Verifying Haskell Programs Using Constructive Type Theory

Andreas Abel Marcin Benke Ana Bove John Hughes
Ulf Norell

Haskell Workshop, 2005

1 Example: Queues

A Specification of Queues

- A queue is simply a list.

```
type Queue a = [a]

empty        = []
add x q      = q ++ [x]
isEmpty q    = null q
front  (x:q) = x
remove (x:q) = 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.

```
type QueueI a = ([a],[a])

retrieve :: QueueI a -> Queue a
retrieve (f,b) = f ++ reverse b
```

- An data *invariant*:

  If the front list is empty, then so is the back list.

Implementation of Queue Operations

- Auxiliary operation `flipQ` restores the invariant.

```
flipQ ([],b)     = (reverse b,[])
flipQ q          = q
```

- Queue operations:

```
emptyI           = ([],[])
addI   x  (f,b) = flipQ (f,x:b)
isEmptyI  (f,b) = null f
frontI  (x:f,b) = x
removeI (x:f,b) = flipQ (f,b)
```

Soundness

- Diagram should commute:

$$\begin{tikzcd} \texttt{QueueI} \arrow[r, "\texttt{opI}"] \arrow[d, "\texttt{retrieve}"'] & \texttt{QueueI} \arrow[d, "\texttt{retrieve}"] \\ \texttt{Queue} \arrow[r, "\texttt{op}"] & \texttt{Queue} \end{tikzcd}$$

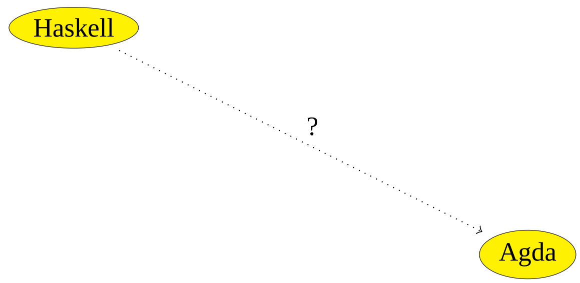
- Example:

```
retrieve (addI x q) == add x (retrieve q)
```

2 From Haskell to Agda

Proofs about Haskell Programs

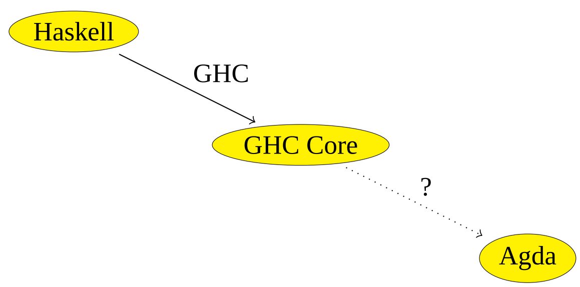
- We need a translation:



- But: Haskell is a rich language!

Translation Outline

- We use GHC Core as an intermediate language.



- (GHC) Core = System F_ω + 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): $\mathsf{Prop} = \mathsf{Set}$.
- Predicates are dependent types, e.g.:

$$\begin{array}{lcl} \mathsf{Even} & : & \mathsf{Nat} \to \mathsf{Prop} \\ \mathsf{lemma} & : & (n : \mathsf{Nat}) \to \mathsf{Even}\, n \to \mathsf{Even}(n+2) \end{array}$$

Agda Programs Must Be...

- predicative,
- terminating,

- and total. Oops!

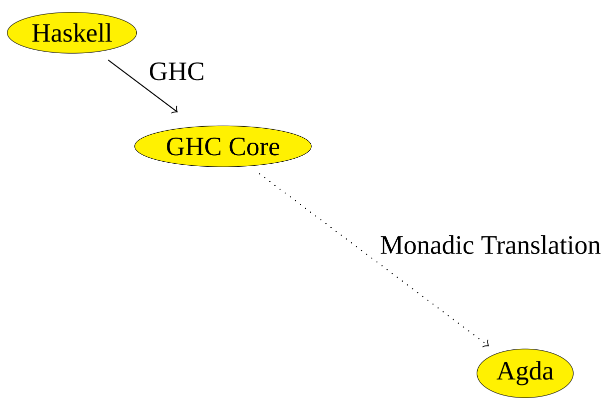
```
front  (x:q) = x
```

- We need to translate each type A by $\mathsf{Maybe}\ A$.

A Monadic Translation

- Partiality involved? Translate A by $\mathsf{Maybe}\ A$.
- Everything total? Translate A by A.
- Maybe is a monad.
- Identity is a monad.
- We do a *monadic* translation.

Translation Outline (refined)



3 Monadic Translation

Monads in Agda

- An abstract monad:

$$
\begin{array}{lcl}
\mathsf{m} & : & \mathsf{Set} \to \mathsf{Set} \\
\\
\mathsf{return}\,(\alpha : \mathsf{Set}) & : & \alpha \to \mathsf{m}\,\alpha \\
(\gg\!=)\,(\alpha, \beta : \mathsf{Set}) & : & \mathsf{m}\,\alpha \to (\alpha \to \mathsf{m}\,\beta) \to \mathsf{m}\,\beta
\end{array}
$$

- Arguments to the right of (:) are implicit.

Translating the λ-Calculus

- Translation of types:

$$\begin{array}{rcl} \tau^{\dagger} & = & \mathsf{m}\,\tau^{*} \\ \\ (\alpha\,\vec{\tau})^{*} & = & \alpha\,\vec{\tau}^{*} \\ (\tau_1 \to \tau_2)^{*} & = & \tau_1^{\dagger} \to \tau_2^{\dagger} \end{array}$$

- Translation of programs (domain-free):

$$\begin{array}{rcl} x^{\dagger} & = & x \\ (\lambda x.e)^{\dagger} & = & \mathsf{return}\,(\lambda x.\, e^{\dagger}) \\ (f\,e)^{\dagger} & = & f^{\dagger} \gg\!= \lambda f'.\, f'\, e^{\dagger} \end{array}$$

Dealing with Polymophism

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

$$\begin{array}{rcl} (\forall\alpha.\sigma)^{\dagger} & = & \mathsf{m}\,((\alpha\!:\!\mathsf{Set}) \to \sigma^{\dagger}) \\ \\ (\Lambda\alpha.e)^{\dagger} & = & \mathsf{return}\,(\lambda\alpha.\, e^{\dagger}) \end{array}$$

- But Agda is predicative: $(\alpha\!:\!\mathsf{Set}) \to \sigma$ is not in Set!

- However, we want to instantiate α with some $\mathsf{m}\,\tau$.

- So, m needs to be in $\mathsf{Set} \to \mathsf{Set}$.

- $\Longrightarrow$ Polytypes are translated non-monadically.

Translating Polymorphism

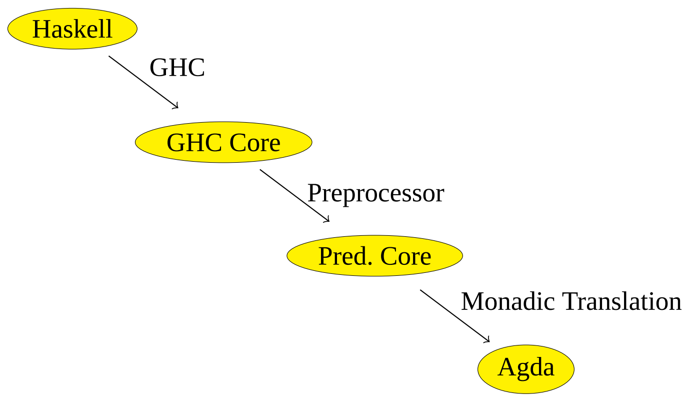
- Our approach:

$$\begin{array}{rcl} (\forall\alpha.\sigma)^{\dagger} & = & (\alpha\!:\!\mathsf{Set}) \to \sigma^{\dagger} \\ \\ (\Lambda\alpha.e)^{\dagger} & = & \lambda\alpha.\, e^{\dagger} \end{array}$$

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

- We need to distinguish between *monotypes* and *polytypes*.

Translation Outline (revised)



Predicative Core

- Predicative F_ω (restriction of Leivant 1991):

$$
\begin{array}{lcll}
\kappa & ::= & * \mid \kappa \to \kappa' & \text{kinds} \\[6pt]
\tau & ::= & \alpha\,\vec{\tau} \mid \tau \to \tau' & \text{monotypes} \\
\sigma & ::= & \tau \mid \forall \alpha\!:\!\kappa.\,\sigma \mid \sigma \mapsto \sigma' & \text{polytypes}
\end{array}
$$

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

$$
\begin{array}{lcl}
(\sigma_1 \mapsto \sigma_2)^\dagger & = & \sigma_1^\dagger \to \sigma_2^\dagger \\[6pt]
(\lambda x\!:\!\sigma.e)^\dagger & = & \lambda x\!:\!\sigma^\dagger.\,e^\dagger \\
(f^{\sigma_1 \mapsto \sigma_2}\, e)^\dagger & = & f^\dagger\, e^\dagger
\end{array}
$$

Translating Datatypes

- Lists . . .

$$
\begin{array}{lcl}
\mathsf{data\ List}\,\alpha & = & \mathsf{Nil} \\
 & \mid & \mathsf{Cons}\ \alpha\ (\mathsf{List}\,\alpha)
\end{array}
$$

- . . . are translated as:

$$
\begin{array}{lcl}
\mathsf{data\ List}\,(\alpha\!:\!\mathsf{Set}) & = & \mathsf{Nil} \\
 & \mid & \mathsf{Cons}\ (mx\!:\!\mathsf{m}\,\alpha)\ (mxs\!:\!\mathsf{m}\,(\mathsf{List}\,\alpha))
\end{array}
$$

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.