Temporal Logic and Functional Reactive Programming

Sergei Winitzki

Bay Area Categories and Types

April 25, 2014

What is “reactive programming”

Transformational programs	**Reactive** programs
example: `pdflatex frp_talk.tex`	example: any GUI program
start, run, then stop	keep running indefinitely
read some input, write some output	wait for signals, send messages
execution: sequential + some parallel	“main run loop” + concurrency
difficulty: algorithms	signal/response sequences
specification: classical logic	temporal logic? flowcharts?
verification: prove it correct	model checking?
synthesis: extract code from proof	temporal synthesis?
type theory: intuitionistic logic	intuitionistic temporal logic

The uses of logic in computer science

1. Logic as a **specification language** - enables automatic verification
   - Automatic synthesis of programs from specifications?
2. (Intuitionistic) logic as **type theory** - guides language design
   - Mathematicians have already minimized the set of axioms!
3. **Logic programming** - use a decidable subset of logic
   - Very high-level language, but limited to its domain
4. **Automated theorem proving** - derive a program as a proof artifact
   - Requires advanced type systems and proof heuristics

Part 1: Introduction to temporal logic

- Let's forget all philosophy, "what is time", "what is true", modal logic...
- We want to see logic as a down-to-earth, computationally useful tool
- We begin with the computational view of classical Boolean logic

Boolean algebra: notation

- Classical propositional (Boolean) logic: T, F, $a \vee b$, $a \wedge b$, $\neg a$, $a \rightarrow b$
- A notation better adapted to school-level algebra: 1, 0, $a + b$, ab, a'
- The only "new rule" is $1 + 1 = 1$
- Define $a \rightarrow b = a' + b$
- Some identities:

$$0a = 0, \quad 1a = a, \quad a + 0 = a, \quad a + 1 = 1,$$
$$a + a = a, \quad aa = a, \quad a + a' = 1, \quad aa' = 0,$$
$$(a+b)' = a'b', \quad (ab)' = a' + b', \quad (a')' = a$$
$$a(b+c) = ab + ac, \quad (a+b)(a+c) = a + bc$$

- DNF = "expand all brackets". Some DNF simplification tricks:

$$a + ab = a, \quad a(a+b) = a,$$
$$(a \rightarrow b)(a' \rightarrow c) = ab + a'c,$$
$$(a \rightarrow x)(b \rightarrow x') = (a' + x)(b' + x') = x'a' + xb'$$

Boolean algebra: example

Of the three suspects A, B, C, only one is guilty of a crime.
Suspect A says: "B did it". Suspect B says: "C is innocent."
The guilty one is lying, the innocent ones tell the truth.

$$\phi = \left(ab'c' + a'bc' + a'b'c\right)\left(a'b + ab'\right)\left(b'c' + bc\right)$$

Simplify: expand the brackets, omit aa', bb', cc', replace $aa = a$ etc.:

$$\phi = ab'c' + 0 + 0 = ab'c'$$

The guilty one is A.

Synthesis of Boolean programs

- Specification of a "Boolean program":

If the boss is in, I need to work unless the telephone rings.
If the boss is not in, I go drink tea.

- $b =$"boss is in", $r =$"telephone rings", $w =$"I work", $w' =$"I drink tea"

$$\begin{aligned}\phi(b, r, w) &= \left(br' \to w\right)\left(b' \to w'\right) \\ &= w'\left(br'\right)' + wb = w'\left(b' + r\right) + wb\end{aligned}$$

- **Goal**: given any b and r, compute w such that $\phi(b, r, w) = 1$.
- One solution is just $\phi(b, r, w = 1)$:

$$w = \phi(b, r, 1) = 0\left(b' + r\right) + 1b = b$$

"I work if and only if the boss is in"

- (Other solutions exist, e.g. $w = br'$)

Propositional linear-time temporal logic (LTL)

- Reactive programs respond to an *infinite stream* of signals
- So let's work with *infinite boolean sequences* ("linear time")

  Boolean operations:

$$a = [a_0, a_1, a_2, ...] ; \quad b = [b_0, b_1, b_2, ...] ;$$

$$a + b = [a_0 + b_0, a_1 + b_1, ...] ; \; a' = \left[a_0', a_1', ...\right] ; \; ab = [a_0 b_0, a_1 b_1, ...]$$

  Temporal operations:

$$\begin{aligned} \text{(Next)} \quad & \mathbf{N}a = [a_1, a_2, ...] \\ \text{(Sometimes)} \quad & \mathbf{F}a = [a_0 + a_1 + a_2 + ..., \; a_1 + a_2 + ..., \; ...] \\ \text{(Always)} \quad & \mathbf{G}a = [a_0 a_1 a_2 a_3 ..., \; a_1 a_2 a_3 ..., \; a_2 a_3 ..., \; ...] \end{aligned}$$

  Other notation (from modal logic):

$$\mathbf{N}a \equiv \bigcirc a; \; \mathbf{F}a \equiv \Diamond a; \; \mathbf{G}a \equiv \Box a$$

Temporal fixpoints and the μ-calculus notation

- LTL admits *only* temporal functions defined *by fixpoints*:

$$\begin{aligned}
\mathbf{F}a &= [a_0 + a_1 + a_2 + a_3 + ..., \; a_1 + a_2 + a_3 + ..., \; ...] \\
\mathbf{F}a &= a + \mathbf{N}(\mathbf{F}a) \\
\mathbf{G}a &= [a_0 a_1 a_2 a_3 ..., \; a_1 a_2 a_3 ..., a_2 a_3 ..., \; ...] \\
\mathbf{G}a &= a\mathbf{N}(\mathbf{G}a)
\end{aligned}$$

- Notation: $\boldsymbol{\mu}$ for the least fixpoint, $\boldsymbol{\nu}$ for the greatest fixpoint

$$\mathbf{F}a = \boldsymbol{\mu}x.\,(a + \mathbf{N}x)\,; \quad \mathbf{G}a = \boldsymbol{\nu}x.\,(a(\mathbf{N}x))$$
$$\text{but} \quad \boldsymbol{\nu}x.\,(a + \mathbf{N}x) = 1; \quad \boldsymbol{\mu}x.\,(a(\mathbf{N}x)) = 0$$

- The most general fixpoints involving *only one* **N**:

$$\begin{aligned}
&(\text{weak Until}) \quad p\mathbf{U}q = \boldsymbol{\nu}x.\,(q + p(\mathbf{N}x)) \\
&(\text{strict Until}) \quad p\dot{\mathbf{U}}q = \boldsymbol{\mu}x.\,(q + p(\mathbf{N}x))
\end{aligned}$$

LTL: interpretation of "Until"

- Weak Until: $p\mathbf{U}q$ = "p holds from now on until q first becomes true"

$$p\mathbf{U}q = q + p\mathbf{N}\left(q + p\mathbf{N}\left(q + ...\right)\right)$$

Example:

$$\begin{aligned} a &= [1, 0, 0, 0, 1, 0, ...] \\ b &= [0, 1, 0, 1, 0, 1, ...] \\ a\mathbf{U}b &= [1, 1, 0, 1, 1, 1, ...] \end{aligned}$$

- Strict Until: $p\dot{\mathbf{U}}q$ = "q *must* become true, and p holds until then"
- Dualities: $(\mathbf{F}a)' = \mathbf{G}(a')$; also $(p\dot{\mathbf{U}}q)' = q'\mathbf{U}(p'q')$

LTL: temporal specification

Whenever the boss comes by my office, I will start working.
Once I start working, I will keep working until the telephone rings.

$$\mathbf{G}\left((b \to \mathbf{F}w)(w \to w\mathbf{U}r)\right) = \mathbf{G}\left(\left(b' + \mathbf{F}w\right)\left(w' + w\mathbf{U}r\right)\right)$$

Whenever the button is pressed, the dialog will appear.
The dialog will disappear after 1 minute of user inactivity.

$$\mathbf{G}\left((b \to \mathbf{F}d)(d \to \mathbf{F}t)\left(d \to d\mathbf{U}td'\right)\right)$$

- The timer t is an external event and is *not specified* here
- Difficult to say "x stays true until further notice"

LTL: disjunctive normal form

- What would be the DNF in LTL? Let's just expand brackets:

$$\begin{aligned}\phi &= \mathbf{G}\left(\left(b' + \mathbf{F}w\right)\left(w' + w\mathbf{U}r\right)\right) = \left(b' + \mathbf{F}w\right)\left(w' + w\mathbf{U}r\right)\mathbf{N}\phi \\ &= \left(b' + w + \mathbf{N}(\mathbf{F}w)\right)\left(w' + r + w\mathbf{N}(w\mathbf{U}r)\right)\mathbf{N}\phi \\ &= \left(b' + w + \mathbf{N}(w + \mathbf{N}(\mathbf{F}w))\right)\left(w' + r + w\mathbf{N}(r + w\mathbf{N}(w\mathbf{U}r))\right)\mathbf{N}\left(...\right) \\ &= \quad ... \quad \mathbf{N}\left(... \quad ...\mathbf{N}(... \quad ...\mathbf{N}\left(...\right))\right)...\end{aligned}$$

  We will *never finish* expanding those brackets!

- But many subformulas under $\mathbf{N}(...)$ are the same:

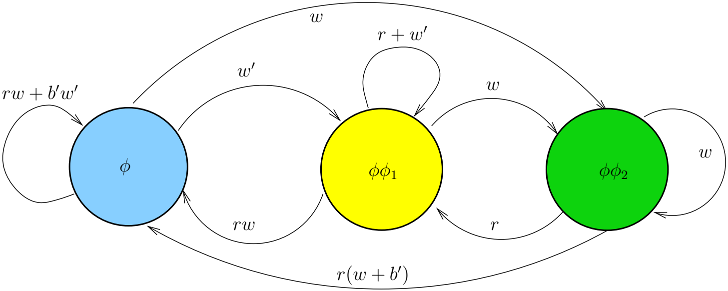
$$\begin{aligned}\phi_1 &= \mathbf{F}w; \quad \phi_2 = w\mathbf{U}r; \\ \phi &= \left(b' + w + \mathbf{N}\phi_1\right)\left(w' + r + w\mathbf{N}\phi_2\right)\mathbf{N}\phi \\ &= \left(rw + b'w'\right)\mathbf{N}\phi + w'\mathbf{N}(\phi\phi_1) + w\mathbf{N}(\phi\phi_2)\end{aligned}$$

LTL: disjunctive normal form

- Let's expand and simplify $\phi\phi_1$ and $\phi\phi_2$: we get simultaneous fixpoints

$$\begin{aligned}
\phi &= \left(rw + b'w'\right)\mathbf{N}(\phi) + w'\mathbf{N}(\phi\phi_1) + w\mathbf{N}(\phi\phi_2);\\
\phi\phi_1 &= rw\mathbf{N}(\phi) + \left(r + w'\right)\mathbf{N}(\phi\phi_1) + w\mathbf{N}(\phi\phi_2);\\
\phi\phi_2 &= r\left(w + b'\right)\mathbf{N}(\phi) + r\mathbf{N}(\phi\phi_1) + w\mathbf{N}(\phi\phi_2).
\end{aligned}$$

- The DNF for LTL is a *graph!*



The failure of LTL program synthesis

- Goal: given b and r, determine w
- The DNF generates a *nondeterministic* finite automaton (NFA) for w
- States of the automaton are ϕ, $\phi\phi_1$, $\phi\phi_2$, ... (**sets of fixpoints** of ϕ)
  - The DNF construction generates these states for us
- Determinizing the automaton may exponentially increase its size
  - Worst case: for ϕ with n fixpoints, DFA will have 2^{2^n} states
- Specifications will generally need to use many fixpoints. Example: *Whenever b is pressed, send query q and show w ("Waiting"). Upon reply r, show d ("Done"). Pressing c ("Cancel") stops waiting.*

$$\phi = \mathbf{G}\left[\left(bw' \to b\mathbf{U}d'w\right)\left(w \to d'w\mathbf{U}\left(c+r\right)\right)\right.$$
$$\left.\left(cw \to c\mathbf{U}w'\right)\left(q \leftrightarrow bw'\right)\left(rw \to r\mathbf{U}dw'\right)\right].$$

  - LTL is not particularly convenient for modular specification
  - Synthesis is *not practical* (I write and debug my automata by hand...)

Part 2: Temporal logic as type theory

- Logic gives a recipe for designing a *minimal* programming language (Curry-Howard isomorphism)
- Typically, we use an **intuitionistic** version of the logic:
  - No negation, no $\bot$; only $a+b$, ab, $a \to b$
  - No law of excluded middle
  - No truth tables, no "simplification"
  - Usually, cannot derive proofs automatically
- Axioms are *predefined terms* needed in the language
  - Example: $(a \to c) \to (b \to c) \to (a + b \to c)$ is the **case** operator
- Proof rules are *predefined constructions* needed in the language
  - Example: modus ponens (a; $a \to b$ so b) is function application
- Natural deduction rules are typing rules for the language

Interpreting values typed by LTL

- What does it mean to have a value x of type, say, $\mathbf{G}(\alpha \to \alpha\mathbf{U}\beta)$?
  - $x : \mathbf{N}\alpha$ means that $x : \alpha$ will be available *only* at the *next* time tick (x is a **deferred value** of type α)
  - $x : \mathbf{F}\alpha$ means that $x : \alpha$ will be available at *some* future tick(s) (x is an **event** of type α)
  - $x : \mathbf{G}\alpha$ means that a (different) value $x : \alpha$ is available at *every* tick (x is an **infinite stream** of type α)
  - $x : \alpha\mathbf{U}\beta$ means a **finite stream** of α that may end with a β
- Some *temporal axioms* of intuitionistic LTL:

$$\begin{aligned}
\text{(deferred apply)} \quad & \mathbf{N}\,(\alpha \to \beta) \to (\mathbf{N}\alpha \to \mathbf{N}\beta)\,; \\
\text{(streamed apply)} \quad & \mathbf{G}\,(\alpha \to \beta) \to (\mathbf{G}\alpha \to \mathbf{G}\beta)\,; \\
\text{(generate a stream)} \quad & \mathbf{G}\,(\alpha \to \mathbf{N}\alpha) \to (\alpha \to \mathbf{G}\alpha)\,; \\
\text{(read infinite stream)} \quad & \mathbf{G}\alpha \to \alpha\mathbf{N}(\mathbf{G}\alpha) \\
\text{(read finite stream)} \quad & \alpha\mathbf{U}\beta \to \beta + \alpha\mathbf{N}(\alpha\mathbf{U}\beta)
\end{aligned}$$

A small FRP language: Elm

- Core `Elm`: polymorphic λ-calculus with `lift2`, `foldp`, `async`

$$\begin{aligned}\texttt{lift2} &: (\alpha \to \beta \to \gamma) \to \mathbf{G}\alpha \to \mathbf{G}\beta \to \mathbf{G}\gamma \\ \texttt{foldp} &: (\alpha \to \beta \to \beta) \to \beta \to \mathbf{G}\alpha \to \mathbf{G}\beta \\ \texttt{async} &: \mathbf{G}\alpha \to \mathbf{G}\alpha\end{aligned}$$

- (`lift2` makes $\mathbf{G}$ an applicative functor)
- `async` is a special *scheduling instruction*
- Limitations:
  - Cannot have a type $\mathbf{G}(\mathbf{G}\alpha)$, also no concept of $\mathbf{N}$ or $\mathbf{F}$
  - Cannot construct temporal values by hand
  - This language is an *incomplete* Curry-Howard image of LTL!
  - *I work after the boss comes by and until the phone rings*:

    let after_until w (b,r) = (w or b) and not r in
       foldp after_until false (boss, phone)

"Legacy" FRP systems: `FrTime`, `Fran`, AFRP, etc.

- Two functors: "continuous behavior" $\mathbf{G}\alpha$ and "discrete event" $\mathbf{F}\alpha$
- Time is conceptually continuous
- Explicit $\mathbf{N}$, delay by time Δt, explicit values of time
- Many predefined combinators that do not follow from type theory:
  `value-now`, `delay-by`, `integral`, ... (`FrTime`)
  `merge`, `switcher`, $\mathbf{G}\,(\mathbf{G}\alpha)$, ... (`Fran`)
- AFRP: temporal values are not available, only combinators!

A lower-level FRP language: `AdjS`

- A lower-level type system: $\mathbf{N}\alpha$ (`next`), $\hat{\boldsymbol{\mu}}\alpha.\Sigma$ ($\boldsymbol{\mu}$+next), $\boxdot\alpha$ (`stable`)
- Explicit one-step temporal fixpoints, for example $\mathbf{F}\tau = \hat{\boldsymbol{\mu}}\alpha.\tau + \alpha$

$$\tau = \hat{\boldsymbol{\mu}}\alpha.\Sigma \cong \hat{\boldsymbol{\mu}}\alpha.\left[\frac{\mathbf{N}\tau}{\alpha}\right]\Sigma$$

- Term assignments, simplified (see Krishnaswamy's paper):

$$\frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \text{next } e : \mathbf{N}\alpha}\ \mathbf{N}I \qquad \frac{\Gamma \vdash f : \mathbf{N}\alpha \quad \Gamma, x : \alpha \vdash e : \beta}{\text{let next } x = f \text{ in } e : \beta}\ \mathbf{N}E$$

$$\frac{\Gamma \vdash e : [\mathbf{N}(\hat{\boldsymbol{\mu}}\alpha.\Sigma)/\alpha]\,\Sigma}{\Gamma \vdash \text{into } e : \hat{\boldsymbol{\mu}}\alpha.\Sigma}\ \hat{\boldsymbol{\mu}}I \qquad \frac{\Gamma \vdash e : \hat{\boldsymbol{\mu}}\alpha.\Sigma}{\Gamma \vdash \text{out } e : [\mathbf{N}(\hat{\boldsymbol{\mu}}\alpha.\Sigma)/\alpha]\,\Sigma}\ \hat{\boldsymbol{\mu}}E$$

$$\frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \text{stable } e : \boxdot\alpha}\ \boxdot I \qquad \frac{\Gamma \vdash f : \boxdot\alpha \quad \Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \text{let stable } x = f \text{ in } e : \beta}\ \boxdot E$$

Dreams of an ideal FRP language

- Requirements for a broadly usable FRP language:
  - "stable" and "temporal" types distinguished statically
  - seamless conversion from `int` to $\mathbf{G}$(`int`) and back, for "stable" types
  - construct values of type $\mathbf{F}\alpha$ by hand: from asynchronous scheduling
  - construct values of type $\mathbf{F}\alpha$ from external sources (environment)
  - tick-free operation: $\mathbf{N}$ is not needed, use $\mathbf{F}$ instead
  - the runloop (UI thread / background threads) needs to be represented
- I guess we are still in the research phase here...

Conclusions and outlook

- LTL is not a good fit as a specification language for reactive programs
- LTL synthesis from specification is theoretical, not practical
- FRP tries to make specification closer to implementation
- There are some languages that implement FRP in various *ad hoc* ways
- The ideal is not (yet) reached

Abstract

In my day job, most bugs come from imperatively implemented reactive programs. Temporal Logic and FRP are declarative approaches that promise to solve my problems. I will briefly review the motivations behind and the connections between temporal logic and FRP. I propose a rather "pedestrian" approach to propositional linear-time temporal logic (LTL), showing how to perform calculations in LTL and how to synthesize programs from LTL formulas. I intend to explain why LTL largely failed to solve the synthesis problem, and how FRP tries to cope.

FRP can be formulated as a λ-calculus with types given by the propositional intuitionistic LTL. I will discuss the limitations of this approach, and outline the features of FRP that are required by typical application programming scenarios.

My talk will be largely self-contained and should be understandable to anyone familiar with Curry-Howard and functional programming.

Suggested reading

E. Czaplicki, S. Chong. Asynchronous FRP for GUIs. (2013)
E. Czaplicki. Concurrent FRP for functional GUI (2012).
N. R. Krishnaswamy.
https://www.mpi-sws.org/∼neelk/simple-frp.pdfHigher-order functional reactive programming without spacetime leaks(2013).
M. F. Dam. Lectures on temporal logic. Slides: Syntax and semantics of LTL, A Hilbert-style proof system for LTL
E. Bainomugisha, et al. A survey of reactive programming (2013).
W. Jeltsch. Temporal logic with Until, Functional Reactive Programming with processes, and concrete process categories. (2013).
A. Jeffrey. LTL types FRP. (2012).
D. Marchignoli. Natural deduction systems for temporal logic. (2002). – See Chapter 2 for a natural deduction system for modal and temporal logics.