Formalization of an FRP language with references

SeSTeRce Day

Jordan Ischard

September 13, 2023

LIFO - University of Orleans

Table of Contents

Table of Contents

Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems*.

Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.*

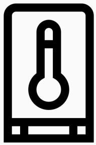
Definition [Pnu77]
A reactive system is a system that maintains an ongoing interaction with its environment [. . .].

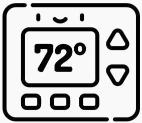
Reactive programming in 80s

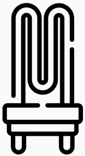
In the 1980s, there was a demand for programming languages to work on *reactive systems.*

Definition [Pnu77]
A reactive system is a system that maintains an ongoing interaction with its environment [. . .].

Radiator example





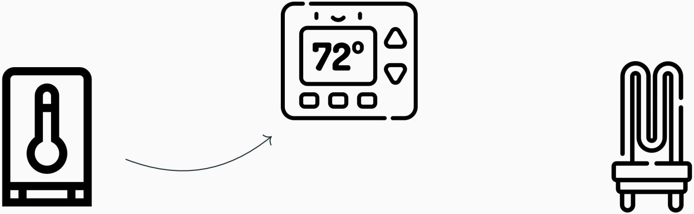


Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.*

Definition [Pnu77]
A reactive system is a system that maintains an ongoing interaction with its environment [. . .].

Radiator example

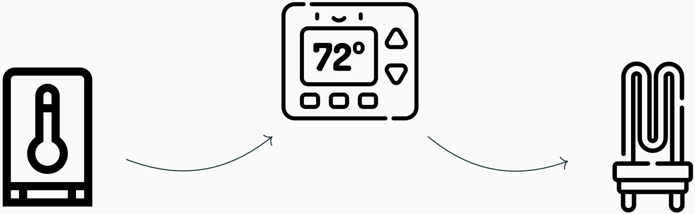


Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.*

Definition [Pnu77]
A reactive system is a system that maintains an ongoing interaction with its environment [. . .].

Radiator example

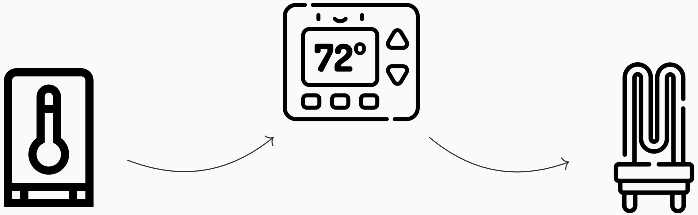


Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.* **An answer to this request was synchronous programming languages.**

Definition [Pnu77]
A reactive system is a system that maintains an ongoing interaction with its environment [. . .].

Radiator example

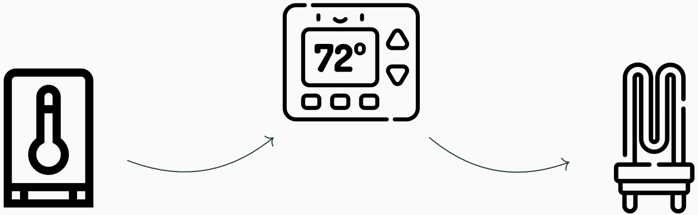


Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.* **An answer to this request was synchronous programming languages.**

Definition [Hal98]
A synchronous programming language is a computer programming language optimized for programming reactive systems.

Radiator example



Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.* **An answer to this request was synchronous programming languages.**

Definition [Hal98]
A synchronous programming language is a computer programming language optimized for programming reactive systems.

Radiator example (using Esterel [BG92])

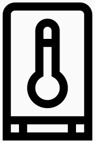
```
input OFF, TEMP(float);
do







watching OFF;
```



Reactive programming in 80s

In the 1980s, there was a demand for programming languages to work on *reactive systems.* **An answer to this request was synchronous programming languages.**

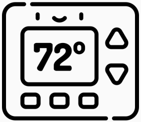
Definition [Hal98]
A synchronous programming language is a computer programming language optimized for programming reactive systems.

Radiator example (using Esterel [BG92])

```
input OFF, TEMP(float);
do
  signal SWITCH = bot in



      if needs_to_switch(?TEMP) then emit SWITCH
                                else nothing end
watching OFF;
```



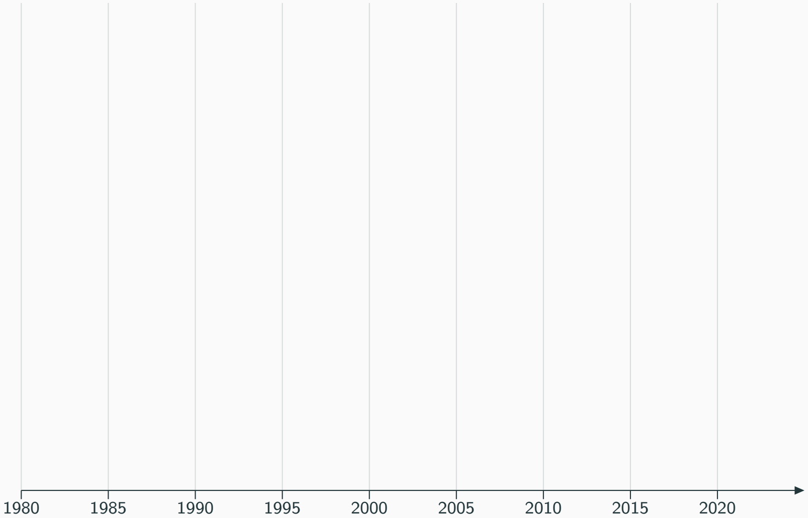
In the 1980s, there was a demand for programming languages to work on *reactive systems.* **An answer to this request was synchronous programming languages.**

Definition [Hal98]
A synchronous programming language is a computer programming language optimized for programming reactive systems.

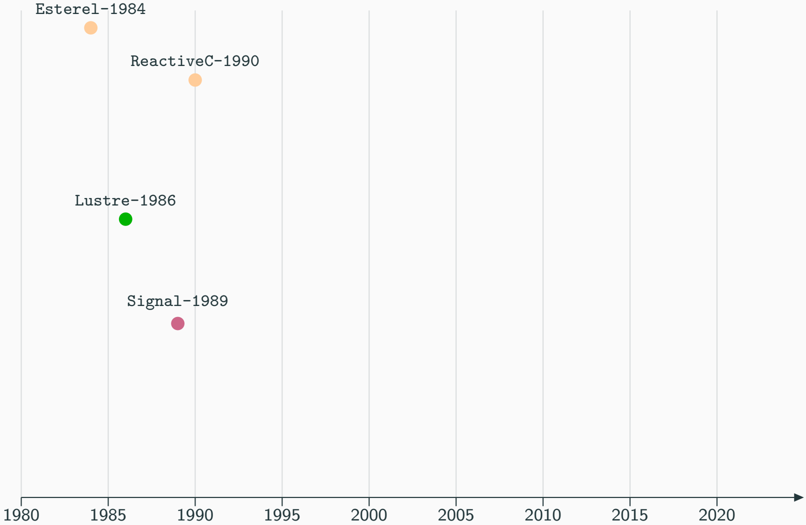
Radiator example (using Esterel [BG92])

```
input OFF, TEMP(float);
do
  signal SWITCH = bot in
      present SWITCH then // ...
                     else nothing end
    ||
      if needs_to_switch(?TEMP) then emit SWITCH
                                else nothing end
watching OFF;
```

Timeline

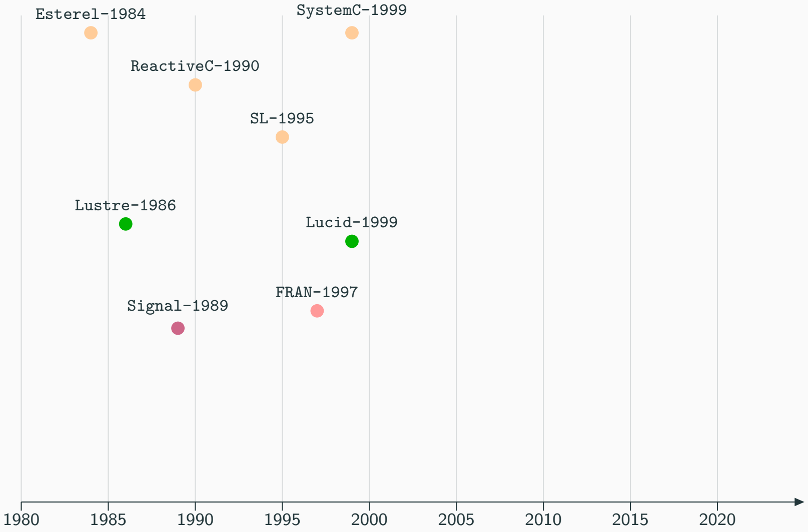


Timeline



Esterel-1984
ReactiveC-1990
Lustre-1986
Signal-1989
1980
1985
1990
1995
2000
2005
2010
2015
2020

Timeline


Esterel-1984
SystemC-1999
ReactiveC-1990
SL-1995
Lustre-1986
Lucid-1999
FRAN-1997
Signal-1989
1980
1985
1990
1995
2000
2005
2010
2015
2020

Reactive Programming in 90's

Functional Reactive Animation [EH97]
FRAn is a functional reactive programming language. The purpose of FRAN was to simplify the development of complex 2D/3D animation program.

Reactive Programming in 90's

Functional Reactive Animation [EH97]
FRAn is a functional reactive programming language. The purpose of FRAN was to simplify the development of complex 2D/3D animation program.

Principle
Data flows are split into two categories:

- $\mathit{Behavior}\ A : \mathit{Time} \to A$
- $\mathit{Event}\ A : \mathit{Time} \times A$

The program has a behavior which is a composition of behaviors and events allow us to modify the behavior.

Reactive Programming in 90's

Functional Reactive Animation [EH97]
FRAn is a functional reactive programming language. The purpose of FRAN was to simplify the development of complex 2D/3D animation program.

Principle
Data flows are split into two categories:

- $\textit{Behavior } A : \textit{Time} \rightarrow A$
- $\textit{Event } A : \textit{Time} \times A$

The program has a behavior which is a composition of behaviors and events allow us to modify the behavior.

Examples

```
slowByTwo :: Behavior Time
slowByTwo = lift (\t → t / 2) time
```

Principle
Data flows are split into two categories:

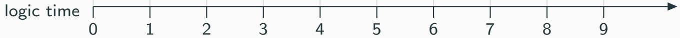
- *Behavior* A : *Time* $\rightarrow A$
- *Event* A : *Time* $\times A$

The program has a behavior which is a composition of behaviors and events allow us to modify the behavior.

Examples

```
slowByTwo :: Behavior Time
slowByTwo = lift (\t → t / 2) time
```

time	0	1	2	3	4	5	6	7	8	···
slowByTwo	0	0	1	1	2	2	3	3	4	···



Reactive Programming in 90's

Principle
Data flows are split into two categories:

- *Behavior* A : *Time* $\rightarrow A$
- *Event* A : *Time* $\times A$

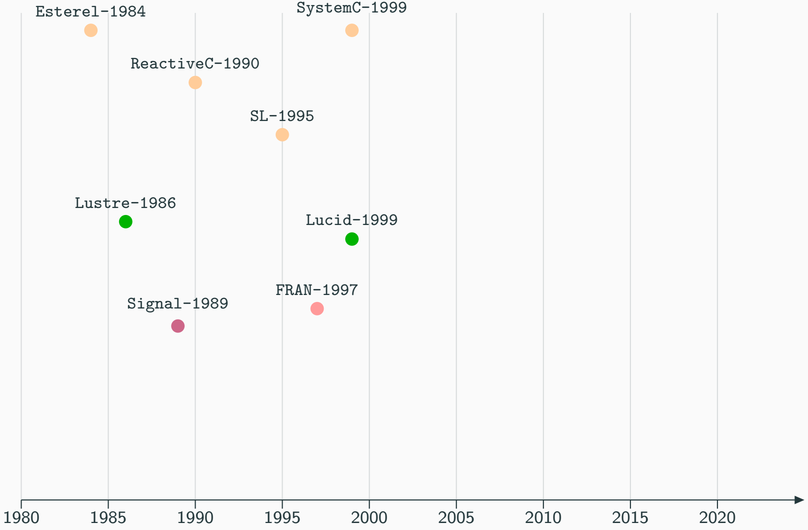
The program has a behavior which is a composition of behaviors and events allow us to modify the behavior.

Examples

```
switchWhenFour :: Behavior Int
switchWhenFour = time untilB (constEv 4 slowByTwo)
```

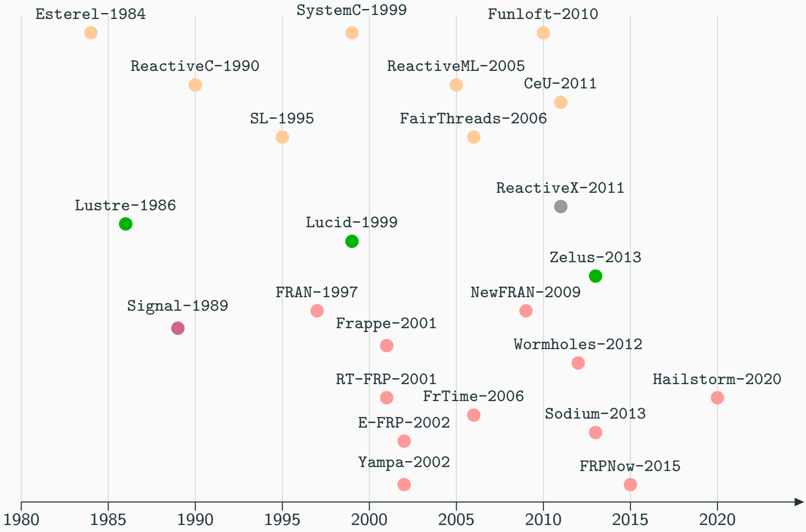
logic time	0	1	2	3	4	5	6	7	8	9
time	0	1	2	3	4	5	6	7	8	···
slowByTwo	0	0	1	1	2	2	3	3	4	···
switchWhenFour	0	1	2	3	2	2	3	3	4	···

Timeline



Esterel-1984
SystemC-1999
ReactiveC-1990
SL-1995
Lustre-1986
Lucid-1999
FRAN-1997
Signal-1989
1980
1985
1990
1995
2000
2005
2010
2015
2020

Timeline


Esterel-1984
SystemC-1999
Funloft-2010
ReactiveC-1990
ReactiveML-2005
CeU-2011
SL-1995
FairThreads-2006
ReactiveX-2011
Lustre-1986
Lucid-1999
Zelus-2013
FRAN-1997
NewFRAN-2009
Signal-1989
Frappe-2001
Wormholes-2012
RT-FRP-2001
Hailstorm-2020
FrTime-2006
Sodium-2013
E-FRP-2002
Yampa-2002
FRPNow-2015
1980
1985
1990
1995
2000
2005
2010
2015
2020

Reactive Programming now

Definition [Wik23]
In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change.

Reactive Programming now

Definition [Wik23]
In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change.

Definition [Tea23]
Reactive programming was created from the observer design pattern whose flaws needed to be corrected.

Reactive Programming now

Definition [Wik23]
In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change.

Definition [Tea23]
Reactive programming was created from the observer design pattern whose flaws needed to be corrected.

Definition [Nol21]
Reactive programming describes a design paradigm that relies on asynchronous programming logic to handle real-time updates to otherwise static content. It provides an efficient means – the use of automated data streams – to handle data updates to content whenever a user makes an inquiry.

Reactive Programming now

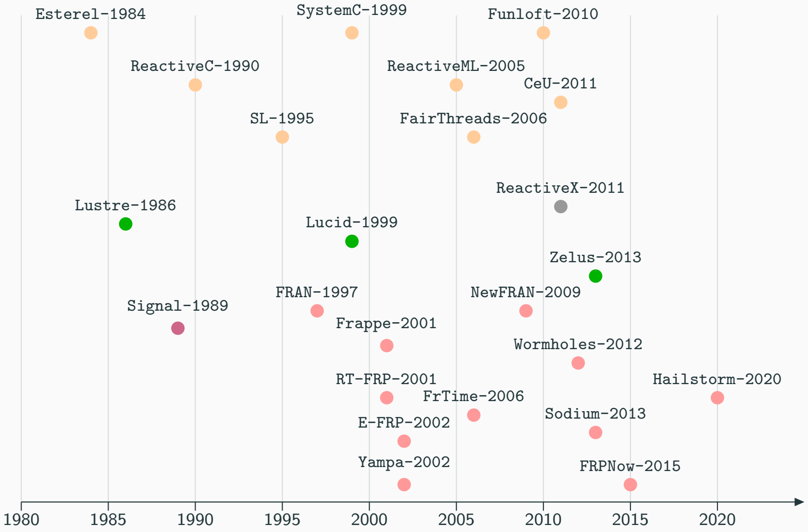
Definition [Wik23]
In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change.

Definition [Tea23]
Reactive programming was created from the observer design pattern whose flaws needed to be corrected.

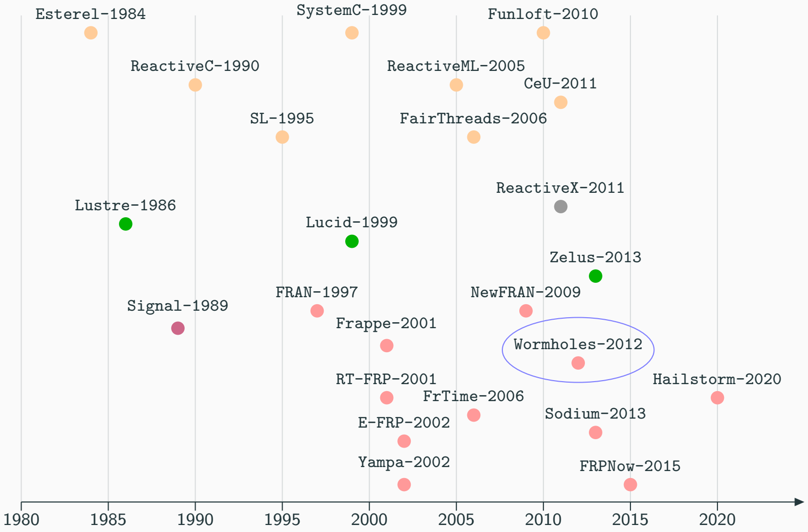
Definition [Nol21]
Reactive programming describes a design paradigm that relies on asynchronous programming logic to handle real-time updates to otherwise static content. It provides an efficient means – the use of automated data streams – to handle data updates to content whenever a user makes an inquiry.

My Definition
Reactive programming is a programming paradigm concerned with data streams handled with a synchronous or an asynchronous style in order to preserve the coherence of the program.

Timeline


Esterel-1984
SystemC-1999
Funloft-2010
ReactiveC-1990
ReactiveML-2005
CeU-2011
SL-1995
FairThreads-2006
ReactiveX-2011
Lustre-1986
Lucid-1999
Zelus-2013
FRAN-1997
NewFRAN-2009
Signal-1989
Frappe-2001
Wormholes-2012
RT-FRP-2001
Hailstorm-2020
FrTime-2006
Sodium-2013
E-FRP-2002
Yampa-2002
FRPNow-2015
1980
1985
1990
1995
2000
2005
2010
2015
2020

Timeline



Esterel-1984
SystemC-1999
Funloft-2010
ReactiveC-1990
ReactiveML-2005
CeU-2011
SL-1995
FairThreads-2006
ReactiveX-2011
Lustre-1986
Lucid-1999
Zelus-2013
FRAN-1997
NewFRAN-2009
Signal-1989
Frappe-2001
Wormholes-2012
RT-FRP-2001
Hailstorm-2020
FrTime-2006
Sodium-2013
E-FRP-2002
Yampa-2002
FRPNow-2015
1980
1985
1990
1995
2000
2005
2010
2015
2020

Table of Contents

Table of Contents

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

The Wormholes syntax

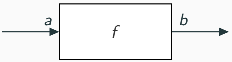
Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

Arrow based FRP
Yampa [Hud+02] is an FRP language that use Arrows in order to prevent some drawbacks of FRAN. The main idea is that all computations on signal are done indirectly via a *signal function* ($SF\ a\ b$).

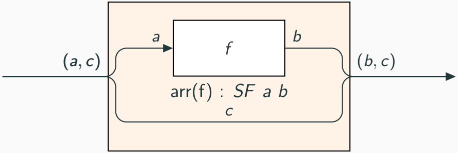
Arrow based FRP

Yampa [Hud+02] is an FRP language that use Arrows in order to prevent some drawbacks of FRAN. The main idea is that all computations on signal are done indirectly via a *signal function* ($SF\ a\ b$).

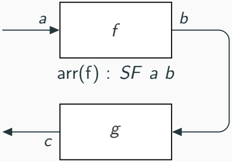
Arrow's syntax



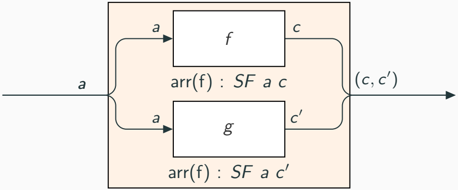
arr : $(a \to b) \to SF\ a\ b$



first : $SF\ a\ b \to SF\ (a, c)\ (b, c)$



arr(g) : $SF\ b\ c$



&&& : $SF\ a\ b \to SF\ (a, c)\ (b, c)$

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

Syntax

$$\begin{array}{llll} \text{Expression} & t, t_1, t_2 & ::= & x \mid () \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \mid \lambda x.t \mid t_1\ t_2 \\ & & & \mid \texttt{arr}(t) \mid \texttt{first}(t) \mid t_1 \ggg t_2 \mid \texttt{rsf}[r] \\ & & & \mid \texttt{wormhole}[r_{read}, r_{write}](t_i; t) \end{array}$$

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

Syntax

$$
\begin{array}{llll}
\text{Expression} & t, t_1, t_2 & ::= & x \mid () \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \mid \lambda x.t \mid t_1\ t_2 \\
 & & & \mid \texttt{arr}(t) \mid \texttt{first}(t) \mid t_1 \ggg t_2 \mid \texttt{rsf}[r] \\
 & & & \mid \texttt{wormhole}[r_{read}, r_{write}](t_i; t)
\end{array}
$$

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

Syntax

$$
\begin{array}{llll}
\text{Expression} & t, t_1, t_2 & ::= & x \mid () \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \mid \lambda x.t \mid t_1\ t_2 \\
 & & & \mid \texttt{arr}(t) \mid \texttt{first}(t) \mid t_1 \ggg t_2 \mid \texttt{rsf}[r] \\
 & & & \mid \texttt{wormhole}[r_{read}, r_{write}](t_i; t)
\end{array}
$$

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

Syntax

$$\begin{array}{llll} \text{Expression} & t, t_1, t_2 & ::= & x \mid () \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \mid \lambda x.t \mid t_1\ t_2 \\ & & & \mid \texttt{arr}(t) \mid \texttt{first}(t) \mid t_1 \ggg t_2 \mid \texttt{rsf}[r] \\ & & & \mid \texttt{wormhole}[r_{read}, r_{write}](t_i; t) \end{array}$$

The Wormholes syntax

Introduction [WH12]
Wormholes is a functional reactive programming language based on arrows, a generalization of Monads like Yampa. It integrates directly the use of information from the outside world with a specific type of signal function named *rsf*.

Syntax

$$
\begin{array}{llll}
\text{Expression} & t, t_1, t_2 & ::= & x \mid () \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \mid \lambda x.t \mid t_1\ t_2 \\
& & & \mid \texttt{arr}(t) \mid \texttt{first}(t) \mid t_1 \ggg t_2 \mid \texttt{rsf}[r] \\
& & & \mid \texttt{wormhole}[r_{read}, r_{write}](t_i; t)
\end{array}
$$

Resource
A resource signal function is a non-local one-way communication channel. Reference can be simulated by two resources: one for reading the other for writing.

The Wormholes syntax

Hypothesis

- `minutes` is a getter resource for the current number of minutes;
- `hours` is a getter resource for the current number of hours;
- `console` is a setter resource for display in the terminal.

The Wormholes syntax

Hypothesis

- `minutes` is a getter resource for the current number of minutes;
- `hours` is a getter resource for the current number of hours;
- `console` is a setter resource for display in the terminal.

Examples

```
displayTime = rsf[minutes] >>> arr (\n → ((),n)) >>>
                 first (rsf[hours]) >>> rsf[console]
```

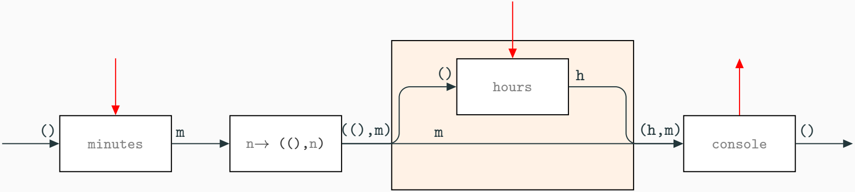
The Wormholes syntax

Hypothesis

- `minutes` is a getter resource for the current number of minutes;
- `hours` is a getter resource for the current number of hours;
- `console` is a setter resource for display in the terminal.

Examples

```
displayTime = rsf[minutes] >>> arr (\n → ((),n)) >>>
                     first (rsf[hours]) >>> rsf[console]
```



The Wormholes syntax

Examples

```
counter = rsf[ctime] >>> arr(\x.((),x)) >>>
            wormhole[cpt_r, cpt_w](0,
               first(rsf[cpt_r])>>>
               arr(\(cpt,ct) → (cpt+ct,cpt) ) >>>
               first(rsf[cpt_w]) >>> arr (fst))
```

The Wormholes syntax

Examples

```
counter = rsf[ctime] >>> arr(\x.((),x)) >>>
                wormhole[cpt_r, cpt_w](0,
                    first(rsf[cpt_r])>>>
                    arr(\(cpt,ct) → (cpt+ct,cpt) ) >>>
                    first(rsf[cpt_w]) >>> arr (fst))
```

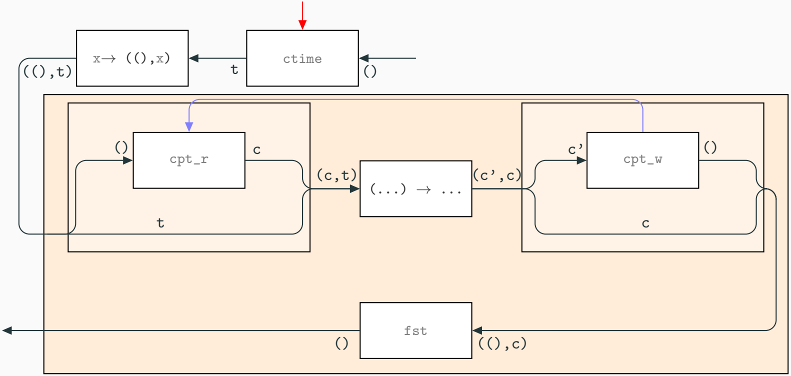


Table of Contents

Typing

Types

$$
\begin{array}{rrcl}
\text{Resource Type} & t & ::= & \langle \tau_{in}, \tau_{out} \rangle \\
\text{Types} & \tau, \tau_1, \tau_2 & ::= & \mathit{unit} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
 & & & \mid \tau_1 \overset{\{r_1;\ldots\}}{\rightsquigarrow} \tau_2
\end{array}
$$

Typing

Types

$$
\begin{array}{rrcl}
\text{Resource Type} & t & ::= & \langle \tau_{in}, \tau_{out} \rangle \\
\text{Types} & \tau, \tau_1, \tau_2 & ::= & \mathit{unit} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
 & & & \mid \tau_1 \overset{\{r_1;\ldots\}}{\rightsquigarrow} \tau_2
\end{array}
$$

Counting used resources
Reactive function type carries the set of used resources. Consequently, the typing serves as a safeguard for the correct use of the language.

Typing

Types

$$
\begin{array}{rrcl}
\text{Resource Type} & t & ::= & \langle \tau_{in}, \tau_{out} \rangle \\
\text{Types} & \tau, \tau_1, \tau_2 & ::= & unit \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\
& & & \mid \tau_1 \overset{\{r_1;\ldots\}}{\rightsquigarrow} \tau_2
\end{array}
$$

Counting used resources

Reactive function type carries the set of used resources. Consequently, the typing serves as a safeguard for the correct use of the language.

Example

`rsf[minutes]` will be typed as follows: $unit \overset{\{minutes\}}{\rightsquigarrow} int$.

Rules

$$\frac{\Gamma, \mathcal{R} \vdash f : \tau_1 \rightarrow \tau_2}{\Gamma, \mathcal{R} \vdash \mathtt{arr}(f) : \tau_1 \overset{\emptyset}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Arr} \qquad \frac{\Gamma, \mathcal{R} \vdash sf : \tau_1 \overset{R}{\rightsquigarrow} \tau_3}{\Gamma, \mathcal{R} \vdash \mathtt{first}(sf) : \tau_1 \times \tau_2 \overset{R}{\rightsquigarrow} \tau_3 \times \tau_2}\ \texttt{Ty-First}$$

$$\frac{\begin{array}{cc} \Gamma, \mathcal{R} \vdash sf_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau_3 & R_1 \cup R_2 = R \\ \Gamma, \mathcal{R} \vdash sf_2 : \tau_3 \overset{R_2}{\rightsquigarrow} \tau_2 & R_1 \cap R_2 = \emptyset \end{array}}{\Gamma, \mathcal{R} \vdash sf_1 \ggg sf_2 : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Comp}$$

$$\frac{}{\Gamma, \mathcal{R} \cup \{r : \langle \tau_{in}, \tau_{out} \rangle\} \vdash \mathtt{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}\ \texttt{Ty-Rsf}$$

$$\frac{\begin{array}{c} \Gamma, \mathcal{R} \vdash t : \tau \quad R = R' \backslash \{r_{read}; r_{write}\} \\ \Gamma, \mathcal{R} \ \cup \ \{r_{read} : \langle (), \tau \rangle ; r_{write} : \langle \tau, () \rangle\} \vdash sf : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma, \mathcal{R} \vdash \mathtt{wormhole}[r_{read}, r_{write}](t; sf) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Wh}$$

Typing

Rules

$$\frac{\Gamma, \mathcal{R} \vdash f : \tau_1 \rightarrow \tau_2}{\Gamma, \mathcal{R} \vdash \texttt{arr}(f) : \tau_1 \overset{\emptyset}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Arr} \qquad \frac{\Gamma, \mathcal{R} \vdash sf : \tau_1 \overset{R}{\rightsquigarrow} \tau_3}{\Gamma, \mathcal{R} \vdash \texttt{first}(sf) : \tau_1 \times \tau_2 \overset{R}{\rightsquigarrow} \tau_3 \times \tau_2}\ \texttt{Ty-First}$$

$$\frac{\begin{array}{cc} \Gamma, \mathcal{R} \vdash sf_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau_3 & R_1 \cup R_2 = R \\ \Gamma, \mathcal{R} \vdash sf_2 : \tau_3 \overset{R_2}{\rightsquigarrow} \tau_2 & R_1 \cap R_2 = \emptyset \end{array}}{\Gamma, \mathcal{R} \vdash sf_1 \ggg sf_2 : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Comp}$$

$$\frac{}{\Gamma, \mathcal{R} \cup \{r : \langle \tau_{in}, \tau_{out} \rangle\} \vdash \texttt{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}\ \texttt{Ty-Rsf}$$

$$\frac{\begin{array}{c} \Gamma, \mathcal{R} \vdash t : \tau \quad R = R' \backslash \{r_{read}; r_{write}\} \\ \Gamma, \mathcal{R} \ \cup \ \{r_{read} : \langle (), \tau \rangle; r_{write} : \langle \tau, () \rangle\} \vdash sf : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma, \mathcal{R} \vdash \texttt{wormhole}[r_{read}, r_{write}](t; sf) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Wh}$$

Rules

$$\frac{\Gamma, \mathcal{R} \vdash f : \tau_1 \rightarrow \tau_2}{\Gamma, \mathcal{R} \vdash \texttt{arr}(f) : \tau_1 \overset{\emptyset}{\rightsquigarrow} \tau_2} \ \texttt{Ty-Arr} \qquad \frac{\Gamma, \mathcal{R} \vdash sf : \tau_1 \overset{R}{\rightsquigarrow} \tau_3}{\Gamma, \mathcal{R} \vdash \texttt{first}(sf) : \tau_1 \times \tau_2 \overset{R}{\rightsquigarrow} \tau_3 \times \tau_2} \ \texttt{Ty-First}$$

$$\frac{\begin{array}{cc} \Gamma, \mathcal{R} \vdash sf_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau_3 & R_1 \cup R_2 = R \\ \Gamma, \mathcal{R} \vdash sf_2 : \tau_3 \overset{R_2}{\rightsquigarrow} \tau_2 & R_1 \cap R_2 = \emptyset \end{array}}{\Gamma, \mathcal{R} \vdash sf_1 \ggg sf_2 : \tau_1 \overset{R}{\rightsquigarrow} \tau_2} \ \texttt{Ty-Comp}$$

$$\frac{}{\Gamma, \mathcal{R} \cup \{r : \langle \tau_{in}, \tau_{out} \rangle\} \vdash \texttt{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}} \ \texttt{Ty-Rsf}$$

$$\frac{\begin{array}{c} \Gamma, \mathcal{R} \vdash t : \tau \quad R = R' \backslash \{r_{read}; r_{write}\} \\ \Gamma, \mathcal{R} \ \cup \ \{r_{read} : \langle (), \tau \rangle; r_{write} : \langle \tau, () \rangle\} \vdash sf : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma, \mathcal{R} \vdash \texttt{wormhole}[r_{read}, r_{write}](t; sf) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2} \ \texttt{Ty-Wh}$$

Typing

Rules

$$\frac{\Gamma, \mathcal{R} \vdash f : \tau_1 \to \tau_2}{\Gamma, \mathcal{R} \vdash \texttt{arr}(f) : \tau_1 \overset{\emptyset}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Arr} \qquad \frac{\Gamma, \mathcal{R} \vdash sf : \tau_1 \overset{R}{\rightsquigarrow} \tau_3}{\Gamma, \mathcal{R} \vdash \texttt{first}(sf) : \tau_1 \times \tau_2 \overset{R}{\rightsquigarrow} \tau_3 \times \tau_2}\ \texttt{Ty-First}$$

$$\frac{\begin{array}{cc} \Gamma, \mathcal{R} \vdash sf_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau_3 & R_1 \cup R_2 = R \\ \Gamma, \mathcal{R} \vdash sf_2 : \tau_3 \overset{R_2}{\rightsquigarrow} \tau_2 & R_1 \cap R_2 = \emptyset \end{array}}{\Gamma, \mathcal{R} \vdash sf_1 \ggg sf_2 : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Comp}$$

$$\frac{}{\Gamma, \mathcal{R} \cup \{r : \langle \tau_{in}, \tau_{out} \rangle\} \vdash \texttt{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}\ \texttt{Ty-Rsf}$$

$$\frac{\begin{array}{c} \Gamma, \mathcal{R} \vdash t : \tau \quad R = R' \backslash \{r_{read}; r_{write}\} \\ \Gamma, \mathcal{R} \ \cup \ \{r_{read} : \langle (), \tau \rangle; r_{write} : \langle \tau, () \rangle\} \vdash sf : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma, \mathcal{R} \vdash \texttt{wormhole}[r_{read}, r_{write}](t; sf) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Wh}$$

Rules

$$\frac{\Gamma, \mathcal{R} \vdash f : \tau_1 \rightarrow \tau_2}{\Gamma, \mathcal{R} \vdash \mathtt{arr}(f) : \tau_1 \overset{\emptyset}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Arr}$$

$$\frac{\Gamma, \mathcal{R} \vdash sf : \tau_1 \overset{R}{\rightsquigarrow} \tau_3}{\Gamma, \mathcal{R} \vdash \mathtt{first}(sf) : \tau_1 \times \tau_2 \overset{R}{\rightsquigarrow} \tau_3 \times \tau_2}\ \texttt{Ty-First}$$

$$\frac{\begin{array}{cc} \Gamma, \mathcal{R} \vdash sf_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau_3 & R_1 \cup R_2 = R \\ \Gamma, \mathcal{R} \vdash sf_2 : \tau_3 \overset{R_2}{\rightsquigarrow} \tau_2 & R_1 \cap R_2 = \emptyset \end{array}}{\Gamma, \mathcal{R} \vdash sf_1 \ggg sf_2 : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Comp}$$

$$\frac{}{\Gamma, \mathcal{R} \cup \{r : \langle \tau_{in}, \tau_{out} \rangle\} \vdash \mathtt{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}\ \texttt{Ty-Rsf}$$

$$\frac{\begin{array}{c} \Gamma, \mathcal{R} \vdash t : \tau \quad R = R' \backslash \{r_{read}; r_{write}\} \\ \Gamma, \mathcal{R} \ \cup \ \{r_{read} : \langle (), \tau \rangle; r_{write} : \langle \tau, () \rangle\} \vdash sf : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma, \mathcal{R} \vdash \mathtt{wormhole}[r_{read}, r_{write}](t; sf) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Wh}$$

Rules

$$\frac{\Gamma, \mathcal{R} \vdash f : \tau_1 \rightarrow \tau_2}{\Gamma, \mathcal{R} \vdash \mathtt{arr}(f) : \tau_1 \overset{\emptyset}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Arr} \qquad \frac{\Gamma, \mathcal{R} \vdash sf : \tau_1 \overset{R}{\rightsquigarrow} \tau_3}{\Gamma, \mathcal{R} \vdash \mathtt{first}(sf) : \tau_1 \times \tau_2 \overset{R}{\rightsquigarrow} \tau_3 \times \tau_2}\ \texttt{Ty-First}$$

$$\frac{\begin{array}{cc} \Gamma, \mathcal{R} \vdash sf_1 : \tau_1 \overset{R_1}{\rightsquigarrow} \tau_3 & R_1 \cup R_2 = R \\ \Gamma, \mathcal{R} \vdash sf_2 : \tau_3 \overset{R_2}{\rightsquigarrow} \tau_2 & R_1 \cap R_2 = \emptyset \end{array}}{\Gamma, \mathcal{R} \vdash sf_1 \ggg sf_2 : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Comp}$$

$$\frac{}{\Gamma, \mathcal{R} \cup \{r : \langle \tau_{in}, \tau_{out} \rangle\} \vdash \mathtt{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}\ \texttt{Ty-Rsf}$$

$$\frac{\begin{array}{c} \Gamma, \mathcal{R} \vdash t : \tau \quad R = R' \backslash \{r_{read}; r_{write}\} \\ \Gamma, \mathcal{R} \ \cup \ \{r_{read} : \langle (), \tau \rangle; r_{write} : \langle \tau, () \rangle\} \vdash sf : \tau_1 \overset{R'}{\rightsquigarrow} \tau_2 \end{array}}{\Gamma, \mathcal{R} \vdash \mathtt{wormhole}[r_{read}, r_{write}](t; sf) : \tau_1 \overset{R}{\rightsquigarrow} \tau_2}\ \texttt{Ty-Wh}$$

Table of Contents

Semantics

Evaluation transition
A small-step operational semantics for non-reactive expression. Rules are not given in the paper but it is defined as a lazy evaluation model.

Semantics

Evaluation transition
A small-step operational semantics for non-reactive expression. Rules are not given in the paper but it is defined as a lazy evaluation model.

Values
All reactive terms are values. So subterms of reactives terms are not reduced.

Semantics

Evaluation transition
A small-step operational semantics for non-reactive expression. Rules are not given in the paper but it is defined as a lazy evaluation model.

Values
All reactive terms are values. So subterms of reactives terms are not reduced.

Functional transition
A big-step operational semantics for reactive expression.

Semantics

Evaluation transition
A small-step operational semantics for non-reactive expression. Rules are not given in the paper but it is defined as a lazy evaluation model.

Values
All reactive terms are values. So subterms of reactives terms are not reduced.

Functional transition
A big-step operational semantics for reactive expression.

Temporal transition
A unique transition to move from the current instant to the next instant.

Functional transition

Functional transition
A big-step operational semantics for reactive expression. The functional transition is defined as follows: $(V, t_v, sf) \Rrightarrow (V', t_v', sf', W)$

- V, V' are resource environments
- t_v, t_v' are stream values
- sf, sf' are signal functions
- W is the set of virtual resources

Functional transition

Functional transition
A big-step operational semantics for reactive expression. The functional transition is defined as follows: $(V, t_v, sf) \Rrightarrow (V', t_v', sf', W)$

- V, V' are resource environments
- t_v, t_v' are stream values
- sf, sf' are signal functions
- W is the set of virtual resources

Rules

$$\frac{}{(V, t_v, \texttt{arr}(f)) \Rrightarrow (V, f\ t_v, \texttt{arr}(f), \emptyset)}\ \texttt{FT-Arr}$$

Functional transition

Functional transition
A big-step operational semantics for reactive expression. The functional transition is defined as follows: $(V, t_v, sf) \Rrightarrow (V', t_v', sf', W)$

- V, V' are resource environments
- t_v, t_v' are stream values
- sf, sf' are signal functions
- W is the set of virtual resources

Rules

$$\frac{sf \mapsto^* sf' \quad (V, t_1, sf') \Rrightarrow (V_1, t_1', sf'', W)}{(V, (t_1, t_2), \texttt{first}(sf)) \Rrightarrow (V_1, (t_1', t_2), \texttt{first}(sf''), W)} \ \texttt{FT-First}$$

Functional transition

Functional transition
A big-step operational semantics for reactive expression. The functional transition is defined as follows: $(V, t_v, sf) \Rrightarrow (V', t'_v, sf', W)$

- V, V' are resource environments
- t_v, t'_v are stream values
- sf, sf' are signal functions
- W is the set of virtual resources

Rules

$$\frac{\begin{array}{c} sf_1 \mapsto^* sf'_1 \quad (V, t_v, sf'_1) \Rrightarrow (V_1, t'_v, sf''_1, W_1) \\ sf_2 \mapsto^* sf'_2 \quad (V_1, t'_v, sf'_2) \Rrightarrow (V_2, t''_v, sf''_2, W_2) \end{array}}{(V, t_v, sf_1 \ggg sf_2) \Rrightarrow (V_2, t''_v, sf''_1 \ggg sf''_2, W_1 \cup W_2)} \ \texttt{FT-Comp}$$

Functional transition

Functional transition
A big-step operational semantics for reactive expression. The functional transition is defined as follows: $(V, t_v, sf) \Rrightarrow (V', t'_v, sf', W)$

- V, V' are resource environments
- t_v, t'_v are stream values
- sf, sf' are signal functions
- W is the set of virtual resources

Rules

$$\frac{}{(V \cup \{(r, t_r, .)\}, t_v, \texttt{rsf}[r]) \Rrightarrow (V \cup \{(r, ., t_v)\}, t_r, \texttt{rsf}[r], \emptyset)}\ \texttt{FT-Rsf}$$

Functional transition

Functional transition
A big-step operational semantics for reactive expression. The functional transition is defined as follows: $(V, t_v, sf) \Rrightarrow (V', t'_v, sf', W)$

- V, V' are resource environments
- t_v, t'_v are stream values
- sf, sf' are signal functions
- W is the set of virtual resources

Rules

$$\frac{sf \mapsto^* sf' \quad (V \cup \{(r_r, t_i, .); (r_w, (), .)\}, t_v, sf') \Rrightarrow (V_1, t'_v, sf'', W)}{(V, t_v, \texttt{wormhole}[r_r, r_w](t_i; sf)) \Rrightarrow (V_1, t'_v, sf'', W \cup \{[r_r, r_w, t_i]\})} \ \texttt{FT-Wh}$$

Table of Contents

Properties of Wormholes

Properties

Properties of Wormholes

- Progress and preservation for the evaluation transition

Properties

Properties of Wormholes

- Progress and preservation for the evaluation transition
- Progress and preservation for the functional transition

Properties

Properties of Wormholes

- Progress and preservation for the evaluation transition
- Progress and preservation for the functional transition
- Progress and preservation for the temporal transition

Properties

Properties of Wormholes

- Progress and preservation for the evaluation transition
- Progress and preservation for the functional transition
- Progress and preservation for the temporal transition
- Safety on the use of resources

Properties

Properties of Wormholes

- Progress and preservation for the evaluation transition
- Progress and preservation for the functional transition
- Progress and preservation for the temporal transition
- Safety on the use of resources

Properties

Properties of Wormholes

- Progress and preservation for the evaluation transition
- Progress and preservation for the functional transition
- Progress and preservation for the temporal transition
- Safety on the use of resources

Table of Contents

Table of Contents

Formalization of a subset of Wormholes

Arrow language

For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

Formalization of a subset of Wormholes

Arrow language
For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

The theorem of Progress
The formalization was going well until the theorem of progress of the functional transition.

Formalization of a subset of Wormholes

Arrow language

For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

The theorem of Progress

The formalization was going well until the theorem of progress of the functional transition.

`Theorem progress :` $\forall\ t\ \ ts\ \ \tau 1\ \ \tau 2,$

$\emptyset \vdash t \in (\tau 1 \rightsquigarrow \tau 2) \to \emptyset \vdash ts \in \tau 1 \to \exists\ \ ts'\ \ t', \mid ts; t \mid\Rrightarrow\mid ts'; t' \mid.$

Formalization of a subset of Wormholes

Arrow language
For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

The theorem of Progress
The formalization was going well until the theorem of progress of the functional transition.

`Theorem progress :` $\forall\ t\ \ ts\ \ \tau 1\ \ \tau 2,$
$\emptyset \vdash t \in (\tau 1 \rightsquigarrow \tau 2) \rightarrow \emptyset \vdash ts \in \tau 1 \rightarrow \exists\ \ ts'\ \ t', \mid ts; t \mid \Rrightarrow \mid ts'; t' \mid.$

Why ?
Two reasons:

Formalization of a subset of Wormholes

Arrow language
For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

The theorem of Progress
The formalization was going well until the theorem of progress of the functional transition.

`Theorem progress :` $\forall\ t\ \ ts\ \ \tau 1\ \ \tau 2,$
$\emptyset \vdash t \in (\tau 1 \rightsquigarrow \tau 2) \rightarrow \emptyset \vdash ts \in \tau 1 \rightarrow \exists\ \ ts'\ \ t', \mid ts; t \mid\Rrightarrow\mid ts'; t' \mid.$

Why ?
Two reasons:

- `FT-First` asks an implicit normalisation of the stream value;

Formalization of a subset of Wormholes

Arrow language
For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

The theorem of Progress
The formalization was going well until the theorem of progress of the functional transition.

`Theorem progress :` $\forall\ t\ \ ts\ \ \tau 1\ \ \tau 2,$
$\emptyset \vdash t \in (\tau 1 \rightsquigarrow \tau 2) \to \emptyset \vdash ts \in \tau 1 \to \exists\ \ ts'\ \ t', \mid ts; t \mid\Rrightarrow\mid ts'; t' \mid.$

Why ?
Two reasons:

- `FT-First` asks an implicit normalisation of the stream value;
- If the term t is not a value, there is no progress;

Formalization of a subset of Wormholes

Arrow language
For the sake of simplicity we start with a subset of Wormholes. We will take only the lambda terms and the arrow terms.

The theorem of Progress
The formalization was going well until the theorem of progress of the functional transition.

`Theorem progress :` $\forall\ t\ \ ts\ \ \tau 1\ \ \tau 2,$
$\emptyset \vdash t \in (\tau 1 \rightsquigarrow \tau 2) \to \emptyset \vdash ts \in \tau 1 \to \exists\ \ ts'\ \ t', \mid ts; t \mid\Rrightarrow\mid ts'; t' \mid.$

Why ?
Two reasons:

- `FT-First` asks an implicit normalisation of the stream value;
- If the term t is not a value, there is no progress;
- The concept of value provoke an interlacing between evaluation transition and functional transition.

The `FT-First` rule

$$\frac{sf \mapsto^* sf' \quad (V, t_1, sf') \Rrightarrow (V_1, t_1', sf'', W)}{(V, (t_1, t_2), \texttt{first}(sf)) \Rrightarrow (V_1, (t_1', t_2), \texttt{first}(sf''), W)} \ \texttt{FT-First}$$

The FT-First rule

$$\frac{sf \mapsto^* sf' \quad (V, t_1, sf') \Rrightarrow (V_1, t_1', sf'', W)}{(V, {\color{red}(t_1, t_2)}, \texttt{first}(sf)) \Rrightarrow (V_1, (t_1', t_2), \texttt{first}(sf''), W)} \; \texttt{FT-First}$$

Behavior of the stream

The stream do not have to be reduced in most case, except in this case where it needs to be normalized.

The $\texttt{FT-First}$ rule

$$\frac{sf \mapsto^* sf' \quad (V, t_1, sf') \Rrightarrow (V_1, t_1', sf'', W)}{(V, {\color{red}(t_1, t_2)}, \texttt{first}(sf)) \Rrightarrow (V_1, (t_1', t_2), \texttt{first}(sf''), W)}\ \texttt{FT-First}$$

Behavior of the stream
The stream do not have to be reduced in most case, except in this case where it needs to be normalized.

Example
The configuration below is stuck with the current version of the rules.

$$(V, \lambda x.(t_1, x)\ t_2, \texttt{first}(sf)) \Rrightarrow (...)$$

The FT-First rule

$$\frac{sf \mapsto^* sf' \quad (V, t_1, sf') \Rrightarrow (V_1, t_1', sf'', W)}{(V, {\color{red}(t_1, t_2)}, \texttt{first}(sf)) \Rrightarrow (V_1, (t_1', t_2), \texttt{first}(sf''), W)} \ \texttt{FT-First}$$

Behavior of the stream
The stream do not have to be reduced in most case, except in this case where it needs to be normalized.

Example
The configuration below is stuck with the current version of the rules.

$$(V, \lambda x.(t_1, x)\ t_2, \texttt{first}(sf)) \Rrightarrow (...)$$

Modification done

$$\frac{{\color{red}t_s \mapsto^* (t_1, t_2)} \quad sf \mapsto^* sf' \quad (V, t_1, sf') \Rrightarrow (V_1, t_1', sf'', W)}{(V, {\color{red}t_s}, \texttt{first}(sf)) \Rrightarrow (V_1, (t_1', t_2), \texttt{first}(sf''), W)} \ \texttt{FT-First}$$

Stuck at start

Using the functional transition implicitly asks the expression to be normalized. Consequently, even a well typed expression can be stuck at the start.

Stuck at start

Using the functional transition implicitly asks the expression to be normalized. Consequently, even a well typed expression can be stuck at the start.

Example

$$\emptyset, \emptyset \vdash \lambda x.(arr(\lambda y.(x + y)))\ 4 \in int \overset{\emptyset}{\rightsquigarrow} int$$

Stuck at start

Using the functional transition implicitly asks the expression to be normalized. Consequently, even a well typed expression can be stuck at the start.

Example

$$\emptyset, \emptyset \vdash \lambda x.(arr(\lambda y.(x + y)))\ 4 \in int \overset{\emptyset}{\rightsquigarrow} int$$

Stuck at start

Using the functional transition implicitly asks the expression to be normalized. Consequently, even a well typed expression can be stuck at the start.

Example

$$\emptyset, \emptyset \vdash \lambda x.(arr(\lambda y.(x + y)))\ 4 \in int \overset{\emptyset}{\rightsquigarrow} int$$

Modification done

We add a rule for lift evaluation transition into functional transition. A side effect of that is a simplification of other rules.

$$\frac{t \mapsto t' \quad (V, t_s, t') \Rrightarrow (V_1, t'_s, sf, W)}{(V, t_s, t) \Rrightarrow (V_1, t'_s, sf, W)}\ \texttt{FT-Eval}$$

The interlacing issue

Explanation
Because of the fact that reactive terms are values, each time we want to use a functional transition we have to apply multiple times the evaluation transition on subterms.

Explanation
Because of the fact that reactive terms are values, each time we want to use a functional transition we have to apply multiple times the evaluation transition on subterms.

Example
Do a little example on a board if possible (otherwise make big gesture).

The interlacing issue

Explanation
Because of the fact that reactive terms are values, each time we want to use a functional transition we have to apply multiple times the evaluation transition on subterms.

Example
Do a little example on a board if possible (otherwise make big gesture).

Can we avoid this ?
The modification chosen was to define reactive terms as values only if their subterms are also values and let the evaluation transition pass through the reactive terms.

Table of Contents

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

- use the locally nameless representation

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

- use the locally nameless representation
- use the HOAS/PHOAS

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

- use the locally nameless representation
- use the HOAS/PHOAS
- define our equivalence

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

- use the locally nameless representation $\longrightarrow$ break because of the `Ty-wh` rule
- use the HOAS/PHOAS
- define our equivalence

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

- use the locally nameless representation $\longrightarrow$ break because of the `Ty-wh` rule
- use the HOAS/PHOAS $\longrightarrow$ not chosen because never used
- define our equivalence

The equivalence issue

Explanation
Adding the remaining terms of the Wormholes's syntax brings with him the feared renaming problem.

Example

$$\texttt{wormhole}[r_r, r_w](i; rsf[r_r]) \iff \texttt{wormhole}[r_{read}, r_w](i; rsf[r_{read}])$$

Possibility
Several representation can possibly saved us:

- use the locally nameless representation $\longrightarrow$ break because of the `Ty-wh` rule
- use the HOAS/PHOAS $\longrightarrow$ not chosen because never used
- define our equivalence

References i

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (Nov. 1, 1992), pp. 87–152. ISSN: 0167-6423. DOI: 10.1016/0167-6423(92)90005-V.

[EH97] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Ed. by Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman. Amsterdam, The Netherlands: ACM, 1997, pp. 263–273. DOI: 10.1145/258948.258973.

[Hal98] Nicolas Halbwachs. "Synchronous programming of reactive systems". In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 1–16. ISBN: 978-3-540-69339-0. DOI: 10.1007/BFb0028726.

[Hud+02] Paul Hudak et al. "Arrows, Robots, and Functional Reactive Programming". In: *Advanced Functional Programming, 4th International School (AFP)*. Ed. by Johan Jeuring and Simon L. Peyton Jones. Vol. 2638. Lecture Notes in Computer Science. Oxford, UK: Springer, 2002, pp. 159–187. DOI: 10.1007/978-3-540-44833-4_6.

References ii

[Nol21] Tom Nolle. *Reactive programming*. https://www.techtarget.com/searchapparchitecture/definition/reactive-programming. 2021.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science (FOCS)*. Providence, Rhode Island, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[Tea23] Ryax Team. *La programmation réactive : c'est quoi ?* https://ryax.tech/fr/la-programmation-reactive-cest-quoi/. 2023.

[WH12] Daniel Winograd-Cort and Paul Hudak. "Wormholes: introducing effects to FRP". In: *ACM SIGPLAN Notices* 47.12 (Sept. 13, 2012), pp. 91–104. ISSN: 0362-1340. DOI: 10.1145/2430532.2364519.

[Wik23] Wikipedia. *Reactive programming*. https://en.wikipedia.org/wiki/Reactive_programming. 2023.

Thanks for your attention !

Do you have any questions ?