Verifying Functional Reactive Programs with Side Effects

Manuel Bärenz[1] and Sebastian Seufert[2]

[1] University of Vienna, Austria
maths@manuelbaerenz.de

[2] Otto-Friedrich-University of Bamberg, Germany
sebastian.seufert@stud.uni-bamberg.de

Temporal Logic and Verification of Functional Reactive Programming In general, desirable properties of reactive programs are expressed in a suited temporal logic. Recently, Alan Jeffrey has shown that Linear Temporal Logic (LTL) can be embedded into Martin-Löf type theory, allowing to write functional reactive programs and expressing their properties as LTL-formulas in a natural way [1].

What is lacking is an equally suitable embedded domain-specific language (EDSL) to express *proofs* of these properties, general enough to comply a broader range of cases while still practicable to the programmer. When employing such an EDSL with support by interactive assistants, ideally program, properties and proofs could then be provided together and thus increase the modularity of the verified code.

Jeffrey's approach covers pure signal processing without side effects. It is a practical programming pattern to allow side effects in FRP, though. Knowledge of the environment time, which distinguishes signal processing from stream processing, can be regarded as a side effect as well. Consequently, stream processing with side effects as described in [2] is a viable approach to hybrid FRP, combining discrete and continuous paradigms.

The work cited above describes a framework for effectful stream processing implemented in Haskell. Side effects are encoded as *monads* there, but other functors are also conceivable. The kind of side effect, represented by the choice of a particular functor, is then a parameter in the type signature of the reactive program. This allows for reasoning about its behaviour to some extent, similar as the EDSL provided by Alan Jeffrey.

However, in the context of verifying effectful FRP, there are propositions Φ whose values depend on the context of a side effect, such that LTL may not be sufficiently expressive:

Safety	"After any possible side effect that can occur, Φ becomes true."
Liveness	"There is a side effect such that if it occurs, Φ becomes true."

These two aspects match the 'all paths' (`A`) and 'exists a path' (`E`) modalities from Computational Tree Logic (CTL) excellently.

Container modalities With side effects encoded as completely positive functors, *container extensions* provide the possibility to express these modalities. A container is a dependent pair `S ▷ P` of a type `S`, the "shapes", and a type family `P`, the "positions". Every container gives a functor, its *extension*:

```
⟦ S ▷ P ⟧ X = Σ[ s ∈ S ] (P s → X)
```

A value of type `⟦ S ▷ P ⟧ X` is a side-effectful computation that produces a value of type `X`. The shape `s` plays the role of a command that is sent to the environment, while the type of positions `P s` encodes the possible response values that the environment can supply in order to yield a definite value.

Implementation Our library is formalised in Agda. There, containers are universe-polymorphic. A value in `Set` encodes a proposition, thus a value in `⟦ S ▷ P ⟧ Set` is an *effectful proposition*, i.e. a proposition with its validity depending on the environment. Consequently, container modalities can be defined which encode the statements that the proposition holds for any response (or position), or that there exists a response such that the proposition holds, respectively.

We implement effectful streams (FStreams) as a coinductive type:

```
FStream : Container → Set → Set
FStream (S ▷ P) X = ν Y . ⟦ S ▷ P ⟧ (X × Y)
```

Each time a value is retrieved from the stream, a side effect is executed.

Our library encompasses the usual utilities to work with streams, such as streams repeating (temporally) constant effectful values, application of functions to streams, a syntax for defining ultimately periodic streams from their prefixes, corecursion and bisimulation. All the elements of the EDSL can also be used to reason about the stream. Values of type `FStream (S ▷ P) Set` then correspond to CTL formulas.

We provide all CTL modalities, i.e. combinations of `A` and `E` with the temporal modalities `G` ("globally"), `F` ("future"), and others, such as "next" and "until". We supply a closely matched EDSL for constructing proofs for the CTL formulas, such that programs, properties and proofs become succinct and readable in our library as test cases demonstrate.

Example Consider the following program:

```
trafficLight : FStream (Reader Bool) Bool
trafficLight = ⟨ return true ▶ read ⟩ ▶···
```

This program encodes a traffic light which unconditionally outputs `true` (encoding "green") in the first tick, and asks for input from a `Reader` environment in the second tick and outputs it. After that, the stream repeats. (The input could be supplied by another stream, or in theory by a physical sensor.) In our DSL, ▶ is a stream constructor and ··· stands for repetition.

We will verify a *responsivity property*: At any moment, the traffic light could be green, given the appropriate input.

```
responsivity : EG (map (true ≡_) trafficLight)
responsivity = mapEG ⟨ refl ▶EG refl ⟩EG ▶EG···
```

Since we prove a "*globally*"-modality, the whole proof will be a stream of proofs for every tick, and can be expressed in a DSL approximating the stream DSL. First, we commute `map` past `EG`. We then prove the tautology `true ≡ true` for the first tick. In the second tick, we prove that the stream can indeed output `true`, and Agda automatically infers the appropriate input that the user needs to make to validate the proof. Finally, the proof repeats.

With little more effort, one can implement a second traffic light and verify, for example, a safety property (the two traffic lights are never green at the same time, under any side effect).

References

[1] Alan Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, pages 49–60. ACM, 2012.

[2] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell*, pages 33–44. ACM, 2016.