Directions in Functional Programming for Real(-Time) Applications*

Walid Taha**, Paul Hudak, Zhanyong Wan

Department of Computer Science,
Yale University, New Haven, CT, USA.
{taha,hudak,zwan}@cs.yale.edu

Abstract. We review the basics of functional programming, and give a brief introduction to emerging techniques and approaches relevant to building real-time software. In doing so we attempt to explain the relevance of functional programming concepts to the real-time applications domain. In particular, we address the use of *types* to classify properties of real-time computations.

"If thought corrupts language, language can also corrupt thought."
George Orwell, *Politics and the English Language*

1 Introduction

An important challenge facing functional programming is the successful application of its principles to the domain of real-time software. Examples of real-time software include controllers in audio systems, video cameras, video games, imaging and data acquisition systems, and telecommunications hardware. The significance of this challenge is two fold: First, real-time applications are constantly growing in complexity, and society is growing more dependent on their correctness. Second, real-time applications possess characteristics that have been traditionally considered outside the scope of functional programming languages. For example, real-time systems are often required to be:

- *Responsive, reactive, or live*: a response must be made to every input.
- *Resource-bounded*: responses must happen in a limited amount of time, using limited hardware.
- *Concurrent*: components may run in parallel and manipulate shared data.
- *Networked or distributed*: communication between components of the system may involve time delays or loss of information.

The purpose of this paper is to illustrate how many fundamental aspects of functional programming are highly relevant not only to programming but also

* Funded by DARPA F33615-99-C-3013 and NSF CCR-9900957.
** Funded by NSF ITR-0113569 and subcontract #8911-48186 from Johns Hopkins University under NSF agreement Grant # EIA-9996430.

to *understanding* real-time systems. Following on a long tradition of mathematical modeling using functions [3, 22, 38, 89, 96], we summarize the key benefits of functional programming. In doing so, we put into context many on-going efforts both in and outside the area of functional programming to address issues that are specific to the real-time domain.

1.1 Audience and Organization of this Paper

The classic paper by Hughes presents excellent motivation for general purpose functional programming [48]. The present paper is aimed at a reader interested in using functional programming for real-time applications. We assume the reader has a background in discrete mathematics, a few years experience with programming in a mainstream language, and an informal familiarity with basic programming language concepts. We focus on:

- Programming by writing and composing *functions* (Section 2),
- Understanding of the notion of *types* and *type systems* (Section 3),
- Using *higher-order* functions (Section 4), and
- Appreciating the presence of a wealth of formal *semantics treatments* for various aspects of functional languages (Section 5).

2 The Big Deal about Functions

What sets functions apart from arbitrary relations is two simple properties: for two given sets A and B, a relation $f \subseteq A \times B$ is a *function* (written $f : A \to B$) if and only if it satisfies:

- *Totality*: for every $a \in A$, there exists an element $b \in B$ such that $(a, b) \in f$, and
- *Uniqueness*: for any $a \in A$ and $b, b' \in B$, if $(a, b) \in f$ and $(a, b') \in f$, then $b = b'$. We write $f(a)$ to denote this unique element b.

A relation that satisfies only uniqueness is called a *partial function*.

A framework that allows us to model programs as functions (from input to output) has a natural appeal in the context of real-time systems, where each of these properties is desirable: Totality means that the program will always respond to its input, and uniqueness means that the program will return the same result whenever it is subjected to the same input. While it may be hard to see how this simple intuition scales to more complex settings, a key quality of functions is that they compose naturally and without surprising interactions.

In this section we begin with a brief review of the state of the art both in functional programming languages and programming languages for real-time applications. We then proceed to illustrate how a "language of functions" can be an expressive tool for the specifying real-time computational models.

2.1 State of the Art

A wide variety of functional programming languages are available today, including Scheme [86], SML [66], Haskell [78], and OCaml [60]. But none of these provides full support for programming with functions in the sense described above, for two reasons. First, all of them permit the definition of partial functions, thus leading to potentially non-terminating computations. Second, except for Haskell, none of these languages are "purely" functional: that is, they allow the expression of imperative programs that violate the uniqueness property. For example, while Scheme is a highly expressive and versatile programming language, historically its main goal was list and symbolic processing, and not so much programming with functions. Scheme is not statically-typed and can express "impure functions" that depend implicitly on interactions with both machine state and the real world. ML and OCaml are typed functional languages, but still permit partial and imperative computations.

Of these popular languages, Haskell comes closest to being "purely" functional, in that both local state and interactions with the real world are modeled without side-effects. Such interactions are also typed explicitly (see the discussion of *monads* below), ensuring that the type still reflects these interactive properties. At the same time, Haskell provides special syntax for "imperative" features. Although this paper presents most concepts in generic mathematical notation, it is fairly straightforward to encode our equations in Haskell.

There are frameworks that fully support programming with functions. Important examples of such systems are Elementary Strong Functional Programming (ESFP) [94], Nuprl [2], Martin-Löf's Type Theory [74], the Calculus of Constructions [21, 28, 73], Charity [19, 20], LEGO [61], and Twelf [80]. These languages are still not in the mainstream, but we predict that they will grow in popularity in the coming years.

Functional Programming for Real-time Applications While many languages developed for the real-time domain are "imperative" (such as ESTEREL [9] and STATECHARTS [37]), a number of these languages are expressly functional, including LUSTRE [15], synchronous Kahn networks [16], SAFL [70] and FRP [30, 99]. But essentially all the "imperative"' ones are deterministic (thus satisfying uniqueness) and terminating (thus satisfying totality), and so it is reasonable to classify them as functional.[1] This statement is less surprising when we note that Haskell also supports and encourages the imperative style, as long as side-effects are properly encapsulated.

To guarantee termination, most real-time languages disallow general recursion. Synchronous Kahn networks were developed as an extension to LUSTRE that provides recursion and higher-order programming, but termination is sacrificed for this expressivity. Implementations of LUSTRE also have a macro-like

[1] ESTEREL first translates programs into circuits (or state machines), then these circuits are analysed to ensure determinism.

facility that supports recursion, but runs the risk of causing the compiler to diverge. FRP is embedded in Haskell [46, 31, 47] and so inherits recursion and non-termination. RT-FRP [100] is a subset of FRP that guarantees resource-bounded program execution: every interactive RT-FRP computation terminates. Using a special type system, RT-FRP still allows two forms of recursion similar in spirit to the idea of tail-recursion [86]. Because RT-FRP is a closed language [56], it is possible to *guarantee* that no partiality is accidentally introduced.

2.2 Functions as a Tool for Analysis and Modeling of Computation

If we temporarily abstract away from issues of performance, practically all interesting features of program behavior can be specified precisely using functions. This means that we can easily *simulate* or *prototype* most interesting computational phenomena in a functional language. We believe that this has tremendous conceptual benefits, and that, in the long term, will have the same impact on software engineering processes. In what follows, we give a brief description of such computational phenomena, and review how they can be described in a purely functional manner.

Input/Output, Uniqueness, and Interaction A function cannot implicitly interact with an "outside world" during the course of its computation. If it did, then it is very possible that each time it is applied to a value it would produce a different output. In other words, this computation would not have the uniqueness property.

A simple approach to modeling such computations is to view them as a chain of functions that determine successive interactions with the outside world. Given two types input and output, a recursive type equation can be written to describe such a computation as follows:

$$\mathsf{computation} \rightleftharpoons \mathsf{input} \rightarrow (\mathsf{output} \times \mathsf{computation}).$$

This equation says that a computation is a function which takes an input and returns a pair. The pair consists of both the output of the first computation and the computation that should be carried out on the next input. As such the type $\mathsf{computation}$ models a strictly infinite sequence of interactions with the outside world (as long as the outside world is providing an input). This model of computations is well-suited for reactive and interactive systems.

A number of reactive languages, such as Lava [18], Hawk [62], and FRP [46], have used streams to implement the idea presented above. *Streams* are infinite datastructures which can be easily defined in a lazy language. However, most of these systems have been developed as languages embedded into Haskell. RT-FRP implements the same model, but as a closed language, which makes it more suitable for direct use in a real-time setting.

Runtime Errors and Partiality A common challenge that one encounters when trying to model various features of computation as functions is dealing with partiality. For example, consider the following definition:

$$f(x) \triangleq 1/x.$$

When $x = 0$, $f(x)$ is not defined. Definitions similar to the one above are allowed in most programming languages, even statically-typed ones that "guarantee that runtime errors do not occur." This is not a flaw with these languages, because what they do guarantee is the preclusion of a *specific* class of runtime errors, not all [13, 14]. Whenever we start with primitives such as division that can themselves generate runtime errors, those errors are generally ignored by type systems. However, in the design of real-time systems where raising errors is simply not acceptable, such primitives become a concern.

It is possible to prevent errors by replacing the faulty set of primitives by a modified set. For example, we can define a new division function:

$$x /\!\!/ y \triangleq \text{ if } y = 0 \text{ then } 0 \text{ else } x/y.$$

A more elegant way to dealing with partiality is to introduce a distinguished value $\perp$ (read "bottom") to be returned in the case of an error. To introduce such a value in a type-sound manner, we use a *lifting* type constructor $\_\_\perp$ defined as follows:

$$\alpha_\perp \ ::= \ \perp \mid \alpha_\perp.$$

Intuitively, this specification can be read as a parameterized BNF definition. The variable α on the left-hand side is a type variable that can be instantiated to a specific type, and on the right-hand side denotes a term of that type. The first variant in this definition tells us that $\perp$ has type $\alpha_\perp$ for any type α. The second variant tells us that a term $e_\perp$ has type $\alpha_\perp$ whenever e is a term of type α. The type constructor $\alpha_\perp$ is a standard concept that allows us to say that we may or may not get a value as a result, and hence is useful for modeling partiality.[2]

Now we can redefine our division function as:

$$x /\!\!/ y \triangleq \text{ if } y = 0 \text{ then } \perp \text{ else } (x/y)_\perp$$

All we need to do is to check that the result of this operation is not $\perp$ before we continue on to other computations.

Infinite Loops In any Turing-complete language, it is well known that one can write programs that run forever without returning a result. This means that we cannot directly treat arbitrary programs as functions. But we can still provide functional models of such programs. In fact, we can do this using a combination

[2] In implementations of typed functional programming languages $\alpha_\perp$ is known as either a "maybe" or an "option" type.

of the two techniques just introduced. We use a *sum* type constructor $\_ + \_$ that takes two type parameters, and is defined as follows:

$$\alpha + \beta \ ::= \ \mathsf{Alpha}\ \alpha \mid \mathsf{Beta}\ \beta.$$

This new constructor allows us to package up values of two different types α and β in one common type $\alpha + \beta$, and still retain the ability to recover the original values.[3] For any term e of some type α, the term $\mathsf{Alpha}\ e$ has type $\alpha + \beta$, for any type β. The Beta variant lets us do things the other way around. Now, we can define a new kind of computation as:

$$\begin{aligned} \mathsf{computation} &\rightleftharpoons \mathsf{input} \to \mathsf{initialized\text{-}computation} \\ \mathsf{initialized\text{-}computation} &\rightleftharpoons \mathsf{output} + \mathsf{initialized\text{-}computation} \end{aligned}$$

Such a computation takes an input and returns an initialized computation. An initialized computation is either an output, in which case we are done, or another initialized computation. In the second case, we have "try again." Intuitively, this model is similar to observing a Turing machine as it performs a step of a program. If a program is finished, it generates an output. If not, we get back a machine that has been advanced by one step, and then we can try again.

Note that this model of computation is, at least intuitively, a special case of the model for interaction presented above. This idea has been used to provide a form of recursion suitable for real-time applications in RT-FRP.

Concurrency, Randomness, Non-determinism, and Sets Possibly the biggest conceptual challenge to programming with functions is *non-determinism*. Applications that try to take advantage of either concurrency or randomness often end up having to deal with non-determinism in one form or another. A basic approach to modeling non-determinism in the functional setting is to use of *sets* [82, 88]. But a naive account of sets can itself get in the way of programming functionally. For example, the order of elements in a set is generally considered irrelevant. If our language provides any mechanism for turning an arbitrary set into an ordered data structure, such as a list, then the language is forcing us to make one of two choices: either the order of elements has to be fixed using some mechanism (which would weaken our ability to model non-determinism accurately), or the mechanism for turning sets into lists would itself be non-deterministic (in which case we would lose the uniqueness property of the language).

A simple approach to soundly modeling a set is by its characteristic function:

$$\mathsf{set}(\alpha) \ \rightleftharpoons \ \alpha \to \mathsf{bool}$$

For example, a specific set of integers can be represented by a characteristic function of type $\mathsf{int} \to \mathsf{bool}$. With this representation, it is easy to define basic

[3] As such, sums provide a kind of overloading.

set-theoretic functions, such as union, intersection, complement, and membership. Slightly richer models allow us to define other operations on sets, such as size. A large body of techniques exist for the specification of functional data structures that have very reasonable performance characteristics [76].

To model concurrent/distributed computation, we must consider the effect of running two or more interactive computations simultaneously. The model would have to explicitly construct the set of all possible interactions between these computations. Although reasoning about such a model is not easier than about any non-deterministic computation, using function can help us identify the sources of complexity.

Stateful Computation Pure functions do not have "memory." So, no matter what a function has been applied to before, it will always return the same result for the same input. How can we write a program that reads a sequence of numbers, and continually prints the sum of the numbers up to this point? The essential idea is to explicitly pass around the relevant *state*. In fact, the interactive model that we presented earlier is also suitable for realizing this idea. That is, a computation of type:

$$\mathsf{computation} \rightleftharpoons \mathsf{int} \to (\mathsf{int} \times \mathsf{computation})$$

can be a function that takes an input and returns the current sum as output together with a computation that continues this process indefinitely. This idea is exploited in RT-FRP, for example, to realize operations like integration.

3 Types and Type Systems

In addition to their "traditional role" in programming, types have an important role in the classification of models and data structures. We begin this section by first presenting a possible application of types as a tool for classifying models of computation for real-time applications, and then move on to discuss their newly emerging role as a tool for making assertions about performance.

3.1 Types as a Tool for Classifying Models of Computation

Model theory was developed partly as a tool for the classification of various mathematical structures [42]. Types can do the same for models of computation.

In general, we can think of a reactive system as a map H that takes an input X from the environment and returns an output Y, that is,

$$Y = H(X).$$

The accuracy, tractability, and over all appropriateness of a particular model depends on the specifics of each of these variables. A particularly important aspect of these models is the treatment of time (or "the clock"), which can be treated

very concretely as a real number ($\mathbb{R}$), more abstractly as a natural number ($\mathbb{N}$), or even more abstractly as a partial order [57]. In what follows we briefly demonstrate how types can capture many of the key characteristics of some common models for reactive systems [59].

Continuous/Differential Model The finest scale which we can use to model the physical world (in the Newtonian sense) is achieved by using a time-line of reals, and measuring features of the world as real numbers. We can also assume that we can respond to these measurements in any manner, as long as it is causal. Mathematically, our model would be:

$$Y(t) \;=\; H(X|_{[0,t]}, t) \quad \text{and} \quad \begin{array}{l} X : \mathbb{R} \to \mathbb{R}^i, \\ Y \;: \mathbb{R} \to \mathbb{R}^o, \end{array}$$

where i and o are natural numbers corresponding to the number of inputs and outputs to the system, and R^j is the Cartesian product of R repeated j times. The restriction $|_{[0,t]}$ on the domain of X captures the standard constraint that H is causal. Finding solutions to such equations where H is unconstrained, however, would be extremely hard. More often we restrict our model to be one where H is a tractable transformation on reals. For example, we may restrict ourselves to ordinary differential equations. In this case, H can be a polynomial of integro-differential operators on the inputs X. While this model is still extremely accurate for many applications, it ignores the digital aspect of the machinery that is typically used to implement such computations.

Discrete/Difference Model A more tractable model is the discrete/difference-equation model. In a formal sense, this model is an approximation of the one presented above. It can be described as follows:

$$Y(n) \;=\; H(X|_{0..n}, n) \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to \mathbb{R}^i, \\ Y \;: \mathbb{N} \to \mathbb{R}^o. \end{array}$$

In this case, H is typically a polynomial of values of X at times $0..n$.

Synchronous/Reactive Model Several real-time programming languages are built upon the notion of *synchronous* computation. In the literature, the word "synchrony" has two meanings, corresponding to two important properties of synchronous languages [7]: first, it implies the ability to share a common time scale (that is, the logic time does not advance until all computations in the current "step" have been completed); second, the elements of the streams are to be consumed as soon as produced, making it possible to allocate only one cell for one stream. The first part of this definition is already present in the discrete model. The second is a performance issue, which is determined by the primitives that we are allowed to use in combining signals. A key change in the model, in our view, is that it becomes possible to use conditionals and partial

functions on signals. From this point of view, it is only a minor variation on the model above:

$$\begin{array}{ll} Y(0) & = Y_0 \\ Y(n+1) & = H(X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to (\mathbb{R}_\perp)^i, \\ Y : \mathbb{N} \to (\mathbb{R}_\perp)^o. \end{array}$$

where Y_0 is some initial state. An important change here is that the memory of the system is restricted to a fixed size (width of Y). With the constant developments of exact-arithmetic methods [29], this model can be realized quite effectively.

Finite State Machine Model Reals can in general require an arbitrary amount of space to represent. This makes analyzing space requirements non-trivial, and makes another highly tractable model very attractive: finite state machines. This model can be described as follows:

$$\begin{array}{ll} Y(0) & = Y_0 \\ Y(n+1) & = H(X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to \mathsf{one\text{-}bit\text{-}input}^i, \\ Y : \mathbb{N} \to \mathsf{one\text{-}bit\text{-}state}^o, \end{array}$$

The main difference between this model and the previous one is that the set of inputs and states is restricted to be finite. This yields a model of computation that is especially tractable and easy to analyze, at least for reasonably small state sizes.

Abstract State Machine Model This model is a generalization of finite state machines, where a state consists of a term rather than just a fixed number of bits. At any n, the size of each term is finite, but there is no static limit on this size.

$$\begin{array}{ll} Y(0) & = Y_0 \\ Y(n+1) & = H(X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} X : \mathbb{N} \to \mathsf{term\text{-}input}^i, \\ Y : \mathbb{N} \to \mathsf{term\text{-}state}^o. \end{array}$$

Discrete Event (or Event-Driven) Model An increasingly popular model is the discrete event model. It can be viewed as a special case of the finite state machine model, where updating a special set of inputs (the "event") is what drives the global clock. This model can be described as follows:

$$\begin{array}{ll} Y(0) & = Y_0 \\ Y(n+1) & = H(I|_{n+1}, X|_{n+1}, Y|_n) \end{array} \quad \text{and} \quad \begin{array}{l} I : \mathbb{N} \to \mathsf{event}, \\ X : \mathbb{N} \to \mathsf{input}^i, \\ Y : \mathbb{N} \to \mathsf{output}^o, \end{array}$$

where $I(n)$ is the nth event that occurs during the system's life. So-called cycle-driven models are special cases of discrete event models, where some events are planned to occur at fixed intervals.

3.2 Types for Performance

Most functional programming languages are based on the lambda-calculus [17, 4]. The untyped lambda-calculus is deterministic but not terminating. In fact, in the absence of recursion and primitives that may introduce partiality, all programs in the simply-typed lambda-calculus are terminating (see Hindley [39] for an excellent introduction). The same is also true for more sophisticated typed lambda-calculi, such as the Calculus of Constructions [5]. As such, typing is essential for using the lambda-calculus as a language for programming with functions. But while these type systems guarantee termination, it might require super-exponential time [101]. This means that a traditional typed lambda-calculus is not suitable for the real-time domain. It is only with the advent of more modern concepts that type systems are proving to be useful for characterizing resource consumption at a finer level. We give a brief introduction to some of these developments in this section.

Abstract Functors and Di-Functors A *functor* is essentially a parameteric data type, such as a list. For any type α, the list functor allows us to form the type $\mathsf{List}(\alpha)$, which is a list of values of type α. Adding functors to a language can extend its expressivity in many ways.

As with data types, functors can be concrete or abstract. List is usually a concrete data type. Modern programming languages allow users to introduce new functors using either data type declarations or class instances. Abstract functors, on the other hand, allow us to go beyond what can be defined in our language, and can provide powerful encapsulation mechanisms. For example, consider a situation where we want to handle values of type $\mathsf{CouldDiverge}(\alpha)$, which are computations that could either yield α when evaluated or diverge. Can we introduce such a data type into a functional setting without losing uniqueness or totality? This and many other interesting problems can be addressed by a variety of functors:

Monads: The very question raised above, for example, turns out to be fundamental. It can be expanded to include many *computational effects* other than non-termination, such as state, concurrency, and exceptions. A particular flavor of abstract functors called monads [68, 97, 79] was shown to be suitable for precisely that. Using a monad, a wide variety of effectful computations can be passed around, combined, and extended within a purely functional setting.

Linear types: Another kind of abstract functor is one based on linear logic [6]. Linear logic provides a means to expressing a notion of a "resource" in the lambda-calculus. Recent work in this area has shown how linear types can be used to build expressive programming languages with dynamic data structures that can be compiled into malloc-free C [45]. Almost exactly the same system has also been used to build a language where all programs run in constant space and can take at most polynomial time to execute [1, 43]. Programs in both languages are terminating.

Reactivity: FRP is built around two functors called Behavior and Event, corresponding to continuous-time behaviors and discrete event occurrences. Semantically, the same type constructors are present in RT-FRP, but they are made implicit in the types. This is achieved by having a specialized type system that only addresses the reactive part of FRP.

Staging: Multi-stage languages [12, 90, 92] are based on the key ideas of multi-level [25, 26, 33] and two-level languages [52, 71], and provide mechanisms for building programs that execute in multiple distinct stages. This is achieved by providing constructs for building, combining, and executing code at runtime. The presence of these constructs provides both hygienic program generation and reflection mechanisms, all in one, statically-typed framework. Multi-stage languages also provide a basis for heterogeneous languages, which combine more than one "traditional" language in the same framework. All of these languages provide a functor for "code". It is an abstract notion of code because it is usually not possible to inspect its text. So, a more accurate way of describing this kind of code is as a "future stage" computation. A variant of the code functor has been used in RT-FRP to model "exportability annotations:" only those variables whose type is marked as "exportable" can participate in computations in the base language. This allows a certain kind of cyclic definitions to be detected statically, and to ensure totality.

We are not aware of many results on multi-stage programming in a resource-bounded setting. McKay and Singh use partial evaluation for the dynamic specialization of FPGAs [63]. We see this as an important direction of future research.

Arrows: More recently, it has been noted that it is useful to distinguish between two kinds of types that a functor can be parameterized by: input types and output types.[4] This gives rise to a particular brand of di-functors called *arrows*. Arrows can be viewed as a generalization of monads, and have been shown to model a variety of interesting kinds of computations. Efforts are under way to explore the utility of the notion of arrows in FRP.

Type-and-Effect Systems and Indexed-Types Another approach to increasing the power of type systems is to enrich types with annotations that capture additional information about a value. An early example of this approach is the type-and-effect system used by Talpin and Jouvelot [93] to introduce side-effects in a functional manner. In this system, the type of every "impure function" is enriched with information about which variables are read or written when it is executed. This approach is as an alternative to monads. In some instances the two are indeed equivalent [54, 98]. Effects were later used to develop region-based memory management [11, 95]. In this approach, each type carries the name of a region where it is allocated, in addition to the standard effect information. This

[4] For the reader familiar with subtyping, the distinction alluded to here is related to co- and contra-variance of type constructors.

enables a safe and high-level form of explicit allocation/deallocation of memory. Explicit memory management can make the execution of programs more predictable than if it is left to a garbage collector. Effects have also been used to build a type system for a multi-threaded language where freedom of deadlocks can be statically tested [32].

Sized types are types enriched to capture the amount of space needed to store a value. For example, a list of length n carrying values of type α would have the type $\mathsf{List}_n(\alpha)$. This idea has been used to build languages for reactive systems, where properties such as termination and resource-boundedness can be verified statically [50]. An especially interesting aspect of this work is that one of the rules explicitly encodes an induction principle, which allows the programmer to use recursion, as long as the type system can check that it is well-founded. This idea is explored in the context of an execution model where reactive systems are viewed as stream-processors where the rates of the various streams can be different, and the key constraint is to ensure the "liveness" of the output.

It has also been shown that sized types and region effects can be combined naturally in a first-order system [49].

Using ideas from dependent typing, Crary and Weirich [24] develop a type system that provides an explicit upper bound on the number of steps needed to complete the computation. Space is conservatively bounded by the same bound as time. The language does not have recursion, rather, provides special iterators that are always guaranteed to terminate. The language supports higher-order functions.

LUSTRE and Synchronous Kahn networks use the notion of a *clock calculus*, which is essentially a type system characterizing the clocks underlying each expression. Programs that are well-typed in this system, and that also pass a cyclic dependency test, are guaranteed to be well-behaved.

4 Higher-Order Functions

Intuitively, higher-order functions are programming patterns. Formally, higher-order functions are ones that can take functions as arguments, or return functions as results. We have already seen the usefulness of returning functions: it was used in various models of computations that we discussed earlier. In particular, it provides the ability to build new functions that are "evolved" versions of older ones. Passing functions as arguments is equally useful, as it provides a mechanism for parameterizing programs by functions.

Consider, for example, the pattern of performing point-wise operations on elements of a container type, such as $\mathsf{List}(\alpha)$. One way to do this is to write a recursive (or iterative) program each time we want to carry out this pattern. But with higher-order functions, we can capture this pattern once and for all with one function (call it map) that has the following type:

$$\mathsf{map} : (\alpha \to \beta) \times (\mathsf{List}(\alpha)) \to (\mathsf{List}(\beta)).$$

The presence of higher-order functions can drastically enhance the expressivity of a programming language [75].

Higher-order programming is generally not supported in programming languages intended for the real-time domain. There are, however, a few notable exceptions, such as FRP and synchronous Kahn networks. The main reason seems to be that it is harder to guarantee resource boundedness in the presence of higher-order functions. In RT-FRP, for example, higher-order functions are not supported directly.

5 Mathematical Semantics

The semantics of a programming language is a mathematical specification of what the programming language is supposed to do. While a semantics can be abstract, it is not necessarily so. There is a colorful spectrum of ways to define the semantics of a programming language. A number of balanced textbook treatments of the various approaches already exist [36, 67, 72], as do more advanced studies [23]. There are two flavors of semantics that are often viewed as being in strong contrast: *denotational* semantics and *operational* semantics.

Denotational semantics is generally presented by a translation from syntax to a more abstract mathematical domain, often called "the meaning" or "denotation" of the syntax. It is concerned with traditional program equivalence, which is a subtle and technical subject. A denotational semantics is frequently chosen as the reference when it is the simplest semantics. This simplicity is often what makes it abstract. While abstractness is perfect if we are only interested in reasoning about program equivalence, it can have two disadvantages: first, it can be alienating for the practitioner who does not have the mathematical background to understand it. Second, it can be too abstract. For example, a denotational semantics does not traditionally describe the cost of a computation, which is a crucial concern in real-time applications.

Operational semantics is generally presented by a set of rules for "building" a computation, and thus can provide a basis for discussing performance. Originally promoted by Plotkin [83, 85, 10], in recent years this approach has gained substantial popularity [35, 69, 81, 87]. When combined with typing, operational techniques can also provide powerful proof techniques [34].

Essentially all of the languages mentioned above have been given some kind of formal semantics. A large effort has been made to develop the semantics of ESTEREL, including a denotational semantics in the point view of Scott's ternary logic, an operational semantics based on an interpretation scheme expressed by term rewriting rules defining microstep sequences, and a circuit semantics based on a translation of programs into circuits. These semantics are shown to correspond in a certain way, constrained only by a notion of stability [8].

A distinguishing characteristic of FRP is that it has a continuous-time based denotational semantics. It has also been shown that a stream-based implemen-

tation of FRP converges to this semantics at the limit as sampling intervals drop to zero, modulo some uniform continuity conditions [99].

RT-FRP has a deterministic operational semantics. This semantics allows a notion of cost to be defined in terms of derivation size, which is necessary for proving that RT-FRP is resource bounded.

6 How Do I Learn More?

Jones [51] presents a language-based account of complexity and computability. Hofmann [44] presents a detailed overview of results in the area of programming languages that capture complexity classes.

While there are few implementations of purely functional programming languages, there is a number of good introductions to programming in Haskell which would still serve as an excellent introduction to the subject nevertheless [47, 46]. Di Cosmo's monograph on isomorphisms of types [27] and Hindley's treatment of the simply-typed lambda-calculus [39] are excellent introductions to types in the sense that we have used them here.

Lee [58] gives an overview of how many ongoing efforts fit into the greater picture of transferring software engineering techniques to the area of embedded and real-time systems.

Finally, we have not discussed traditional real-time techniques like priority-based and rate-monotonic scheduling [55]. We expect that these approaches can fit within the traditional frameworks for concurrency [64, 84, 40, 41, 65], and that the same approaches discussed above for the encapsulation of concurrency apply.

Acknowledgments We would like to thank Françoise Bellegarde, Adriana Compagnoni, Bill Harrison, Gordon Pace and John Peterson for reading and giving us feedback on an earlier draft of the paper.

References

1. Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *the Symposium on Logic in Computer Science (LICS' 00)*, pages 84–94. IEEE, June 2000.
2. Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In D. McAllester, editor, *the International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer-Verlag, 2000.
3. John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
4. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

5. Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1991.
6. Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *the Symposium on Logic in Computer Science (LICS '96)*, New Brunswick, 1996. IEEE Computer Society Press.
7. Gerard Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.
8. Gerard Berry. The constructive semantics of pure Esterel (draft version 3). Draft Version 3, Ecole des Mines de Paris and INRIA, July 1999.
9. Gerard Berry and the Esterel Team. *The Esterel v5.21 System Manual*. Centre de Mathématiques Appliquées, Ecole des Mines de Paris and INRIA, March 1999. Available at http://www.inria.fr/meije/esterel.
10. Manfred Broy. A fixed point approach to applicative multi-programming. In *Lecture Notes*. International Summer School on Theoretical Foundations of Programming Methodology, 1981.
11. Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *the Symposium on Principles of Programming Languages (POPL'01)*, 2001.
12. Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36, Geneva, 2000. Springer-Verlag.
13. Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, 1991.
14. Luca Cardelli. Type systems. In Allen B. Jr Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.
15. Paul Caspi, Halbwachs Halbwachs, Nicolas Pilaud, and John A. Plaice. Lustre: A declarative language for programming synchronous systems. In *the Symposium on Principles of Programming Languages (POPL '87)*, January 1987.
16. Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *the International Conference on Functional Programming (ICFP'96)*, pages 226–238, Philadelphia, Pennsylvania, 24–26 May 1996.
17. Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.
18. Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers, 2001.
19. J. Robin B. Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *Proceedings International Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, volume 13 of *Canadian Mathematical Society Conf. Proceedings*, pages 141–169. American Mathematical Society, Providence, RI, 1992.
20. J. Robin B. Cockett and Dwight Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theoretical Computer Science*, 139(1–2):69–113, 1995.
21. Thierry Coquand and Gérard Huet. A theory of constructions. Presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis, 1984.
22. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the Haskell Workshop*, September 2001.

23. Patrik Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *Mathematical Foundations of Programming Semantics*, 1997.
24. Karl Crary and Stephanie Weirich. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)*, pages 184–198, N.Y., January 19–21 2000. ACM Press.
25. Rowan Davies. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.
26. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270, St. Petersburg Beach, 1996.
27. Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhäuser, 1995.
28. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
29. Abbas Edalat and Peter John Potts. A new representation for exact real numbers. *Electronical Notes in Theoretical Computer Science*, 6:14 pp., 1997. Mathematical foundations of programming semantics (Pittsburgh, PA, 1997).
30. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
31. John Peterson et. al. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, Mar 1997. World Wide Web version at http://haskell.cs.yale.edu/haskell-report.
32. Cormac Flanagan and Martín Abadi. Types for safe locking. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 1999.
33. Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.
34. Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.
35. Andrew D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, September 1994.
36. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.
37. David Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
38. Peter Henderson. Functional programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engineering*, 12(2):241–250, 1986.
39. J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.
40. C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.
41. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
42. Wilfred Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.
43. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *the Symposium on Logic in Computer Science (LICS '99)*, pages 464–473. IEEE, July 1999.

44. Martin Hofmann. Programming languages capturing complexity classes. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 31, 2000.
45. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4), Winter 2000.
46. Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.
47. Paul Hudak and Joe Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.
48. R.J.M. Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, Chalmers University of Technology, November 1984.
49. R.J.M. Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 70–81, N.Y., September 27–29 1999. ACM Press.
50. R.J.M. Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Guy L. Steele Jr, editor, *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, volume 23, St Petersburg, Florida, 1996. ACM Press.
51. Neil D. Jones. *Computability and Complexity From a Programming Perspective*. Foundations of Computing. The MIT Press, Cambridge, MA, USA, 1997.
52. Neil D Jones and C. K. Gomard. A partial evaluator for the untyped lambda calculus. DIKU report, University of Copenhagen, Copenhagen, Denmark, 1990. Extended version of [53].
53. Neil D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE International Conference on Computer Languages*, pages 49–58, 1990.
54. Richard Kieburtz. Taming effects with monadic typing. In *the International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 51–62. ACM, June 1999.
55. Richard Kieburtz. Real-time reactive programming for embedded controllers. Available from author's home page, March 2001.
56. Richard B. Kieburtz. Implementing closed domain-specific languages. In *[91]*, pages 1–2, 2000.
57. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.
58. Edward A. Lee. What's ahead for embedded software? In *IEEE Computer*, September 2000.
59. Edward A. Lee. Computing for embedded systems. In *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, 2001.
60. Xavier Leroy. Objective Caml, 2000. Available from http://caml.inria.fr/ocaml/.
61. Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
62. John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.
63. Nicholas McKay and Satnam Singh. Dynamic specialization of XC6200 FPGAs by partial evaluation. In Reiner W. Hartenstein and Andres Keevallik, editors,

International Workshop on Field-Programmable Logic and Applications, volume 1482 of *Lecture Notes in Computer Science*, pages 298–307. Springer-Verlag, 1998.
64. Robin Milner. *A Calculus of Communicating Systems*, volume 81 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
65. Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
66. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
67. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, 1996.
68. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
69. Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *the Symposium on Principles of Programming Languages (POPL '99)*, pages 43–56, San Antonio, Texas, January 1999. ACM.
70. Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.
71. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.
72. Hanne Rijs Nielson and Flenning Nielson. *Semantics with Applications : A Formal Introduction*. John Wiley & Sons, Chichester, 1992. Available online from http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html.
73. Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.
74. Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7. Oxford University Press, New York, NY, 1990.
75. Chris Okasaki. Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2):195–199, March 1998.
76. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
77. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.
78. Paul Hudak Simon Peyton Jones, Philip Wadler, Brian Boutel, John Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.
79. Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *the Symposium on Principles of Programming Languages (POPL '93)*. January 1993. 71–84.
80. Frank Pfenning and Carsten Schürmann. System description: Twelf — A metalogical framework for deductive systems. In H. Ganzinger, editor, *the International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
81. Andrew M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and Andrew M. Pitts, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1995. Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.

82. Gordon D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, September 1976.
83. Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981.
84. Gordon D. Plotkin. An operational semantics for CSP. Technical report, University of Edinburgh, Department of Computer Science, 1982.
85. Jean-Claude Raoult and Jean Vuillemin. Operational and semantic equivalence between recursive programs. *JACM*, 27(4):772–796, October 1980.
86. Jonathan Rees, William Clinger, H. Abelson, N. I. Adams IV, D. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised^4 report on the algorithmic language Scheme. Technical Report AI Memo 848b, MIT Press, 1992.
87. David Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4), 1995.
88. Michael B. Smyth. Powerdomains. In *the Mathematical Foundations of Computer Science Symposium*, volume 45 of *Lecture Notes in Computer Science*, pages 537–543. Springer-Verlag, 1976.
89. Joseph E. Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*, pages 217–252. Cambridge University Press, 1982.
90. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [77].
91. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
92. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.
93. Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In A. Scedrov, editor, *Proceedings of the 1992 Logics in Computer Science Conference*, pages 162–173. IEEE, 1992.
94. Alastair Telford and David Turner. Ensuring Streams Flow. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, 6th International Conference, AMAST '97, Sydney Australia, December 1997*, volume 1349 of *Lecture Notes in Computer Science*, pages 509–523. AMAST, Springer-Verlag, December 1997.
95. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.
96. David A. Turner. Functional programs as executable specifications. *Philosophical Transactions of the Royal Society of London*, A312:363–388, 1984.
97. Philip Wadler. The essence of functional programming. In *the Symposium on Principles of Programming Languages (POPL '92)*, pages 1–14. ACM, January 1992.
98. Philip Wadler. The marriage of effects and monads. In *the International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 63–74. ACM, June 1999.
99. Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *the Symposium on Programming Language Design and Implementation (PLDI '00)*. ACM, 2000.
100. Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *the International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.
101. Hongwei Xi. Upper bounds for standardizations and an application. *Journal of Symbolic Logic*, 64(1):291–303, March 1999.