align*}
\text{flipQ} ([],b) &= (\text{reverse } b, []) \\
\text{flipQ} q &= q
\end{align*}
\]

- Queue operations:

\[
\begin{align*}
\text{emptyI} &= ([], []) \\
\text{addI} x (f, b) &= \text{flipQ} (f, x:b) \\
\text{isEmptyI} (f, b) &= \text{null } f \\
\text{frontI} (x:f, b) &= x \\
\text{removeI} (x:f, b) &= \text{flipQ} (f, b)
\end{align*}
\]

Soundness

- Diagram should commute:

\[
\begin{array}{c}
\text{QueueI} \xrightarrow{\text{opI}} \text{QueueI} \\
\text{retrieve} \downarrow \quad \quad \quad \quad \quad \quad \uparrow \text{retrieve} \\
\text{Queue} \xrightarrow{\text{op}} \text{Queue}
\end{array}
\]

- Example:

\[
\text{retrieve} \ (\text{addI} \ x \ q) = \text{add} \ x \ (\text{retrieve} \ q)
\]

2 From Haskell to Agda

Proofs about Haskell Programs

- We need a translation:
• But: Haskell is a rich language!

Translation Outline

• We use GHC Core as an intermediate language.

• (GHC) Core = System $F_\omega$ + data types + mutual recursion.

• Type classes and nested patterns are translated away by GHC.

Target: Agda

• Purely functional, dependently typed language.

• Propositions are sets (types): $\text{Prop} = \text{Set}$.

• Predicates are dependent types, e.g.:

  Even : $\text{Nat} \rightarrow \text{Prop}$

  lemma : ($n : \text{Nat}$) $\rightarrow$ Even $n \rightarrow$ Even($n + 2$)

Agda Programs Must Be...

• predicative,

• terminating.
• and total. Oops!

\[ \text{front \ (x:q) = x} \]

• We need to translate each type \( A \) by \( \text{Maybe} \ A \).

A Monadic Translation

• Partiality involved? Translate \( A \) by \( \text{Maybe} \ A \).

• Everything total? Translate \( A \) by \( A \).

• Maybe is a monad.

• Identity is a monad.

• We do a \emph{monadic} translation.

Translation Outline (refined)

3 Monadic Translation

Monads in Agda

• An abstract monad:

\[
\begin{align*}
m & : \text{Set} \to \text{Set} \\
\text{return \ (\alpha : \text{Set})} & : \alpha \to m \alpha \\
(\gg=) \ (\alpha, \beta : \text{Set}) & : m \alpha \to (\alpha \to m \beta) \to m \beta
\end{align*}
\]

• Arguments to the right of (\:) are implicit.
Translating the λ-Calculus

- Translation of types:

\[ \tau^\dagger = m\tau^* \]
\[ (\alpha \tau)^* = \alpha \tau^* \]
\[ (\tau_1 \rightarrow \tau_2)^* = \tau_1^\dagger \rightarrow \tau_2^\dagger \]

- Translation of programs (domain-free):

\[ x^\dagger = x \]
\[ (\lambda x. e)^\dagger = \text{return} (\lambda x. e^\dagger) \]
\[ (fe)^\dagger = f^\dagger \gg= \lambda f'. f'e^\dagger \]

Dealing with Polymorphism

- In the literature (Barthe, Hatcliff, Thiemann 1997):

\[ (\forall \alpha. \sigma)^\dagger = m ((\alpha:\text{Set}) \rightarrow \sigma^\dagger) \]
\[ (\Lambda \alpha. e)^\dagger = \text{return} (\lambda \alpha. e^\dagger) \]

- But Agda is predicative: \((\alpha:\text{Set}) \rightarrow \sigma\) is not in \text{Set}!

- However, we want to instantiate \(\alpha\) with some \(m\tau\).

- So, \(m\) needs to be in \(\text{Set} \rightarrow \text{Set}\).

- \(\Rightarrow\) Polytypes are translated non-monadically.

Translating Polymorphism

- Our approach:

\[ (\forall \alpha. \sigma)^\dagger = (\alpha:\text{Set}) \rightarrow \sigma^\dagger \]
\[ (\Lambda \alpha. e)^\dagger = \lambda \alpha. e^\dagger \]

- Consistent with Haskell semantics:
  
  - Type abstraction and applications are \textit{not computations}, but information for the compiler.
  
  - \((\Lambda \alpha. \bot) = \bot\).

- We need to distinguish between \textit{monotypes} and \textit{polytypes}.
Translation Outline (revised)

Haskell
GHC
GHC Core
Preprocessor
Pred. Core
Monadic Translation
Agda

Predicative Core

• Predicative $F_\omega$ (restriction of Leivant 1991):

\[
\begin{align*}
\kappa & ::= * \mid \kappa \rightarrow \kappa' & \text{kinds} \\
\tau & ::= \alpha \vec{\tau} \mid \tau \rightarrow \tau' & \text{monotypes} \\
\sigma & ::= \tau \mid \forall \alpha : \kappa, \sigma \mid \sigma \mapsto \sigma' & \text{polytypes}
\end{align*}
\]

• Translation of poly-function types (arise from dictionaries):

\[
\begin{align*}
(\sigma_1 \mapsto \sigma_2)^\dagger & = \sigma_1^\dagger \rightarrow \sigma_2^\dagger \\
(\lambda x : \sigma, e)^\dagger & = \lambda x : \sigma^\dagger, e^\dagger \\
(f^{\sigma \mapsto \sigma'}, e)^\dagger & = f^\dagger e^\dagger
\end{align*}
\]

Translating Datatypes

• Lists . . .

\[
\begin{align*}
data \text{ List } \alpha & = \text{ Nil} \\
& \mid \text{ Cons } \alpha (\text{ List } \alpha)
\end{align*}
\]

• . . . are translated as:

\[
\begin{align*}
data \text{ List } (\alpha : \text{ Set}) & = \text{ Nil} \\
& \mid \text{ Cons } (m x : m \alpha) (m x s : m (\text{ List } \alpha))
\end{align*}
\]

Demo
Conclusions

- New monadic translation.

- Pragmatic approach to Haskell program verification.

- Drawbacks:
  - Monads everywhere.
  - GHC Core designed as frontend for compiler, not theorem prover.

- But:
  - Lightweight translation (easy to get right).
  - “Core-ification” preserves most names.
  - Proofs about the de-facto semantics of Haskell programs.