Interactive Computation in an Open World

Stefan K. Muller  
*Carnegie Mellon University*

Umut A. Acar  
*Carnegie Mellon University*

William Duff  
*Carnegie Mellon University*

Robert Harper  
*Carnegie Mellon University*

Follow this and additional works at: [http://repository.cmu.edu/compsci](http://repository.cmu.edu/compsci)  
Part of the [Computer Sciences Commons](http://repository.cmu.edu/compsci)
Abstract

Interactive applications must operate correctly and efficiently while reacting to events in an unpredictable external world. Currently, most such programs are written in an event-driven style that can often be difficult to understand and maintain because callbacks defeat modularity and often share global state. Elliott and Hudak proposed Functional Reactive Programming (FRP), which tames this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, this complexity by representing interactive values as functions of time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, its realizations limit the sampling of external events and signals to time. Although FRP has an elegant denotational semantics, it...
used; otherwise, sampling blocks (waits) until the value becomes available. Asynchrony of sampling means that there is no external time coordinate, and so there can be no violations of causality (as there can be in classical FRP) stemming from mixing “current” with “future” values—sampling may itself be viewed as advancing time.

To model the external world with which a program interacts, we employ Brouwer’s concept of a free choice sequence [37]. A free choice sequence may be seen as the intuitionistic analogue of the concept of an arbitrary function in classical set theory—one that is not necessarily given by any formula, law, or algorithm governing its behavior. The lawless aspect of choice sequences is ideal for modeling the unpredictability of the external world; we regard external agents as Brouwerian “creative subjects” whose behavior need not be governed by any algorithm. Such a choice sequence may represent input such as the characters entered on a keyboard, the packets on a network, or the movements of a mouse pointer. The dynamics is parameterized by a collection of such choice sequences, which are chosen arbitrarily for each run. Because the semantics allows for all possible choice sequences, individual factors (e.g. the input from the keyboard) may be instantiated with an element of a set of choice sequences that make sense for that factor (e.g. a sequence of bytes representing a piece of text). But this need not be the case; quantifying over all possible choices represents an entirely unpredictable world for which the program (and our reasoning techniques) must be prepared.

Allowing for the possibility of entirely unpredictable behavior yields some surprising results, which we capture in an equational semantics. We make the somewhat counterintuitive observation that programs designed to be run in (and be robust to) an unpredictable world display more structure than those designed to run under assumptions about the real world. Part of the reason for this appears to be that instantiating each factor with an arbitrary choice sequence yields quite a strong assumption: all factors are independent of each other. A not-so-surprising result of this independence assumption is that queries to distinct factors may be re-ordered without changing the behavior of the program. A more surprising result is that, under some circumstances, a program that is designed to have predictable behavior in an unpredictable world (for example, one that counts the number of times the mouse is clicked in 10 seconds) can actually be shown to be independent of its input! Of course, this counterintuitive result reflects the fact that we cannot always assume total independence of input sources in the real world, but this assumption can be applicable in certain instances. For example, the result about re-ordering queries to unrelated factors can be used to show that two program components working with plausibly independent input sources (e.g. the keyboard and the network) may safely be run concurrently in separate threads.

Based on these ideas, we present the static and dynamic semantics for \( \lambda^1 \) (Section 2). We then present a metatheory for the language which shows type safety properties and also enables deriving equations between programs written in \( \lambda^1 \) (Section 3). Because the equational semantics is quantified over an arbitrary assignment of free choice sequences, reasoning using this theory makes the assumption of an entirely unpredictable world. The technical machinery for the metatheory is based on a logical relations technique of Ahmed, Fluet, and Morrisett [3], which allows strong updates to linear memory locations. We adopt their approach to our setting by relaxing the linearity requirement to allow for weakening in the context of linear variables and by naming interactive values in a manner analogous to locations in memory (but no storage effects are involved).

To assess the practicality of \( \lambda^1 \), we describe an implementation as an OCaml library (Section 4). Because of the limitations of OCaml’s type system, affinity is not enforced statically, but multiple uses of an I/O factor will be caught at runtime and result in an exception. This is sufficient for building a fairly broad set of applications including interactive physics simulations, a Unix shell, and a multiplayer game. These examples illustrate a variety of modes of interaction with the user, operating system, and the network.

2. The Language

In this section, we present \( \lambda^1 \), a simple, mostly-functional language extended with factors for interaction. Before presenting the language in detail, we will describe and motivate several of its important features.

Our approach revolves around a single construct called a factor for representing and operating on interactive values. A factor is an abstract type that models interaction as a two-way exchange of information. Intuitively, a factor can be thought of as an interactive agent that responds to queries while maintaining an internal state that accumulates information over time based on queries received.

To support input-output and interaction between different agents in a computation, we distinguish between two kinds of factors: internal factors and I/O factors. For simplicity, we usually refer to internal factors simply as factors.

An internal factor is a coinductive data type or data structure; it is defined by an (internal) state and a generator, which is a function that maps the state and a prompt to a new state and a response. The key operation on factors is a query, which supplies the factor with a prompt and returns a response and a new updated continuation factor, or simply a continuation, which can be used for further interaction. The response is computed by applying the generator to the prompt and the internal state, thus taking into account both the present and the past. Querying a factor updates its state, which is then passed on to the continuation.

As mentioned in the introduction, the semantics of factors are based on Brouwerian free choice sequences [37]. To illustrate the connection between factors and Brouwer’s choice sequences, consider the following exercise. Let \( F_0 \) be an internal factor and query \( F_0 \) with some prompt to obtain a response \( r_0 \) and continuation factor \( F_1 \). Now repeat the experiment with \( F_1 \) to obtain \( r_2 \), and then with \( F_2 \) to obtain \( F_3 \) and so on for \( n \) times. Thinking of the continuations of \( F_n \) as its instances, we can represent the sequence of responses as a lawlike choice sequence \( (r_0,r_1,\ldots,r_{n-1}) \). This sequence is lawlike because it is determined by the algorithm (as specified by the generator and the initial internal state) governing the factor and the prompts issued by the queries.

I/O factors enable interaction with the external world. They are defined similarly to internal factors: they can be queried like internal factors and can be thought of as maintaining some internal state. However, they cannot be introduced by the programmer. The important semantic difference between internal factors and I/O factors is that I/O factors are lawless in that they cannot be assumed to be governed by any law or algorithm. For example, repeating the same experiment as above with an I/O factor \( F'_0 \) would return a lawless sequence \( (r'_0,r'_1,\ldots,r'_{n-1}) \), the values of which may not obey any law or be computed by any algorithm. I/O factors capture the unpredictability of the external world from the perspective of the interactive program. For example, we may think of an I/O factor for interacting with a user as an oracle that determines the successive inputs provided by a user responding to queries.

The language \( \lambda^1 \) relies on an affine type system to ensure that a program does not accumulate, or memoize, state over time. The affine type system rules out programs that hold on to outdated I/O factors. This means that after a factor is queried, it cannot be used again, though its continuation can be used to continue interaction.
2.1 Abstract Syntax

The abstract syntax for $\lambda^I$ is presented in Figure 1. Our language has an affine type system (described in more detail in Section 2.2), which is reflected in the types. These include natural numbers $\mathbb{N}$, functions $\tau \to \tau$, positive pair $\tau \otimes \tau$, and unrestricted values. We define some syntactic sugar which we will use standard infix notation for. We also define the syntax of three contexts. Persistent contexts $\Gamma$ map persistent (unrestricted) variables to types. Affine contexts $\Delta$ map affine variables to types. Input contexts $\Phi$ contain the I/O factor names which may be used in an expression.

### Figure 1. The abstract syntax of $\lambda^I$.###

<table>
<thead>
<tr>
<th>Persistent Variables $x, y, \ldots \in Var$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affine Variables $a, b, \ldots \in AVar$</td>
</tr>
<tr>
<td>I/O Factor Names $i \in Sym$</td>
</tr>
<tr>
<td>Natural Numbers $n, m \in \mathbb{N}$</td>
</tr>
</tbody>
</table>

### Type $\tau$ ::= $\text{nat}$ $\tau \to \tau$ $\tau \otimes \tau$ $\text{!}\tau$ $\text{eftr}$ $\text{ftr}$

### Value $v ::= x$ $a$ $\lambda a : \tau. e$ $\text{!}v$ $v \otimes v$ $\text{let} a \otimes b = v$ in $e$ $\text{let} \text{!x} = v$ in $e$ $\text{eq} \text{v} v$ $\text{query} v v$ $\text{let} a = e$ in $c$

### Expression $e ::= v$ $\text{rec}(v; v, a.e)$ $\text{v v}$ $\text{let} a \otimes b = v$ in $e$ $\text{let} \text{!x} = v$ in $e$ $\text{let} \text{v} v$ $\text{eq} \text{v} v$ $\text{let} a = e$ in $c$ $\text{fst}(e) \triangleq \text{let} a = e$ in $\text{let} b \otimes c = a$ in $b$ $\text{snd}(e) \triangleq \text{let} a = e$ in $\text{let} b \otimes c = a$ in $c$

Finally, we include an introduction form for internal factors. Recall that factors are a coinductive type; querying a factor produces a value and a new factor, which can continue the interaction. The new factor is generated by feeding forward some “internal state.” The internal state must be sufficient to, in turn, generate the next response and next internal state. However, it can be of arbitrary, including function, type. In the form $\text{ftr}[\tau]$ with $v$ as s.p.e., the parameter $\tau$ gives the type of this internal state, and $v$ is the state itself. The body of the factor, $e$, accepts two arguments: $x$, which will be bound to the current state, and $p$ which will be bound to the prompt, and must produce a pair of the response and the new state.

Expressions $e$ include values. Primitive recursion on natural numbers, noted $\text{rec}(n; v, x.e)$, is defined inductively over the natural number $n$, producing $v$ if $n$ is zero and, if $n$ is nonzero, evaluating $v$ with $x$ bound to the result of recursively calling $\text{rec}(n - 1; v, x.e)$. Positive products are eliminated by pattern matching: let $a \otimes b = v$ in $e$ evaluates with $a$ and $b$ bound to the first and second components of the pair $\tau$, respectively. Pattern matching is used in place of projection operators, which would eliminate product types in a non-affine language, since an affine pair can only be used once, and so, if projection were used, only one component could be projected. The standard projection operators can be defined (and are under “Syntactic Sugar” below) if desired. The $\text{l!}$-elimination form let $\text{l!x} = v$ in $e$ (pronounced “let-bang”) substitutes the unrestricted value $v$ for the unrestricted variable $x$ in $e$. The constructs $\text{eq} \text{v} v$ and query $v_1 v_2$ query the (I/O or internal, respectively) factor $v_1$ with prompt $v_2$.

We also define the syntax of three contexts. Persistent contexts $\Gamma$ map persistent (unrestricted) variables to types. Affine contexts $\Delta$ map affine variables to types. Input contexts $\Phi$ contain the I/O factor names which may be used in an expression.

### Syntactic Sugar

We define some syntactic sugar which we will occasionally use for clarity in examples. We may use forms which are not limited to “2/3-cps”, since these can easily be defined in our language using let. We also define syntactic sugar for the standard product projection operators:

- $\text{fst}(e) \triangleq \text{let} a = e$ in $\text{let} b \otimes c = a$ in $b$
- $\text{snd}(e) \triangleq \text{let} a = e$ in $\text{let} b \otimes c = a$ in $c$

Finally, we will refer to (and use standard infix notation for) operations on natural numbers, such as addition, which can be defined using primitive recursion.

2.2 Statics

The type system of $\lambda^I$ is an affine type system inspired by, and borrowing notation from, computational interpretations and judgmental formulations of linear logic [2, 8, 38]. Our typing judgment, $\Gamma; \Delta \vdash e : \tau$, indicates that under affine context $\Delta$ and persistent context $\Gamma$, $e$ is well-typed with type $\tau$ and may contain I/O factor names drawn from input context $\Phi$. The rules for this judgment are given in Figure 2. The rules for variables, let, affine implication ($\to$), multiplicative product ($\otimes$), and “of course” ($!$) are largely standard. In rules with multiple subexpressions, the affine context is generally split between subexpressions to reflect the fact that affine variables may not be duplicated. All subexpressions are typed using the same persistent context, since persistent variables may be freely duplicated. It may seem surprising that all subexpressions are also typed using the same input context. This is because the affinity restriction governs only variables and not I/O factor names. Since I/O factor names are not present in source programs and arise only through substitution for affine variables, they will not be duplicated. The (eftr-I) rule states that an I/O factor can be referred to by name as long as the name is contained in $\Phi$. The rule (eftr-E) states that an I/O factor can be read, producing a (non-affine) natural number and a new I/O factor. Internal factors are coinductively
generated as in the rule (frt-I). As described above, an internal factor consists of some state of arbitrary type \((v:\tau)\), and a method of generating a (persistent) natural number and a new state given the current state (of type \(\tau\), affine) and a prompt (persistent, of type nat). The generator (body) of the factor is typed under an affine context containing only \(s\), since it may be executed multiple times and therefore must not capture affine variables. An internal factor can then simply be queried with a natural number to produce a (persistent) natural number and a new internal factor, as in rule (frt-E). The primitive recursion operator is typed using rule (nat-E). The base case and the inductive case must be typed with the same type \(\tau\). Like the generator of an internal factor, the inductive case must not capture any affine variables.

### 2.3 Dynamics

The dynamic semantics of \(\lambda^i\) are defined in Figure 3. The transition judgment is of the form \(e \mapsto e'\). The input map \(I\) maps each I/O factor name to a free choice sequence, a predetermined infinite sequence of natural numbers whose values cannot be predicted by the program and aren’t necessarily generated by any computable function or law. Since the free choice sequence factors out the nondeterminism of interaction, evaluation with a fixed \(I\) is deterministic. In developing the metatheory of Section 3, most of our results will quantify over all maps \(I\) whose domains contain the free I/O factor names of the expression, and therefore over all possible resolutions of the nondeterminism.

We briefly describe the rules. The primitive recursion operator \(\text{rec}(n; v; x.e)\) recons on \(n\). If \(n = 0\), it takes the base case \(v\). Otherwise, it recons on \(n - 1\) and then substitutes this result for \(x\) in \(e\) using standard capture-avoiding substitution. Application substitutes the argument into the body of the function. A product pattern match substitutes the components of the pair for the two arguments of the body. In all of the above cases, the variables are affine and occur at most once in the body. By contrast, let \(x = v\) in \(e\) substitutes the unrestricted value \(v\) for the unrestricted variable \(x\) in the body \(e\). Querying an I/O factor with \(\text{query}((i, n))\) takes the \(n\)th element of the free choice sequence \(i\) and returns a pair consisting of this value and the new I/O factor, which contains the same name and the cursor advanced by one. Querying an internal factor with internal state \(v_1\) and prompt \(v_2\) substitutes these values for \(s\) and \(p\) respectively in the factor’s body \(e\). When this evaluates to a pair, the first component will be the response and the second component will be the new internal state. At this point, a new pair is returned, with the first component being the response and the second being a new factor created from the new state and the same body. The only form that needs to evaluate a subexpression is \(let\), which evaluates \(e_1\) until it is a value, and then substitutes it into the body.

### 3. Metatheory

The goal of this section is to develop a logical relation interpretation of the semantics of \(\lambda^i\). This interpretation allows us to prove properties of the language and also to derive equations between terms in \(\lambda^i\). Since these terms can possibly perform interaction, these equations allow us to reason about the behavior of programs in the presence of interactive effects. As described in the introduction, our metatheory quantifies over all assignments of free choice sequences for external factors, resulting in the assumption of unpredictable, independent input sources.

We define several terms describing relations over expressions. A well-typed relation is parameterized by the type and may relate open terms. If two expressions \(e_1\) and \(e_2\) typed at \(\tau\) under contexts \(\Gamma, \Delta\) and \(\Phi\) are related by \(R\), this will be denoted \(\Gamma; \Phi \vdash e_1 \to e_2 : \tau\). A well-typed relation is consistent if it never relates terms of type nat that evaluate to unequal values. It is compatible if it is preserved by all expression-forming constructs of the language.

**Definition 1** (Consistency). A well-typed relation \(R\) is consistent if \(\Gamma; \Phi \vdash e \to e' : \tau\) : nat implies that, for all \(I : \Phi \to (\Pi, \Pi)\), there exists some \(n \in \mathbb{N}\) such that \(e \to^* n\) and \(e' \to^* n\).

**Definition 2** (Compatibility). A relation \(R\) is compatible if:

(i) \(\Gamma, x : \tau; \Delta \vdash f x : \tau x : \tau\)

(ii) \(\Gamma, \Delta; a : \tau \vdash \Phi a : \tau\)

(iii) For all \(n \in \mathbb{N}\), \(\Gamma, \Delta ; n \vdash R n : \text{nat}\)

(iv) If \(\Gamma, \Delta; a : \tau \vdash e \to e' : \tau'\), then \(\Gamma, \Delta \vdash \lambda a : \tau.f a : \tau. e\)

(v) If \(\Gamma, \Delta \vdash v_1 : \tau_1 \text{ and } \Gamma, \Delta \vdash v_2 : \tau_2\), then \(\Gamma, \Delta, \Delta_1 \vdash v_1 \otimes v_2 : \tau_1 \otimes \tau_2\)

(vi) If \(\Gamma, \Delta \vdash v R v' : \tau\), then \(\Gamma, \Delta \vdash v' R v : \tau\)

(vii) For all \(i \in \Phi \text{ and } n \in \mathbb{N}\), \(\Gamma, \Delta \vdash (i, n) : (i, n) : \text{efr}\)

(viii) If \(\Gamma, \Delta \vdash v R v' : \tau \text{ and } \Gamma, \Phi, p : \text{nat} s : \tau \vdash e R e' : \text{nat} \otimes \text{frt}\), then \(\Gamma, \Delta \vdash \text{ftr}[\tau] [\tau] v R v' [\tau] v R v' [\tau] v R v' [\tau]\).

(ix) If \(\Gamma, \Delta \vdash v_1 : \tau' \text{ and } \Gamma, \Delta \vdash v_2 : \tau\), then \(\Gamma, \Delta \vdash \text{rec}(v_1; v_2; a.e) : \text{rec}(v_1; v_2; a.e) : \tau\).

(x) If \(\Gamma, \Delta \vdash v_1 \tau_0 v_2 : \tau \to \tau'\), then \(\Gamma, \Delta, \Delta_1 \vdash v_1 v_2 \tau_0 v_2 : \tau'\)

(xi) If \(\Gamma, \Delta \vdash v R v' : \tau_1 \otimes \tau_2\), then \(\Gamma, \Delta, \Delta_2 \vdash a : \tau_1, b : \tau_2 : \tau \to \tau'\), then \(\Gamma, \Delta, \Delta_2 \vdash \text{let } a \otimes b = v e R e \text{ let } a \otimes b = v' e' : \tau'\)

(xii) If \(\Gamma, \Delta \vdash v R v' : \tau\), then \(\Gamma, \Delta \vdash \text{let } x = v \text{ in } e R \text{ let } x = v' : \tau'\)

(xiii) If \(\Gamma, \Delta \vdash v_1 R v_1' \text{ and } \Gamma, \Delta \vdash e R : \tau\), then \(\Gamma, \Delta, \Delta_2 \vdash \text{let } v_1 R v_1' \text{ in } \text{let } x = v \text{ in } e R \text{ let } x = v' : \tau'\)

(xiv) If \(\Gamma, \Delta \vdash v_1 R v_1' \text{ and } \Gamma, \Delta \vdash e R : \tau\), then \(\Gamma, \Delta, \Delta_2 \vdash \text{query} v_1 R v_1' \text{ query } v_1' R v_1 \text{ let } a \otimes b = v e R e' : \tau'\)

(xv) If \(\Gamma, \Delta \vdash e_1 : \tau\), then \(\Gamma, \Delta, \Delta_2 \vdash \text{let } a = e_1 \text{ in } \text{let } a = e'_1 : e_2 \text{ R } e'_2 : \tau'\)

Figure 3. Dynamics of \(\lambda^i\)
It is straightforward to see that two contextually equivalent terms cannot be differentiated by composition with any other expressions in the language. To formally define contextual equivalence, we appeal to the standard notion of evaluation contexts. An evaluation context $C[\cdot]$ is an arbitrary $\lambda^I$ expression which contains the placeholder $\cdot$ in place of any one subexpression. The notation $C[e]$ denotes the context $C[\cdot]$ where the "hole", $\cdot$, is replaced by the expression $e$.

**Definition 3 (Contextual Equivalence).** We say $e$ and $e'$ are contextually equivalent, written $e =_{ctx} e'$, iff, for all contexts $C[\cdot]$ such that $\vdash \Gamma \vdash \Phi C[e] : \tau$ and $\vdash \Gamma \vdash \Phi C[e'] : \tau$ and all $(I : \Phi \rightarrow (\mathbb{N} \rightarrow \mathbb{N}))$, it is the case that, for some $n \in \mathbb{N}$, $C[e] \rightarrow^*_I n$ and $C[e'] \rightarrow^*_I n$.

**Lemma 1.** Contextual equivalence, as defined in Definition 3, is an equivalence relation. It is consistent and compatible. Furthermore, it is the largest consistent, compatible relation in the sense that if $R$ is a consistent, compatible relation and $\Gamma; \Delta \vdash e : \tau$, then $e =_{ctx} e'$.

**Proof.** It is straightforward to see that $=_{ctx}$ is an equivalence relation.

To show that it is consistent, let $e$ and $e'$ be such that $\vdash \Gamma \vdash \Phi e : \tau$ and $\vdash \Gamma \vdash \Phi e' : \tau$. If $e =_{ctx} e'$, then for all $I$, (taking $C[\cdot]$ to be the trivial context) $e \rightarrow^*_I n$ and $e' \rightarrow^*_I n$.

Showing that $=_{ctx}$ is compatible proceeds by induction on the cases of Definition 2. The cases with no premises hold by reflexivity. As an example of a case with one premise, consider (iv). Let $C[\cdot]$ be a context. Note that $C[\lambda x . \tau \vdash e]$ is itself a context, so the result follows since $e =_{ctx} e'$. As an example of a case with two or more premises, consider (v). By the above argument, we have that $v_1 \otimes v_2 =_{ctx} v'_1 \otimes v_2$ and $v_1' \otimes v_2 =_{ctx} v'_1 \otimes v'_2$. The result follows from transitivity.

Let $R$ be a consistent, compatible relation. Suppose $\Gamma; \Delta \vdash \Phi e : \tau$ and let $C[\cdot]$ be a context such that $\vdash \Gamma \vdash \Phi C[e] : \tau$ and $\vdash \Gamma \vdash \Phi C[e'] : \tau$. Since $C[e]$ and $C[e']$ are built by composing expressions around $e$ and $e'$, respectively, compatibility of $R$ gives that $\vdash \Gamma \vdash \Phi C[e] R C[e'] : \tau$. Next, consistency of $R$ ensures that, for some $n$, $C[e] \rightarrow^*_I n$ and $C[e'] \rightarrow^*_I n$, so by definition, $e =_{ctx} e'$.

We now define the logical relation. The definition follows the semantics of Ahmed et al. [3] for a language that enforces linear use of references to store locations. The elements of the relation are pairs $(\Phi, e)$ of an input context and an expression. The input context $\Phi$ may be thought of as the set of I/O factors that are exclusively "owned" by $e$. This ownership represents semantically the fact that, in our affine type system, I/O factors may not be aliased. Because weakening is allowed in our system, $e$ need not actually make use of all of the I/O factors in $\Phi$.

We proceed to define the relation in two parts: a relation $E$ on expressions and a relation $V$ on values. Both are parametrized by the type. These relations are defined in Figure 4. The expression relation $E$ simply requires that both expressions evaluate to values that are related at the appropriate type. The value relation is defined inductively on the structure of $\tau$. At type nat, the value relation simply requires that the values be equal as natural numbers. At type str, $\psi$ relates an I/O factor to itself if it is contained in both input contexts. The relations at type $\rightarrow \psi$ and $\tau_1 \otimes \tau_2$ are defined inductively on the subexpressions in a straightforward way. The most complex definitions are those for function types and internal factors. Intuitively, two functions $\lambda x : \tau \vdash e$ and $\lambda x : \tau \vdash e'$ are related if for all related arguments $v$ and $v'$ of the appropriate type, the function bodies are related after the application is performed. The complication is that the arguments may themselves "own" sets of I/O factors, $\Phi_v$ and $\Phi_{e'}$, respectively. In order for the application to be well-typed, it must be the case that the argument and the function do not share I/O factors, so we require that their input contexts be disjoint (indicated by the judgment $\Phi \not\equiv \Phi_v$ defined).

The definition of the relation at internal factor type is based on the observation that internal factors, like values of existential type, are parametrized by a type (the type of the internal state) which is abstract to the outside world. Two internal factors with states of type $\tau$ and $\tau'$ are related if there exists a relation $R$ relating values of types $\tau$ and $\tau'$ such that the states of the two factors are related by $R$ and the factor bodies take identical prompts and $R$-related states to identical responses and $R$-related new states.
We now show that the expression and value relations coincide on values.

**Lemma 4.** $E_r$ and $V_r$ coincide on values: For values $v, v' : \tau$, $((\Phi, v), (\Phi', v')) \in E_r \Leftrightarrow ((\Phi, v), (\Phi', v')) \in V_r$.

**Proof.** Suppose $((\Phi, v), (\Phi', v')) \in E_r$. If $v \mapsto v_j$, then $v = v_j$ (and similarly for $v'$), so we have $((\Phi, v), (\Phi', v')) \in V_r$.

Suppose $((\Phi, v), (\Phi', v')) \in V_r$. We have that $v \mapsto v$ and $v' \mapsto v'$, so $((\Phi, v), (\Phi', v')) \in E_r$. \(\Box\)

Before proceeding, we must make $E$ into a well-typed relation by extending it to operate on open terms.

**Definition 4.** We define two operations that produce closing substitutions from contexts. Each produces a pair of closing substitutions whose values are pointwise related by $\forall \gamma$. The operator for affine contexts also generates input contexts containing any free I/O factor names in the values.

$G() \triangleq \{ (\emptyset, 0) \}$

$G(\Gamma, x : \tau) \triangleq \{ (\gamma, \gamma') \in G(\Gamma) \land ((\{\}, v), (\{\gamma, \gamma'\})) \in V_r \}$

$D() \triangleq \{ (\emptyset, 0, 0, 0) \}$

$D(\Delta, a : \tau) \triangleq \{ (\Phi \uplus \Phi', \delta[a \mapsto v], \Phi' \uplus \Phi', \delta'[a \mapsto v']) | (\Phi, \delta, \Phi', \delta') \in D(\Delta) \land ((\Phi, v), (\Phi', v')) \in V_r \}$

Note that if $(\Phi, \delta, \Phi', \delta') \in D(\Delta, a : \tau)$, then there exist $\Phi_1, \Phi_2, \delta_1, \delta_2, \Phi'_1, \delta'_1, \Phi'_2, \delta'_2$ such that $\Phi = \Phi_1 \uplus \Phi_2$ and $\Phi' = \Phi'_1 \uplus \Phi'_2$ and $\delta = \delta_1 \uplus \delta_2$ and $\delta' = \delta'_1 \uplus \delta'_2$.

We define open logical equivalence, $\Gamma; \Delta \vdash_{o} E \vdash \tau : \tau'$, to mean that for all $(\gamma, \gamma') \in G(\Gamma)$ and all $(\Phi, \delta, \Phi', \delta') \in D(\Delta)$, it is the case that

$((\Phi, \delta(a \mapsto v)), (\Phi', \delta'(a \mapsto v'))) \in E_r$

We are now almost ready to show the correspondence between $E$ and $\vdash_{cxt}$. Note that one direction holds trivially by Lemma 1 if we can show that $E$ is consistent and compatible.

**Theorem 1.** The relation $E$ is compatible.
Proof. By cases on Definition 2. The full proof appears in Appendix A. □

Note that compatibility implies reflexivity; this shows the fundamental property of the logical relation $E$.

**Theorem 2 (Fundamental Property).** If $\Gamma; \Delta \vdash e : \tau$, then $\Gamma; \Delta \vdash e \in E e : \tau$.

**Proof.** By induction on the structure of $e$, applying Theorem 1. □

This, in turn, shows that $\lambda^!$ is type-safe and terminating.

**Corollary 1.** If $\Gamma; \Delta \vdash e : \tau$, then for all closing substitutions $\gamma$ and $\delta$ such that $\vdash \gamma(\delta(e)) : \tau$ and all $I : \Phi \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$, there exists $v$ such that $\gamma(\delta(e)) \mapsto I v$.

**Proof.** By Theorem 2, $\Gamma; \Delta \vdash e \in E e : \tau$, and so $((\Phi, \gamma(\delta(e))),(\Phi, \gamma(\delta(e))) ) \in E_{\tau}$. The result follows from the definition of $E$. □

We are now prepared to show the major result of this section, that the relations $E$ and $\equiv_{cte}$ are equivalent.

**Theorem 3.** Let $\Gamma; \Delta \vdash e : \tau$ and let $\Gamma; \Delta \vdash e' : \tau$.

$\Gamma; \Delta \vdash e =_{cte} e' \Rightarrow \Gamma; \Delta \vdash e \in E e' : \tau$

**Proof.** The reverse direction follows from Lemma 1. We know that $E$ is compatible by Theorem 1. It is consistent since if $((\Phi, e),(\Phi', e')) \in E_{\tau}$, then for all $I$, we have $e \mapsto^* v$ and $e' \mapsto^* v'$ and so $((\Phi, v),(\Phi', v')) \in E_{\tau}$, so $v = v'$.

For the forward direction, proceed by induction on the structure of $\tau$. The full proof appears in Appendix B. □

### 3.1 Examples

**Counting Clicks.** We now show an application of Theorem 2. Suppose we have one I/O factor, clicks, which, when (synchronously) read, blocks until the mouse is clicked and then returns the number of times the mouse has been clicked. The internal state of $e$ is a pair of the counter and the current handle to clicks. On the other hand, $e'$ does not query clicks. It simply increments the counter each time it is queried, returning the sequence of natural numbers in order starting at 1. Despite the fact that $e$ performs interaction and $e'$ does not, they have the same external behavior; they both return the sequence of natural numbers when queried. We can make this precise and show that the two are equivalent.

$\begin{align*}
\vdash e & \triangleq \mathsf{frt}[\mathsf{nat} \otimes \mathsf{eftr}] \text{ with } \!0 \otimes (\text{clicks}, 0) \text{ as } s.p. \\
& \text{let } n \in \mathbb{N} = a \text{ in } \\
& (!n') \mapsto n \text{ in } let \cdot \vdash !c' = \mathsf{equry } c \text{ 0 in } \\
& !(n' + 1) \otimes !(n' + 1) \\
\vdash e' & \triangleq \mathsf{frt}[\mathsf{nat}] \text{ with } \!0 \text{ as } n.p. \\
& \text{let } n' \mapsto n \text{ in } !(n' + 1) \\
\end{align*}$

We show $\vdash e \mapsto \cdot (\text{clicks}) e \in E e' : \mathsf{frt}$. To do this, we need to define a relation $R \subseteq \mathsf{Rel}(\mathsf{nat} \otimes \mathsf{eftr}, \mathsf{nat})$ and then show that it meets the conditions specified in the definition of $V_{\eta}$. Let

$R = \{(n \otimes (\text{clicks}, n), n) \mid n, n_1 \in \mathbb{N}\}$. Intuitively, the states of $e$ and $e'$ are related if their counters are equal. Let $\alpha$ be a free choice sequence and let $I \triangleq \{\text{clicks} \mapsto \alpha\}$. Let $(v, v') \in R$. We have $v = n \otimes (\text{clicks}, n_1)$ and $v' = n + 1 \otimes \cdot (\text{clicks}) v : \mathsf{nat} \otimes \mathsf{eftr}$ and $\cdot \vdash \cdot (\text{clicks}) v' : \mathsf{nat}$. We have

$\begin{align*}
\vdash & n \otimes c = v \text{ in } \\
& \text{let } !n' \mapsto n \text{ in } let \cdot \vdash !c' = \mathsf{equry } c \text{ 0 in } \\
& !(n' + 1) \otimes !(n' + 1) !c' \\
\Rightarrow & i \mapsto i !n + 1 \otimes \cdot (\text{clicks}) v : \mathsf{nat} \otimes \mathsf{eftr} \\
\vdash & n + 1 \otimes n + 1 \mapsto i !n + 1 \otimes n + 1 \\
\end{align*}$

Finally, we have $(0 \otimes \text{clicks}, 0) \in R$, so the two factors are equivalent.

The above result may seem quite surprising. Since the second program doesn’t perform interaction, it cannot possibly be counting mouse clicks, so what does it mean to say that a program that counts clicks is equivalent to one that doesn’t? The answer to this question lies in a more careful analysis of what the two programs do. Recall that $\lambda^!$ lives in a world with no global coordinate system for time. Thus, the two programs can only be compared for the values that they output, not when they output these values. In this setting, a program like $e$, which only reports the number of clicks seen so far in some entirely unspecified time period, is not particularly useful. We know a priori what its output will be: the sequence $2, 3, 4, 3$, and so on. Instead, what we are more likely interested in is the number of clicks in a specified period of time. To make this explicit, we could introduce a second I/O factor, time, which at each query reports the amount of time that has passed since the previous query (in some unspecified units). A non-negative value will always be reported since responses are drawn from $\mathbb{N}$. The factor $e_t$, defined below, reports the number of clicks seen until time 100, at which point it remains constant (the result may be off by one, since the resolution of time cannot be controlled or predicted). The factor $e'_t$ simply counts until the time reaches 100, and does not sample clicks. In defining $e_t$ and $e'_t$, we assume (definable) syntax for 3- and 4-tuples and pattern matching, as well as a conditional branch on a comparison of two natural numbers.

$\begin{align*}
e_t & \triangleq \mathsf{frt}[\mathsf{nat} \otimes \mathsf{nat} \otimes \mathsf{eftr} \otimes \mathsf{eftr}] \text{ with } \\
& !0 \otimes !0 \otimes (\text{clicks}, 0) \otimes (\text{time}, 0) \text{ as } \\
& !(\mathsf{ln} \otimes t \otimes f_1 \otimes f_2).p. \\
& \text{let } !t' \otimes f_2 \mapsto \mathsf{equry } f_2 \text{ 0 in } \\
& \text{if } t + t' > 100 \text{ then } \\
& !\mathsf{ln} \otimes !(\mathsf{ln} \otimes !(t + t') \otimes f_1 \otimes f_2) \text{ else } let \cdot \vdash f_1' = \mathsf{equry } f_1 \text{ 0 in } \\
& !(n' + 1) \otimes !(n' + 1) \otimes !(t + t') \otimes f_1 \otimes f_2' \\
\vdash e'_t & \triangleq \mathsf{frt}[\mathsf{nat} \otimes \mathsf{nat} \otimes \mathsf{eftr}] \text{ with } \\
& !0 \otimes !0 \otimes (\text{time}, 0) \text{ as } \\
& !(\mathsf{ln} \otimes t \otimes f).p. \\
& \text{let } !t' \otimes f' \mapsto \mathsf{equry } f \text{ 0 in } \\
& \text{if } t + t' > 100 \text{ then } \\
& !\mathsf{ln} \otimes !(\mathsf{ln} \otimes !(t + t') \otimes f') \text{ else } let \cdot \vdash f' \mapsto \mathsf{equry } f' \text{ 0 in } \\
& !(n' + 1) \otimes !(n' + 1) \otimes !(t + t') \otimes f' \\
\end{align*}$

The careful reader will note that the same argument used above to show $e$ and $e'$ equivalent appears to apply to $e_t$ and $e'_t$: the result of clicks is not used by $e_t$ and so, other than the time it takes them to return, the queries of clicks should not affect the program’s behavior. This paradoxical result is explained by
showing our assumptions. We can no longer ignore the time that the queries take to return, because time is itself another factor used in the code. Because the amount of time a query to clicks blocks affects the next value returned by a query to time, the factors clicks and time cannot be viewed as independent! This example thus shows both the power and limitations of assuming an unpredictable world.

**Interchanging queries** The next example shows that queries to two independent I/O factors can be commuted without changing the result. Consider a two-player game played over a network. Two factors that are queried on each iteration of the game loop are the factor representing the keyboard, which is used to determine if the local user is attempting to move, and the factor representing a network socket carrying key presses from the remote player. Since the two players’ behaviors should be independent (other than coordination through external channels), it should be the case that the order in which the two factors are queried is irrelevant. We show this formally by defining two expressions, $e$ and $e'$, which query the two I/O factors in different orders and produce a pair of the local keyboard response and the network response (both expressions produce the pair in this order so that the outputs will be equal).

$$
eq \text{let } r_1 \otimes . = \text{query (keyboard, 0)} \text{ in}\neq \text{let } r_2 \otimes . = \text{query (network, 0)} \text{ in}\neq r_1 \otimes r_2$$

$$
e' \text{ let } r_2 \otimes . = \text{query (network, 0)} \text{ in}\neq \text{let } r_1 \otimes . = \text{query (keyboard, 0)} \text{ in}\neq r_1 \otimes r_2$$

We show that $e$ and $e'$ are equivalent:

$$e \Rightarrow \lambda !\text{keyboard}(0) \otimes !\text{network}(0)$$

and

$$e' \Rightarrow \lambda !\text{keyboard}(0) \otimes !\text{network}(0)$$

By Theorem 2 and Lemma 4, we have

$$((\text{keyboard, network}), !\text{keyboard}(0) \otimes !\text{network}(0)),
((\text{keyboard, network}), !\text{keyboard}(0) \otimes !\text{network}(0)))
\in \text{Eval} \otimes \text{Eval}$$

so, by repeated applications of Lemma 3, we can show

$$\vdash \lambda (\text{keyboard, network}) \in E e' : \text{Eval} \otimes \text{Eval}$$

While commuting two queries may seem like a trivial example, its motivation is quite relevant. One important application of showing that effects are benign is parallelism: two operations that have no externally visible effects may be easily parallelized without the possibility of data races or other concurrency bugs. If we extend our interactive language with support for concurrency (as our implementation, described in Section 4, does), we may wish to write a program in which two threads handle different interactions. The argument that these threads may safely be run in parallel comes down to an argument that their effects (queries) may be safely interleaved, a result much like the one shown above.

### 4. Implementation and Applications

We briefly describe an implementation of $\lambda^3$ and applications that we have developed. The implementation shows that the proposed techniques can be realized in practice and that they are compatible with the full array of features in a full-fledged programming language. The example applications considered show that the proposed techniques are expressive and flexible, allowing the implementation of a relatively broad range of applications involving a variety of modes of interaction. The applications show that the language is able to provide fine-grained control over sampling of inputs while also retaining expressive powers of functional programming.

#### 4.1 Implementation

The proposed language $\lambda^3$ can probably be implemented in any functional language; we implemented it as an OCaml library. Figure 5 shows the core of the interface for the library, in a style similar to an ML module definition. The implementation follows the syntax and semantics described in Section 2, with a few differences:

- OCaml’s type system is not powerful enough to enforce the affine type system of $\lambda^3$. We therefore check dynamically that no factor is used more than once by furnishing the runtime representations of I/O factors with additional facilities that raise an exception when they are used more than once.

- Our implementation includes a set of standard libraries with built-in I/O factors for interacting with the console and network as well as graphics and media libraries. It also includes a primitive for defining new external factors and deriving multiple factors from one. The standard library also includes many useful functions, such as map, for building factors.

- Prompts and responses of (internal and I/O) factors are not constrained to be integers, but may be any type, including functions.

- The implementation provides a mechanism for allowing concurrent querying of factors.

We describe the interface and the use of the OCaml library by showing code excerpts from a program we have developed using the library: an arcade-style game (similar to “Breakout”) in which two players, at separate computers connected via a network, collaborate to complete the objective. The players can also communicate using a chat window. Figure 6 shows a snapshot from the implementation of the game. Many forms of interaction are present here: the game window must, at regular intervals, sample the keyboard

---

Figure 5. The interface for the library implementation.
(for the local user’s commands), send the user’s action over the network and wait for the remote user’s action on the network. At the same time, the chat window must respond to lines entered on standard input and chat messages received over the network. Since these events will be relatively infrequent, the chat window should be able to block and awake when an event occurs without blocking the game window.

**Generators and defining factors.** Programmers can create a factor by supplying a generator to the function ftr, or ioftr for I/O factors. A generator is a function that takes a prompt and produces a response and a continuation. For example, the I/O factor stdin, when prompted with a unit value, returns a line from standard input. It is defined using a generator that calls the OCaml function read_line and returns its result along with a continuation created by a recursive use of the generator.

```ocaml
let std_in : (unit, string) ftr =  
  let rec std_in_gen () = (read_line (), ftr std_in_gen)  
  in ioftr std_in_gen
```

We can similarly define network.resp, a function which, when given a network socket, produces an I/O factor that, when queried with a prompt consisting of a string s and an integer n, sends s over the network and listens for n bytes in response, which it returns.

```ocaml
let rec network_resp sock =  
  ioftr (fun (msg, n) ->  
    ignore (sock_send sock msg);  
    (sock_recv sock n, network_resp sock))
```

**Querying factors.** The synchronous query function, as in our formal development, queries a factor with a given prompt, waits for the response and continuation, and returns them when they become available. In our multiplayer game, the main loop uses the I/O factor network.resp to inform the other player of this player’s actions. To this end, it sends a byte to the other player indicating the key currently pressed and waits for a byte in response. Since it is important for both players to operate in synchrony, we use query (rather than the asynchronous query discussed later).

```ocaml
let (my_key, key_presses') = (get currently pressed key*) in  
let query network_resp (String.make 1 my_key, 1) in
```

**Concurrency: asynchronous queries** The above example considered use synchronous queries, which block until the queried factor returns. Such behavior can be desirable—even necessary—in many cases. But sometimes it can be important to proceed without waiting for a response. For example, in the above code, we wish to proceed executing the game loop even if the user does not press a key to be assigned to my_key, but we have access only to key_presses, an I/O factor that blocks until a key is pressed.

We therefore provide an asynchronous query operation, aquery, which takes the same arguments as query. This operation spawns a lightweight thread in which the factor is queried with the prompt. If the response is not available immediately, a future factor is returned. A future factor is a handle on the concurrently running computation. Like factors, future factors can be queried synchronously or asynchronously. However, since a future factor represents a running computation that has already been given a prompt, it cannot accept a new prompt. Thus, both query and aquery on future factors accept only unit prompts. Calling query on a future factor blocks until the original response is ready. Calling aquery on a future factor polls the computation. It will return the response and continuation if available, or the original future factor otherwise.

The current key of the above example can be obtained by asynchronously querying key_presses, which will return immediately even if no key is pressed (indicated by the constructor Later), in which case we fill in a space and continue.

```ocaml
let (my_key, key_presses') =  
  match aquery key_presses () with  
  | Later (key_presses' →  
    | no key is pressed*) (' ', key_presses')  
  in ...```

**Asynchronously joining factors.** The primitives described so far turn out to be sufficiently powerful to express a rich class of interactive computations. In our empirical evaluation, we found a particular operation for asynchronously joining (merging) two factors into one, interleaving responses as they are available, to be very helpful. When queried, the joined factor asynchronously queries the two component factors and returns the first of the two responses. Even though such an operation is expressible based on the primitives described thus far, the resulting implementation is rather inefficient. We therefore include in our core library the primitive ajoin for joining two factors asynchronously.

In the multiplayer game, we create a factor for running the game loop, another for writing chat messages from standard input over the network, and yet another to listen for incoming chat messages. Since these three operations should be performed concurrently, the implementation joins them asynchronously as follows:

```ocaml
ajoin gameloop  
  (ajoin (output_on_network  
    (map (fun s → s"\n") stdin)))  
  (output_on_stdout  
    (map (fun s → "[other player] ""s""\n")  
      (input_lines in_channel))))
```

### 4.2 Example Applications

We briefly describe some of the more interesting applications that we have implemented. Many examples exploit the full range of OCaml features, such as mutable references and foreign function interface (FFI).

**Multiplayer Game.** We implemented a multiplayer arcade game as described in the previous section. This code uses the graphics and network factors of our standard library to display and react to GUI events, and communicate between players over the network. The features of $\lambda$, for example controlling sampling, are useful for this example. The game consists of 259 lines of OCaml code.

![Figure 6](image-url) Screenshots of a simplified multiplayer game: the main window (top) and the terminal chat window (bottom).
**Tax “Factor”**. As an example console applications, we implemented an application guiding the user through completing the IRS W-4 form. The applications prompts the user for information to calculate the user’s withholding allowances, which are expressed as a simple mathematical formula. The application involves non-trivial data and control dependencies on the responses of the user but still can be written in a natural, functional style using our library.

**GUI Calculator**. To test how well the library scales to many factors, we developed a simple GUI calculator, where each button consists of two factors, one which determines if the mouse is hovering over the button and one which determines if the mouse is clicking on the button, resulting in 38 factors to handle button events, all sampled asynchronously. The arithmetic logic of the calculator is a factor which combines the values of the button-click factors and changes its state according to which buttons are pressed, producing numbers to be displayed on the screen. A separate factor draws the buttons, using information from the mouse-hover factors to shade a button if the mouse is over it. A screenshot is shown in Figure 7. The calculator involves 214 lines of code.

**Physics Simulation**. To test how well our techniques handle continuous calculations combined with complex user interaction, we developed a physics simulation involving two balls moving in a bounded 2D space on 2D trajectories. Each ball can be “free” or “caught.” In the “free” mode, the balls move according to the laws of physics under gravitational force, bouncing off of the walls of the box and each other. The user can “catch” a ball with the mouse and drag it. As a ball is dragged, its factor updates its position by sampling the mouse, and also computes the velocity by continuously taking the derivative of the mouse position. When the ball is released, the simulation resumes with the ball moving with the velocity at which it was released. This allows the user to catch and toss the balls around the screen.

For increased accuracy and efficiency, we vary the sampling frequency of the system time roughly linearly with the velocity of the balls (higher velocities lead to more sampling). By sampling infrequently when the balls are moving slowly, this policy ensures efficiency. By sampling 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. This example demonstrates the power of allowing sampling frequency to be determined dynamically at run time. This example totals 224 lines of code.

**Screaming Music**. We implement a streaming music server and client. The server opens a music file stored locally and streams it over the network using the network factors of our standard library. The client reads bytes from the network and plays the music, but to ensure that temporary network delays will not affect playback, a fixed amount of data is buffered on the client side. To make this work, two factors work asynchronously, sharing a mutable buffer whose implementation will be described below. One factor requests bytes from the network and writes them onto the end of the buffer. The other takes a number \( n \) as a prompt, removes \( n \) bytes from the start of the buffer and returns them. A third factor reads console input. These three factors are asynchronously joined into a factor that continuously fills the buffer, pauses and resumes playback based on commands typed into the console, and returns requested bytes from the buffer when the music is playing. This factor is then passed to a foreign function that allows the Simple and Fast Multimedia Library\(^1\) to use the factor as an input stream to play the music.

We use I/O factors to manipulate the shared mutable buffer at a high level without the need for explicit synchronization. We use OCaml references to define a function `ref_ftr` which creates an I/O factor wrapping a new reference, initialized to a given value. Querying the factor performs an atomic read and update on the reference, as directed by a function passed as the prompt. Locking is handled by the code of the I/O factor. Splitting this factor (using the primitive for making multiple factors from one) corresponds to sharing the underlying mutable state, which can be done safely because operations are atomic. Operations performed concurrently may be interleaved arbitrarily, but the shared state will never be corrupted by partial reads or writes.

The (combined) OCaml code for the server and client consists of 163 lines of code. The function `ref_ftr` is an additional 11 lines of code, and the \( \lambda \) sound libraries developed for this example consist of 160 lines of C++ code and 5 lines of OCaml code to perform the FFI.

**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` performs interaction with the user. To make sure that keystrokes will be captured while simultaneously monitoring for signals indicating that background processes have terminated, we asynchronously join standard input with the factor of signals and query this in a loop:

```ocaml
1 let rec input_loop ... children input_and_signals =
2     match query input_and_signals (i, o) with
3     | (Left c, is') → process_char c ... is'
4     | (Right s, is') →
5     let c' =
6     if s = Sys.sigchild then reap children
7 else children in
8 (None, ftr (input_loop ... c' aj'))
```

Low-level system operations are encapsulated within separate functions, while much of the code handles high-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 functional style of our library. The shell consists of approximately 400 lines of OCaml code.

**Yampa**. Yampa is a Haskell library based on Arrowized FRP [29], in which interactive programs are built up by combining signal functions, functions that transform signals. We have embedded a substantial subset of Yampa in \( \lambda \), and used the embedding to implement the tailgating detection example of Nilsson et al. [29]. Our implementations of Yampa and of the tailgating example consist of approximately 300 and 120 lines of code respectively. When implemented directly in our implementation of \( \lambda \), the tailgating example is approximately 75 lines of code.

**Elm**. We also implemented the calculus called ELem that constitutes the core of the Elm language [12] for functional reactive programming. Unlike other FRP languages, ELem allows a special form of asynchrony in which long-running signal-processing functions can be run asynchronously at the top level. The widely available Elm implementation, a compiler that targets Javascript,

\(^1\)http://sfml-dev.org/
does not implement fully the asynchronous features of FEIm because Javascript has limited concurrency support. We implemented all features of FEIm using our library implementation of \( \lambda ^{x} \) in less than 100 lines of code.

**Futures.** We implemented in our library an encoding of futures [15, 17], a classic technique for parallelism and concurrency. This shows that the concurrency features of the library implementation are quite powerful. In the encoding, creation of a future maps to creation of a factor (with unit prompt type) followed by an asynchronous query, and forcing a future maps to a blocking query. To represent futures faithfully, the representation also holds an already-computed value in the event that the asynchronous query returns a value immediately.

5. Related Work

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

5.1 Event Driven Programming

Event-driven programming can lead to responsive interactive programs, but, as documented in the literature, this efficiency comes at the cost of a low-level style of programming (e.g., [9, 13, 16]). Some prior research therefore proposed improvements to event-driven programming that can alleviate some of the complexities [9, 16]. Notably, the responders of Chin and Millstein [9] share the idea of providing an abstraction that combines two-way communication with state. Responders are methods of classes in an object-oriented language. In contrast, our main abstraction for interaction, factors, are first-class values in a higher-order functional language. In addition, the work on responders does not consider concurrence or the metatheory of the proposed abstractions, which are important aspects of our work.

The query-response-continuation mechanisms of factors are in some ways similar to coroutines [23] but they are also different from co-routines, because the call structure is not symmetric.

5.2 Functional Reactive Programming

Functional Reactive Programming (FRP) offers language-based abstractions for writing programs that interact with the external world. Elliott and Hudak [14] first introduced functional reactive programming by providing primitives for time-varying values, along with their elegant and powerful denotational semantics. Elliott and Hudak’s proposal turned out to be too powerful to implement safely, disallowing time travel and as a result does not suffer from time and space leak. We therefore do not restrict the use of interactive values: they are, just like any other value, first-class values in a higher order language. We note that the type system does not rule out constructing dynamic and rich data structures, which can be necessary to implement sophisticated algorithms, but only prevents inadvertent memoization of interactive state, i.e., space and time leaks. For example, a program is allowed to explicitly construct a list of previous elements of a factor if so desired.

FRP implementations typically couple the time-mapped stream implementation of interactive values with a synchronous evaluation strategy. Streams are sampled at periodic intervals and the computation is updated accordingly. This leads to questions about the frequency of sampling, which is difficult to determine because it depends on the application [11, 12]. Furthermore, as shown by Wan and Hudak [39], such synchronous implementation are consistent with the denotational semantics of FRP only when the sampling rate approaches zero, which is not practically feasible because programs take time to execute. The Elm language made important progress on this problem by allowing certain computations to run asynchronously. This is a significant advance, but Elm’s model of computation remains fundamentally synchronous: asynchronous computations operate in the context of a globally synchronous model. In contrast, in our approach, evaluation is fully asynchronous. An interactive program samples interactive values as needed by the demands of the application. This is essential in allowing finer-grained control over sampling to ensure that interactive data can be sampled at the right frequency that is demanded by the application. For example, a rarely used network connection or a slowly moving object can be sampled less frequently, whereas a pointing device or a fast moving object in an interactive game may be sampled frequently. We use these techniques in the examples that we implemented to improve precision of calculated results and to improve efficiency and responsiveness (e.g., interactive physics simulation in Section 4). We note that in our approach, the programmer can implement a global clock or evaluation loop if needed. We implemented such a loop in the our implementation of the Yampa [29] interface (Section 4).

5.3 Metatheory for Effects

There is a substantial body of work based around semantic models of effectful languages and language features in order to allow reasoning about programs in the presence of effects. Pitts and Stark [31] developed a semantics for an ML-like language in which functions may allocate local state. Their approach is based around a logical relation parametrized by a relation on stores, which they use to show, for example, that memoization of a function is a benign effect that does not change the external behavior of the function. The development of our metatheory is inspired by theirs. Building on the work of Pitts and Stark, Johann et al. [22] developed a similar metatheory for a language with a very general notion of algebraic effects. As one of several running examples, they consider a language with input and output channels. Their semantics uses the elegant notion of computation trees [32] to model the nondeterminism of general algebraic effects. The metatheoretic results are parametrized over a relation on computation trees which can be used to capture many interesting properties of an evaluation. In
developing a metatheory for interactive effects (as opposed to more general notions of effects), we instead factor out the nondeterminism using choice sequences to model interaction.

More recent work (e.g. [5, 6]) has used the technique of proof-relevant logical relations to develop richer equational theories for languages with effects such as name generation and store effects. This theory can be used to validate certain effect-based program transformations, such as commuting independent writes and removing writes whose values are not read, which may be viewed as analogous to some of the transformations we are interested in for interactive applications.

5.4 Process Calculi and Concurrency

Process and concurrency calculi such as CSP [20], π-calculus [27, 28] and actors [19], have been proposed to model computations involving interaction between many processes via some form of communication medium, typically called channels. Many variants of these calculi have been studied (several surveys exist, e.g. [4, 18]), and several programming languages such as Concurrent ML [34] and Pict [30] have been designed based on these calculi. Our factors are similar to Plotkin’s resumptions [33], which have an isomorphic type, but are used to model processes and non-determinism.

Process calculi are powerful and can in principle be used to express interactive computation, including the sorts that we consider here. As evidenced by the research on interaction, we feel that interaction is an important problem that deserves to be studied on its own. In this paper, we therefore propose abstractions tailored to the interaction problem by considering the problem of a single process interacting with an external unpredictable world, rather than modeling the interactions among many processes. Such tailored abstractions enable avoiding the full power and the complexity of concurrent programming where such complexity is not needed; instead we build on the conceptually and practically simpler lambda calculus. Tailored abstractions as we develop here are able to offer a simpler interface that avoids the complexity of concurrency and facilitate crucial meta reasoning facilities (Section 3) that enable the programmer to reason about higher order code.

6. Conclusion

We have presented abstractions for writing programs that interact with the external world using a coinductive data type, called a factor, that is a first-class value in a higher-order language. The language uses an affine type system to ensure that interactive programs remain “in sync” with the external world by advancing the “internal clock” every time the language interacts with the external world. The same mechanism prohibits use of interactive values from the past and thus prevents by construction the inadvertent blowup in time and space demands that can otherwise happen interactive programs.

The dynamic semantics of our proposed language uses Brouwer’s free and lawlike choice sequences to assign a precise meaning to interactive values. Free choice sequences correspond to the non-determinism of the external world and allow us to present a meta-theory of the language. The meta-theory enables reasoning about the equational properties of the language by quantifying over all possible instances of choice sequences, effectively enabling us to state and prove equations about programs that hold in worlds of independent free choice sequences (interactions). This theory can make it possible to prove formally and precisely non-trivial relations such as commutativity of interaction under corresponding independence assumptions.

The presented implementation and example applications show that the proposed techniques are practical, allowing the implementation of sophisticated modes of interaction in a high-level functional style. While the core of the implementation closely corresponds to the presented language and semantics, our metatheoretical results make assumptions that do not always hold true in the real world. Specifically, interactive programs whose interactions involve multiple interdependent choice sequences may not benefit from our theory, although components of such applications, such as individual functions, may. Natural future research directions include scaling the metatheory to handle interdependent choice sequences and performing an empirical study to quantify the effectiveness of the techniques, a nontrivial task since the performance of interactive applications is shown to be relatively difficult to quantify.

References

A. Proof of compatibility

Proof. (i) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta)\). We have \( \gamma(\delta(x)) = \gamma(x) \) and \( \gamma'(\delta'(x)) = \gamma'(x) \), so by definition of \( \Gamma \vdash \gamma \sim \gamma' \), \( \Gamma, \Delta \vdash \gamma(\delta(x)) E \gamma'(\delta'(x)) \). 

(ii) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta)\). We have \( \gamma(\delta(a)) = \delta(a) \) and \( \gamma'(\delta'(a)) = \delta'(a) \), so by definition of \( \Gamma \vdash \delta \sim \delta' \), \( \Gamma, \Delta \vdash \gamma(\delta(a)) E \gamma'(\delta'(a)) \).

(iii) By Lemma 4

(iv) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta)\). By assumption, \( \Gamma, \Delta, a : \tau \vdash b \in E e' : \tau' \). Let \((\Phi_4, v)(\Phi_4', v') \in V_\tau\) such that \( \Phi_3 \vdash \Phi_4 \) \& \( \Phi_4 \vdash \Phi_4' \), \( \delta(\delta(a \rightarrow v')(e')) \in E \). Since \( \delta(a \rightarrow v')(e') = \delta(e'[a/v]) \) and similarly \( e' \), this is what we wished to show.

(v) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta) \) and let \( \delta = \delta_1 \circ \delta_2 \) and \( \delta' = \delta'_1 \circ \delta'_2 \). By assumption and Lemma 4, we have \( (((\Phi_3, \gamma_1(\delta(v_1))), (\Phi_4, \gamma'_1(\delta'(v'_1)))) \in V_{\tau_1} \) and \( (((\Phi_3, \gamma_2(\delta(v_2))), (\Phi_4, \gamma'_2(\delta'(v'_2)))) \in V_{\tau_2} \). Apply the definition of \( V \) and Lemma 4.

(vi) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta)\). By assumption and Lemma 4, we have \( (((\Phi_3, \gamma_1(\delta(v))), (\Phi_4, \gamma'_1(\delta'(v')))) \in V_\tau \), so \( (((\Phi_3, \gamma_1(\delta(v))), (\Phi_4, \gamma'_1(\delta'(v')))) \in V_\tau \).

(vii) By Lemma 4.

(viii) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta) \). By assumption \( \Gamma \vdash \gamma \sim \gamma' \), \( \Gamma \vdash \delta \sim \delta' \). By assumption and Lemma 4, we have \( (((\Phi_3, \gamma_1(\delta(v))), (\Phi_4, \gamma'_1(\delta'(v')))) \in V_\tau \). By assumption and the definition of open logical equivalence, \( (((\Phi_3, \vdash \Phi_4), \gamma(\delta(a \rightarrow v')(e')) \in E, \delta(\delta(a \rightarrow v')(e')) \in E \). Since \( \delta(a \rightarrow v')(e') = \delta(e'[a/v]) \) and similarly \( e' \), this is what we wished to show.

(ix) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta) \). By assumption \( \Gamma \vdash \gamma \sim \gamma' \), \( \Gamma \vdash \delta \sim \delta' \). By assumption and Lemma 4, we have \( (((\Phi_3, \gamma_1(\delta(v))), (\Phi_4, \gamma'_1(\delta'(v')))) \in V_\tau \). Proceed by induction on \( n \).

• \( n = 0 \) By assumption, \( ((\Phi_3, \gamma_2(\delta_2(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). The result follows from Lemma 3.

• \( n > 0 \) By induction, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). By assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).

(x) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta, \Delta_2) \) and let \( \delta = \delta_1 \circ \delta_2 \). By assumption, we have \( (((\Phi_3, \gamma_1(\delta(v_1))), (\Phi_4, \gamma'_1(\delta'(v'_1)))) \in V_{\tau} \). Proceed by induction on \( n \).

• \( n = 0 \) By assumption, \( ((\Phi_3, \gamma_2(\delta_2(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). The result follows from Lemma 3.

• \( n > 0 \) By induction, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). By assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).

(xi) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta, \Delta_2) \) and let \( \delta = \delta_1 \circ \delta_2 \). By assumption and Lemma 4, we have \( \gamma(\delta(v)) = v_1 \circ v_2 \) and \( \gamma'(\delta'(v')) = v'_1 \circ v'_2 \). By assumption and Lemma 4, we have \( \gamma(\delta(v)) = v_1 \circ v_2 \) and \( \gamma'(\delta'(v')) = v'_1 \circ v'_2 \). By assumption and the definition of \( V \). Also by assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).

(xii) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta, \Delta_2) \) and let \( \delta = \delta_1 \circ \delta_2 \). By assumption and Lemma 4, we have \( \gamma(\delta(v)) = v_1 \circ v_2 \) and \( \gamma'(\delta'(v')) = v'_1 \circ v'_2 \). By assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).

(xiii) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta, \Delta_2) \) and let \( \delta = \delta_1 \circ \delta_2 \). By assumption and Lemma 4, we have \( \gamma(\delta(v)) = v_1 \circ v_2 \) and \( \gamma'(\delta'(v')) = v'_1 \circ v'_2 \). By assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).

(xiv) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta, \Delta_2) \) and let \( \delta = \delta_1 \circ \delta_2 \). By assumption and Lemma 4, we have \( \gamma(\delta(v)) = v_1 \circ v_2 \) and \( \gamma'(\delta'(v')) = v'_1 \circ v'_2 \). By assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).

(xv) Let \( (\gamma, \gamma') \in G(\Gamma) \) and \((\Phi_3, \delta, \Phi_4, \delta') \in D(\Delta, \Delta_2) \) and let \( \delta = \delta_1 \circ \delta_2 \). By assumption and the definition of open logical equivalence, \( \gamma(\delta(v)) = v_1 \circ v_2 \) and \( \gamma'(\delta'(v')) = v'_1 \circ v'_2 \). By assumption and the definition of open logical equivalence, \( ((\Phi_3, \vdash \Phi_4, \vdash \gamma_1(\delta_1(v))), (\Phi_4, \gamma'_2(\delta'_2(v')))) \in V_{\tau} \). Proceed by induction on \( n \).
\[ \gamma'(\delta'_2[x \mapsto v'_1(v'_2)]) \mapsto v'. \] This gives \( \gamma(\delta(\text{let } e_1 = x \text{ in } e_2)) \mapsto v \) and similar for \( e'_1 = x \in e'_2 \). The result follows from the definition of \( \gamma' \).

B. Proof of correspondence of logical and contextual equivalence

Proof. The reverse direction follows from Lemma 1. We know that \( \gamma' \) is compatible by Theorem 2. It is consistent since if \( ((\Phi, e), (\Phi', e')) \in \mathcal{E}_{\text{nat}} \), then we have \( e \mapsto v \) and \( e' \mapsto v' \) and \( ((\Phi, v), (\Phi', v')) \in \mathcal{E}_{\text{nat}} \), so \( v = v' \).

For the forward direction, we first show a restricted result for closed expressions: if \( :: \vdash \Phi \vdash e : \tau \) and \( :: \vdash \Phi' \vdash e' : \tau \) and \( e =_\text{cts} e' \), then \( ((\Phi, e), (\Phi', e')) \in \mathcal{E}_e \). We will then extend this result for open expressions. Let \( I : \Phi \Rightarrow \mathbb{N} \rightarrow \mathbb{N} \). Proceed by induction on the structure of \( \tau \).

- \( \tau = \text{nat} \). Then, considering the identity context, for some \( n \in \mathbb{N} \), \( e \mapsto n \) and \( e' \mapsto n' \) and \( ((\Phi, n), (\Phi', n)) \in \mathcal{E}_{\text{nat}} \), so the result follows from the definition of \( \mathcal{E} \).

- \( \tau = \tau_1 \otimes \tau_2 \). We wish to show that \( ((\Phi, e), (\Phi', e')) \in \mathcal{E}_{\tau_1} \Rightarrow \mathcal{E}_{\tau_2} \). Let \( ((\Phi, v), (\Phi', v')) \in \mathcal{E}_{\tau_1} \) such that \( \Phi \cong \Phi' \) and \( \Phi' \vdash \Phi'_e \) defined. By the reverse direction, \( \Phi \cong \Phi' \) and \( \Phi' \cong \Phi'_e \) defined. By the reverse direction, \( e_2 =_\text{cts} e_2' \), so by compatibility of \( =_\text{cts} \), we have let \( a \otimes b = c \) in \( e_2 =_\text{cts} e_2' \) and let \( a \otimes b = c' \) in \( e_2' \).

Suppose \( e \mapsto v \mapsto ((\lambda a : \tau_1, e_0) v \mapsto e_0[v/a] \mapsto v_1 \) and \( e' \mapsto v' \mapsto ((\lambda a : \tau_1, e'_0) v' \mapsto e'_0[v'/a] \mapsto v'_1 \) of the definition of \( \mathcal{E} \). We have \( ((\Phi \cong \Phi, v), (\Phi' \cong \Phi'_e, v')) \in \mathcal{E}_{\tau_2} \). This gives \( ((\Phi \cong \Phi, e_0[v/a]), (\Phi' \cong \Phi'_e, e'_0[v'/a])) \in \mathcal{E}_{\tau_2} \), so \( ((\Phi, \lambda a : \tau_1, e_0), (\Phi', \lambda a : \tau_1, e'_0)) \in \mathcal{E}_{\tau_1} \). The result follows from the definition of \( \mathcal{E} \).

- \( \tau = \tau_1 \swarrow \tau_2 \). Let \( ((\Phi, e), (\Phi', e')) \) such that \( \Phi \cong \Phi' \) and \( \Phi' \vdash \Phi'_e \) defined. By the reverse direction, \( e_2 =_\text{cts} e_2' \), so by compatibility of \( =_\text{cts} \), we have let \( a \otimes b = c \) in \( e_2 =_\text{cts} e_2' \) and let \( a \otimes b = c' \) in \( e_2' \).

Suppose \( e \mapsto v \mapsto v_1 \) and \( e' \mapsto v' \) of the definition of \( \mathcal{E}_{\tau_1} \). By definition of \( \mathcal{E} \), we have \( \text{fst}(e) =_\text{cts} \text{fst}(e') \) and \( \text{snd}(e) =_\text{cts} \text{snd}(e') \), so \( v_1 =_\text{cts} v'_1 \) and \( v_2 =_\text{cts} v'_2 \). By induction, \( ((\Phi, v), (\Phi', v')) \in \mathcal{E}_{\tau_1} \) and \( ((\Phi, v_2), (\Phi', v'_2)) \in \mathcal{E}_{\tau_2} \). The result follows from the definition of \( \mathcal{E}_{\tau_1} \) and \( \mathcal{E} \).

- \( \tau = \text{lnat} \). By compatibility of \( =_\text{cts} \), let \( ![x] = e \in x =_\text{cts} \text{let } ![x] = e' \in x \). By induction, \( (\{ \}, \text{let } ![x] = e \in x \), (\{ \}, \text{let } ![x] = e' \in x \)) \in \mathcal{E}_{\tau}, \) so let \( ![x] = e \in x \mapsto ![x] = v \in x \mapsto ![x] = e' \in x \mapsto ![x] = v' \in x \mapsto ![y] = ![y]' \in x \). But this gives \( e \mapsto v \) and \( e' \mapsto v' \) of the definition of \( \mathcal{E} \).

- \( \tau = \text{etr} \). Suppose \( e \mapsto v \) and \( e' \mapsto v' \). We have \( v = \text{etr}[\tau] \text{ with } v_1 = s.p.e.1 \) and \( v' = \text{etr}[\tau'] \text{ with } v'_1 = s.p.e.1' \). We wish to show that there exists a relation \( R \in \text{Rel}(\tau, \tau') \) such that \( (v_1, v'_1) \in R \) and for all \( (v_1, v'_1) \in R \) and \( n \in \mathbb{N} \), there exist \( n', v'_1 \) and \( v'_1 \) such that \( [v_1, n, s, p]e \mapsto v'_1 \otimes v_1 \) and \( [v'_1, n, s, p]e' \mapsto v'_1 \otimes v'_1 \) and \( (v_1, v'_1) \in R \). Let \( R = \{ (v_1, v'_1) \mid \exists C[-], C[v] \mapsto \text{etr}[\tau] \} \). It is easy to see that \( (v_1, v'_1) \in R \), taking the context to be the trivial context. Let \( (v_1, v'_1) \in R \). By the definition of \( R \), there exists \( C[-] \) such that \( C[v] \mapsto \text{etr}[\tau] \) with \( v_1 = s.p.e.1 \) and \( C[v'] \mapsto \text{etr}[\tau] \) with \( v_1' = s.p.e.1' \). Let \( n \in \mathbb{N} \). Suppose \( [v_1, n, s, p]e \mapsto n_1 \otimes v_1 \) and \( [v_1', n, s, p]e' \mapsto n'_1 \otimes v_1' \). By the dynamic rules, \( \text{fst}(\text{query} C[v]' n) \mapsto n_1 \) and \( \text{fst}(\text{query} C[v]' n) \mapsto n'_1 \). By the dynamic rules, \( \text{snd}(\text{query} C[v]' n) \mapsto \text{etr}[\tau] \) with \( v_1' = s.p.e.1' \) and \( \text{snd}(\text{query} C[v]' n) \mapsto \text{etr}[\tau] \) with \( v_1' = s.p.e.1' \). The result then follows from \( \text{etr} \) since \( e \mapsto v \) and \( e' \mapsto v' \) of the definition of \( \text{etr} \).

Now, let \( (\gamma, \gamma') \in \mathcal{G}(\Gamma) \) and \( (\delta, \delta') \in \mathcal{D}(\Delta) \). Using λ-abstractions and applications, we can construct a context \( C[-] \) such that \( C[e] \mapsto \gamma(\delta(\epsilon)) \) and similar for \( e' \). By the above result for closed expressions, we have \( \gamma(\delta(e)) =_\text{cts} \gamma'(\delta'(e')) \). Since contexts compose, closure of contextual equivalence under converse evaluation gives \( e =_\text{cts} e' \).