A Lightweight Approach to Start Time Consistency in Haskell

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
February 9, 2012

Functional reactive programming

- declarative approach to programming reactive systems
- functional programming extended with support for temporal processes
- examples of processes:
  - **behaviors** time-varying values:

$$[\![\mathit{Behavior}\ \alpha]\!] \approx \mathrm{Time} \to [\![\alpha]\!]$$

  - **events** values at points in time:

$$[\![\mathit{Event}\ \alpha]\!] \approx \mathrm{Time} \times [\![\alpha]\!]$$

Start times

- processes have associated start times:
  - behaviors provides values only at their start times and later
  - events can only fire at their start times or later
- processes appearing within other processes at some time t must start at t
- introduce a start time parameter to the meanings of types:
  - behaviors:

$$[\![\mathit{Behavior}\ \alpha]\!](t) = \Pi t' : \mathrm{Time}\ .\ (t \leqslant t') \to [\![\alpha]\!](t')$$

  - events:

$$[\![\mathit{Event}\ \alpha]\!](t) = \Sigma t' : \mathrm{Time}\ .\ (t \leqslant t') \times [\![\alpha]\!](t')$$

  - start time parameter passed downwards for ordinary type constructors

Temporal categories

- basic constructions in Haskells type system:
  - finite products
  - finite sums
  - function spaces
- modelled by bicartesian closed categories (BCCCs):
  - objects correspond to types
  - morphisms correspond to functions
- support for FRP by extending BCCCs to temporal categories (TCs):
  - objects correspond to types
  - morphisms correspond to families of functions with one function per time:

$$\Pi t : \mathsf{Time}\ .\ [\![\alpha]\!](t) \rightarrow [\![\beta]\!](t)$$

  - *Behavior* and *Event* correspond to functors $\Box$ and $\Diamond$

FRP operations in temporal categories

- natural transformations for operations where all involved processes have the same start time:

$$m_{A,B} : \Box A \times \Box B \to \Box(A \times B)$$
$$\mu_A : \Diamond\Diamond A \to \Diamond A$$
$$s_{A,B} : \Box A \times \Diamond B \to \Diamond(A \times B)$$

  etc.
- transforming values inside behaviors and events:
  - for every $f : A \to B$, we have:

$$\Box f : \Box A \to \Box B$$
$$\Diamond f : \Diamond A \to \Diamond B$$

  - safe, because $f : A \to B$ includes a function for every time

Tensorial strength

- two natural transformations:

$$t^{\square}_{A,B} : A \times \square B \to \square(A \times B)$$
$$t^{\diamond}_{A,B} : A \times \diamond B \to \diamond(A \times B)$$

- disallowed, because they would have to shift values to different times

A straightforward implementation approach

- polymorphic functions for natural transformations:

$$
\begin{aligned}
\mathit{fuse} &:: (\mathit{Behavior}\ \alpha, \mathit{Behavior}\ \beta) \rightarrow \mathit{Behavior}\ (\alpha, \beta) \\
\mathit{join} &:: \mathit{Event}\ (\mathit{Event}\ \alpha) \rightarrow \mathit{Event}\ \alpha \\
\mathit{sample} &:: (\mathit{Behavior}\ \alpha, \mathit{Event}\ \beta) \rightarrow \mathit{Event}\ (\alpha, \beta)
\end{aligned}
$$

- Haskell's *Functor* class for functors:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\
&\quad \mathit{fmap} :: (\alpha \rightarrow \beta) \rightarrow (f\ \alpha \rightarrow f\ \beta) \\
&\textbf{instance}\ \mathit{Functor}\ \mathit{Behavior}\ \textbf{where}\ \ldots \\
&\textbf{instance}\ \mathit{Functor}\ \mathit{Event}\ \textbf{where}\ \ldots
\end{aligned}
$$

Tensorial strength through the backdoor

- *fmap* is a Haskell function
- so it corresponds to a morphism itself
- for each functor F, we have the following:

$$\varphi^F_{A,B} : B^A \to FB^{FA}$$

- allows us to construct tensorial strength:

$$\begin{aligned} \Lambda\mathrm{id}_{A\times B} &: A \to (A\times B)^B \\ \varphi^F_{B,A\times B}(\Lambda\mathrm{id}_{A\times B}) &: A \to F(A\times B)^{FB} \\ (\varphi^F_{B,A\times B}(\Lambda\mathrm{id}_{A\times B}) \times \mathrm{id}_{FB}) &: A\times FB \to F(A\times B)^{FB} \times FB \\ e(\varphi^F_{B,A\times B}(\Lambda\mathrm{id}_{A\times B}) \times \mathrm{id}_{FB}) &: A\times FB \to F(A\times B) \end{aligned}$$

- the same in Haskell:

$$\mathit{strength} :: (\mathit{Functor}\ f) \Rightarrow (\alpha, f\ \beta) \to f\ (\alpha, \beta)$$
$$\mathit{strength}\ (x, f) = \mathit{fmap}\ ((,)\ x)\ f$$

Applicative functors

- functors support lifting of unary functions:

$$fmap :: (Functor\ f) \Rightarrow (\alpha \to \beta) \to (f\ \alpha \to f\ \beta)$$

- applicative functors support lifting of functions of arbitrary arity:

$$\begin{aligned} liftA_n :: (Applicative\ f) \Rightarrow \\ (\alpha_1 \to \cdots \to \alpha_n \to \beta) \to \\ (f\ \alpha_1 \to \cdots \to f\ \alpha_n \to f\ \beta) \end{aligned}$$

- *Applicative* class contains two methods:

$$pure = liftA_0 \qquad (\circledast) = liftA_2\ (\$)$$

- for arbitrary $n \in \mathbb{N}$, $liftA_n$ can be derived:

$$liftA_n\ f\ f_1\ \ldots\ f_n = pure\ f \circledast f_1 \circledast \cdots \circledast f_n$$

Timed values

- no functions that directly work with behaviors and events
- user can construct only values that are valid independently of time
- type constructor *At* for encapsulating values that are fixed to some time:
  - *At* has a phantom parameter t that represents a time
  - $At\ t\ \alpha$ contains all values of type α that are valid at t
  - in particular:
    - $At\ t\ (Behavior\ \alpha)$ contains all behaviors that start at t
    - $At\ t\ (Event\ \alpha)$ contains all events that start at t
- for every t, $At\ t$ is an applicative functor:

$$\begin{aligned} liftA_n :: (\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta) &\rightarrow \\ (At\ t\ \alpha_1 \rightarrow \cdots \rightarrow At\ t\ \alpha_n \rightarrow At\ t\ \beta) \end{aligned}$$

Operations with timed values

- functions on *At* for natural transformations:

$$\begin{aligned}
\mathit{fuse} &:: \mathit{At}\ t\ (\mathit{Behavior}\ \alpha, \mathit{Behavior}\ \beta) \rightarrow \\
&\ \mathit{At}\ t\ (\mathit{Behavior}\ (\alpha, \beta)) \\
\mathit{join} &:: \mathit{At}\ t\ (\mathit{Event}\ (\mathit{Event}\ \alpha)) \rightarrow \\
&\ \mathit{At}\ t\ (\mathit{Event}\ \alpha) \\
\mathit{sample} &:: \mathit{At}\ t\ (\mathit{Behavior}\ \alpha, \mathit{Event}\ \beta) \rightarrow \\
&\ \mathit{At}\ t\ (\mathit{Event}\ (\alpha, \beta))
\end{aligned}$$

- functor application requires an argument with universally quantified time:

$$\begin{aligned}
&\mathbf{class}\ \mathit{SafeFunctor}\ f\ \mathbf{where} \\
&\quad \mathit{safeMap} :: (\forall t\ .\ \mathit{At}\ t\ \alpha \rightarrow \mathit{At}\ t\ \beta) \rightarrow \\
&\quad\qquad \mathit{At}\ t\ (f\ \alpha) \rightarrow \mathit{At}\ t\ (f\ \beta) \\
&\mathbf{instance}\ \mathit{SafeFunctor}\ \mathit{Behavior}\ \mathbf{where}\ \ldots \\
&\mathbf{instance}\ \mathit{SafeFunctor}\ \mathit{Event}\ \mathbf{where}\ \ldots
\end{aligned}$$

No tensorial strength anymore

- remember our implementation of *strength*:

$$\mathit{strength} :: (\mathit{Functor}\ f) \Rightarrow (\alpha, f\ \beta) \to f\ (\alpha, \beta)$$
$$\mathit{strength}\ (x, f) = \mathit{fmap}\ ((,)\ x)\ f$$

- for any $x :: \alpha$, we have $((,)\ x) :: \beta \to (\alpha, \beta)$
- not suitable as an argument of *safeMap*

Really?

- we can lift that function:

$$liftA_1\ ((,)\ x) :: At\ t\ \beta \rightarrow At\ t\ (\alpha, \beta)$$

- allows us to construct a "safe strength":

$$\begin{aligned} safeStrength &:: (SafeFunctor\ f) \Rightarrow \\ &\qquad (\alpha, At\ t\ (f\ \beta)) \rightarrow At\ t\ (f\ (\alpha, \beta)) \\ safeStrength\ (x, f) &= safeMap\ (liftA_1\ ((,)\ x))\ f \end{aligned}$$

- no problem if values of α are valid independently of time
- but we can transform values that are already under an *At*:

$$\begin{aligned} liftA_1\ safeStrength :: (SafeFunctor\ f) \Rightarrow \\ At\ t\ (\alpha, At\ t'\ (f\ \beta)) \rightarrow \\ At\ t\ (At\ t'\ (f\ (\alpha, \beta))) \end{aligned}$$

- solution:

  At-values are only assumed to be consistent
  if they do not appear under another *At*

The Q-functor

- values of types $\forall t\ .\ At\ t\ \alpha \to At\ t\ \beta$ everywhere
- conversion from $\forall t\ .\ At\ t\ \alpha \to At\ t\ \beta$ to $\forall t\ .\ At\ t\ (\alpha \to \beta)$ should be safe
- opposite conversion is safe, because it is possible via application of $(\circledast)$
- introduce a type constructor Q with $Q\ \alpha = \forall t\ .\ At\ t\ \alpha$
- instead of $\forall t\ .\ At\ t\ \alpha \to At\ t\ \beta$ use $Q\ (\alpha \to \beta)$
- Q is an applicative functor, because the $liftA_n$ of $At\ t$ with type

$$\begin{array}{l}(\alpha_1 \to \cdots \to \alpha_n \to \beta) \to \\ (At\ t\ \alpha_1 \to \cdots \to At\ t\ \alpha_n \to At\ t\ \beta)\end{array}$$

can be turned into a $liftA_n$ of Q, which has type

$$\begin{array}{l}(\alpha_1 \to \cdots \to \alpha_n \to \beta) \to \\ ((\forall t\ .\ At\ t\ \alpha_1) \to \cdots \to (\forall t\ .\ At\ t\ \alpha_n) \to (\forall t\ .\ At\ t\ \beta))\end{array}$$

No universal quantification at the surface

- using applicative functor operations is enough for working with Q
- so we can hide the implementation of Q
- no universal quantification at the surface anymore:

$$
\begin{aligned}
\mathit{fuse} &:: Q\,((\mathit{Behavior}\ \alpha, \mathit{Behavior}\ \beta) \to \mathit{Behavior}\ (\alpha, \beta)) \\
\mathit{join} &:: Q\,(\mathit{Event}\ (\mathit{Event}\ \alpha) \to \mathit{Event}\ \alpha) \\
\mathit{sample} &:: Q\,((\mathit{Behavior}\ \alpha, \mathit{Event}\ \beta) \to \mathit{Event}\ (\alpha, \beta))
\end{aligned}
$$

class *SafeFunctor f* **where**

$$\mathit{safeMap} :: Q\,(\alpha \to \beta) \to Q\,(f\ \alpha \to f\ \beta)$$

No universal quantification internally

- library implementor can take care of ensuring consistency, since implementation of Q is hidden
- no need for using universal quantification internally anymore
- Q can just be implemented as the identity functor:

  newtype $Q\ \alpha = Q\ \alpha$

Conclusions and outlook

- lightweight technique for ensuring start time consistency:
  - easy to use
  - easy to implement
  - no need for language extensions
- open problems:
  - Does it really work?
  - Why does it work?
  - What kind of “effect” is represented by the Q-functor?
  - Is Q also a monad?
- future work:
  - Q-functor technique seems to be more generally applicable
  - use it to encode the Curry–Howard analog of linear logic in Haskell
  - leads to a more functional way of dealing with I/O (hopefully)
  - see next theory seminar