FOP Away Day 2007
Scalable Functional Reactive Programming

Neil Sculthorpe

School of Computer Science and Information Technology
University of Nottingham, United Kingdom
Outline

• The current Yampa implementation
• The problem, by example
• Proposed solution
• Difficulties of the solution
The current Yampa implementation

- A Yampa program can be represented as a dynamic network of signal functions.
The current Yampa implementation

- A Yampa program can be represented as a dynamic network of signal functions.

- At each time step, the value of each signal function is re-calculated.
The current Yampa implementation

- Yampa makes extensive use of events.

```haskell
data Event a = Event a | NoEvent
```
The current Yampa implementation

- Yampa makes extensive use of events.
  \[
  \text{data } \text{Event } a = \text{Event } a \mid \text{NoEvent}
  \]

- Those signal functions that produce \text{Event} values will be producing \text{NoEvent} most of the time.
The current Yampa implementation

- Yampa makes extensive use of events.
  
  data Event a = Event a | NoEvent

- Those signal functions that produce Event values will be producing NoEvent most of the time.

- Any stateless signal functions that have unchanged input will remain unchanged.
The current Yampa implementation

- Yampa makes extensive use of events.
  \[
  \text{data } \text{Event } a = \text{Event } a \mid \text{NoEvent}
  \]

- Those signal functions that produce Event values will be producing NoEvent most of the time.

- Any stateless signal functions that have unchanged input will remain unchanged.

- The same is true of some, but not all (eg. integral), stateful signal functions.
The current Yampa implementation

- Yampa makes extensive use of events.
  \[
  \text{data } \text{Event } a = \text{Event } a \mid \text{NoEvent}
  \]
- Those signal functions that produce Event values will be producing NoEvent most of the time.
- Any stateless signal functions that have unchanged input will remain unchanged.
- The same is true of some, but not all (eg. integral), stateful signal functions.
- Re-calculating them all every time step is a waste of computational resources.
The current Yampa implementation

- It would be better to re-calculate only the signal functions that need updating.
The current Yampa implementation

- It would be better to re-calculate only the signal functions that need updating.
- We can construct a graph recording:
The current Yampa implementation

- It would be better to re-calculate only the signal functions that need updating.
- We can construct a graph recording:
  - Which signal functions will output a constant signal while their input remains unchanged.
The current Yampa implementation

- It would be better to re-calculate only the signal functions that need updating.
- We can construct a graph recording:
  - Which signal functions will output a constant signal while their input remains unchanged.
  - The dependencies of each signal function.
The current Yampa implementation

- It would be better to re-calculate only the signal functions that need updating.
- We can construct a graph recording:
  - Which signal functions will output a constant signal while their input remains unchanged.
  - The dependencies of each signal function.
- At each time interval, we can propagate changes through the network.
The current Yampa implementation

• It would be better to re-calculate only the signal functions that need updating.

• We can construct a graph recording:
  - Which signal functions will output a constant signal while their input remains unchanged.
  - The dependencies of each signal function.

• At each time interval, we can propagate changes through the network.

• Unfortunately, the Yampa implementation creates a lot of incidental dependencies.
The problem, by example

\[ \text{sfDisF, sfDisR, sfDisL :: SF Input Distance} \]
\[ \text{sfLampCol :: SF Distance Colour} \]
\[ \text{sfOut :: SF (Colour, Direction) \rightarrow Output} \]
\[ \text{turnDir :: Distance \rightarrow Distance \rightarrow Distance \rightarrow Direction} \]
\[ \text{robot :: SF Input Output} \]
\[ \text{robot = proc inp \rightarrow do} \]
\[ \text{\hspace{1cm} fDis \leftarrow sfDisF \leftarrow inp} \]
\[ \text{\hspace{1cm} lDis \leftarrow sfDisR \leftarrow inp} \]
\[ \text{\hspace{1cm} rDis \leftarrow sfDisL \leftarrow inp} \]
\[ \text{\hspace{1cm} dir \leftarrow arr \text{ turnDir} \leftarrow (fDis, lDis, rDis} \]
\[ \text{\hspace{1cm} col \leftarrow sfLampCol \leftarrow fDis} \]
\[ \text{\hspace{1cm} sfOut} \hspace{1cm} (\text{col, dir}) \]
The problem, by example

- Ideally, we’d like a dependency graph that looks like:
The problem, by example

• Ideally, we’d like a dependency graph that looks like:

• But the code so far has been syntactic sugar.
The problem, by example

After translation into point free arrow code, it becomes:

\[
\text{robot} = \\
\text{arr id} \bowtie \text{sfDisF} \\
\text{arr id} \bowtie \left( \lambda (\text{inp}, f\text{Dis}) \to \text{inp} \right) \bowtie \text{sfDisL} \\
\text{arr id} \bowtie \left( \lambda ((\text{inp}, f\text{Dis}), \text{lDis}) \to \text{inp} \right) \bowtie \text{sfDisR} \\
\text{arr id} \bowtie \left( \lambda (((\text{inp}, f\text{Dis}), \text{lDis}), \text{rDis}) \to (f\text{Dis}, \text{lDis}, \text{rDis}) \right) \bowtie \text{arr turnDir} \\
\text{arr id} \bowtie \left( \lambda (((\text{inp}, f\text{Dis}), \text{lDis}), \text{rDis}, \text{dir}) \to f\text{Dis} \right) \bowtie \text{sfLampCol} \\
\text{arr } (\lambda (((\text{inp}, f\text{Dis}), \text{lDis}), \text{rDis}, \text{dir}, \text{col}) \to (\text{col}, \text{dir})) \bowtie \text{sfOut}
\]
The problem, by example

• After each signal function, all the values so far are tupled together, and then passed on.
The problem, by example

- After each signal function, all the values so far are tupled together, and then passed on.
- It is this that creates the incidental dependencies.
The problem, by example

• After each signal function, all the values so far are tupled together, and then passed on.

• It is this that creates the incidental dependencies.
Proposed Solution

- Abandon the Arrow framework for implementation purposes.
Proposed Solution

- Abandon the Arrow framework for implementation purposes.
- But try to keep the advantages of arrows, which include:
Proposed Solution

• Abandon the Arrow framework for implementation purposes.

• But try to keep the advantages of arrows, which include:
  - A syntax similar to the syntactic sugar.
Proposed Solution

• Abandon the Arrow framework for implementation purposes.

• But try to keep the advantages of arrows, which include:
  - A syntax similar to the syntactic sugar.
  - A clean, modular semantics that supports reasoning.
Proposed Solution

• Abandon the Arrow framework for implementation purposes.

• But try to keep the advantages of arrows, which include:
  - A syntax similar to the syntactic sugar.
  - A clean, modular semantics that supports reasoning.

• We can then create dependency graphs without incidental dependencies.
Difficulties of the solution

- Yampa’s dynamic nature:
Difficulties of the solution

- Yampa’s dynamic nature:
  - Dependencies will change as the network structure changes.
Difficulties of the solution

• Yampa’s dynamic nature:
  - Dependencies will change as the network structure changes.
  - Signal functions are first class entities, and thus can be created during runtime.
Difficulties of the solution

• Yampa’s dynamic nature:
  - Dependencies will change as the network structure changes.
  - Signal functions are first class entities, and thus can be created during runtime.

• How do you incorporate feedback into a dependency graph?
Summary

• The current Yampa implementation is not as efficient as it could be.

• This is due to the restrictions of the Arrow Framework.

• A new implementation is needed, but it should keep the strengths of Arrows.