# Research Statement

Daniel Winograd-Cort

December 2014

My primary research interest is in designing programming languages that are expressive, concise, efficient, and robust. In my graduate career, I have focused on functional programming, and, more specifically, *functional reactive programming* (FRP) to achieve this goal. Although still a relatively new concept, FRP has already received a significant amount of attention in the Programming Languages community. FRP combines the benefits of functional programming, such as its declarative style and equational reasoning, with reactive programming, which provides a way to interface with a live and changing world. FRP has been used in a wide variety of domains such as robotics, animation, networking, games, and music and sound synthesis, and it is generally applicable to any real-time system.

My work with FRP has focused on incorporating effects, such as mutation and physical resource interaction, both in how to keep the system safe and secure but also to retain good performance. Without effects, FRP has limited power and a constrictive design, but so far, work has only been on either the programmer interface level [1] or the system's underlying connection to imperative-style effects libraries [2]. My work aims to bridge the gap between these by providing arbitrary effect usage directly to the front interface in a clear and safe way. I devised a system of *resource types* as an extension to a standard type system which segregates effects from each other, enforcing appropriate effect usage at the type level and thereby avoiding any possible inconsistencies. This segregation also serves as a secure barrier, allowing one portion of a program to "own" a resource and preventing any other portions of the program from accessing it. Additionally, these resource types provide a static view of which resources a given program uses, giving a programmer an overview of the extent of possible resource usage directly from the type of the expression. Using the resource type system, I showed how to achieve a form of non-local communication within FRP programs that can give rise to locally-deterministic concurrency. Finally, I demonstrated how adding a non-interference law to FRP provides new expressive power without inhibiting typical optimization techniques.

Functional reactive programming is a model of programming currently in active use, and enhancing its expressiveness, safety, security, and performance provides a real benefit to actual programmers. Furthermore, beyond the practical results, this research serves as a platform for pushing the boundaries of type level computation and allows us to continue to explore abstractions from category theory and their impact on programming design.

## Past and Present Research: Enhancements to FRP

Functional reactive programming facilitates programming with streams of time-varying data. Thus, one can think of FRP as an inversion of flow control from the structure of the program to the structure of the data itself. In a typical (say, imperative) program, the structure of the program governs how the program will behave over time; as time moves forward, the program sequentially executes its statements, and at any line of code, one can make a clear distinction between code that has already been run (the past) and code that has yet to be run (the future). However, in FRP, the program acts as a signal transformer, and, as such, we are allowed to assume that the program executes *continuously* on its time-varying inputs—essentially, it behaves as if it is running infinitely fast and infinitely often. This

is what I consider to be the core principle of the design and what I have dubbed the fundamental abstraction of FRP. The whole program must be able to be perceived to run simultaneously, and it is in the data itself that one can examine the past, present, and future.

## Design and Performance

My work is specifically rooted in *Arrowized* FRP, which uses the category theoretic abstraction of arrows for modeling computation. In this realm, the arrows themselves, which we refer to as *signal functions* in the FRP domain, become the first-class values rather than the signals they act on. These signal functions remain static as they process the dynamic signals they act upon, but in practice, it is often valuable to be able to dynamically alter the way that a signal function behaves over time. For this reason, many FRP formulations extend the arrows with enough extra features to make them synonymous with monads (or they simply use monads as the base abstraction). However, this is an overkill: although it provides the necessary power, it also erodes the abstraction that time is based entirely in the data.

I developed a modest extension to arrows called *non-interfering choice* [7], which allows what I call "predictably dynamic" behavior. This extension, along with a notion of settability that I designed and codified, recovers the necessary dynamic power for most use cases without permitting the full realm of dynamic operations (i.e. that which are achievable with monads). I further demonstrated that optimizations designed specifically for arrowized FRP and which do not apply to monadic FRP, such as those for Causal Commutative Arrows [3], are perfectly applicable to my system. Thus, my system can be powerfully optimized.

## Adding Effects

In its purest form, functional reactive programming permits no side effects (e.g. mutation, state, interaction with the physical world), and as such, all effects must be performed outside of the FRP scope. Thus, in practice, pure FRP programs are forced to route input data streams to where they are internally used and likewise route output data streams back out to the edge of the FRP context. I call this the FRP *I/O bottleneck*. This design inhibits modularity and also creates a security vulnerability whereby parent signal functions have complete access to their children's inputs and outputs. Allowing signal functions themselves to perform effects would alleviate this problem, but it can interfere with the fundamental abstraction.

I developed the notion of *resource types* [8] to address this issue and allow the fundamental abstraction to hold in the presence of effects. Resource types are phantom type parameters that are added to the type signatures of signal functions that indicate what effects those signal functions are performing and leverage the type-checker to prevent resource usage that would break the abstraction. I designed type judgments and operational semantics for a resource-typed model and then implemented the system in Haskell.

## Supporting Asynchrony

FRP typically relies on a notion of *synchronicity*, or the idea that all streams of data are synchronized across time. In fact, this synchronicity is a key component of maintaining the fundamental abstraction: it ensures that two disparate portions of the program will receive the same deterministically associated (synchronous) input values and that their separate results will coordinate in the same output values. However, in many applications, this synchronicity is too strong.

I devised a notion of treating time not as a global constant that governs the entire program uniformly, but rather as *relative to a given process* [5]. In one process, time will appear to progress at one rate, but in another, time can proceed differently. Although we forfeit the global impact of the fundamental

abstraction, this allows us to retain its effects on a per-process scale. That is, we can assume each process processes its inputs continuously despite the whole network having different notions of time.

Along with this notion of relative time, I introduced a feature called a *wormhole* [6], which allows non-local communication. Wormholes leverage resource types to keep their access safe and secure, but also have an internal system of *time dilation* to allow them to act as communication interfaces between asynchronous processes. Additionally, they can be used to subsume other common FRP operations such as looping and causality.

In total, I created a new form of asynchronous FRP that retains local determinism in the presence of global non-determinism. I defined the type system and a set of operational semantics for the design and built a prototype in Haskell.

## Future Research

The motivation for my research stems from the simple idea that communication is confounded by language. Without a language, there is no means to communicate at all, but with one, the communicator is burdened by the syntax, vocabulary, and idioms inherent in it. Thus, I feel a need to enhance programming languages, to make them clearer and more expressive and to build into them idioms and abstractions that promote good style and safe, reliable, and efficient programming. Here, I will mention a few examples of the kinds of problems that I intend to pursue in the future:

- In my recent work, I have used arrows as the basis for FRP design, but others have designed FRP systems based on applicative functors, monads, and even combinations of different abstractions. That said, I do not believe that an optimal approach has been found. Arrows themselves do not naturally convey the causality, commutativity, or dynamism that FRP typically requires, but monads are too strong and applicative functors severely limit dynamic decisions. My work with non-interfering choice is a step in the right direction, but I would like to explore this realm further by experimenting with alternative abstractions and (more concretely) different syntaxes for expressing FRP.

- There are portions of my work with resource types that I have been unable to build in a current programming language. Specifically, the most abstract form of wormholes require something that I call a *unique-existential type parameter*, or an existential type that can unify with itself but not with any other type. In addition to their use in implementing general resource types, these unique-existentials may be useful in other type-level computations that require phantom type parameters that fill a concrete purpose but must remain abstract, for example in implementing type-polymorphic heterogeneous sets. I would like to more rigorously explore this idea and see how it fits into the broader class of advanced type-level operations.

- With cloud-based systems gaining in popularity and the continued growth of data sets, it is imperative that we have languages that can scale efficiently and automate multi-process communication protocols. My current work with asynchronous FRP is one example of this, in that the notion of relative time with wormholes allows for easy multi-process communication, and non-interfering choice allows a straightforward performance boost with little loss of expressive power. However, my design needs to be more carefully engineered and aggressively optimized before it can gain broad appeal. Thus, in addition to the theoretical foundations, I intend to continue working on the practical aspects of the design.

- My work with FRP has been generally limited to graphical user interfaces and music and sound synthesis, but FRP has many more applications. For example, some work has been done to incorporate FRP into software-defined networking (SDN) [4], but due to FRP's absence of internally

usable effects and its strongly enforced synchronization, the FRP aspects are typically dropped when concurrency is added (such as in the update from Nettle to Multi-Core Nettle). I believe my work with resource types and wormholes can address this, and I plan on exploring SDN from my asynchronous FRP perspective.

## Conclusions

Researching demands a blend of both theory and practice, of design and engineering, but *being* a researcher also requires the more subtle skills of picking the right problem to solve and identifying the value of a solution. Research projects are not simply whimsical explorations into topics that the researcher finds interesting; rather, they should be tractable, goal-oriented problems which provide a clear utility to the field.

When first identifying a problem domain, it is important to consider the relevance of the topic. One should ask of any potential research: Will solving this problem be useful to the community? Furthermore, this question must remain a driving force behind the work. In many cases, the answer will change over the course of the project, and problems can introduce tangents whose solutions yield new and independently useful results. However, research must ultimately be justifiable to others in the field, and we should never be working on a problem simply "because we can."

Although I trust my own intuition in this matter, I refrain from relying on it wholeheartedly. Thus, I additionally make use of my network of fellow researchers to gauge the potential of my projected research ideas. Are others interested in my work? Can I help them apply my results to their own projects? Exploring these questions can lead to new research opportunities and collaborative projects. Computer science research as a whole has trends, and although I do not intend to switch fields as one topic grows in popularity over another, it is important to stay aware of the priorities of the community.

## References


[1] A. Courtney and C. Elliott. Genuinely Functional User Interfaces. In *2001 Haskell Workshop*, September 2001.

[2] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 259–276. Springer Berlin Heidelberg, 2006.

[3] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *Journal of Functional Programming*, 21(4–5):467–496, September 2011.

[4] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *PADL*, pages 235–249, 2011.

[5] **D. Winograd-Cort** and P. Hudak. Communicating Functional Reactive Processes. Unpublished.

[6] **D. Winograd-Cort** and P. Hudak. Wormholes: Introducing Effects to FRP. In *Haskell Symposium*, pages 91–103. ACM, September 2012.

[7] **D. Winograd-Cort** and P. Hudak. Settable and Non-Interfering Signal Functions for FRP. In *ICFP*, pages 213–225. ACM, September 2014.

[8] **D. Winograd-Cort**, H. Liu, and P. Hudak. Virtualizing Real-World Objects in FRP. In *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, January 2012.