Practical and Safe Abstractions for Interactive Computation via Linearity

Stefan K. Muller **William A. Duff**
Umut A. Acar

August 2015
CMU-CS-15-130

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a version of a paper that was submitted to the 42^{nd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015) in July 2014. Apart from typographical corrections, the difference between the July 2014 submission and this technical report concerns terminology: here, we use the term "factor" instead of "interactible". We also include some proof details that were omitted from the submission for space reasons, and have made one minor correction as noted in the text.

This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940, by the European Research Council under grant number ERC-2012-StG-308246, and by Microsoft Research.

Keywords: interaction, reactive programming, equational reasoning, linear type systems

Abstract

We propose abstractions and techniques for interactive or reactive computation that are general-purpose, expressive, flexible, and that can enable writing efficient interactive programs. Our key interactivity abstraction is a *factor*: a co-inductive type that abstracts interaction as exchange of information and internal state change. To enable expressive and flexible development of interactive programs, we extend a linearly typed, call-by-value lambda calculus with factors as first-class values. The resulting language, called $\lambda^{\mathbf{i}}$, allows input and output through factors that reflect the changing state of the world. To ensure correctness and efficiency in the presence of such effects, $\lambda^{\mathbf{i}}$ treats interaction as a consumable, linear resource. The small-step semantics of $\lambda^{\mathbf{i}}$ takes advantage of the linearity to prevent unsafe behavior as well as guarantee efficiency. We formalize $\lambda^{\mathbf{i}}$, and establish its type safety and additional meta-theoretical results that show that interaction and purely functional programming are not always inconsistent. We formalize all of our results in the Coq proof assistant. We give an implementation of $\lambda^{\mathbf{i}}$ as an OCaml library along with a relatively broad range of example applications.

1 Introduction

Some of the most interesting and complex software involves interaction with an external agent such as a user, another software system, or a device such as a sensor. Such programs, called interactive or sometimes reactive, can be challenging to design, implement, and reason about because they are usually designed to run forever, and because their semantics and runtime behavior depend on their interaction with the environment.

Interactive programs can be expressed using an event-driven style of programming, where a main event loop waits for events to take place and handles them by scheduling for execution the appropriate *callback functions* or *event handlers*. Event-driven programming offers a general-purpose approach to writing efficient interactive programs. Unfortunately, writing event-driven programs is notoriously difficult for several reasons (e.g. [8]). Perhaps the most important issues are that event-driven programs rely on side effects for communication and that they break abstractions such as the function call abstraction. For example, callbacks don't always return to their caller and they can be scheduled by interactive events or by other callbacks. Due to its complexity, event-driven programming is sometimes described by colorful terms such as "callback hell" [8].

There have been many prior efforts to tame the complexity of interactive computation by developing abstractions. Closely related prior work includes process calculi and functional reactive programming.

Process calculi such as Hoare's CSP [12] (Communicating Sequential Processes) and Milner's π-calculus [21] model the synchronous interactions of large numbers of independent processes. While they could technically be used for writing programs that interact with an external world, the full power and complexity of concurrent programming are not necessary for writing interactive programs, where interaction is between a single program and its environment.

Functional Reactive Programming (FRP) has focused on the problem of interaction between a program and the external world. Elliott and Hudak [9] first introduced functional reactive programming by providing primitives for time-varying values, along with their denotational semantics. Elliott and Hudak's proposal turned out to be difficult to implement safely, due to problems of *causality* [22, 17, 13], and efficiency called *time leaks* and *space leaks* [22, 16, 7]. There has been significant follow-up work on implementing FRP safely and efficiently [5, 7, 16, 18, 13, 17, 22]. At the highest level of abstraction, implementations of FRP operate in synchronous steps, each of which takes a "snapshot" of the external world, computes the desired property on the snapshot, and returns a result, and possibly a new program to evaluate at the next step. Implementing FRP, however, remains challenging: to ensure safety and efficiency, state-of-the-art implementations such as Yampa [22] and Elm [7] limit the expressiveness of the language. As discussed in prior work [7, 22] and in Section 6, even if these problems could be solved, FRP inherits a number of limitations by adopting a synchronous evaluation strategy. These stem from the requirement to establish a sampling rate globally on all parts of a program, leading to response latency, unnecessary computation, imprecision, and even possibly unsoundness—a synchronous implementation converges to the desired semantics of FRP only when the frequency of sampling approaches infinity, usually a physical impossibility [7, 29].

In this paper, we propose general-purpose, flexible, and provably safe abstractions for interactive computation. One key concept behind our approach is the *factor*, a co-inductive type that, at

any time, can be *queried* by supplying a value called a *prompt*. When queried, the factor returns a *response* and a *continuation*, a new factor to continue the interaction. To enable expressing interesting interactive computations compositionally, we extend the simply typed, call-by-value lambda calculus with factors. In the resulting language, called $\lambda^{\mathbf{i}}$, programs can use factors to interact with the external world through, for example, the mouse and the keyboard, and to perform more complex computations that depend directly or indirectly on interaction. Since factors are first-class values in $\lambda^{\mathbf{i}}$, programmers may define higher-order functions that operate on and produce factors.

Since factors can be queried as needed by the program, they offer a fundamentally asynchronous model for interaction. This provides two major benefits. First, because there is no notion of a global clock, and because factors supply their current value only when it becomes available, $\lambda^{\mathbf{i}}$ programs are causal: it is not possible to depend on the future value of a factor. Second, $\lambda^{\mathbf{i}}$ programs can flexibly choose the frequency at which they operate and utilize different frequencies for different components to match the desired precision and efficiency requirements by computing only the results needed at any time, i.e., on demand.

Factors can be viewed as related to the process abstraction in process calculi [12, 21]. Like a process, a factor accepts and responds to messages. Unlike a process, a factor is restricted to communicate only in the specific prompt-and-response fashion, which makes it more similar to resumptions [25]. Beyond this high-level similarity, our work is quite different from process calculi: it does not include the notions of concurrently operating processes but rather relies on function abstraction as the main form of composition. While it is inspired by functional reactive programming, our work differs from it in several fundamental ways. The factor abstraction, which is based on two-way-information exchange, is different from the signals or streams of functional reactive programming. Since it is asynchronous and demand-driven, our evaluation model is different from the synchronous models adopted in functional reactive programming. Finally, our model is inherently imperative as it allows programs to accept input from the external environment during evaluation.

When interacting with the external world via a factor, an interactive program must take care to take into account the changes in the environment. To enable this, our factors return a continuation, which itself is a factor, informing the program of the changes in the external environment. A program can thus remain up to date by discarding a factor after each query in favor of the returned continuation. To enforce proper usage of factors statically, we develop a linear type system that treats factors as linear resources, ensuring that old factors may not be used again. In addition, we take advantage of linearity to ensure that we never hold on to outdated data or factors by presenting an operational semantics for $\lambda^{\mathbf{i}}$ that discards outdated information, preventing efficiency problems analagous to the time and space leaks of functional reactive programming.

By making the side effects explicit, factors make it possible to reason about interaction, which is naturally imperative, in a more functional way. Specifically, we establish as a meta-theoretical result that $\lambda^{\mathbf{i}}$ is functional modulo the external input (Section 4). We show this by factoring out the input using a trace-based simulation technique and showing that evaluation with such inputs can be simulated by purely functional evaluation. In addition, we show that, under sufficient assumptions about the external environment, $\lambda^{\mathbf{i}}$ programs may be reasoned about equationally in ways generally only expected from functional programs. These results suggest that it may be possible to tame the

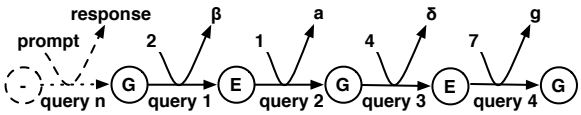

Figure 1: A factor being queried.

imperative nature of interaction and may be of independent interest.

The contributions of this paper include:

- We propose a set of interactivity abstractions, as part of the language $\lambda^{\mathbf{i}}$ (Section 3).
- We establish several meta properties of $\lambda^{\mathbf{i}}$, including its type safety, its consistency with functional programming, and its equational reasoning properties. (Sections 3 and 4).
- We show that $\lambda^{\mathbf{i}}$ is practical by implementing it as an OCaml library (Section 5) and developing a number of examples with it. The examples illustrate the expressiveness and the flexibility of the proposed approach, allowing us to implement a relatively broad range of applications with relative ease.

Supplementary material. All of the theoretical results of this paper have been formalized in the Coq proof assistant. Proof scripts can be obtained from the authors upon request.

2 Overview

We present a brief, informal overview of the important abstractions and ideas using several simple examples. The rest of the paper makes these ideas precise. For the examples, we use an ML-like language with constructs such as integers, floating-point numbers and booleans; such constructs can be added to our calculus (Section 3) in the straightforward way, as for example supported by our OCaml implementation (Section 5).

Our key interactivity abstraction is a *factor*, which is an infinite data structure represented finitely by a function, called a *generator*. A generator takes a *prompt* as an argument, and returns a *response* and a new factor, the *continuation.* A factor can be *queried* by supplying a prompt. When queried, the factor applies its underlying generator to the prompt, and returns the response and the continuation, as determined by the generator. As an example, Figure 1 illustrates a factor (drawn as a circle) that when prompted with an integer returns the matching letter of the alphabet, transitioning between English and Greek alphabets.

We distinguish between two kinds of factors in a program: internal and I/O. An *internal factor* is created within the program by supplying a generator, which may query other (internal and I/O) factors. The simplest case is an internal factor whose generator is a pure function that does not query other factors. On the other hand, *I/O factors* are inputs of the program that are created outside of the program. They can be queried to interact with the outside world by, for example, displaying a command prompt or supplying the mouse position. The generator of an I/O factor is an imperative function that performs the requested measurement or interaction with the outside world, generally through a system call. Since I/O factors reside in the external world, they can capture the state of the world in their continuation. We therefore consider I/O factors stateful or impure and keep their generators as abstract functions unavailable to the program.

Defining factors. As a concrete example, we define the infinite sequence of natural numbers as an internal factor that returns the next natural number every time it is queried. The type `(unit, int) ftr` specifies that when queried with the unit value `()`, `nats` returns the next natural number (an integer). The factor is defined as `natsfrom 0`, where `natsfrom` is a recursive function that takes an integer `n0` and constructs a factor by applying the constructor `ftr` to a generator that, for unit prompt, returns `n0` as the response and `natsfrom (n0 + 1)` as the new factor, effectively incrementing the integer stored in the internal state.

```
let nats : (unit, int) ftr =
  let rec natsfrom (n0: int) =
    ftr (fun () → (n0, natsfrom (n0 + 1)))
  in natsfrom 0
```

Querying factors. Once defined, factors can be queried by supplying a prompt using the `query` construct. For example, we can create a new factor by querying another internal factor and mapping a supplied function over its responses as follows. We can map over I/O factors in the same way by changing the argument type to `(α, β) eftr`. The function `map` is defined as a recursive function that constructs a factor from a generator, which accepts a prompt `p`. It queries the factor `i` with `p`, binding the response and continuation to `h` and `i'`, respectively. The response is `f` applied to `h` and the continuation is generated by calling `map` recursively on continuation `i'`.

```
let rec map (f: β → γ) (i: (α, β) ftr) =
  ftr (fun (p: α) →
       let (h, i') = query i p in (f h, map f i'))
```

Two-way interaction. By allowing a prompt to be specified as part of a query, factors enable information exchange rather than merely information consumption. Here we present a simple example to illustrate an I/O factor that uses prompts in an interesting way. We define a type `direction` with constructors `North`, `South`, `East` and `West`, and assume we have an I/O factor of type `(direction, !(float * float)) eftr` that controls a Mars rover. When prompted with a direction, the factor moves the rover in that direction and returns its new location as a persistent pair of floating-point numbers. We can then define an internal factor that moves the robot in a specified direction and passes the location to a function that determines whether the robot is nearing a known obstacle.

```
let rec avoid_obstacles (r: (direction, !(float * float)) eftr) =
  ftr (fun d → let (p, r') = query r d in
        (near_obstacle p, avoid_obstacles r'))
```

Linearity for soundness. Since the rover moves when `r` is queried, the state of the physical world changes at each query. In this sense, $\lambda^{\mathbf{i}}$ is fundamentally imperative. To provide more control over this imperative nature of interactivity, our factors return a continuation (`r'` in the example above) that represents the changed state of the world. We use this ability to distinguish between the old and the new state of the world to enforce a critical soundness property in interactive programs—that they keep up to date with the changing world—by treating factors as linear resources that can

be used (at most) once. For example, suppose that, as shown below, we ask the rover to move north, and as a result drive it over a cliff (an accident that it survives).

```
let (p1, r') = query r North in (* rover drives off cliff *)
let (p2, _) = query r South in ...
```

Now, even though it may appear that we have an unaffected copy `r` of the rover, which we can continue using, this is not true, because the rover has undergone an unfortunate and irreversible physical process. Our type system therefore rejects this programs, because it uses the rover `r` (a linear resource) twice. If we wish to continue using the rover, we can do so by using `r'` instead of `r`; this would allow us to continue exploring from where the rover is at the bottom of the cliff. As a consequence of this typing discipline, we prove, in addition to type safety, that interactive programs in $\lambda^{\mathbf{i}}$ satisfy interesting properties of functional programs (Section 4).

We note that promotion allows factors and other expressions that do not depend on I/O factors (like `nats` of our first example) to be used in an unrestricted fashion.

Linearity for efficiency. The soundness problem above corresponds to the issue of time and space leaks in FRP which, while not leading to impossible behavior, result in the accumulation of large and expensive computations. Our approach prevents such accumulation by employing an operational semantics that keeps only the most recent continuation of an I/O factor. This is safe because, as described above, our linear type system prevents the use of outdated factors. Note that the programmer can hold on to responses as permitted by the type of the response. For example, if the response is of "of course" type, as in the rover example above, then it can be used in an unrestricted fashion. If the response contains other factors, however, the mechanisms of linearity would apply. As a result, the syntax and the type system make explicit the cost of the operations and do not implicitly hold on to any past computations.[1]

Splitting streams. Preventing multiple uses of I/O factors may seem like a harsh restriction. Returning to the rover example, we may have two functions, `avoid_obstacles` and `explore`, which both require access to the rover factor.

```
(avoid_obstacles r, explore r)
```

Unfortunately, by using `r` twice, this code violates linearity. We could, of course, merge these two features into one large function that explores *and* avoids obstacles, passing the continuation `r'` to the next operation each time `r` is used. As an alternative to this single-threading approach, we allow multiple uses of factors by permitting them to be *split*. The key observation is that, while it is not safe for both functions to access the rover in the state captured by `r` (since one function may move the rover, invalidating that state), we can create two independent handles to access the rover as long as they both remain up-to-date. For example, we can use the construct `split` provided in $\lambda^{\mathbf{i}}$ as follows:

```
let (r1, r2) = split r in (avoid_obstacles r1, explore r2)
```

Operationally, splitting makes two independent factors with the same generator as the original. In our rover example, both resulting factors will have the same behavior on a query (move the rover and return its position), but they will interact with the rover independently.

[1] We appeal to garbage collection to reclaim unused memory objects such as outdated factors.

We note that splitting of *internal* factors is not necessary because the program can define internal factors and their copies as needed and as guided by the type system. It is possible to define an operation that generates two copies of an internal factor by (transitively) splitting all factors on which they depend. However, this can lead to unexpected behavior. For example, splitting an internal factor which counts mouse clicks would result in two internal factors with *separate* counters, rather than two factors sharing the same counter. Requiring the splitting and copying to be done manually makes this behavior explicit. [2]

3 The Language

We present a small linearly typed language for interactive programming. The language includes familiar abstractions of simply typed lambda calculus and several constructs for operating on factors. In the rest of the section, we present the abstract syntax, the statics and the dynamics of the language and prove the main type safety theorem.

3.1 Abstract Syntax

Figure 2 shows the abstract syntax of a subset of the $\boldsymbol{\lambda}^{\mathbf{i}}$ language. The types include units (1), linear functions ($\tau_1 \multimap \tau_2$), positive binary product ($\tau_1 \otimes \tau_2$), and negative binary product ($\tau_1 \mathbin{\&} \tau_2$), contractibles (read "of course" or "bang") ($!\tau$), and finally internal factors ($\mathsf{ftr}(\tau_1; \tau_2)$) and I/O factors ($\mathsf{eftr}(\tau_1; \tau_2)$). The types of internal and I/O factors are differentiated so that the type system can enforce that only I/O factors are split. Both types are parametrized by a *prompt type* τ_1, the type of value that must be supplied by the program to interact with a factor and a *response type* τ_2, the type of value with which the factor will respond.

The expressions of $\boldsymbol{\lambda}^{\mathbf{i}}$ contain the unit value (empty tuple), lambda abstraction, application, let binding and a fixed point operator to provide general recursion. Our linear type system requires the use of both persistent variables (denoted by metavariables x and y) and linear variables (a and b). Variables bound by λ-abstraction, pattern matching and let binding are linear and may be used only once. Variables bound in a fixed point, however, are unrestricted. Closed values of any type τ may be promoted to type $!\tau$ using the explicit introduction form $!$. These types are eliminated by bindings of the form $\mathsf{let}\ !x = e_1\ \mathsf{in}\ e_2$, which binds the value to a persistent variable whose use is unrestricted. Positive pairs $\langle e_1, e_2 \rangle$ may be constructed from expressions e_1 and e_2, and may be eliminated with pattern matching. Pattern matching is used in place of projection for positive pairs because otherwise, linearity would enforce that only one component of a pair could be projected. This is the case for negative pairs, $\langle e_1 \mid e_2 \rangle$, where e_1 and e_2 are unevaluated expressions, only one of which is computed when it is projected. The other component is discarded.

The language also contains the primitives $\mathsf{ftr}\ e$, $\mathsf{query}\ e_1\ e_2$ and $\mathsf{split}\ e$ for operating on factors. We assume that a set of I/O factors is available to the programmer and index them by the metavariable i. The type system and the proofs in Section 4 require that when an I/O factor is split, the two resulting factors be distinguishable. To enable this property, I/O factor literals are implicitly decorated with a unique integer representing the branch. Branch n of factor i will therefore sometimes be written as i^n. When the branch is unimportant, the integer annotation may be omitted. Before

[2] In the original version of this paper, this paragraph stated without the above clarification that "If desired, however, internal interactibles [factors] can be split by (transitively) splitting all interactibles on which they depend. For the sake of simplicity, we don't define such a split operation."

Type	τ	::=	1	nullary product
			$\tau \multimap \tau$	function
			$\tau \otimes \tau$	positive binary product
			$\tau \,\&\, \tau$	negative binary product
			$!\tau$	contractible
			$\mathsf{ftr}(\tau;\tau)$	internal factor
			$\mathsf{eftr}(\tau;\tau)$	I/O factor
Expression	e	::=	x	persistent variable
			a	linear variable
			$\langle\rangle$	empty tuple
			i	I/O factor
			$\lambda\, a : \tau.\, e$	abstraction
			$e\ e$	application
			$\langle e, e\rangle$	positive pair
			$\mathsf{let}\ \langle a, b\rangle = e\ \mathsf{in}\ e$	product pattern match
			$\langle e \mid e\rangle$	negative pair
			$e \cdot \mathsf{l}$	left projection
			$e \cdot \mathsf{r}$	right projection
			$!e$	promotion
			$\mathsf{let}\ !x = e\ \mathsf{in}\ e$	replication
			$\mathsf{ftr}\ v$	factor creation
			$\mathsf{query}\ e\ e$	query
			$\mathsf{split}\ e$	asyncronous split
			$\mathsf{let}\ a = e\ \mathsf{in}\ e$	let
			$\mathsf{fix}\ x\ \mathsf{is}\ e$	fixed point

Figure 2: The abstract syntax of $\boldsymbol{\lambda^{\mathbf{i}}}$.

Linear Context	Δ	::=	$\cdot \mid \Delta, a : \tau$
Persistent Context	Γ	::=	$\cdot \mid \Gamma, x : \tau$
Input Context	Φ	::=	${i_1}^{n_1} : \tau_1, \ldots, {i_m}^{n_m} : \tau_m$

Figure 3: Syntax for contexts

evaluation (i.e. in a source-level program), only one branch of each I/O factor is available, so the branch is implicit and need not be exposed to programmers. Without loss of generality, we may assume that the one branch available for each i is i^1, though this is not necessary for correctness.

3.2 Statics

In formalizing the linear type system of $\boldsymbol{\lambda^{\mathbf{i}}}$, we use a presentation inspired by the calculus of Turner and Wadler [28] and judgmental formulations of linear logic [6]. We extend lambda calculus with a linear type system and factors. The lambda calculus and linear type system fragments are standard, and we will discuss primarily the added features for handling factors. The typing judgment $\Gamma; \Delta; \Phi \vdash e : \tau$ uses three contexts, whose syntax is given in Figure 3. The persistent

context Γ maps persistent variables to types. Weakening and contraction are allowed within Γ as in standard type systems, and so persistent variables may be used zero or more times. The linear context Δ maps linear variables to types, but does not allow contraction, so linear variables may be used at most once, but may be dropped. The type system may therefore more precisely be called affine instead of linear. The input context Φ assigns types to I/O factor literals i. As in the linear context, contraction is not allowed in the input context, forcing I/O factors to be used at most once. The usages of the input and linear contexts are complementary. While the input context assigns types and enforces linearity when the literals are used directly, if a factor literal is bound to a variable and then used, the variable will be added to the linear context, which will be used to assign it a type and enforce the linearity restriction on the use of a variable until the factor literal is substituted for the variable.

The typing rules are presented in Figure 4. For most rules involving two subexpressions, the linear and input contexts are split between the typing judgments for the subexpressions so that linear variables and I/O factors may be used to type only one of these subexpressions. The notable exception is the introduction rule for negative pairs, (&**-I**). Since only one component of a negative pair is evaluated, both components can be typed using the same linear and input contexts.

The rules **(Var)** and **(LVar)** allow variable lookup in the nonlinear and linear contexts, respectively. An expression e of type τ may be *promoted* to type $!\tau$ using rule **(!-I)**. This is only possible if e does not refer to any linear values, a restriction which is enforced by typing e under empty linear and input contexts. These types can be eliminated using the binding construct $\mathsf{let}\ !x = e_1\ \mathsf{in}\ e_2$ and rule **(!-E)**. For e_1 of type $!\tau$, e_2 is typed under a context that includes $x : \tau$ in the *persistent* context, so it may be used multiple times.

The construct $\mathsf{query}\ e_1\ e_2$ may be typed with one of two rules. The rule (eftr**-E)** eliminates I/O factors, and (ftr**-E)** eliminates internal factors. In both cases, the prompt must have a type matching the prompt type of the factor and a response and continuation of the appropriate types are produced. Split expressions, however, may be typed with only one rule, **(Split)**, which, as discussed earlier, requires that the factor be an I/O factor.

As an example, we return to the program described earlier which requests old data from an I/O factor and show that it is ill-typed.

```
1 let (p, _) = query r North in
2 let (p, _) = query r South in ...
```

We assume that `r` is an I/O factor literal contained in the context Φ. The analysis is the same if it is instead a linear variable to which such a literal is bound. Rule **(Let)** requires that the expression `query r North` and the remainder of the program be typed under two subcontexts, Φ_1 and Φ_2, respectively. Because of linearity, `r` may be in only one of these contexts. If it is in Φ_1, the remainder of the program cannot be typed because rule (eftr**-I)** will not find `r` in the input context. Similarly, `query r North` cannot be typed if `r` is in Φ_2.

3.3 Dynamics

Before presenting the dynamics of $\boldsymbol{\lambda}^{\mathbf{i}}$, we define execution traces and user strategies [23], two concepts used in the evaluation of expressions. Informally, a trace keeps a record of the program's interaction with the outside world by listing queries and splits. The user strategy models the state-

(Var)
$$\frac{\Gamma(x) = \tau}{\Gamma; \Delta; \Phi \vdash x : \tau}$$

(LVar)
$$\frac{\Delta(a) = \tau}{\Gamma; \Delta; \Phi \vdash a : \tau}$$

(!-I)
$$\frac{\Gamma; \cdot; \cdot \vdash e : \tau}{\Gamma; \Delta; \Phi \vdash !e : !\tau}$$

(!-E)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : !\tau \qquad \Gamma, x : \tau; \Delta_2; \Phi_2 \vdash e_2 : \tau'}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash \mathsf{let}\ !x = e_1\ \mathsf{in}\ e_2 : \tau'}$$

(1-I)
$$\frac{}{\Gamma; \Delta; \Phi \vdash \langle\rangle : 1}$$

(⊸-I)
$$\frac{\Gamma; \Delta, x : \tau; \Phi \vdash e : \tau'}{\Gamma; \Delta; \Phi \vdash \lambda\, a : \tau.\, e : \tau \multimap \tau'}$$

(⊸-E)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : \tau \multimap \tau' \qquad \Gamma; \Delta_2; \Phi_2 \vdash e_2 : \tau}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash e_1\ e_2 : \tau'}$$

(⊗-I)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_2; \Phi_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash \langle e_1, e_2\rangle : \tau_1 \otimes \tau_2}$$

(⊗-E)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad \Gamma; \Delta_2, a : \tau_1, b : \tau_2; \Phi_2 \vdash e_2 : \tau}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash \mathsf{let}\ \langle a, b\rangle = e_1\ \mathsf{in}\ e_2 : \tau}$$

(&-I)
$$\frac{\Gamma; \Delta; \Phi \vdash e_1 : \tau_1 \qquad \Gamma; \Delta; \Phi \vdash e_2 : \tau_2}{\Gamma; \Delta; \Phi \vdash \langle e_1 \mid e_2\rangle : \tau_1 \,\&\, \tau_2}$$

(&-E1)
$$\frac{\Gamma; \Delta; \Phi \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma; \Delta; \Phi \vdash e \cdot \mathsf{l} : \tau_1}$$

(&-E2)
$$\frac{\Gamma; \Delta; \Phi \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma; \Delta; \Phi \vdash e \cdot \mathsf{r} : \tau_2}$$

(eftr-I)
$$\frac{\Phi(i^n) = \mathsf{eftr}(\tau_1; \tau_2)}{\Gamma; \Delta; \Phi \vdash i^n : \mathsf{eftr}(\tau_1; \tau_2)}$$

(eftr-E)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : \mathsf{eftr}(\tau_1; \tau_2) \qquad \Gamma; \Delta_2; \Phi_2 \vdash e_2 : \tau_1}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash \mathsf{query}\ e_1\ e_2 : \tau_2 \otimes \mathsf{eftr}(\tau_1; \tau_2)}$$

(Split)
$$\frac{\Gamma; \Delta; \Phi \vdash e : \mathsf{eftr}(\tau_1; \tau_2)}{\Gamma; \Delta; \Phi \vdash \mathsf{split}\ e : \mathsf{eftr}(\tau_1; \tau_2) \otimes \mathsf{eftr}(\tau_1; \tau_2)}$$

(ftr-I)
$$\frac{\Gamma; \Delta; \Phi \vdash e : \tau_1 \multimap \tau_2 \otimes \mathsf{ftr}(\tau_1; \tau_2)}{\Gamma; \Delta; \Phi \vdash \mathsf{ftr}\ e : \mathsf{ftr}(\tau_1; \tau_2)}$$

(ftr-E)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : \mathsf{ftr}(\tau_1; \tau_2) \qquad \Gamma; \Delta_2; \Phi_2 \vdash e_2 : \tau_1}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash \mathsf{query}\ e_1\ e_2 : \tau_2 \otimes \mathsf{ftr}(\tau_1; \tau_2)}$$

(Let)
$$\frac{\Gamma; \Delta_1; \Phi_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_2, a : \tau_1; \Phi_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2; \Phi_1, \Phi_2 \vdash \mathsf{let}\ a = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

(Fix)
$$\frac{\Gamma, x : \tau; \cdot; \cdot \vdash e : \tau}{\Gamma; \Delta; \Phi \vdash \mathsf{fix}\ x\ \mathsf{is}\ e : \tau}$$

Figure 4: Statics of $\boldsymbol{\lambda^{\mathsf{i}}}$

$$\textbf{(1-V)}\ \frac{}{\langle\rangle\ \mathsf{val}} \qquad \textbf{(}\multimap\textbf{-V)}\ \frac{}{\lambda\ a : \tau.\ e\ \mathsf{val}} \qquad \textbf{(}\otimes\textbf{-V)}\ \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\langle e_1, e_2\rangle\ \mathsf{val}} \qquad \textbf{(\&-V)}\ \frac{}{\langle e_1 \mid e_2\rangle\ \mathsf{val}} \qquad \textbf{(!-V)}\ \frac{e\ \mathsf{val}}{!e\ \mathsf{val}} \qquad \textbf{(ftr-V)}\ \frac{e\ \mathsf{val}}{\mathsf{ftr}\ e\ \mathsf{val}} \qquad \textbf{(}i\textbf{-V)}\ \frac{}{i\ \mathsf{val}}$$

Figure 5: Value judgment

ful, nondeterministic behavior of the outside world by factoring out all effects using a deterministic function of the query and the current execution trace, which allows dependence on history, in much the same way that a pseudorandom number generator simulates true randomness.

Definition 1 (Execution trace)**.** An *execution trace* T is a comma-separated list of zero or more trace elements, with the most recent event on the right:

$$T ::= \epsilon \mid T, Q(i, e, e') \mid T, Sp(i)$$

- ϵ represents the empty trace
- $T, Q(i^n, e, e')$ indicates that factor i^n was queried with prompt e and returned e' after all the events in T.
- $T, Sp(i^n)$ indicates that factor i^n was split after all of the events in T.

Two traces T_1 and T_2 may be concatenated by writing T_1, T_2.

Definition 2 (User strategy)**.** A *user strategy* ω is a deterministic function that, given a triple of an I/O factor literal, an execution trace and a value (the prompt), produces a value as a response.

The dynamics of $\boldsymbol{\lambda}^{\mathbf{i}}$ involve two judgments. The judgment $e\ \mathsf{val}$ indicates that e is a value. The rules for this judgment are shown in Figure 5. The notable rule is the one for $\langle e_1 \mid e_2\rangle$, which is always a value. Because only one component can be evaluated, evaluation must not occur inside negative pairs until a component is projected.

The small-step evaluation judgment $T; e \overset{\omega}{\mapsto} e'; T'$, shown in Figure 6, indicates that e steps to e' and produces trace T' from trace T. The initial trace T is used in rule **(QueryI-E)** when an I/O factor i^n is queried using prompt e. The user strategy is applied to i (the branch is irrelevant), the current trace T and e, and the response is returned as the result, along with the same factor, i^n. By reusing the same factor literal and not allocating any new data structures, the dynamics ensure that repeatedly querying I/O factors will not cause space to leak over time. A new element is added on to the trace indicating that i^n produced response e' at this point in the computation. By contrast, rule **(Query-E)** queries an internal factor by simply applying the generator to the prompt. The rule **(Split-E)** splits an I/O factor i^n by returning a pair of copies of i, but with branches $2n$ and $2n+1$ (this numbering ensures that branches will always be unique.) Note that while the dynamics manipulate the branch annotations to maintain invariants required for the preservation proof and the results of Section 4, the operational rules themselves are not sensitive to these annotations, and the branches could be erased at runtime, assuming it is not necessary to type-check the running code. In addition, while the rules add on to the trace, the dynamics do not read the trace except through their dependence on the input strategy. The evaluation rules for binding constructs use

(Ftr-S)
$$\frac{T; e \mapsto e'; T'}{T; \mathsf{ftr}\ e \mapsto \mathsf{ftr}\ e'; T'}$$

(Query-S1)
$$\frac{T; e_1 \mapsto e_1'; T'}{T; \mathsf{query}\ e_1\ e_2 \mapsto \mathsf{query}\ e_1'\ e_2; T'}$$

(Query-S2)
$$\frac{e_1\ \mathsf{val} \qquad T; e_2 \mapsto e_2'; T'}{T; \mathsf{query}\ e_1\ e_2 \mapsto \mathsf{query}\ e_1\ e_2'; T'}$$

(Query-E)
$$\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{T; \mathsf{query}\ (\mathsf{ftr}\ e_1)\ e_2 \mapsto e_1\ e_2; T}$$

(QueryI-E)
$$\frac{e\ \mathsf{val} \qquad \omega(i, T, e) = e'}{T; \mathsf{query}\ i^n\ e \mapsto \langle e', i^n \rangle; T, Q(i^n, e, e')}$$

(Split-S)
$$\frac{T; e \mapsto e'; T'}{T; \mathsf{split}\ e \mapsto \mathsf{split}\ e'; T}$$

(Split-E)
$$\frac{}{T; \mathsf{split}\ i^n \mapsto \langle i^{2n}, i^{2n+1} \rangle; T, Sp(i^n)}$$

(!-I-S)
$$\frac{T; e \mapsto e'; T'}{T; !e \mapsto !e'; T'}$$

(!-E-S)
$$\frac{T; e_1 \mapsto e_1'; T'}{T; \mathsf{let}\ !x = e_1\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ !x = e_1'\ \mathsf{in}\ e_2; T'}$$

(!-E-E)
$$\frac{e_1\ \mathsf{val}}{T; \mathsf{let}\ !x = !e_1\ \mathsf{in}\ e_2 \mapsto [e_1/x]e_2; T'}$$

(⊸-E-S1)
$$\frac{T; e_1 \mapsto e_1'; T'}{T; e_1\ e_2 \mapsto e_1'\ e_2; T'}$$

(⊸-E-S2)
$$\frac{e_1\ \mathsf{val} \qquad T; e_2 \mapsto e_2'; T'}{T; e_1\ e_2 \mapsto e_1\ e_2'; T'}$$

(⊸-E)
$$\frac{e_2\ \mathsf{val}}{T; (\lambda\ a : \tau.\ e_1)\ e_2 \mapsto [e_2/a]e_1; T}$$

(⊗-I-S1)
$$\frac{T; e_1 \mapsto e_1'; T'}{T; \langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle; T'}$$

(⊗-I-S2)
$$\frac{e_1\ \mathsf{val} \qquad T; e_2 \mapsto e_2'; T'}{T; \langle e_1, e_2 \rangle \mapsto \langle e_1, e_2' \rangle; T'}$$

(⊗-E-S)
$$\frac{T; e_1 \mapsto e_1'; T'}{T; \mathsf{let}\ \langle a, b \rangle = e_1\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ \langle a, b \rangle = e_1'\ \mathsf{in}\ e_2; T'}$$

(⊗-E)
$$\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{T; \mathsf{let}\ \langle a, b \rangle = \langle e_1, e_2 \rangle\ \mathsf{in}\ e \mapsto [e_1, e_2/a, b]e; T}$$

(&-E1-S)
$$\frac{T; e \mapsto e'; T'}{T; e \cdot \mathsf{l} \mapsto e' \cdot \mathsf{l}; T'}$$

(&-E2-S)
$$\frac{T; e \mapsto e'; T'}{T; e \cdot \mathsf{r} \mapsto e' \cdot \mathsf{r}; T'}$$

(&-E1)
$$\frac{}{T; \langle e_1 \mid e_2 \rangle \cdot \mathsf{l} \mapsto e_1; T}$$

(&-E2)
$$\frac{}{T; \langle e_1 \mid e_2 \rangle \cdot \mathsf{r} \mapsto e_2; T}$$

(Let-S)
$$\frac{T; e_1 \mapsto e_1'; T'}{T; \mathsf{let}\ a = e_1\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ a = e_1'\ \mathsf{in}\ e_2; T'}$$

(Let-E)
$$\frac{e_1\ \mathsf{val}}{T; \mathsf{let}\ a = e_1\ \mathsf{in}\ e_2 \mapsto [e_1/a]e_2; T}$$

(Fix-D)
$$\frac{}{T; \mathsf{fix}\ x\ \mathsf{is}\ e \mapsto [\mathsf{fix}\ x\ \mathsf{is}\ e/x]e; T}$$

Figure 6: Dynamics of $\boldsymbol{\lambda^{\mathbf{i}}}$

two distinct capture-avoiding substitution operations, $[v/x]e$ for persistent variables and $[v/a]e$ for linear variables. Both are defined using straightforward induction on the structure of e.

The reader may notice at this point that there is an issue with the dynamic rule for split. The expression $\mathsf{split}\ i^n$ steps to $\langle i^{2n}, i^{2n+1}\rangle$. The use of $2n$ and $2n+1$ ensures that branch annotations will always be unique. However, if i only appears once in the input context under which $\mathsf{split}\ i^n$ is typed, the resulting expression will not be well-typed because of linearity. Therefore, to allow splitting, the input context may be populated with multiple branches of an input factor.

3.4 Type Safety

Before stating the type safety theorem, we present three definitions regarding input contexts and user strategies. We first require that input contexts are well-formed. At a high level, Definition 3 requires that all types in an input context are I/O factors, and that all branches of the same factor have the same type. It also requires that, if splitting factors are viewed as a tree, all of the branches contained in the input context are leaves of the tree, so that a factor and the factor from which it was split may never be present in an expression simultaneously. Finally, well-formed input contexts may only contain i^n for any $n > 0$. Zero may not be used as a branch number since if i^0 is split, its left child would be itself.

Definition 3 (Well-formed input context)**.** Input context Φ is well-formed, written $\Phi\ \mathsf{ok}$ if, for all $i^n \in \Phi$, all of the below hold:

- $n > 0$
- $\exists \tau_1, \tau_2. \Phi(i^n) = \mathsf{eftr}(\tau_1; \tau_2)$
- for all $i^m \in \Phi$, $\Phi(i^m) = \Phi(i^n)$
- for all $i^m \neq i^n$ such that $desc(i^m, i^n)$, we have $i^m \notin \Phi$

where the predicate $desc(\cdot, \cdot)$ is defined inductively:

$$\begin{gathered} desc(i^n, i^n) \\ \text{for all } i^m,\ desc(i^m, i^n) \rightarrow desc(i^{2m}, i^n) \\ \text{for all } i^m,\ desc(i^m, i^n) \rightarrow desc(i^{2m+1}, i^n) \end{gathered}$$

To indicate that a user strategy is consistent with the input context, we write $\Phi \vdash \omega$. Informally, this requires that for all factors in Φ, ω provides values for i of the type indicated by Φ when given closed values of the correct type as prompts. Further, the returned values must be closed with respect to the linear, persistent and input contexts; they may not have free variables or refer to input factors. Note, however, that we make no assumptions on the trace; a user strategy must produce a value for any syntactically valid trace it is given, even if the trace could not have arisen from a valid execution. In practice, of course, such traces will never be passed to user strategies and the value they would return for an ill-formed trace is immaterial.

Definition 4 (Well-typed user strategy)**.** A user strategy ω is well-typed with respect to an input context Φ, written $\Phi \vdash \omega$, if for all i, T, e such that $i \in \Phi$ and $\Phi(i) = \mathsf{eftr}(\tau_1; \tau_2)$ and $e\ \mathsf{val}$ and $\cdot; \cdot; \cdot \vdash e : \tau_1$, we have $\omega(i, T, e)\ \mathsf{val}$ and $\cdot; \cdot; \cdot \vdash \omega(i, T, e) : \tau_2$.

Finally, we define that two input contexts Φ and Φ' are related if all factors in Φ are present in Φ', but may have split.

Definition 5 (Related input contexts)**.** Input contexts Φ and Φ' are related, written $rel(\Phi, \Phi')$ if:

$$\text{for all } i^n \in \Phi, i^n \in \Phi' \text{ or } (i^{2n} \in \Phi' \text{ and } i^{2n+1}\Phi')$$

and

$$\text{for all } i^n \in \Phi', i^n \in \Phi \text{ or } i^{\lfloor n/2 \rfloor} \in \Phi$$

Theorem 1 (Type safety)**.**

- *Progress: If* $\cdot;\cdot;\Phi \vdash e : \tau$ *and* Φ ok*, then either* e val*, or, for any* T *and* ω *such that* $\Phi \vdash \omega$*, there exist* e' *and* T' *such that* $T; e \overset{\omega}{\mapsto} e'; T'$*.*
- *Preservation: If* $\cdot;\cdot;\Phi \vdash e : \tau$ *and* Φ ok *and* $\Phi \vdash \omega$ *and* $T; e \overset{\omega}{\mapsto} e'; T'$*, then there exists* Φ' *such that* Φ' ok *and* $rel(\Phi, \Phi')$ *and* $\cdot;\cdot;\Phi' \vdash e' : \tau$*.*

Proof. Progress is a straightforward induction on the step derivation. Preservation is proven by induction on the typing derivation, but requires a number of lemmas regarding the correctness of substitution for persistent and linear variables. Both are fully proven in Coq. □

It is worth noting an important feature of the preservation theorem. One would expect preservation to state that if $\cdot;\cdot;\Phi \vdash e : \tau$ and $T; e \overset{\omega}{\mapsto} e'; T'$, then $\cdot;\cdot;\Phi \vdash e' : \tau$. This statement is false, because the step from e to e' might split an I/O factor i^n. This means that e' will contain references to i^{2n} and i^{2n+1}, which are not in Φ. Thus, the input context under which the expression is typed must change as the program evaluates. We can, however, guarantee that if e steps to e' then there exists *some* Φ' under which e' is well-typed. Furthermore, we can guarantee that Φ' is related to Φ.

4 Functional Simulation

Although the syntax and semantics of $\boldsymbol{\lambda^{\mathbf{i}}}$ are similar to those of functional languages, $\boldsymbol{\lambda^{\mathbf{i}}}$ is not purely functional because it interacts with the outside world. In this section, we show two main results restoring functional guarantees to $\boldsymbol{\lambda^{\mathbf{i}}}$. First, we show that $\boldsymbol{\lambda^{\mathbf{i}}}$ programs can be viewed as functional programs *modulo* the responses of I/O factors. In other words, interaction is the only effectful feature of $\boldsymbol{\lambda^{\mathbf{i}}}$. Next, we show that by making certain assumptions about I/O factors, we can restore some forms of equational reasoning to $\boldsymbol{\lambda^{\mathbf{i}}}$ which would otherwise not be possible in general for imperative languages.

4.1 Input simulators and functional semantics

We begin by defining an alternate semantics for $\boldsymbol{\lambda^{\mathbf{i}}}$, which we call $\boldsymbol{\lambda^{\mathbf{i}}_{\mathbf{f}}}$, that evaluates purely functionally using *input simulators* instead of I/O factors. Input simulators are purely functional tree-like data structures that produce values when queried and branch when split. Input simulators will encapsulate all of the state and non-determinism of the interaction with the outside world. In this section, we will show that any run of a $\boldsymbol{\lambda^{\mathbf{i}}}$ program can be simulated by a run of the equivalent $\boldsymbol{\lambda^{\mathbf{i}}_{\mathbf{f}}}$ program and vice versa.

The syntax for input simulators is shown below:

$$S \quad ::= \quad \cdot \mid e, S \mid S \parallel S$$

$$
\begin{array}{rcll}
Sim(i^n;\epsilon) & = & \cdot & \\
Sim(i^n;Q(i^n,e_1,e),T) & = & e, Sim(i^n;T) & \\
Sim(i^n;Q(j^m,e_1,e),T) & = & Sim(i^n;T) & i \neq j \\
Sim(i^n;Q(i^m,e_1,e),T) & = & Sim(i^n;T) & n \neq m \\
Sim(i^n;Sp(i^n),T) & = & Sim(i^{2n};T) \parallel Sim(i^{2n+1};T) & \\
Sim(i^n;Sp(j^m),T) & = & Sim(i^n;T) & i \neq j \\
Sim(i^n;Sp(i^m),T) & = & Sim(i^n;T) & n \neq m
\end{array}
$$

Figure 7: Generation of input simulators

(QueryS-E)

$$\frac{e_1 \text{ val} \qquad e_2 \text{ val}}{T; \mathsf{query}\ (i^n, e_1, S)\ e_2 \mapsto \langle e_1, (i^n, S)\rangle; T, Q(i^n, e_2, e_1)}$$

(SplitS-E)

$$\frac{}{T; \mathsf{split}\ (i^n, S_1 \parallel S_2) \mapsto \langle (i^{2n}, S_1), (i^{2n+1}, S_2)\rangle; T, Sp(i^n)}$$

Figure 8: Semantic rules for handling input simulators

An input simulator can provide a response e and a continuation S, written e, S, or can split into two simulators S_1 and S_2, written $S_1 \parallel S_2$. For ease of presentation, we define a new syntactic class $\hat{e}$ which is identical to the syntactic class e of expressions except that I/O factors i are replaced by a pair (i, S) of the factor literal and an input simulator. In order to determine the exact input simulator that replaces a factor, we must look at a particular evaluation of a $\boldsymbol{\lambda}^{\mathbf{i}}$ program since input simulators encapsulate a particular selection of values for input factors. The information to generate an input simulator is contained in the execution trace produced by $\boldsymbol{\lambda}^{\mathbf{i}}$ evaluation. The function $Sim(i^n;T)$, defined in Figure 7, recurs over the trace T to generate a simulator for branch n of I/O factor i. The relation $e \sim_T \hat{e}$ (read "$\hat{e}$ corresponds to e under trace T") establishes a correspondence between the $\boldsymbol{\lambda}^{\mathbf{i}}$ program e and the $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ program $\hat{e}$ under trace T. This relation requires that $\hat{e}$ be identical to e with I/O factors i replaced by $(i, Sim(i;T))$. The formal definition is quite straightforward and can be found in Appendix A.

Expressions $\hat{e}$ in $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ evaluate using the judgment $T; e \mapsto e'; T'$. This judgment uses all of the rules of Figure 6 except **(QueryI-E)** and **(Split-E)**, which are replaced by the two rules in Figure 8. None of the rules now use ω, so this is omitted from the judgment, as it should be since this input strategy reflects non-functional interaction with the outside world. In rule **(QueryS-E)**, the value at the head of the input simulator is taken as the response, regardless of the prompt. This value is of the syntactic class of $\boldsymbol{\lambda}^{\mathbf{i}}$ expressions, but it is an invariant that it will contain no I/O factors[3], and so can also be treated as a member of the syntactic class $\hat{e}$, performing the obvious conversion implicitly.

While the dynamics of $\boldsymbol{\lambda}^{\mathbf{i}}$ are driven by the input strategy, the dynamics of $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ are driven by the

[3]Input simulators are generated from traces produced by $\boldsymbol{\lambda}^{\mathbf{i}}$ evaluations. For any ω used in such an evaluation, our theorems will require $\Phi \vdash \omega$, which ensures that values of I/O factors can be typed under empty input contexts.

$$\frac{}{\omega \rightsquigarrow \epsilon} \qquad \frac{\omega(i, T, e') = e \qquad \omega \rightsquigarrow T}{\omega \rightsquigarrow T, Q(i^n, e_1, e)} \qquad \frac{\omega \rightsquigarrow T}{\omega \rightsquigarrow T, Sp(i^n)}$$

Figure 9: The compatibility judgment $\omega \rightsquigarrow T$

simulators corresponding to each I/O factor. As we have seen, input simulators can be generated from imperative runs using the function $Sim(\cdot;\cdot)$. Expressing the equivalence between a $\boldsymbol{\lambda}^{\mathbf{i}}$ evaluation and a $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ evaluation also requires the reverse operation. That is, given a $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ evaluation, we wish to generate an input strategy that will produce an equivalent imperative evaluation. For this reason, $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ evaluations also produce an execution trace recording the splits and queries of input simulators. So that both evaluations produce traces of the same form, $\boldsymbol{\lambda}^{\mathbf{i}}_{\mathbf{f}}$ evaluation records in the trace the factor literal corresponding to the input simulator that is queried or split. The judgment $\omega \rightsquigarrow T$ (read "ω is compatible with T") indicates that ω can produce trace T. It is defined inductively on the trace by the rules in Figure 9. The only nontrivial constraint imposed on ω is that if, at a point after T has been generated, i is queried with prompt e' to produce value e, then $\omega(i, T, e')$ must produce e.

4.2 Bisimulation

Our goal is now to show a correspondence between an imperative run $\epsilon; e \overset{\omega}{\mapsto}{}^* e'; T$ and a functional run $\epsilon; \hat{e} \mapsto^* \hat{e}'; T$ when $e \sim_T \hat{e}$. We show this essentially by proving that $\sim_T$ is a bisimulation, with a caveat. In a standard bisimulation, one would assume that $e \sim_{T_0} e'$ and show that e and e' simulate each other. That is, they can take identical steps and the resulting expressions are related. Formally, if $T; e \overset{\omega}{\mapsto} e'; T, t$, then there exists $\hat{e}'$ such that $T; \hat{e} \mapsto \hat{e}'; T, t$ and if $T; \hat{e} \mapsto \hat{e}'; T, t$ then there exists e' such that $T; e \overset{\omega}{\mapsto} e'; T, t$, and in both cases $e' \sim_{T_0} \hat{e}'$. In our case, however, the bisimulation relation is parametrized by a trace T_0 that must be the trace of events *still to come* in the evaluation of e and $\hat{e}$. This means that e' and $\hat{e}'$ should not be related by T_0. Instead, it must be the case that $T_0 = t, T_1$ and $e' \sim_{T_1} \hat{e}'$. With this in mind, we present the bisimulation lemma.

Lemma 1 (Bisimulation)**.** *Suppose* $\cdot;\cdot;\Phi \vdash e : \tau$ *and* Φ ok*. Let* T' *and* t *be such that* $e \sim_{t,T'} \hat{e}$*.*

- *If* $\Phi \vdash \omega$ *and* $T; e \overset{\omega}{\mapsto} e'; T, t$ *then there exists* $\hat{e}'$ *such that* $T; \hat{e} \mapsto \hat{e}'; T, t$ *and* $e' \sim_{T'} \hat{e}'$*.*
- *If* $\omega \rightsquigarrow T, t$ *and* $T; \hat{e} \mapsto \hat{e}'; T, t$ *then there exists* e' *such that* $T; e \overset{\omega}{\mapsto} e'; T, t$ *and* $e' \sim_{T'} \hat{e}'$*.*

Proving this lemma requires two important properties of the relation $e \sim_T \hat{e}$. Lemma 2 states that the relation between two expressions continues to hold when an element is removed from the beginning of the trace if the I/O factor to which that element relates is not contained in e. This lemma, specifically its corollary, Corollary 1, is used in cases of the bisimulation proof where an expression involving two subexpressions takes a step by stepping one of the subexpressions, for example if $T; \langle e_1, e_2\rangle \overset{\omega}{\mapsto} \langle e_1', e_2\rangle; T, a$. The corollary states that, if $e_2 \sim_{T,a} \hat{e_2}$, then $e_2 \sim_T \hat{e_2}$. This corollary, and thus the bisimulation result, depend critically on our linear type system. If the step $T; e_1 \overset{\omega}{\mapsto} e_1'; T, a$ involved a factor i that was also allowed to appear in e_2, the result would no longer hold.

Lemma 2 (Trace element removal preserves relation)**.** *Let* $e, \hat{e}, \Gamma, \Delta, \Phi$ *and* τ *be such that* $\Gamma; \Delta; \Phi \vdash e : \tau$*. If* $e \sim_{Q(i^n, e_1, e), T} \hat{e}$ *or* $e \sim_{Sp(i^n), T} \hat{e}$ *and* $i^n \notin \Phi$ *then* $e \sim_T \hat{e}$*.*

Proof. By induction on the derivation of $e \sim_{t,T} e'$, where $t = Q(i^n, e_1, e)$ or $t = Sp(i^n)$. The nontrivial case is $i'^{n'} \sim_{t,T} Sim(i'^{n'}; t, T)$. Since $\cdot;\cdot;\Phi \vdash e : \tau$ and $i^n \notin \Phi$, we know that $i'^{n'} \neq i^n$. By inspection of the rules for $Sim(\cdot;\cdot)$, this means that $Sim(i'^{n'}; t, T) = Sim(i'^{n'}; T)$, so $i'^{n'} \sim_T Sim(i'^{n'}; T)$. □

Corollary 1 (Relation on independent subexpressions)**.** *Let* $e_1, e_2, e_1', \hat{e_2}, \Gamma, \Delta_1, \Delta_2, \Phi_1, \Phi_2, \Phi, \tau_1$ *and* τ_2 *be such that* $\Gamma;\Delta_1;\Phi_1 \vdash e_1 : \tau_1$ *and* $\Gamma;\Delta_2;\Phi_2 \vdash e_2 : \tau_2$ *and* $\Phi = \Phi_1, \Phi_2$ *(the last premise ensures that* Φ_1 *and* Φ_2 *are disjoint.) If* $e_2 \sim_{T,a} \hat{e_2}$ *and* $T'; e_1 \overset{\curvearrowright}{\omega} e_1'; T', a$ *then* $e_2 \sim_T \hat{e_2}$.

Proof. By induction on the derivation of the step $T'; e_1 \overset{\curvearrowright}{\omega} e_1'; T', a$, it can be shown that the factor to which a pertains must be contained in e_1, and, by inversion on the typing derivation of e_1, must therefore be in Φ_1. The factor is therefore not contained in Φ_2 and Lemma 2 applies. □

It was noted above that a $\boldsymbol{\lambda^{i}}$ expression containing no I/O factor literals can also be viewed as a member of the syntactic class of $\boldsymbol{\lambda^{i}_{f}}$ expressions. Lemma 3 states that if e is such an expression, e treated as a $\boldsymbol{\lambda^{i}}$ expression is related to e treated as a $\boldsymbol{\lambda^{i}_{f}}$ expression under any trace.

Lemma 3 (Relation reflexivity)**.** *Let* e, Γ, Δ, τ *be such that* $\Gamma;\Delta;\cdot \vdash e : \tau$*. For all* t*,* $e \sim_t e$.

Proof. By induction on the structure of e. We need not consider the case where $e = i$, since this could not be typed under an empty input context. □

We can use the bisimulation result to show that the evaluation of a $\boldsymbol{\lambda^{i}}$ program and the functional evaluation of the related $\boldsymbol{\lambda^{i}_{f}}$ expression remain related throughout the multistep evaluation of a full program starting with an empty trace. This shows that a $\boldsymbol{\lambda^{i}}$ program is indistinguishable up to bisimulation from a functional expression that factors out external behavior. Theorem 2 shows that any program e that runs to produce a trace T can be simulated by a $\boldsymbol{\lambda^{i}_{f}}$ program $\hat{e}$ such that $e \sim_T \hat{e}$, in that the latter can also evaluate to produce the trace T and a related final result. Theorem 3 shows the reverse direction: if $\hat{e}$ evaluates, then e can evaluate to produce a corresponding trace and final result.

Theorem 2 (Forward multistep simulation)**.** *Suppose that* Φ ok *and* $\cdot;\cdot;\Phi \vdash e : \tau$ *and* $\Phi \vdash \omega$ *and* $e \sim_T \hat{e}$*. If* $\epsilon; e \overset{\curvearrowright}{\omega}{}^{*} e''; T$ *then there exists* $\hat{e}''$ *such that* $\epsilon; \hat{e} \mapsto^{*} \hat{e}''; T$ *and* $e'' \sim_\epsilon \hat{e}''$.

Proof. We prove a slightly stronger result by assuming that $T_0; e \overset{\curvearrowright}{\omega}{}^{*} e''; T_0, T$ and showing that $T_0; \hat{e} \mapsto \hat{e}''; T_0, T$. The result for $T_0 = \epsilon$ follows as a special case.

Let $\hat{e}$ be the result of substituting $Sim(i^n; T)$ for i^n in e, for all i^n. We have $e \sim_T \hat{e}$ by construction. We prove the lemma by induction on the derivation of $T_0; e \overset{\curvearrowright}{\omega}{}^{*} e'; T$. If $e = e'$, the result is trivial. Otherwise, $T_0; e \overset{\curvearrowright}{\omega} e'; T_0, T_1$ and $T_0, T_1; e' \overset{\curvearrowright}{\omega}{}^{*} e''; T_0, T_1, T_2$. To apply the induction hypothesis, we must show that $T_0; \hat{e} \mapsto \hat{e}'; T_0, T_1$ and $\cdot;\cdot;\Phi \vdash e' : \tau$ for some valid input context, and $e' \sim_{T_2} \hat{e}'$. We show this by induction on the derivation of $T_0; e \overset{\curvearrowright}{\omega} e'; T_0, T_1$, considering only the cases that don't follow from straightforward induction.

- **(Query-S1)**. Then $\hat{e} = \mathsf{query}\ \hat{e_1}\ \hat{e_2}$. By induction, $T_0; \hat{e_1} \mapsto \hat{e_1}'; T_0, T_1$ and $e_1' \sim_{T_2} \hat{e_1}'$. For the result $e \sim_{T_2} \hat{e}$, it remains to show that $e_2 \sim_{T_2} \hat{e_2}$. This follows from Corollary 1. **(Query-S2)**, **(QueryI-S)**, **(⊸-E-S1)**, **(⊸-E-S2)**, **(⊗-I-S1)**, **(⊗-I-S2)**, **(⊗-E-S)** and **(Let-S)** are similar.

- **(Query-E)**. Then $T_1 = \epsilon$ and $\hat{e} = \mathsf{query}\ \mathsf{ftr}\ (\lambda\ x : \tau.\ \hat{e_1})\ \hat{e_2}$, where $e_1 \sim_{T_2} \hat{e_1}$ and $e_2 \sim_{T_2} \hat{e_2}$, so $[e_2/x]e_1 \sim_{T_2} [\hat{e_2}/x]\hat{e_1}$. Applying **(Query-E)**, $T_0; \hat{e} \mapsto [\hat{e_2}/x]\hat{e_1}; T_0$. **(⊸-E)**, **(⊗-E)**, **(&-E1)**, **(&-E2)**, **(Let-E)** and **(Fix-D)** are similar.
- **(QueryI-E)**. Then $e = \mathsf{query}\ i^n\ e_0$ and

$$\hat{e} = \mathsf{query}\ Sim(i^n; Q(i^n, e_0, e_1), T_2)\ e_0 = \mathsf{query}\ e_1, Sim(i^n; T_2)\ e_0$$

  and $e' = \langle e_1, i^n \rangle$. Applying **(QueryS-E)** gives $T_0; \hat{e} \mapsto \langle e_1, Sim(i^n; T_2) \rangle; T_0, T_1$. By the definition of $\Phi \vdash \omega$, we have that $\cdot; \cdot; \cdot \vdash e_1 : \tau'$ for some τ', so Lemma 3 gives $e_1 \sim_{T_2} e_1$. By applying the rules for the relation, we can see that $\langle e_1, i^n \rangle \sim_{T_2} \langle e_1, Sim(i^n; T_2) \rangle$.
- **(SplitI-E)**. Then $e = \mathsf{split}\ i^n$ and

$$\hat{e} = \mathsf{split}\ Sim(i^n; Sp(i^n), T_2) = \mathsf{split}\ Sim(i^{2n}; T_2) \parallel Sim(i^{2n+1}; T_2)$$

  and $e' = \langle i^{2n}, i^{2n+1} \rangle$. Applying **(SplitS-E)** gives $T_0; \hat{e} \mapsto \langle Sim(i^{2n}; T_2) \parallel , Sim(i^{2n+1}; T_2) \rangle; T_0, T_1$. By applying the rules for the relation, we can see that $\langle i^{2n}, i^{2n+1} \rangle \sim_{T_2} \langle Sim(i^{2n}; T_2), Sim(i^{2n+1}; T_2) \rangle$.

□

Theorem 3 (Backward multistep simulation)**.** *Suppose* Φ ok *and* $\cdot; \cdot; \Phi \vdash e : \tau$ *and* $e \sim_T \hat{e}$ *and let* ω *be such that* $\omega \rightsquigarrow T$. *If* $\epsilon; \hat{e} \mapsto^* \hat{e''}; T$ *then there exists* e'' *such that* $\epsilon; e \overset{\omega}{\mapsto}{}^* e''; T$ *and* $e'' \sim_\epsilon \hat{e''}$.

Proof. As above, we prove a stronger result, assuming $T_0; \hat{e} \mapsto^* \hat{e'}; T_0, T_1$ and $\omega \rightsquigarrow T_0, T_1$ and concluding that $T_0; e \overset{\omega}{\mapsto} e'; T_0, T_1$. The proof is similar to that of Theorem 2. The case where $e = e'$ is trivial, so we prove the result in the case where $T_0; \hat{e} \mapsto \hat{e'}; T_0, T_1$ and $T_0, T_1; \hat{e'} \mapsto^* \hat{e''}; T_0, T_1, T_2$. We consider the two nontrivial cases for the bottommost rule in the derivation of the step.

- **(QueryS-E)**. Then $\hat{e} = \mathsf{query}\ \hat{e_1}, S\ \hat{e_2}$ and $e = \mathsf{query}\ i^n\ e_2$ and $\hat{e'} = \langle \hat{e_1}, S \rangle$ and $T_1 = Q(i^n, e_2, e_1)$. By the rules for $\cdot \rightsquigarrow \cdot$, we have $\omega(i, T_0, e_2) = \hat{e_1}$. Suppose $\omega \rightsquigarrow T_0, T_1, T_2$. Applying **(QueryI-E)** gives $T_0; e \overset{\omega}{\mapsto} \langle \hat{e_1}, i^n \rangle; T_0 T_1$. Since $e \sim_{T_1, T_2} \hat{e}$, we know that $S = Sim(i^n; T_2)$, so $i^n \sim_{T_2} S$ and the result follows from this and Lemma 3.
- **(SplitS-E)**. Then $\hat{e} = \mathsf{split}\ S_1 \parallel S_2$ and $e = \mathsf{split}\ i^n$ and $\hat{e'} = \langle S_1, S_2 \rangle$ and $T_1 = Sp(i^n)$. Suppose $\omega \rightsquigarrow T_0, T_1, T_2$. Applying **(SplitI-E)** gives $T_0; e \overset{\omega}{\mapsto} \langle i^{2n}, i^{2n+1} \rangle; T_0, T_1$. By the rules for the relation, we have that $S_1 = Sim(i^{2n}; T_2)$ and $S_2 = Sim(i^{2n+1}; T_2)$. This gives the desired result.

□

4.3 Equational reasoning

Many of the beneficial properties of functional programming, including referential transparency, are related to the ability to perform equational reasoning. For example, if `query` were a pure function, the following two programs that query two external sensors for the current temperature would have identical behavior:

```
let prog1 =
  let (t1, _) = query sensor1 () in
  let (t2, _) = query sensor2 () in
  (t1, t2)

let prog2 =
  let (t2, _) = query sensor2 () in
  let (t1, _) = query sensor1 () in
  (t1, t2)
```

In $\boldsymbol{\lambda^{\mathbf{i}}}$, the sensors may depend arbitrarily on each other, and so these programs will not, in general, behave identically. In certain cases, however, they will. Assume for the moment that the two sensors are in different rooms, so their temperature readings are independent of each other, and that the temperature in both rooms is constant, so the readings are independent of time.[4] In this case, we would expect the programs to behave identically. We now show how the results of Section 4.2 give us the ability to justify this sort of reasoning about $\boldsymbol{\lambda^{\mathbf{i}}}$ programs formally. Suppose we have two $\boldsymbol{\lambda^{\mathbf{i}}_{\mathbf{f}}}$ expressions, $\hat{e_1}$ and $\hat{e_2}$, which evaluate to the same value (but may produce different traces in the process.) If e_1 and e_2 are $\boldsymbol{\lambda^{\mathbf{i}}}$ programs from which $\hat{e_1}$ and $\hat{e_2}$ are derived, we want the guarantee that e_1 and e_2 will also evaluate to the same value. This is the result of Theorem 4.

Theorem 4 (Equational reasoning)**.** *Let e_1 and e_2 be $\boldsymbol{\lambda^{\mathbf{i}}}$ programs, and let $\hat{e_1}$ and $\hat{e_2}$, respectively, be the corresponding $\boldsymbol{\lambda^{\mathbf{i}}_{\mathbf{f}}}$ programs, that is, $e_1 \sim_{T_1} \hat{e_1}$ and $e_2 \sim_{T_2} \hat{e_2}$. Suppose $\cdot;\cdot;\Phi \vdash e_1 : \tau_1$ and $\cdot;\cdot;\Phi \vdash e_2 : \tau_2$. If $\epsilon;\hat{e_1} \mapsto^* \hat{e'};T_1$ and $\epsilon;\hat{e_2} \mapsto^* \hat{e'};T_2$ and $\omega \rightsquigarrow T_1$ and $\omega \rightsquigarrow T_2$ then there exists e' such that $\epsilon;e_1 \overset{\omega}{\mapsto}{}^* e';T_1$ and $\epsilon;e_2 \overset{\omega}{\mapsto}{}^* e';T_2$ and $e' \sim_\epsilon \hat{e'}$.*

Proof. By Theorem 3, $\epsilon;e_1 \overset{\omega}{\mapsto}{}^* e'_1;T_1$ and $\epsilon;e_2 \overset{\omega}{\mapsto}{}^* e'_2;T_2$ such that $e'_1 \sim_\epsilon \hat{e'}$ and $e'_2 \sim_\epsilon \hat{e'}$. The fact that $e'_1 = e'_2$ follows from a straightforward induction on the structure of $\hat{e'}$. □

Theorem 4 requires that ω be compatible with the traces of *both* functional executions. The strength of the assumptions we make about ω governs the class of programs we can prove equivalent (or, if e_2 is viewed as the result of applying some transformation to e_1, the class of transformations we can prove valid.) Any transformation made to e_1 to produce e_2 will have a corresponding effect on the trace. For example, reversing the order in which two factors are queried, as in the examples `prog1` and `prog2` earlier in this section, will result in the order of the query events being swapped between the traces t_1 and t_2. For any pair of expressions, we can use Theorem 4 to guarantee that the expressions will behave identically if we can prove a "trace compatibility" lemma stating that for traces T_1 and T_2 resulting from this pair of expressions, $\omega \rightsquigarrow T_1$ implies $\omega \rightsquigarrow T_2$. If the trace compatibility lemma makes certain assumptions about ω, then the result about the program equivalence will hold under the same assumptions.

A particularly strong assumption we might make about a user strategy is that it is independent of the trace; this corresponds to a world that has no memory of past history. Formally, for all i, e, T_1, T_2,

$$\omega(i, T_1, e) = \omega(i, T_2, e)$$

[4] In general, I/O factors may depend upon features, such as time, that are outside our model. Our semantics, and our metatheoretic results, assume that this dependence is factored out as part of the user strategy.

This strong assumption allows us to prove quite a strong trace compatibility lemma.

Lemma 4 (Trace compatibility for memoryless strategies)**.** *Suppose ω is "memoryless" as defined above, and that T_1 and T_2 are permutations of each other. If $\omega \rightsquigarrow T_1$, then $\omega \rightsquigarrow T_2$.*

Proof. By induction on T_2. The case for ϵ imposes no constraints on ω and so is trivial. If $T_2 = T_2, Q(i^n, e, e')$, then T_1 must contain $Q(i^n, e, e')$. Since $\omega \rightsquigarrow T_1$, we know that $\omega(i, T, e) = e'$ for some trace, and so trace-independence gives us $\omega(i, T_2, e) = e'$. The remainder of the result follows by induction. □

This correspondence between trace-independent strategies and permuted traces allows us to prove that, assuming a trace-independent input strategy, two programs produce the same value if the corresponding functional programs produce the same value and permuted traces. This result follows immediately from Theorem 4 and Lemma 4.

Corollary 2. *Suppose ω is memoryless and $\Phi \vdash \omega$. Suppose $\cdot; \cdot; \Phi \vdash e_1 : \tau_1$ and $\cdot; \cdot; \Phi \vdash e_2 : \tau_2$, and let T_1, T_2 be traces that are permutations of each other. If $e_1 \sim_{T_1} \hat{e_1}$ and $e_2 \sim_{T_2} \hat{e_2}$ and $\epsilon; \hat{e_1} \mapsto^* \hat{e}'; T_1$ and $\epsilon; \hat{e_2} \mapsto^* \hat{e}'; T_2$ and $\omega \rightsquigarrow T_1$, then there exists e' such that $\epsilon; e_1 \overset{\omega}{\mapsto}{}^* e'; T_1$ and $\epsilon; e_2 \overset{\omega}{\mapsto}{}^* e'; T_2$ and $e' \sim_\epsilon \hat{e}'$.*

This result is strong enough to guarantee the equivalence of the example programs `prog1` and `prog2`, since these programs will produce permuted traces. However, we can also prove these programs equivalent using a weaker assumption on the user strategy than full trace-independence. Instead, we consider "locally trace-sensitive" user strategies in which I/O factors are independent of each other's history but not their own. Formally, for all i, e, T_1, T_2,

$$(T_1)\restriction_i = (T_2)\restriction_i \Rightarrow \omega(i, T_1, e) = \omega(i, T_2, e)$$

where $(T)\restriction_i$ is the restriction of a trace to events pertaining to i:

$$\begin{array}{rcll} (\epsilon)\restriction_i & := & \epsilon & \\ (Q(i,e,e'),T)\restriction_i & := & Q(i,e,e'),(T)\restriction_i & \\ (Q(i',e,e'),T)\restriction_i & := & (T)\restriction_i & \text{if } i' \neq i \\ (Sp(i),T)\restriction_i & := & Sp(i),(T)\restriction_i & \\ (Sp(i'),T)\restriction_i & := & (T)\restriction_i & \text{if } i' \neq i \end{array}$$

This assumption on strategies allows the traces of Theorem 4 to have the events pertaining to different I/O factors interleaved arbitrarily, as long as the traces restricted to one factor remain in the same order. We first prove the appropriate trace compatibility lemma.

Lemma 5 (Trace compatibility for local strategies)**.** *Let T_1 and T_2 be such that for all i, $(T_1)\restriction_i = (T_2)\restriction_i$. If ω is locally trace-sensitive, as defined above and $\omega \rightsquigarrow T_1$, then $\omega \rightsquigarrow T_2$.*

Proof. We first prove three intermediate results for our strategy ω:

- If a and a' pertain to different I/O factors and $\omega \rightsquigarrow a, a', T$, then $\omega \rightsquigarrow a', a, T$.
- If a and a' pertain to different I/O factors and $\omega \rightsquigarrow T, a, a', T'$, then $\omega \rightsquigarrow T, a', a, T'$.

- If a pertains to i and $(T') \restriction_i = (T_2) \restriction_i$ and for all i', $(T_1, a, T_2) \restriction_{i'} = (a, T') \restriction_{i'}$ and $\omega \rightsquigarrow T_1, a, T_2$, then $\omega \rightsquigarrow T_1, T_2, a$.

Each of these results builds on the previous one by straightforward induction. We show the main result by induction on T_2. If $T_2 = T_2', Q(i, e, e')$, then $(T_1) \restriction_i = (T) \restriction_i, Q(i, e, e')$ for some T, so there exist T and T' such that $T_1 = T, Q(i, e, e'), T'$ and $(T_2') \restriction_i = (T) \restriction_i$ and for all i, $(T_2') \restriction_i = (T, T') \restriction_i$. We know that $\omega \rightsquigarrow T, Q(i, e, e'), T'$. By result 3 above, we have $\omega \rightsquigarrow T, T', Q(i, e, e')$, so $\omega(i, (T, T'), e) = e'$ and $\omega \rightsquigarrow T, T'$. The assumption on ω then gives $\omega(i, T_2', e) = e'$ and $\omega \rightsquigarrow T_2'$ follows by induction, as does the case for $T_2 = T_2', Sp(i)$. □

A result about programs with such traces then follows directly from Theorem 4 and Lemma 5.

Corollary 3. *Suppose ω is independent of interleavings and $\Phi \vdash \omega$. Suppose $\cdot; \cdot; \Phi \vdash e_1 : \tau_1$ and $\cdot; \cdot; \Phi \vdash e_2 : \tau_2$, and let T_1, T_2 be such that for all i, $(T_1) \restriction_i = (T_2) \restriction_i$. If $e_1 \sim_{T_1} \hat{e_1}$ and $e_2 \sim_{T_2} \hat{e_2}$ and $\epsilon; \hat{e_1} \mapsto^* \hat{e}'; T_1$ and $\epsilon; \hat{e_2} \mapsto^* \hat{e}'; T_2$ and $\omega \rightsquigarrow T_1$, then there exists e' such that $\epsilon; e_1 \overset{\omega}{\mapsto}{}^* e'; T_1$ and $\epsilon; e_2 \overset{\omega}{\mapsto}{}^* e'; T_2$ and $e' \sim_\epsilon \hat{e}'$.*

5 Implementation and Examples

Since it requires just a few additions to lambda calculus, $\boldsymbol{\lambda^{\mathbf{i}}}$ can probably be implemented in any functional language. We implemented it as an OCaml library and used the library to develop a number of examples, ranging from simple to more sophisticated examples that demonstrate the flexibility and the expressivity of the proposed techniques.

Our implementation follows closely the operational semantics (Section 3.3). Since OCaml does not have a linear type system, however, our implementation cannot enforce the invariants required by our type system, i.e., that all I/O factors are used linearly. We therefore check this property dynamically by furnishing the runtime representations of I/O factors with additional facilities that raise an exception when they are used more than once.

5.1 The Interface

Figure 10 shows the core of the interface for our interaction library. The interface allows the creation and use of internal and I/O factors, defined by the abstract types `('p, 'r) ftr` and `('p, 'r) eftr` respectively, where `'p` is the prompt type and `'r` is the response type.

A factor can be created from a generator using the function `ftr`. The `query` function queries an internal factor, returning a response and a new factor of the same type. The library provides the `eftr` function for implementing user-defined I/O factors, the `equery` function for querying them, and the `split` function for splitting them.

For the examples presented in the rest of the section, we assume an implementation `I` of the interface `Interactive`.

5.2 Examples

Standard library. While the core of the implementation remains quite small, we provide a standard library containing many I/O factors useful for writing actual applications, such as mouse input, keyboard input and system time. The library also contains basic operations over factors, such as map and fold, and their variants.

```
module type Interactive = sig
  type ('p, 'r) ftr
  type ('p, 'r) eftr

  (* Create a new factor from a generator. *)
  val ftr : ('p → 'r * ('p, 'r) ftr) → ('p, 'r) ftr
  (* Query a factor. *)
  val query : ('p, 'r) ftr → 'p → 'r * ('p, 'r) ftr

  (* Create a new I/O factor from a generator. *)
  val eftr : ('p → 'r * ('p, 'r) eftr) → ('p, 'r) eftr
  (* Query an I/O factor. *)
  val equery : ('p, 'r) eftr → 'p → 'r * ('p, 'r) eftr

  (* Split an I/O factor into two. *)
  val split : ('p, 'r) eftr → ('p, 'r) eftr * ('p, 'r) eftr
end
```

Figure 10: The $\lambda^{\mathbf{i}}$ core interface

A top-level loop. Evaluation in $\lambda^{\mathbf{i}}$ is driven by the querying of factors within a program rather than, as in many interactive languages, a top-level loop that synchronously advances all time-varying values. Such a top-level can be defined *within* $\lambda^{\mathbf{i}}$ if desired using a recursive function that repeatedly queries the desired factor, perhaps performing some action like printing the responses to the screen as output. Because the loop is under the control of the programmer, the frequency at which the factor is queried can be controlled by inserting the desired delays between recursive calls. We use this technique in several of the examples described in this section.

A physics simulation. In this example, we consider simulating free-falling objects such as perfectly elastic balls within a rigid box. This example illustrates interesting interaction patterns, for example allowing the user to insert new balls to the system, while also performing some non-trivial computation involving moving objects. The simulation detects collisions with the floor and the ceiling, bouncing a colliding ball off the wall by adjusting its velocity vector. The simulation also computes the "high-point" marker, a ball that tracks the highest ball at any time. Figure 12 shows a snapshot from the simulation.

The code below illustrates the `simulate` function. The function takes a specification of the box and several unit-prompted factors representing, respectively, time, acceleration, mouse position, mouse clicks and the balls currently in the simulation. Since balls can be added dynamically, the balls (each represented by a unit-prompted factor returning positions) are passed in as a list. We note that representing the collection of balls requires higher-order factors. The function `simulate` checks whether the mouse has been clicked. If so, a new ball is added to the simulation whose initial velocity and position are determined from the mouse position by function `make_ball` (details of which are not shown here). This allows the user to insert new balls by clicking the mouse. Regardless of whether a ball is added, `simulate` generates a renderer (a

function that displays the ball) for each ball by mapping the function `render` (not shown here) over the collection. The renderers are then combined, calculating the high-point marker in the process. Another function called `graphics_loop` (not shown here), renders the final simulation by querying the returned factor and passing the collection of renderers to OCaml's graphics library.

```
(* simulate: float*float → (float, unit) I.ftr → (float, unit) I.ftr →
           (int*int, unit) I.eftr → (bool, unit) I.ftr →
           ((float*float, unit) I.ftr) list → (unit → unit, unit) I.ftr *)
let simulate box time acc mouse_pos mouse_clicks balls =
  let rec gen clicks mpos balls () =
    let (c, clicks') = I.equery clicks () in
    let balls' =
      if c then (* Mouse is clicked; add a new ball *)
        let (pos, mpos) = V.equery mpos () in
        let (v, x) = make_ball pos in
        let ball = velocity_position box v x time acc in
          (ball::balls)
      else
        balls
    in
    let (renderers, balls'') = List.unzip (map render balls')
    in
    let render_all () = combine_renderers renderers in
      (render_all, I.ftr (gen click' mpos' balls''))
in  I.ftr (gen mouse_clicks mouse_pos balls)
```

Since velocity depends on the position (because a ball bounces back when it collides with the box, changing its velocity), we compute the velocity and the position of the ball together. The code for function `velocity_position` is shown below. The function integrates acceleration and velocity, to compute the velocity and the position of a ball, taking care to reverse the velocity when the ball collides with the box. The `fold2` function folds over a pair of factors to build a factor of responses, providing `f` with the pair of values and the current accumulator. The function `f` returns the new accumulator and either `Some` and a response or `None` to produce no response.

```
(* velocity_position: (float*float) → float → float →
                  (float, unit) I.ftr → (float, unit) I.ftr) →
                   (float*float, unit) I.ftr *)
let velocity_and_position (left, right) v0 x0 time acc =
  let bounce (v, x) =
    if x < left then (~-. v, left)
    else if (x > right) then (~-. v, right)
    else  (v, x)
  in let f (t, a, (v, x, pt)) =
       match pt with
       | None → ((v, x, Some t), Some (v, x))
       | Some pt' →
            let v' = v +. (a *. (t -. pt')) in
```

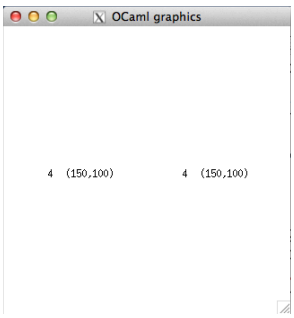

Figure 11: The number of (left) mouse-button clicks and the position of the mouse pointer displayed twice.

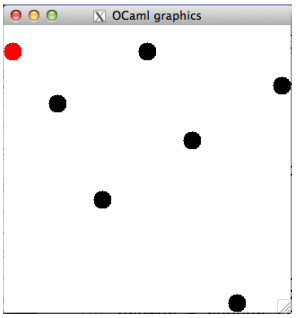

Figure 12: An interactive physics simulation, high-point marker drawn in red.

```
14              let x' = x +. (v' *. (t -. pt')) in
15              let (v'', x'') = bounce (v',x') in
16                ((v'', x'', Some t), Some (v'', x''))
17  in fold2 f (v0, x0, None) time acc
```

Since interactive computations in our approach are performed on demand, `graphics_loop` can query the factor of renderers at any desired frequency, indirectly determining the frequency at which the positions are updated.

Physics Simulation 2D. This example generalizes the 1D physics simulation to 2D, allowing objects move in a bounded 2D space on 2D trajectories, and extends it to enable 1) collision detection between moving objects and 2) input from the user that alters the visualization and the dynamics of the simulation. The example employs multiple and varying polling frequencies for improved efficiency and accuracy, performs certain computations in a demand-driven fashion to avoid unnecessary work, and employs multithreading.

In the normal state, two balls proceed according to the simulation, bouncing off of the walls of the box and each other. We vary the polling frequency of the system time roughly linearly with the velocity of the balls (higher velocities lead to more polling). By polling infrequently when the balls are moving slowly, this policy ensures efficiency. By polling frequently when the balls are moving quickly, the policy ensures accuracy. We further increase accuracy by predicting collision times (based on velocity and acceleration) and perform an update exactly at the time of a collision, ensuring that no collisions are missed.

The example allows the user to interact with the system by catching a ball with the mouse and dragging it. When a ball is being dragged, its factor polls the mouse position in order to update the position of the ball, but does not compute the integrals for velocity and position. When the ball is released, the computation of the integrals resumes but the mouse position is no longer queried. Demanding input values only when they are needed reduces unnecessary polling and computation.

This example also contains different parts operating at different frequencies. Specifically, we use a separate thread for allowing the user to change the color of the balls by pressing certain keys. We configure the thread querying the I/O factor of key presses to block until a key is pressed. Since

the thread handles only this task, it is acceptable to block, and is efficient, because the thread does not have to poll continuously, which would results in unnecessary work.

Tax Assistance. Our library appears to be particularly useful for building console applications, even ones involving complex interactions where inputs to the program affect the control flow. As an example of such an interaction familiar to many US taxpayers, we implemented a console application guiding the user through completing the IRS W-4 form. The I/O factor `prompt_input`, which displays its prompt to the user, reads a line from standard input and returns the line as its result, is split and used to build several unit-prompt factors, each of which asks a particular question (e.g. "Are you married?" or "What is your income?") when queried. The main function calculating the user's withholding allowances is written essentially as a mathematical formula, querying the appropriate factors to get the answers to each question when they are needed. Our framework easily allows the low-level details of performing the interaction to be factored out. These details include prompting the user, reading the input, processing the input and asking again if the input was invalid. At the end, the user is given the opportunity to change the answer to a question, in which case the appropriate factor is queried and the computation run again.

Arrowized FRP. Arrowized FRP (AFRP) [22], notably implemented in the Yampa package for Haskell, is an efficient and causal variation on Elliott and Hudak's FRP [9]. We embedded a substantial subset of Yampa, which is one of the most mature and complete implementations of FRP, in $\lambda^{\mathbf{i}}$, and used the embedding to implement the tailgating detection example of Nilsson et al. [22]. Our implementations of Yampa and of the tailgating example consist of approximately 300 and 120 lines of code respectively. More details of the embedding of Yampa in $\lambda^{\mathbf{i}}$ are available in Appendix B. This example shows that the proposed approach is expressive enough to support an encoding of previous prior work. We also implemented directly the same example in our $\lambda^{\mathbf{i}}$ library in approximately 75 lines of code.

Elm. Elm [7] allows programmers to write computations by manipulating time-varying values called *signals*, and provides an `async` construct, which allows long-running computations to run outside of the otherwise synchronous, global update model. We implemented in $\lambda^{\mathbf{i}}$ an example from the paper of Czaplicki and Chong [7] which uses `async` to prevent an expensive machine translation task from delaying the mouse-position updates. Because our model does not include concurrency, our implementation depends on the compatibility of our model with basic concurrency primitives. Other than the asynchrony, the implementation was straightforward, and we are currently working on embedding an interface similar to that of Elm, as we did with AFRP.

Unix Shell. As an example of a real-world program with many low-level interactions, we implemented `fsh`, a Unix shell that handles foreground and background jobs and supports history, command line editing and tab completion. If a foreground job is running, `fsh` periodically queries the factor `signals` to poll for signals from the operating system. Otherwise, `fsh` queries standard input to perform the interaction with the user. These functions are therefore encapsulated in the $\lambda^{\mathbf{i}}$ library. Other low-level operations, such as forking new processes through the OCaml Unix library and limited C interface, are encapsulated within separate functions. Much of the code, however, handles higher-level operations, such as command line processing, operations on the data

structures that store job status and command line history, and functions to support tab completion. These tasks are programmed quite naturally in the high-level, functional style of our library.

6 Related Work

We discuss most closely related work in the relevant sections of the paper. Here, we present a broader review of related work.

6.1 Process Calculi

Process calculi, which model concurrent systems, have been studied extensively, leading to a vast body of work, and many different calculi (e.g., [2, 11]). Programming languages based on process calculi include Concurrent ML [26], which is based on CSP, and Pict [24], which is based on π-calculus. Our factor abstraction is related to the process abstraction in process calculi: like a process, a factor can be viewed as accepting and responding to messages. Since factors "communicate" only via the query-response interface, however, they are more similar to Plotkin's resumptions [25], which have the same type, but are used to model processes and non-determinism [25, 1]. The key difference between process calculi and our work is that we are interested in the interaction between a computation and the external world but not between concurrently executing processes. We therefore base our calculus on a variant of the lambda calculus rather than a process calculus, such as π-calculus.

6.2 Functional Reactive Programming

Introduced by Elliot and Hudak in 1997 [9], Functional Reactive Programming (FRP), provides powerful primitives for operating on continuously changing values called *behaviors*, and discretely changing values called *events*. Elliott and Hudak specified the semantics of FRP by presenting a denotational semantics. Much of the follow-up work on FRP concerns techniques for implementing the denotational semantics in practice. We present a brief overview of this work, touching upon the important milestones. More details and more citations can be found in recent papers [7, 16, 13], which include excellent bibliographies that together span a broader scope than we can here. As described in Section 1, our work differs from FRP in several ways, including in the abstractions used (factors), in its asynchronous, demand-driven evaluation strategy, and perhaps more fundamentally in its (carefully controlled) imperativeness.

Implementations of FRP. Wan and Hudak [29] proposed a stream-based implementations of the original FRP semantics [9], where behaviors and events were represented as streams of values. Much like synchronous dataflow languages such as Signal [10], Lustre [4] and Esterel [3], these implementations of FRP adopted a *synchronous* strategy, where a program executes by periodically sampling the values of signals from the environment, processing them, and returning an output. Unlike in synchronous dataflow languages, which evaluate a mostly static, unchanging dataflow graph, in FRP, the program changes over time, requiring changes to the dataflow graph itself after each step. Wan and Hudak [29] proved that the synchronous stream-based implementation of FRP that they proposed is consistent with the original denotational semantics for many (but not all) programs, when the size of the sampling interval approaches zero (the sampling frequency approaches infinity).

Subsequent work identified safety and efficiency problems in FRP: *causality* [22, 17, 13] and

space-time leaks [22, 16, 7], leading to much follow-up work on safe and efficient implementations. Initial approaches restricted the language to prevent writing unsafe and inefficient programs. Real-time FRP [30] proposed techniques for eliminating time and space leaks by using signals as a uniform representation of behaviors and events. By using a two-tiered reactive and non-reactive language, and by prohibiting higher order signals, real-time FRP can bound space and time usage in the reactive portion of the program (thus eliminating space and time leaks), while providing no bounds on the non-reactive portion. Following up on real-time FRP work, event-driven FRP [31] introduced discrete signals that change only at events.

Arrowized FRP [22, 20, 27, 19] aims at regaining the expressiveness lost in real-time FRP while keeping its efficiency benefits. To this end, arrowized FRP disallows operating on signals directly, offering instead a programming model centered around the idea of signal functions that can transform their input signals. Signal functions can be viewed as circuit elements in synchronous dataflow languages. But unlike such languages, arrowized FRP supported dynamic switching operators that can reconfigure the circuit (network of signal functions), regaining much of the expressiveness of the original FRP while avoiding time and space leaks and also preserving causality. Ensuring that arrowized FRP programs are correct, safe and efficient, especially in the presence of dynamic switching, can be difficult, however [27], e.g., dynamic switching operators can also make time and space leaks possible [16].

More recent approaches turned to semantics and type systems for ensuring that FRP programs are safe and efficient. Schulthorpe and Nilsson [27] use dependent types to enforce statically the safety of arrowized FRP programs. Krishnaswami, Benton, and Hoffman [18] use linear types to eliminate space leaks from FRP programs. Jeffrey [13], and independently Jeltsch [15] use linear temporal logic as a type system to guarantee causality. Jeffrey [14] presents a type system for FRP that guarantees liveness—that every input event leads to an output event. Krishnaswami [16] proposes a type system based on temporal logic that can prevent inadvertent space and time leaks. Krishnaswami's results are similar to ours and the prior work on real-time FRP [30] (which considers a more restrictive language) in the sense that the programmer can still hold on to computations and data but can only do so explicitly. Cave et al. [5] use linear temporal logic to provide for an expressive FRP language and precise liveness guarantees.

Synchrony. Much of the work on FRP can be viewed as a generalization of synchronous dataflow languages [10, 4, 3] to enable richer computations where the dataflow graph can change from one step to another. Unfortunately synchronous evaluation leads to a number of limitations, some of which are discussed previously [7, 22, 31], and which our proposal appears to avoid. One limitation is that a step cannot be started before the previous one finishes, especially in more interesting instances with dynamic switching [5]. This leads to a range of practical difficulties, including an inherent latency, and the need to choose the right frequency for updates: each step must be long enough for the execution of a step to complete but cannot be too large because it can lead to soundness problems [31]. A large step size also increases latency and can lead to external buffering of the streams that operate at a higher frequency. Unfortunately, it is impossible to determine the right step size (frequency) because the evaluation time of a step depends on the input for that step, which cannot be known *a priori*. Even if a step size can be established, it would

[5] In static networks, steps can be pipelined, an least in principle.

have to be applied globally to all the different parts of the program regardless of their specific requirements. For example, in many applications, a mouse might need to be polled at a higher frequency than a rarely-used button, such as the on/off button of the computer, which might even be polled completely asynchronously by, for example, blocking.

Czaplicki and Chong [7] point out some of the limitations of the synchronous evaluation strategy, and propose techniques for allowing certain computations to span multiple time steps. While not fully asynchronous, their approach gives some flexibility to the programmer in avoiding the strictures of synchrony. Based on this idea, they develop the Elm language, which is arguably one of the most-well developed FRP languages, offering an impressive array of examples and features. As with much other work on FRP, however, Elm does not allow higher-order streams.

7 Conclusion

This paper presents techniques for interactive computation based on three key ideas: 1) abstraction of interaction with a single primitive called a factor that models interaction as exchange of information and internal change, 2) writing interactive programs compositionally using higher-order functions as in simply typed lambda calculus, 3) using a linear type system that treats interaction as consumable resource, leading to a safe and efficient implementation. Perhaps surprisingly, we also show that interaction is not necessarily imperative by proving that certain forms of interaction are in fact consistent with purely functional programming. Our implementation and the examples considered show the techniques to be effective in practice, enabling expression of a relatively diverse set of examples succinctly. While not a focus of this paper, our implementations employ several different forms concurrency favorably, which we plan to research in the future.

References

[1] Samson Abramsky. Retracting some paths in process algebra. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 1–17, 1996.

[2] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005.

[3] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, 1987.

[5] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair reactive programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 361–372, 2014.

[6] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131, Carnegie Mellon University, April 2003.

[7] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 411–422, 2013.

[8] Jonathan Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, 2009.

[9] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

[10] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, UK, 1987. Springer-Verlag.

[11] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[13] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, pages 49–60, 2012.

[14] Alan Jeffrey. Functional reactive programming with liveness guarantees. In *Proceedings of the 18th ACM International Conference on Functional Programming*, 2013. forthcoming.

[15] Wolfgang Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 69–78, 2013.

[16] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.*, 48(9):221–232, September 2013.

[17] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *LICS '11: Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 257–266, 2011.

[18] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 45–58, 2012.

[19] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, 2009.

[20] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, November 2007.

[21] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

[22] Henrik Nilsson, Antony Courtney, and Joh Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[23] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201, Piscataway, NJ, USA, July 2006. IEEE Press.

[24] Benjamin C. Pierce and David N. Turner. Proof, language, and interaction. chapter Pict: A Programming Language Based on the Pi-Calculus, pages 455–494. 2000.

[25] Gordon D Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[26] John H. Reppy. CML: a higher concurrent language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 293–305, 1991.

[27] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. *SIGPLAN Not.*, 44(9):23–34, August 2009.

[28] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231 – 248, 1999.

[29] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[30] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *SIGPLAN Not.*, 36(10):146–156, 2001.

[31] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, 2002.

A Definition of Bisimulation Relation

$$\frac{}{\langle\rangle \sim_T \langle\rangle} \qquad \frac{}{x \sim_T x} \qquad \frac{Sim(i^n; T) = S}{i^n \sim_T (i^n, S)} \qquad \frac{e \sim_T \hat{e}}{\lambda\, x : \tau.\, e \sim_T \lambda\, x : \tau.\, \hat{e}} \qquad \frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{e_1\ e_2 \sim_T \hat{e_1}\ \hat{e_2}}$$

$$\frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{\langle e_1, e_2\rangle \sim_T \langle \hat{e_1}, \hat{e_2}\rangle} \qquad \frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{\mathsf{let}\ \langle x, y\rangle = e_1\ \mathsf{in}\ e_2 \sim_T \mathsf{let}\ \langle x, y\rangle = e_2\ \mathsf{in}\ \hat{e_2}}$$

$$\frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{\langle e_1 \mid e_2\rangle \sim_T \langle \hat{e_1} \mid \hat{e_2}\rangle} \qquad \frac{e \sim_T \hat{e}}{e \cdot \mathsf{l} \sim_T \hat{e} \cdot \mathsf{l}} \qquad \frac{e \sim_T \hat{e}}{e \cdot \mathsf{r} \sim_T \hat{e} \cdot \mathsf{r}} \qquad \frac{e \sim_T \hat{e}}{!e \sim_T\ !\hat{e}}$$

$$\frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{\mathsf{let}\ !x = e_1\ \mathsf{in}\ e_2 \sim_T \mathsf{let}\ !x = \hat{e_1}\ \mathsf{in}\ \hat{e_2}} \qquad \frac{e \sim_T \hat{e}}{\mathsf{ftr}\ e \sim_T \mathsf{ftr}\ \hat{e}} \qquad \frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{\mathsf{query}\ e_1\ e_2 \sim_T \mathsf{query}\ \hat{e_1}\ \hat{e_2}}$$

$$\frac{e \sim_T \hat{e}}{\mathsf{split}\ e \sim_T \mathsf{split}\ \hat{e}} \qquad \frac{e_1 \sim_T \hat{e_1} \qquad e_2 \sim_T \hat{e_2}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \sim_T \mathsf{let}\ x = \hat{e_1}\ \mathsf{in}\ \hat{e_2}} \qquad \frac{e \sim_T \hat{e}}{\mathsf{fix}\ x\ \mathsf{is}\ e \sim_T \mathsf{fix}\ x\ \mathsf{is}\ \hat{e}}$$

B Details of AFRP Encoding

We show how to encode AFRP signal functions as factors in $\boldsymbol{\lambda^{\mathbf{i}}}$, allowing an AFRP library similar to Yampa to be implemented within $\boldsymbol{\lambda^{\mathbf{i}}}$. One implementation of AFRP [22] implements a signal function in approximately the following way.

```
type ('a, 'b) sf = time -> 'a -> 'b * ('a, 'b) sf
```

In this implementation, a signal function is a concrete function whose arguments are the amount of time that has passed since the last time the signal was sampled, and the current value of the input signal. The result is the new value of the output factor and the continuation of the signal function, which is ready to be used at the next time step. This type can be directly implemented using $\boldsymbol{\lambda^{\mathbf{i}}}$ factors.

```
type ('a, 'b) sf = ('b, time * 'a) ftr
```

Under this encoding, however, signal functions may only be used once, a restriction that doesn't exist in AFRP. To allow multiple uses, a signal function in our implementation must be *a function that produces* a signal function of the type above. This leads to the following implementation.

```
type ('a, 'b) raw_sf = ('b, time * 'a) ftr
type ('a, 'b) sf = unit -> ('a, 'b) raw_sf
```

Standard AFRP signal functions and combinators can be programmed using this type. For example, `integral`, which in AFRP is simply a signal function that transforms real-valued signals[6] to their integrals, can be implemented as follows.

[6] In fact, the integral function allows integration of more complicated types, but we do not implement this here.

```
let integral () =
  let rec integral_gen a (dt, h) =
    let v = a +. dt *. h in
    (v, ftr (integral_gen v))
  in
  ftr (integral_gen 0.)
```

Note that `integral` is a function taking a unit argument, as required by our signal function type. To run top-level signal functions, we provide an outer loop that applies the given signal function to produce the underlying factor and then queries it continuously, calling a function `f` to handle each value in turn (`f` is likely an effectful function that, for example, prints the value to the screen.)

```
let run (f: 'a -> unit) (sf: (unit, 'a) sf) =
  let rec run_rec s =
    let (h, t) = query s (delta, ()) in
      (f h; run_rec t)
  in
  run_rec (sf ())
```

Note that a refresh rate (`delta` above) must be specified and cannot be viewed or set by the program, though it could be varied dynamically by the runtime.