Temporal Higher-Order Contracts

Tim Disney
University of California, Santa Cruz

Cormac Flanagan
University of California, Santa Cruz

Jay McCarthy
Brigham Young University

Abstract

Behavioral contracts are embraced by software engineers because they document module interfaces, detect interface violations, and help identify faulty modules (packages, classes, functions, etc). This paper extends prior higher-order contract systems to also express and enforce temporal properties, which are common in software systems with imperative state, but which are mostly left implicit or are at best informally specified. The paper presents both a programmatic contract API as well as a temporal contract language, and reports on experience and performance results from implementing these contracts in Racket.

Our development formalizes module behavior as a trace of events such as function calls and returns. Our contract system provides both non-interference (where contracts cannot influence correct executions) and also a notion of completeness (where contracts can enforce any decidable, prefix-closed predicate on event traces).

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.3 [*Language Constructs and Features*]: Constraints

General Terms Languages, Reliability, Security, Verification.

Keywords Higher-order Programming, Temporal Contracts

1. Contract Expressiveness

Large software systems typically consist of many modules (e.g., packages, classes, functions) produced by different development teams. When the system fails, an initial difficulty is *fault localization*: identifying the module that failed to perform as expected. Undocumented module interfaces are problematic for various reasons, not least because they lead to disagreements about which module is considered "at fault" and should be fixed.

Software engineers embrace behavioral contracts because they address many of these problems. In particular, behavioral contracts provide a mechanism to explicitly document each module's assumptions and guarantees; to dynamically detect contract violations; and to identify faulty modules. Behavioral contracts are widely used in procedural, object-oriented, and functional languages, including Eiffel [36], C [42], C# [32], Haskell [26], Java [28], Python [44], Scheme [19, 15, 22], and Smalltalk [10].

Existing contract systems can express a range of interface specifications. Below, we consider a range of possible specifications for a `sort` routine, not all of which are supported by existing contract systems.

1. *The* `sort` *function takes two arguments, an array of positive integers and a comparison function* `cmp`*.*

   This standard, first-order precondition constrains *how* `sort` should be called, that is, what arguments are valid. These kinds of basic first-order contracts are supported by most contract systems, for example, Eiffel [36].

2. *The argument function* `cmp` *in turn requires two arguments, both positive integers.*

   This higher-order precondition constrains how the `sort` module can call the function argument `cmp`, and so is a guarantee provided by `sort` rather than an obligation on the client. Higher-order contract systems [19, 15, 22, 24, 45] support such preconditions by wrapping the `cmp` argument to enforce this property dynamically.

3. *The* `sort` *function is not re-entrant–it can only be called after all previous* `sort` *invocations have completed.*

   Unlike the previous contracts that constrain how functions may be called, this temporal contract constrains *when* `sort` can be called [12, 13]. This constraint implies that `sort` must be used carefully in a multithreaded setting. Moreover, it also imposes restrictions in a sequential setting, since for example, `cmp` is not allowed to call `sort`. A variety of prior systems support such first-order temporal contracts: for example, MOP [33], Tracematches [39], PathExplorer [25], Eagle [6], RuleR [7], and others.

4. `sort` *is granted a capability to call the* `cmp` *function only until* `sort` *returns.*

   This last contract is a *higher-order temporal contract* in that it combines both higher-order and temporal aspects. It expresses a temporal constraint, not on `sort` itself, but on its higher-order interaction with the `cmp` argument it was passed.

   To enforce this contract, it is not sufficient to instrument all `cmp` functions passed to `sort`, since these may legally be called from other call sites after `sort` returns. It is also insufficient to instrument only calls to `cmp` within `sort`, since `sort` might pass a reference to `cmp` to a third party, who in turn could call `cmp` after `sort` returns.

Our study of a widely-used standard library (see Section 8) indicates that higher-order temporal constraints are common in software systems with imperative state, but are mostly left implicit or are at best informally specified. In this paper, we study how to express and enforce such higher-order temporal contracts. Our development leverages and combines techniques from prior higher-order contract systems [19, 15, 22] and from first-order temporal contract systems [33, 39, 25, 6, 7]. We also address blame assignment in a temporal setting. For the example considered above where `sort`

ICFP'11, September 19–21, 2011, Tokyo, Japan.

passes `cmp` to a third party that later calls `cmp`, our temporal contract system immediately detects the error, halts execution, and correctly blames the `sort` module, even though it is neither the caller (the third party) nor the callee (`cmp`) of the erroneous function call.

In a very general sense, contracts provide a means to specify and restrict the *behavior of a module*. In a higher-order imperative setting, this notion of a module's behavior is quite complex, as it can involve higher-order callbacks, mutual recursion, and sharing of imperative state between modules. Understanding the temporal and higher-order aspects of module behavior is a prerequisite to developing expressive contracts for constraining this behavior.

The first step in our investigation is to formalize the behavior of each module as a sequence of *events*, namely remote procedure calls to other modules and matching returns, etc. We start with an operational abstract machine that extends a control-store machine [16, 17] with a remote procedure call mechanism. This abstract machine then generates structural and scoping constraints on event sequences, and so "bootstraps" the game semantics. Module *linking* involves running abstract machines in parallel and appropriately routing events between them.

In this setting, the observable behavior of a module is captured by the set of possible event sequences it can generate. Consequently, the most general notion of a *contract* is a predicate describing permitted event sequences (that is, permitted observable behaviors of the module). We require that the set of event sequences permitted by a contract is prefix-closed, so that errors are detected as soon as they occur. We also require that contracts are computable predicates.

Based on this formalism, we provide a *monitor interface* whereby programmers can express arbitrary contracts as predicates over event sequences. This monitor interface provides:

1. *non-interference*, since the monitor code does not gain references (a.k.a. capabilities [37]) to mutable data or to functions, and so the monitor cannot perturb correct program executions, but can only halt erroneous executions;[1] and
2. a notion of completeness that we call *trace completeness*, meaning that a monitor can impose any computable prefix-closed predicate on a module's event trace (i.e., on its observable behavior).

It is straightforward to achieve one of non-interference or trace completeness; achieving both simultaneously requires a more subtle design since we need to provide monitors with all information about the observed trace (to achieve trace completeness) without giving it access to function references or to mutable state (in order to ensure non-interference).

In addition to this programmatic monitor interface, we also provide a *declarative contract language* for expressing both structural and temporal specifications of a module's behavior. The structural specifications expresses constraints on function arguments and results, and also names each function (including function arguments such as `cmp` above). The temporal specifications is a regular grammar[2] over calls and returns of functions named in the structural component, and includes *dependent sequences* to express, for example, that a call to `release(`x`)` must be preceded by a call to `acquire(`x`)`, for any resource x. Each constraint in this language is compiled into a monitor, and thus leverages the monitor interface to provide non-interference and blame assignment.

[1] Earlier work on higher-order contracts [20] also proposed non-interference as a design goal, but did not achieve a notion of completeness.

[2] We focus on checking safety but not liveness properties, and so do not need the additional expressiveness of temporal logics such as LTL [40].

Contributions The main contributions of this paper are:

- We formalize the behavior of imperative higher-order modules as event sequences, using an operational semantics (section 3).
- We formalize a *contract* as a computable, prefix-closed predicate on event sequences (section 4).
- We present a *monitor interface* that satisfies the twin design goals of non-interference and trace completeness (section 5).
- We show how to track and assign blame for complex temporal contracts (section 6).
- We present a declarative contract language for expressing mixed structural and temporal contracts (section 7).
- We describe a Racket [22][3] implementation of both the monitor interface and the contract language, and include preliminary performance and usability results (section 8).[4]

2. Motivating Examples

We first illustrate how our proposed contract language succinctly captures important structural and temporal constraints of software interfaces.

To start, the following formal contract captures all four of the desired constraints for `sort` informally outlined above:

```
SortContract =
  sort : (List Pos)                                    // 1
         (cmp : Pos → Pos → Bool)                      // 2
       → (List Pos)
  where not ... call(sort,_) !ret(sort,_)*
                 call(sort,_)                          // 3
    and not ... ret(sort,_)  ... call(cmp,_)           // 4
```

By convention, we use Initial Capitals to denote contract names such as `Bool` and `Pos` (for positive numbers) and `SortContract`. The notation "$name : Dom \rightarrow Rng$" describes a function contract where the argument and result values should satisfy contracts *Dom* and *Rng* respectively. The optional "$name$:" prefix associates a name with each function, which can then be referenced in the `where` clause. This clause expresses a regular constraint over the sequence of *events* (calls and returns of these named functions). The pattern "`...`" matches any sequence of events, and the `not` operator has low precedence, so its argument extends to the end of the `where` clause. The pattern "`_`" matches any argument in a `call` or any return value in a `ret`. The prefix "`!`" denotes the negation of a single event (unlike "`not`", which negates a pattern), and the postfix "`*`" denotes a sequence matching the preceding pattern. Thus, "`!ret(sort,_)*`" denotes any sequence of events that does not include a return from `sort`. Whereas, "`not ret(sort,_)*`" matches any traces that are not only returns from `sort`, such as traces that start with a return from `sort`, a call to something else, then any other events.

Constraint (3) declares `sort` to be non-reentrant; it cannot be called a second time before the first call returns. Constraint (4) is a little more complex in that it is a higher-order temporal contract. Each call of `sort` introduces a binding for `cmp` in the above contract, and so `cmp` denotes not just one function, but any comparison function ever passed to `sort`. Each such comparison function has an associated `sort` invocation, and constraint (4) explicates that the comparison function cannot be called after the associated `sort` invocation returns.

[3] Formerly known as PLT Scheme.

[4] The implementation is available in the core Racket distribution as of July 2011 in the "unstable" collection.

As a simpler example, an implicit property of many library functions is that they are expected to return immediately after being called, with no intervening callbacks to client code. We document this *atomicity* property with the following contract, which detects an error if a call to `f` is not immediately followed by a matching return.

```
AtomicContract =
  f : Any → Any
  where not ... call(f,_) !ret(f,_)
```

Since internal calls within `f` are not externally visible, this contract still permits arbitrary internal computation, but forbids callbacks to outside code. Note that this atomicity guarantee is useful even in a sequential setting.

The following interface describes a file system whose `open` method returns a `File` object corresponding to the given file name. Each `File` object contains methods for reading, writing, and closing that file, with the temporal constraint that once the file is closed these methods should not be called. Note that this temporal constraint is scoped to its enclosing `File` object: calling `close` on one file does not influence the ability to access other open files. Put differently, the alphabet of close events is unbounded.

```
FileSystemContract =
  open : String → FileContract

FileContract =
  Record read  : Unit → String
         write : String → Unit
         close : Unit → Unit
  where not
    ... ret(close,_) ...
    (call(close,_) | call(read,_) | call(write,_))
```

Finally, consider the following contract that documents a standard `alloc`/`free` interface on some resource, where each resource instance has an associated integer handle z. After a resource handle z has been freed, it should not be freed again until after it has been re-allocated. The pattern "?z" binds a variable z to an argument/result, allowing z to be referenced later in the grammar.

```
AllocFreeContract =
  Record alloc : Unit → Int
         free  : Int  → Unit
  where not
    ...  call(free,?z) !ret(alloc,z)* call(free,z)
```

We now proceed to investigate a contract system that can express these kinds of mixed structural and temporal constraints.

3. A Model of Module Interactions

The starting point for our investigation of contract expressiveness is an imperative higher-order language, namely the untyped imperative lambda calculus with constants (c) and mutable reference cells.

$$e \quad ::= \quad x \mid \lambda x.\, e \mid e\ e \mid c \mid \texttt{ref}\ e$$

To simplify our presentation, we use standard lambda-calculus encodings of `let` expressions, sequencing, n-ary function definitions and calls, pairs (with accessors `fst` and `snd`), n-ary records, sums, booleans, conditionals, recursive definitions, and assertions. Thus, despite its simplicity, this language is sufficient to model many aspects of modern languages, including encapsulated behavior (via λ-expressions) and both encapsulated and shared imperative state (via `ref`).[5]

[5] The language does not support concurrency, whose interaction with temporal higher-order contracts remains an important topic for future work.

To facilitate our technical development, we take an "interface-oriented" view to reference cells, as proposed by Reynolds [41], whereby `ref` e returns a pair of getter and setter functions for reading and updating the newly created reference cell. This representation allows calls to getter/setter functions to also model operations on shared mutable data, and so all inter-module interaction is modeled by function calls and returns.

A module generally denotes a syntactic construct such as a package, class, function, etc. In the setting of the λ-calculus, we take a *module* to simply mean a closed expression e. Two modules e_1 and e_2 can be *linked* via function application $(e_1\ e_2)$.

As an example, consider the following module $twice$ with an associated test harness H:

$$\begin{array}{rcl} twice & \stackrel{\text{def}}{=} & (\lambda f.\, \lambda x.\, f\ (f\ x)) \\ H & \stackrel{\text{def}}{=} & (\lambda t.\, t\ (\lambda x.\, x{+}1)\ 4) \end{array}$$

The linked program $(H\ twice)$ passes the result of $twice$ into H, which then calls $twice$ with appropriate arguments $(\lambda x.\, x{+}1)$ and 4. The module $twice$ in turn calls back into the $(\lambda x.\, x{+}1)$ function provided by H. Thus, despite its simplicity, the imperative lambda calculus permits rich patterns of module interaction and callbacks. The goal of this paper is to formalize (and subsequently constrain, via contracts) these interactions.

3.1 Semantics of Modules

We formalize the semantics of a module using the operational abstract machine called the CSI machine defined in Figure 1. This machine includes both code (C) and a store (S), and so in part functions much like Felleisen's $\langle C, S\rangle$ machine [16, 17]. The [CALL] rule performs standard β-reduction within an evaluation context $\mathcal{E}$. The rule [PRIM] leverages an auxiliary δ function to define the semantics of primitives. For example, $\delta(\texttt{constant?}, 4) = \texttt{true}$. The rule [REF] for $(\texttt{ref}\ v)$ creates a new reference cell by extending the store S with a triple "$(x, y) \mapsto v$", where v is the initial value of the cell, and x and y are binding occurrences for functions that read and write the value of this cell via the [GET] and [SET] rules, respectively.

The last four rules of the CSI machine add inter-module interactions, in a manner analogous to remote procedure calls. Specifically, the code term C may contain free variables not bound in the store S; these free variables are external references to functions defined by other modules. The CSI machine communicates with these other modules via an event stream.

The CSI machine does not transmit λ-expressions themselves; instead, each transmitted value is first translated into a *handle*, which is either a constant or a fresh variable. The following operation $I[h \triangleright v]$ performs the appropriate translation of a value v into a handle h, and returns an appropriately updated interface component. Constants c are transmitted by-value, and functions $(\lambda y.\, e)$ and imported variables y are transmitted by exporting a fresh variable z that is bound by I to that value. As usual, $I[z \mapsto v]$ denotes a function that is identical to I except that it maps z to v.

$$\begin{array}{lcll} I[c \triangleright c] & \stackrel{\text{def}}{=} & I & \\ I[z \triangleright (\lambda y.\, e)] & \stackrel{\text{def}}{=} & I[z \mapsto \lambda y.\, e] & z \text{ fresh} \\ I[z \triangleright y] & \stackrel{\text{def}}{=} & I[z \mapsto y] & z \text{ fresh} \end{array}$$

In a function application $\mathcal{E}[x\ v]$ where x is externally defined, the rule [SEND-CALL] first translates the argument value v into an external handle h via the operation $I[h \triangleright v]$, and then transmits the event $\texttt{send.call}(x, h)$.

After transmitting the event $\texttt{send.call}(x, h)$, the module becomes inactive or *quiescent*, with code $\mathcal{E}[\texttt{send.call}_x\ \bot]$ indicating that the module is waiting for a matching return $\texttt{rcv.ret}(x, h')$, after which evaluation continues with $\mathcal{E}[h']$: see [RCV-RET].

Figure 1: CSI Machine

Domains

State	CSI	$\in$	$Code \times Store \times Interface$	*Evaluation context*	$\mathcal{E}$	$::=$	$\mathcal{E}[\bullet\ e] \mid \mathcal{E}[v\ \bullet] \mid \mathcal{E}[\texttt{ref}\ \bullet] \mid \mathcal{Q}[\texttt{rcv.call}_x\ \bullet]$
				Quiescent context	$\mathcal{Q}$	$::=$	$\bullet \mid \mathcal{E}[\texttt{send.call}_x\ \bullet]$
Code	C	$::=$	$\mathcal{E}[e] \mid \mathcal{Q}[\bot]$	*Value*	v	$::=$	$c \mid x \mid \lambda y.\ e$
Store	S	$\in$	$\mathcal{P}(Var \times Var \times Value)$	*Handle*	h	$::=$	$c \mid x$
Interface	I	$\in$	$Var \to Value$	*Event*	a	$::=$	$\rho.\texttt{ret}(x,h) \mid \rho.\texttt{call}(x,h)$
				Direction	ρ	$::=$	$\texttt{send} \mid \texttt{rcv}$
				Trace	t	$::=$	$\vec{a}$

Transition relation $(\to) \subseteq State \times Event_\bot \times State$

$$\langle\ \mathcal{E}[(\lambda x.\ e)\ v],\ S,\ I\ \rangle \to \langle\ \mathcal{E}[e[x := v]],\ S,\ I\ \rangle \quad \text{[CALL]}$$

$$\langle\ \mathcal{E}[c\ v],\ S,\ I\ \rangle \to \langle\ \mathcal{E}[v'],\ S,\ I\ \rangle \quad v' = \delta(c,v) \quad \text{[PRIM]}$$

$$\langle\ \mathcal{E}[\texttt{ref}\ v],\ S,\ I\ \rangle \to \langle\ \mathcal{E}[\texttt{pair}\ x\ y],\ S[(x,y) \mapsto v],\ I\ \rangle \quad x, y \text{ fresh} \quad \text{[REF]}$$

$$\langle\ \mathcal{E}[x\ v],\ S[(x,y) \mapsto v'],\ I\ \rangle \to \langle\ \mathcal{E}[v'],\ S[(x,y) \mapsto v'],\ I\ \rangle \quad \text{[GET]}$$

$$\langle\ \mathcal{E}[y\ v],\ S[(x,y) \mapsto v'],\ I\ \rangle \to \langle\ \mathcal{E}[v],\ S[(x,y) \mapsto v],\ I\ \rangle \quad \text{[SET]}$$

$$\langle\ \mathcal{E}[x\ v],\ S,\ I\ \rangle \to^{\texttt{send.call}(x,h)} \langle\ \mathcal{E}[\texttt{send.call}_x\ \bot],\ S,\ I[h \triangleright v]\ \rangle \quad x \notin BV(S) \quad \text{[SEND-CALL]}$$

$$\langle \mathcal{E}[\texttt{send.call}_x\ \bot],\ S,\ I\ \rangle \to^{\texttt{rcv.ret}(x,h)} \langle\ \mathcal{E}[h],\ S,\ I\ \rangle \quad \text{[RCV-RET]}$$

$$\langle\ \mathcal{Q}[\bot],\ S,\ I\ \rangle \to^{\texttt{rcv.call}(x,h)} \langle \mathcal{Q}[\texttt{rcv.call}_x\ (v\ h)],\ S,\ I\ \rangle \quad I(x) = v \quad \text{[RCV-CALL]}$$

$$\langle\ \mathcal{Q}[\texttt{rcv.call}_x\ v],\ S,\ I\ \rangle \to^{\texttt{send.ret}(x,h)} \langle\ \mathcal{Q}[\bot],\ S,\ I[h \triangleright v]\ \rangle \quad \text{[SEND-RET]}$$

Figure 2: Example of Linking and Running Two CSI Machines Concurrently

The two CSI machines shown below cooperate to evaluate $linkRun([\![H]\!], [\![twice]\!])$. After the initial bootstrapping, `send` events of one machine match `rcv` events of the other and vice-versa. The final $\texttt{send.ret}(y, 6)$ event reports the result of the execution is 6.

Evaluation of $H = (\lambda t.\ t\ (\lambda x.\ x{+}1)\ 4)$
$I_1 = [y \mapsto (\lambda t.\ t\ (\lambda x.\ x{+}1)\ 4)]$
$J_1 = [y \mapsto (\lambda t.\ t\ (\lambda x.\ x{+}1)\ 4),\ f \mapsto (\lambda x.\ x{+}1)]$

$$\begin{array}{ll}
\langle \texttt{rcv.call}_{start}\ (\lambda t.\ t\ (\lambda x.\ x{+}1)\ 4), & \emptyset, \emptyset\ \rangle \to^{\texttt{send.ret}(start,y)} \\
\langle \bot, & \emptyset, I_1\ \rangle \to^{\texttt{rcv.call}(y,t)} \\
\langle \texttt{rcv.call}_y\ ((\lambda t.\ t\ (\lambda x.\ x{+}1)\ 4)\ t), & \emptyset, I_1\ \rangle \to \\
\langle \texttt{rcv.call}_y\ ((t\ (\lambda x.\ x{+}1))\ 4), & \emptyset, I_1\ \rangle \to^{\texttt{send.call}(t,f)} \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_t\ \bot)\ 4), & \emptyset, J_1\ \rangle \\
 & \quad \to^{\texttt{rcv.ret}(t,g)} \\
\langle \texttt{rcv.call}_y\ (g\ 4), & \emptyset, J_1\ \rangle \to^{\texttt{send.call}(g,4)} \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ \bot), & \emptyset, J_1\ \rangle \\
 & \quad \to^{\texttt{rcv.call}(f,4)} \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ (\texttt{rcv.call}_f\ ((\lambda x.\ x{+}1)\ 4))), & \emptyset, J_1\ \rangle \to^2 \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ (\texttt{rcv.call}_f\ 5)), & \emptyset, J_1\ \rangle \to^{\texttt{send.ret}(f,5)} \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ \bot), & \emptyset, J_1\ \rangle \to^{\texttt{rcv.call}(f,5)} \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ (\texttt{rcv.call}_f\ ((\lambda x.\ x{+}1)\ 5))), & \emptyset, J_1\ \rangle \to^2 \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ (\texttt{rcv.call}_f\ 6)), & \emptyset, J_1\ \rangle \to^{\texttt{send.ret}(f,6)} \\
\langle \texttt{rcv.call}_y\ (\texttt{send.call}_g\ \bot), & \emptyset, J_1\ \rangle \to^{\texttt{rcv.ret}(g,6)} \\
\langle \texttt{rcv.call}_y\ 6, & \emptyset, J_1\ \rangle \to^{\texttt{send.ret}(y,6)} \\
\langle \bot, & \emptyset, J_1\ \rangle
\end{array}$$

Evaluation of $twice = (\lambda f.\ \lambda x.\ f\ (f\ x))$
$I_2 = [t \mapsto (\lambda f.\ \lambda x.\ f\ (f\ x))]$
$J_2 = [t \mapsto (\lambda f.\ \lambda x.\ f\ (f\ x)),\ g \mapsto (\lambda x.\ f\ (f\ x))]$

$$\begin{array}{ll}
 & \langle \texttt{rcv.call}_{start}\ (\lambda f.\ \lambda x.\ f\ (f\ x)), \emptyset, \emptyset\ \rangle \\
\to^{\texttt{send.ret}(start,t)} & \langle \bot, \emptyset, I_2\ \rangle \\
 & \\
\to^{\texttt{rcv.call}(t,f)} & \langle \texttt{rcv.call}_t\ ((\lambda f.\ \lambda x.\ f\ (f\ x))\ f), \emptyset, I_2\ \rangle \\
\to & \langle \texttt{rcv.call}_t\ (\lambda x.\ f\ (f\ x)), \emptyset, I_2\ \rangle \\
\to^{\texttt{send.ret}(t,g)} & \langle \bot, \emptyset, J_2\ \rangle \\
\to^{\texttt{rcv.call}(g,4)} & \langle \texttt{rcv.call}_g\ ((\lambda x.\ f\ (f\ x))\ 4), \emptyset, J_2\ \rangle \\
\to & \langle \texttt{rcv.call}_g\ (f\ (f\ 4)), \emptyset, J_2\ \rangle \\
\to^{\texttt{send.call}(f,4)} & \langle \texttt{rcv.call}_g\ (\texttt{send.call}_f\ (f\ \bot)), \emptyset, J_2\ \rangle \\
 & \\
\to^{\texttt{rcv.ret}(f,5)} & \langle \texttt{rcv.call}_g\ (f\ 5), \emptyset, J_2\ \rangle \\
\to^{\texttt{send.call}(f,5)} & \langle \texttt{rcv.call}_g\ (\texttt{send.call}_f\ \bot), \emptyset, J_2\ \rangle \\
 & \\
\to^{\texttt{rcv.ret}(f,6)} & \langle \texttt{rcv.call}_g\ 6, \emptyset, J_2\ \rangle \\
\to^{\texttt{send.ret}(g,6)} & \langle \bot, \emptyset, J_2\ \rangle
\end{array}$$

Since the code component encodes both active and quiescent states, we define both standard evaluation contexts $\mathcal{E}$ and *quiescent contexts* $\mathcal{Q}$. A code component of $\mathcal{Q}[\bot]$ means that control is external to the module, but the module can still receive external calls via a $\texttt{rcv.call}(x, h)$ event. Here, x should be some previously exported function whose definition v is defined by I, and evaluation continues with $\mathcal{Q}[\texttt{rcv.call}_x\ (v\ h)]$, as described by [RCV-CALL]. Once $(v\ h)$ reduces to a value v', the $\texttt{rcv.call}_x$ marker indicates that control should return to the caller module via [SEND-RET].

For a module e, we initialize its execution with the code $(\texttt{rcv.call}_{start}\ e)$, where $start$ is an artificial variable name denoting the initial evaluation of a module. Once the evaluation of e terminates, it will return $\texttt{send.ret}(start, h)$ and wait for incoming events. The *meaning* of a module e is then the set of all possible event sequences $\vec{a}$ that e can generate under the CSI machine:

$$\begin{array}{rcl} [\![-]\!] & : & \mathit{Module} \to \mathcal{P}(\mathit{Trace}) \\ [\![e]\!] & = & \{\vec{a} \mid \langle \texttt{rcv.call}_{start}\ e, \emptyset, \emptyset\rangle \to^{\vec{a}} \mathit{CSI}\ \} \end{array}$$

We use t to range over traces or finite event sequences $\vec{a}$; we use T to range over sets of traces; and $t \cdot t'$ to denote trace concatenation. The *exported variables* (EV) and *imported variables* (IV) in a CSI machine state, and the *bound variables* (BV) of a store S, are:

$$\begin{array}{rcl} EV(\ C, S, I\ ) & \stackrel{\text{def}}{=} & \mathit{dom}(I) \\ IV(\ C, S, I\ ) & \stackrel{\text{def}}{=} & (FV(C) \cup FV(\mathit{rng}(S)) \cup\ FV(\mathit{rng}(I))) \\ & & \setminus\ BV(S) \\ BV(S) & \stackrel{\text{def}}{=} & \{x, y \mid S \text{ contains } (x, y) \mapsto v\} \end{array}$$

Thus, the initial state $\langle \texttt{rcv.call}_{start}\ e, \emptyset, \emptyset\rangle$ of a module's computation has no imported variables or *capabilities* [37]; they must be passed explicitly as arguments to functions defined by the module.

3.2 Running a Single Module

Suppose a program consists of a single module e, which performs some computation and then returns a final result, with no subsequent interactions. We assume that the result is first-order data (*i.e.*, a constant) since a function result would be opaque and thus meaningless. The following function $\mathit{run}(T)$ extracts from the trace set $T = [\![e]\!]$ the result of running the program e, which is simply the constant it returns.

$$\mathit{run}(T) \stackrel{\text{def}}{=} \{c \mid \texttt{send.ret}(\mathit{start}, c) \in T\}$$

Thus, we run a program e by starting the CSI machine in state $\langle \texttt{rcv.call}_{start}\ e, \emptyset, \emptyset\rangle$ and run its computation until it emits a $\texttt{send.ret}(start, c)$ event containing the result of that computation.

3.3 Linking Two Modules

More generally, a program may consist of multiple (reactive) modules that must be linked before running. For simplicity, we consider the case where the program consists of just two modules e_1 and e_2, where the module e_1 is a function that expects the result of e_2 as an argument. Subsequently, the two modules may continue to interact by communicating events, where every `send` event of e_2 becomes a `rcv` event by e_1, and vice versa. More generally, every event a of e_2 must match a dual event $\overline{a}$ of e_1, where the *dual* operation $\overline{\rho}$ on directions swaps sends and receives, and this dual operation extends in a compatible manner to events and to event sequences:

$$\overline{\texttt{send}} \stackrel{\text{def}}{=} \texttt{rcv} \qquad \overline{\texttt{rcv}} \stackrel{\text{def}}{=} \texttt{send}$$

Based on this dual operation, we formalize the notion of linking and running two trace sets T_1 and T_2 (where $T_i = [\![e_i]\!]$) as follows:

$$\begin{array}{l} \mathit{linkRun}(T_1, T_2) \stackrel{\text{def}}{=} \\ \{c \mid \texttt{send.ret}(\mathit{start}, h) \cdot t_2 \in T_2 \\ \quad \texttt{send.ret}(\mathit{start}, x) \cdot \texttt{rcv.call}(x, h) \cdot \overline{t_2} \cdot \texttt{send.ret}(x, c) \in T_1\} \end{array}$$

Here, trace set T_2 (for the function argument) starts by returning a handle h. Trace set T_1 returns a function x that is then immediately applied to handle h from T_2. Subsequent events from the two trace sets must then correspond, with each send event from T_1 matching a receive event from T_2, and vice versa. Eventually, after the function x completes its computation and its interaction with its argument h, trace set T_1 finally returns a result c. We assume that this result is first-order data (*i.e.*, a constant) since a function result would be opaque and thus meaningless.

3.4 Example

To illustrate module linking, Figure 2 revisits the $(H\ \mathit{twice})$ example to show how each module can run on a separate CSI machine, using the function *linkRun* to link up the event traces from these two machines. In particular, the handle t returned by the evaluation of twice is passed as the argument to the function y produced by the evaluation of H. Subsequent events from the two modules must match exactly, with each `send` event from one module matching a corresponding `rcv` event from the other module. Finally, the H module returns the constant 6 as the result of applying y to twice; thus 6 is the final result of running this program.

For this test harness, the module twice produces the trace shown in the right column of Figure 2. However, twice can generate many additional traces if linked with other modules. The trace set $[\![\mathit{twice}]\!]$ captures the set of all possible traces generated by the CSI machine when executing twice.

3.5 Semantic Equivalence

We now have two notions of module linking. We can link two modules e_1 and e_2 via function application, and run the resulting expression $(e_1\ e_2)$ on a single CSI machine according to *run*. Alternatively, we can run e_1 and e_2 on separate CSI machines that communicate according to $\mathit{linkRun}$. The following theorem proves that these two notions coincide, which implies that $\mathit{linkRun}$ correctly captures our notion of module composition as function application:

THEOREM 1 (Module Transparency). *For any closed expressions e_1 and e_2, $\mathit{linkRun}([\![e_1]\!], [\![e_2]\!]) = \mathit{run}([\![e_1\ e_2]\!])$.*

PROOF: By proving a bisimulation relation between the CSI machine state of run and the *CSI state pairs* of $\mathit{linkRun}$. □

3.6 Well-Formed Traces

An inspection of the CSI machine reveals that it generates traces satisfying the following structural and scoping properties; we refer to such traces as being *well-formed*.

1. Traces consist of alternating send events, which produce a quiescent state, and receive events, which re-initiate computation.
2. Every $\texttt{send.ret}(x, h)$ matches a preceding $\texttt{rcv.call}(x, h')$, and every $\texttt{rcv.ret}(x, h)$ matches a preceding $\texttt{send.call}(x, h')$, so these events form "balanced bracket pairs", with the exception of the initial $\texttt{send.ret}(start, h)$ that starts each trace.
3. Exporting a variable x via $\texttt{send.call}(y, x)$ or $\texttt{send.ret}(y, x)$ adds x to the interface I, where it is in scope for subsequent $\texttt{rcv.call}(x, h)$ events.

4. Importing a variable x via $\texttt{rcv.call}(y, x)$ or $\texttt{rcv.ret}(y, x)$ means that x occurs free in the CSI machine and can be used in subsequent $\texttt{send.call}(x, h)$ events.

Properties 3 and 4 provide an implicit notion of variable scoping, where the second variable in each event is a binding occurrence, and the first argument must be a bound occurrence. We consider two traces to be α-equivalent if they are identical modulo consistent renaming of bound variables, and we implicitly perform α-conversion to avoid name collisions.

The set T of traces generated by the CSI machine also satisfies the following properties:

1. It is prefix-closed.
2. It is *input-enabled*, in that whenever it can receive one event it is willing to receive any well-formed event: if $t.a \in T$, a and a' are both receive events, and $t.a'$ is well-formed then $t.a' \in T$.
3. It is *output-deterministic*, in that at most one send event is possible at any point. Thus, if $t.a \in T$, $t.a' \in T$, and a and a' are both send events, then $a = a'$.

4. Contracts as Trace Predicates

The CSI machine semantics exposes each module's behavior as a trace set. We now use this formalism to study contract enforcement.

Consider a module e whose behavior is the set of traces $[\![e]\!]$. Each trace represents a behavior that could be observed on a particular execution. A contract is a mechanism for restricting the set of observable behaviors to those satisfying some safety predicate.

Formally, a *contract* is a set of traces $K \subseteq \mathit{Trace}$ that is prefix-closed, computable, and that contains only well-formed traces. The execution of e under contract K should only generate traces in the intersection $K \cap [\![e]\!]$. Since K restricts both `rcv` and `send` events, it explicates e's assumptions and guarantees.

For a contract K, we want to implement a function e_K that takes as argument the module e. The function e_K then observes all of e's behaviors and halts program execution if any of those behaviors violate K. Thus, e_K should satisfy a requirement like $[\![e_K\ e]\!] = K \cap [\![e]\!]$. However, $[\![e_K\ e]\!]$ must be input-enabled but K (and hence $K \cap [\![e]\!]$) may not be input-enabled, so more precisely $[\![e_K\ e]\!]$ should be the *input-enabled closure* of $K \cap [\![e]\!]$, that is, the smallest superset of $K \cap [\![e]\!]$ that is input-enabled. Intuitively, $[\![e_K\ e]\!]$ has to receive "bad" events from its enclosing context but can block them from being sent on to e.

4.1 The Universal Contract

The identity function $(\lambda x.\, x)$ behaves like a universal contract, in that $((\lambda x.\, x)\ e)$ exposes all the behaviors of module e. We start by studying the semantics $[\![\lambda x.\, x]\!]$ of this universal contract.

Consider the possible executions of the CSI machine starting from the initial state $S_0 = \langle \texttt{rcv.call}_{\mathit{start}}\ (\lambda x.\, x), \emptyset, \emptyset \rangle$. Whenever the machine imports a variable y via [RCV-CALL] or [RCV-RET], it immediately exports a fresh name, say $\widehat{y}$, for that variable, via [SEND-CALL] or [SEND-RET]. Here, the hat $(\widehat{\cdot})$ operator maps between imported and exported variable names, and is an involution (its own inverse). We extend this hat operator to handles by defining $\hat{c} = c$. The set of identity function traces $[\![\lambda x.\, x]\!]$ is then the well-formed subset of the prefix-closure of the following regular grammar. Note that the last two productions essentially copy events from one side of the identity function to the other, renaming variables appropriately.

$$
\begin{array}{rcl}
T_{id} & ::= & \texttt{send.ret}(\mathit{start}, f) \\
& | & T_{id} \cdot \texttt{rcv.call}(f, h) \cdot \texttt{send.ret}\ (f, \widehat{h}) \\
& | & T_{id} \cdot \texttt{rcv.call}(y, h) \cdot \texttt{send.call}(\widehat{y}, \widehat{h}) \\
& | & T_{id} \cdot \texttt{rcv.ret}\ (y, h) \cdot \texttt{send.ret}\ (\widehat{y}, \widehat{h})
\end{array}
$$

4.2 Copycat Traces

We next consider contracts that are more restrictive than the identity function. Suppose the target module e generates the trace:

$$t = \texttt{send.ret}(\mathit{start}, h) \cdot a_1 \cdot \ldots \cdot a_n$$

If the contract K permits this trace then $[\![e_K]\!]$ should include:

$$
\begin{array}{l}
\mathit{copycat}(t) \stackrel{\text{def}}{=} \\
\quad \texttt{send.ret}(\mathit{start}, f) \cdot \texttt{rcv.call}(f, h) \cdot \texttt{send.ret}(f, \widehat{h}) \cdot \\
\quad \mathit{copy}(a_1) \cdot \ldots \cdot \mathit{copy}(a_n)
\end{array}
$$

This trace returns a function f that is applied to the handle h produced by t. It then returns a renamed handle $\widehat{h}$, and subsequently copies every event a_i from one side of e_K to the other via the following event copying function $\mathit{copy}(a)$:

$$
\begin{array}{rcl}
\mathit{copy}(\texttt{send.call}(y, h)) & = & \texttt{rcv.call}(y, h) \cdot \texttt{send.call}(\widehat{y}, \widehat{h}) \\
\mathit{copy}(\texttt{send.ret}\ (y, h)) & = & \texttt{rcv.ret}\ (y, h) \cdot \texttt{send.ret}\ (\widehat{y}, \widehat{h}) \\
\mathit{copy}(\texttt{rcv.call}(y, h)) & = & \texttt{rcv.call}(\widehat{y}, \widehat{h}) \cdot \texttt{send.call}(y, h) \\
\mathit{copy}(\texttt{rcv.ret}\ (y, h)) & = & \texttt{rcv.ret}\ (\widehat{y}, \widehat{h}) \cdot \texttt{send.ret}\ (y, h)
\end{array}
$$

We extend *copycat* from traces to trace sets in a pointwise fashion:

$$\mathit{copycat}(K) \stackrel{\text{def}}{=} \{\mathit{copycat}(t) \mid t \in K\}$$

This *copycat* function allows us to formalize the relationship between the permissible traces K and the contract implementation e_K. We require that e_K copies only K-traces:

$[\![e_K]\!]$ is the prefix and input-enabled closure of $\mathit{copycat}(K)$

5. The Programmatic Monitor Interface

Based on the above specification, it is possible to hand-code an implementation e_K for each contract K, but these implementations are difficult to read and write, and can violate non-interference.

Instead, we propose to express e_K as a pair of a *guard* and a *monitor*. The predefined `guard` function performs the necessary copycat behavior on event streams and guarantees non-interference. It also encodes the observed event stream and communicates it to the monitor M, which detects errors and reports them to the guard, which then halts execution. Thus, we implement e_K as:

$$e_K \stackrel{\text{def}}{=} (\texttt{guard}\ M)$$

Traces contain scope and binding information, which is tricky to communicate to the monitor M. Our chosen encoding is that, when an event such as $\texttt{rcv.call}(\cdot, x)$ introduces a new variable x, the monitor M creates two new sub-monitors, say M_{call} and M_{ret}, which handle events $\rho.\texttt{call}(x, \_)$ and $\rho.\texttt{ret}(x, \_)$, respectively. This encoding avoids passing function references (or getters/setters for mutable reference cells) to M, which helps ensure non-interference. We also require that the monitor M is closed, so that it does not gain accesses to function references through its environment.

Based on this discussion, the type τ of the monitor M is:

$$\tau = (\mathit{Constant} \to \mathit{Bool}) \times (\mathit{Unit} \to (\mathit{False} + (\tau \times \tau)))$$

When an event transmits a constant, the first component of M is called to decide if that event is permitted. That function can use mutable internal state to track and enforce temporal properties. Thus, temporal properties of the contracts are implemented via imperative state in the monitor.

When an event transmits a function x, the second component of M is called, which again has the opportunity to halt execution by retuning `false`. Otherwise, the function should return a pair of monitors for handling `call` and `ret` events on the function x.

The function guard below then converts its monitor argument M into an appropriate contract implementation that queries M to verify that each transmitted event is permitted:

```
guard M =
 λx. if (constant? x) then
       assert ((fst M) x) != false
       x
     else
       let MM = (snd M)()
       assert MM != false
       λy. (guard (snd MM) (x (guard (fst MM) y)))
```

5.1 Example Monitors

Using the monitor interface, it is straightforward to implement a universal monitor Any that permits all communication:

```
rec Any = pair (λx. true) (λ. pair Any Any)
```

As a more interesting example, the following code shows how to implement *revokable membranes* [37]. In this code, `ref true` returns a pair of `get` and `set` methods. As long as `get()` returns `true`, the monitor `M` behaves like the universal contract. Calling `revoke` puts `false` into the reference cell, after which all further communication is prohibited. The top-level `membrane` function returns a pair of the membrane monitor and its revoke function.

```
membrane =
  λ. let (get,set) = ref true
     let revoke    = λ. set false
     letrec M      = pair (λx. get())
                          (λ . if (get())
                               then (pair M M)
                               else false)
     in pair M revoke
```

Note that `M` is not strictly speaking a monitor, since it calls a free variable `get`, which could (in theory, but not in this case) interfere with the enclosing program.

5.2 Deep Tracing

A particularly interesting monitor is `deepTrace`, which performs "deep" or higher-order tracing. The code `((guard deepTrace) g)` prints all calls and returns of `g`. Additionally, if `g` takes or returns functions, then those functions are also traced. Thus, deep tracing reports both first-order and higher-order interactions between the module `g` and the rest of the program.

Revisiting the example *twice* from Section 3.4, the program

```
(H ((guard deepTrace) twice))
```

prints out the event sequence shown in the right half of Figure 2 (modulo naming of bound variables). Alternatively, the program

```
(((guard deepTrace) H) twice)
```

prints out the event sequence shown in the left half of that figure. In this manner, deep tracing allows the programmer to observe the trace semantics of a module as explored during a particular run.

The implementation of `deepTrace` uses an auxiliary function `helper` to generate the appropriate monitor. The `helper` function takes four arguments: the first two indicate whether this monitor is observing send or receive events; the third whether it is receiving `ret` or `call` events; and the fourth argument is a function name. Every function that passes through the guard is assigned a new name via `gensym`, and the generated monitor then prints each event.

```
deepTrace = (helper "send" "rcv" "ret" "start")

helper send rcv op fn =
  pair (λx. print (send+"."+op+"("+fn+","+x+")")
            true)
       (λ.  let name = gensym()
            print (send+"."+op+"("+fn+","+name+")")
            pair (helper rcv send "call" name)
                 (helper send rcv "ret"  name))
```

5.3 Non-Interference and Trace Completeness

We now revisit the design goals of non-interference and trace completeness, to show how they are achieved by the guard/monitor architecture.

First, for any monitor M, the function $(\texttt{guard}\ M)$ satisfies non-interference, in that it behaves like the identity function except that it may abruptly halt certain (erroneous) executions. Thus, `guard` behaves like an "information diode", in that information can flow into the monitor M but can never flow back out to the program. In a richer language with exceptions, `guard` would need to appropriately catch any thrown exceptions to ensure that they do not constitute an additional information channel between the monitor and the enclosing program.

THEOREM 2 (Non-Interference). *For any closed expression M,*

$$[\![\texttt{guard}\ M]\!] \subseteq [\![\lambda x.\, x]\!]$$

PROOF: By showing that the trace set $[\![\lambda x.\, x]\!]$ from section 4.1 is a superset of $[\![\texttt{guard}\ M]\!]$. The behavior of M is irrelevant, since non-interference is guaranteed entirely by `guard`. □

Our argument for trace completeness is based on showing that the monitor observes all events that cross the contract boundary. Specifically, consider the semantics of an arbitrary module e under the `deepTrace` monitor, *i.e.*,

$$[\![(\texttt{guard deepTrace})\ e]\!]$$

An investigation of the reachable states of the corresponding CSI machine shows that this module prints each `send` event before transmitting it, and prints each `rcv` event right after receiving it. Instead of printing these events, `deepTrace` could feed them into an algorithm that decides a computable contract K over event traces, and could halt execution once that algorithm detects a trace that violates K. Consequently, any computable predicate over traces could be implemented using the monitor interface.

5.4 Dependent Contracts

Dependent contracts allow a function's range contract R to depend on the function's argument x. If the argument x is itself a function, then R could violate non-interference by calling x.

Our guard/monitor architecture avoids this difficulty by only allowing dependency when the argument x is a first-order value (that is, a constant). Essentially, the argument x is saved on a stack by the domain contract, and is then popped by the matching range contract. The following monitor illustrates how to implement such dependent contracts. It specifies a function contract where the integer result should be greater than the integer argument. The monitor accepts only functions, where the function's argument `x` must be an integer constant that is pushed on a stack that records the arguments of currently active calls, and the function's result `y` must be an integer that is greater than the argument.

```
increasingMonitor =
  let (push,pop) = newStack()
  pair (λx. false) // no constants allowed
       (λ. pair (pair (λx. (push x); (integer? x))
                      (λ.  false))
                (pair (λy. (integer? y) && y > pop())
                      (λ.  false)))
```

This explicit stack design also avoids subtle difficulties with blame assignment in traditional dependent contracts [15].

6. Blame Assignment

The discussion so far has focused on detecting contract violations; we next address how to assign blame for these violations. In particular, we show that *blame assignment*, previously studied in higher-order contract systems [19], extends in a consistent fashion to the more general setting of temporal higher-order contracts.

Suppose the program consists of two modules e_1 and e_2, with corresponding trace sets $T_i = [\![e_i]\!]$ for $i \in 1, 2$, and suppose that K is a contract on e_2. The semantics of correct executions is defined as:

$$\mathit{linkRun}(T_1, K \cap T_2)$$

This formula links together the modules e_1 and e_2, with the requirement that all interaction between the two modules must be permitted by K.

We now extend that semantics to assign blame, via the function $\mathit{linkMonitorRun}(T_1, K, T_2)$ shown below. The first case in this definition is analogous to the function *linkRun* defined earlier, with the additional requirement that the trace $\texttt{send.ret}(\mathit{start}, h) \cdot t_2$ in T_2 must be permitted by K. The second case deals with assigning blame, where T_1 and T_2 want to exchange a message a that is not permitted by K, but where the prefix of a is accepted by all parties. In this situation, the module i sending that message is blamed.

$$\begin{array}{l}
\mathit{linkMonitorRun}(T_1, K, T_2) \stackrel{\text{def}}{=} \\
\{c \mid \texttt{send.ret}(\mathit{start}, h) \cdot t_2 \in K \cap T_2 \\
\quad \texttt{send.ret}(\mathit{start}, x) \cdot \texttt{rcv.call}(x, h) \cdot \overline{t_2} \cdot \texttt{send.ret}(x, c) \in T_1\} \\
\cup \{\mathit{blame}_i \mid \texttt{send.ret}(\mathit{start}, h) \cdot t_2 \cdot a \in T_2 \\
\qquad \texttt{send.ret}(\mathit{start}, x) \cdot \texttt{rcv.call}(x, h) \cdot \overline{t_2} \cdot \overline{a} \in T_1 \\
\qquad \texttt{send.ret}(\mathit{start}, h) \cdot t_2 \in K \\
\qquad \texttt{send.ret}(\mathit{start}, h) \cdot t_2 \cdot a \notin K \\
\qquad \texttt{if } |t_2| \textit{ odd } \texttt{then } i = 1 \texttt{ else } i = 2 \quad \}
\end{array}$$

Thus $\mathit{linkMonitorRun}(T_1, K, T_2)$ extends $\mathit{linkRun}(T_1, K \cap T_2)$ to return a blame label in situations where *linkRun* would fail silently.

To incorporate blame assignment into the guard implementation, we introduce additional arguments to track the module and enclosing context of each monitor, where these two arguments are swapped for contravariant domain checks (as in [19]). We assume a primitive `blame` for reporting blame, and an appropriate representation for module labels (e.g., strings).

```
guard module context M =
 λx. if (constant? x) then
       if ((fst M) x) == false then blame module
       x
     else
       let MM = (snd M)()
       if MM == false then blame module
       λy. (guard module context (snd MM)
             (x (guard context module (fst MM) y)))
```

Figure 3: Declarative HOT Contracts

M	::=	S `where` R	HOT contract
S	::=	$\mathit{flat}(e) \mid n{:}S_1 \mapsto S_2$	Structural contract
R	::=	$A \mid !A \mid RR \mid R^* \mid \texttt{not}\ R \mid R \cup R$	Temporal contract
		$\mid \ldots \mid \texttt{call}(n, ?x)\ R \mid \texttt{ret}(n, ?x)\ R$	
A	::=	$\texttt{call}(n, p) \mid \texttt{ret}(n, p)$	Event patterns
p	::=	$\_ \mid x \mid c$	Value patterns
n	$\in$	*Name*	Function names

7. The Declarative Contract Language

The programmatic monitor interface provides trace completeness, non-interference, and blame assignment. We now leverage that interface to develop a declarative contract language for writing mixed higher-order temporal (HOT) contracts. Our declarative language sacrifices trace completeness for ease-of-expression, but still interoperates with the monitor interface for situations where additional expressiveness is required.

Figure 3 summarizes the contract syntax. A *HOT contract*

$$S\ \texttt{where}\ R$$

contains both a structural component (S) that binds function names (n), and a temporal component (R) that imposes constraints on when those functions can be called or return.

The flat structural contract $\mathit{flat}(e)$ describes the set of constants c that satisfy the predicate e. The higher-order structural contract $(n : S_1 \mapsto S_2)$ describes functions where S_1 and S_2 specify the function's domain and range contracts, respectively, and n provides a *name* for this function that can be referenced in the temporal component.

In the temporal component, event patterns $\texttt{call}(n, p)$ and $\texttt{ret}(n, p)$ denote calls and returns of the function named n, where the argument or result matches pattern p. Patterns include constants, variables x, and "_", which matches any argument.

Temporal contracts (R) include events (A), negated events ($!A$), concatentation (RR), Kleene closure (R^*), negation of trace sets ($\texttt{not}\ R$), union ($R \cup R$), and the universal temporal contract ("..."), which matches any trace. Temporal contracts also include *dependent sequencing* patterns such as $\texttt{call}(n, ?x)\ R$, where the argument from the first event is bound to x, and can be referred to from within R. Dependent sequencing captures common constraints on function arguments and returns, for example, that the argument passed to `free` must previously have been returned from `alloc`.

7.1 Semantics of HOT Contracts

To formalize the semantics of a structural contract S, we define an abstract machine called the EF machine that describes what traces S permits.

This EF machine contains an environment E and a stack F. The environment E associates each variable x with a direction ρ (describing whether x was sent or received) and a structural contract S (describing permitted uses of x). The stack F contains the variable names of inter-module calls (or stack frames).

$$\begin{array}{rcl} E & ::= & \epsilon \mid E, x : \rho S \\ F & ::= & \mathit{Variable}^* \end{array}$$

The EF machine generates traces according to the following transition relation $EF \Rightarrow^a EF'$. The stack length $|F|$ indicates whether the contracted module is active or quescient, so if $|F|$ is odd we require that $\rho = \texttt{send}$, and otherwise that $\rho = \texttt{rcv}$, in both

of these rules.

$$\langle E, F\rangle \quad \Rightarrow^{\rho.\texttt{call}(x,h)} \quad \langle E \oplus (h : \rho S_1), F.x\rangle \quad \text{where } E(x) = \overline{\rho}(n : S_1 \mapsto S_2) \qquad \text{[S-CALL]}$$

$$\langle E, F.x\rangle \quad \Rightarrow^{\rho.\texttt{ret}(x,h)} \quad \langle E \oplus (h : \rho S_2), F\rangle \quad \text{where } E(x) = \rho(n : S_1 \mapsto S_2) \qquad \text{[S-RET]}$$

Assuming $\rho = \texttt{send}$, the rule [S-CALL] generates a call event $\texttt{send.call}(x, h)$, provided x was previously received and has a function contract. (If $\rho = \texttt{rcv}$ then the dual situation applies.) The [S-RET] rule generates a return event that must return to the top variable on the stack F.

The operation $E \oplus (h : \rho S_1)$ checks if a sent handle h is compatible with the argument contract S_1, and extends the environment E appropriately. Note that the check $run([\![e\ c]\!]) = \texttt{true}$ ensures that the constant c satisfies the structural contract $\mathit{flat}(e)$.

$$\begin{array}{lcll} E \oplus (c : \rho\, \mathit{flat}(e)) & = & E & \text{provided } run([\![e\ c]\!]) = \texttt{true} \\ E \oplus (x : \rho S) & = & E, x : \rho S & \end{array}$$

The meaning of a structural contract S is then defined as the set of all traces that first return a handle satisfying S, and where subsequent interactions satisfy the requirements of the EF machine:

$$[\![S]\!] \quad = \quad \{\ \texttt{send.ret}(\mathit{start}, h).t \mid \langle E_0, \mathit{start}\rangle \Rightarrow^t \langle E, F\rangle\ \} \quad \text{where } E_0 = \emptyset \oplus (h : \texttt{send}\ S)$$

The meaning of a temporal contract R is defined with respect to the environment E that is produced by the EF machine, where E is used to map each variable x in the trace to a name n that is referenced in the temporal contract. The relation $p \sim h$ defines when a pattern p matches a handle h. Constants match constants ($c \sim c$), and the pattern "_" matches any handle ($\_ \sim h$).

$$\begin{array}{rcl} [\![\bullet]\!]_\bullet : R \times E & \to & \mathcal{P}(\mathit{Trace}) \\ [\![\texttt{call}(n,p)]\!]_E & = & \{\rho.\texttt{call}(y,h) \mid E(y) = n : \ldots \text{ and } p \sim h\} \\ [\![\texttt{ret}(n,p)]\!]_E & = & \{\rho.\texttt{ret}(y,h) \mid E(y) = n : \ldots \text{ and } p \sim h\} \\ [\![!A]\!]_E & = & \mathit{Event} \setminus [\![A]\!]_E \\ [\![R_1 R_2]\!]_E & = & [\![R_1]\!]_E \cdot [\![R_2]\!]_E \\ [\![R^*]\!]_E & = & [\![R]\!]^*_E \\ [\![\texttt{not}\ R]\!]_E & = & \text{any trace} \setminus [\![R]\!]_E \\ [\![R_1 \cup R_2]\!]_E & = & [\![R_1]\!]_E \cup [\![R_2]\!]_E \\ [\![\ \ldots\ ]\!]_E & = & \text{any trace} \\ [\![\texttt{call}(n, ?x)\ R]\!]_E & = & \{\rho.\texttt{call}(y,c) \cdot t \mid E(y) = n{:}..,\ t \in [\![R[x := c]]\!]_E\} \\ [\![\texttt{ret}(n, ?x)\ R]\!]_E & = & \{\rho.\texttt{ret}(y,c) \cdot t \mid E(y) = n{:}..,\ t \in [\![R[x := c]]\!]_E\} \end{array}$$

Finally, the meaning of a contract (S `where` R) is defined as any trace that satisfies S, providing a resulting environment E, and where the trace also satisfies R with respect to E. The function *prefixes* performs prefix-closure on a set of traces.

$$\begin{array}{l} [\![S\ \texttt{where}\ R]\!] \\ \stackrel{\text{def}}{=} \{\ (\texttt{send.ret}(\mathit{start}, h) \cdot t) \in \mathit{prefixes}([\![R]\!]_E) \mid \\ \quad \langle E_0, \mathit{start}\rangle \Rightarrow^t \langle E, F\rangle \text{ and } E_0 = \emptyset \oplus (h : \texttt{send}\ S)\ \} \end{array}$$

Thus, a module e under contract S `where` R yields the trace set:

$$[\![e]\!] \cap [\![S\ \texttt{where}\ R]\!]$$

7.2 Compiling HOT Contracts

We enforce each contract (S `where` R) by compiling it into an appropriate monitor. We convert the temporal component R into a finite state automaton, where `s` ranges over the state space of the automaton, s_0 denotes the initial state, and the handlers $\texttt{call}_n$ and $\texttt{ret}_n$ for each function name n imperatively update `s` appropriately and return `true` if the automaton is in an accepting state and `false` otherwise. The code is roughly:

$$\begin{array}{lcl} \texttt{let s} & = & s_0 \\ \quad \texttt{call}_n & = & \lambda i. \cdots \text{check and update s appropriately } \cdots \\ \quad \ldots & & \ldots \\ \quad \texttt{ret}_n & = & \lambda o. \cdots \text{check and update s appropriately } \cdots \\ \quad \ldots & & \ldots \\ \texttt{in}\ \mathit{compile}(\lambda\texttt{x.true}, S) & & \end{array}$$

The function $\mathit{compile}(f, S)$ generates a monitor that ensures that the trace satisfies S, and which is parameterized over a number of $\texttt{call}_n$ and $\texttt{ret}_n$ functions that communicate to the temporal code above. The additional argument

$$f : (\mathit{Constant} \cup \{\lambda\texttt{x.x}\} \mapsto \mathit{Bool})$$

is an "observer function" that is called and can signal an error whenever a value passes through this S boundary; it is used in the following recursive definition of *compile*.

$$\mathit{compile} : (\mathit{Constant} \cup \{\lambda\texttt{x.x}\} \mapsto \mathit{Bool}) \times S \mapsto \mathit{Monitor}$$

$\mathit{compile}(f, \mathit{flat}(e)) \stackrel{\text{def}}{=}$

```
pair (λx. (e x) && (f x))
     (λ.  false)
```

$\mathit{compile}(f, n{:}S_1 \mapsto S_2) \stackrel{\text{def}}{=}$

```
pair (λx. false)
     (λ.  (f (λx.x)) && (pair compile(call_n, S_1)
                              compile(ret_n,  S_2))))
```

$\mathit{compile}(f, y) \stackrel{\text{def}}{=}$

```
let (chkconst,chkfn) = y
pair (λx. (f x)      && (chkconst x))
     (λ.  (f (λx.x)) && (chkfn()))
```

For the contract $\mathit{flat}(e)$, the generated monitor accepts only constants; it checks that the constant satisfies both the observer function f and the predicate e. For the function contract $n : S_1 \mapsto S_2$, the generated monitor accepts only functions. It immediately notifies the observer that a function is passing through this contract, and then returns a pair of monitors that monitor calls and returns of this function, in each case notifying $\texttt{call}_n$ or $\texttt{ret}_n$ and enforcing the sub-contracts S_1 or S_2, as appropriate.

Our implementation extends the structural contract language to include variable references (y) for referring to predefined monitors such as the `Any` monitor defined in section 5.1. The compilation of such monitor references extends the referenced monitor to call the observer function f appropriately.

Compilation Correctness The *compile* function compiles a structural contract S into a monitor in a manner than respects the intended meaning $[\![S]\!]$ of the structural contract.

THEOREM 3 (Compilation Correctness). *For any structural contract S,* $[\![S]\!] = [\![\texttt{guard}\ \mathit{compile}(\lambda\texttt{x.true}, S)]\!]$.

PROOF SKETCH: We proceed by structural induction on S.

In the case that $S = \mathit{flat}(e)$: we substitute into the theorem, then focus on the right-hand side. We expand the definition of *compile* and do beta reduction. The application of `(f x)` in the definition reduces to true with the given f, so we remove it and the *and*-expression. This and an application of eta produces the right-hand side: $[\![\texttt{guard (pair}\ e\ (\lambda\texttt{.false}))]\!]$. If we expand this

application of `guard`, it only succeeds if the value sent to the contract is a constant, c, such that $(e\ c)$ evaluates to `true`. We now focus on the left-hand side. By expanding the definition of $[\![S]\!]$ and $\oplus$, we get the condition that $run([\![e\ c]\!]) = \texttt{true}$ where c is the constant sent to the contract. Clearly these two constraints are the same, so this case is completed.

In the case that $S = n : S_1 \mapsto S_2$, we use the inductive hypothesis twice. However, getting to that point requires a few subtle steps. First, we assume that $\texttt{call}_n$ and $\texttt{ret}_n$ are $\lambda\texttt{x.true}$ for all n, since we are only considering structural contracts, so there are no temporal properties to enforce. Next, we consider the behavior of both sides after it has been applied. Only the case where it is actually a function is relevant, because if it is a constant, both trivially reject any subsequent events. After assuming we've received a function, we then assume that the function is called and returns. The contracts S_1 and S_2 protect the call and return through the S-CALL and S-RET rules on the left and through the recursive calls to `guard` on the right—and the inductive hypothesis ensures that these have identical semantics. □

Compiling Dependent Sequences For dependent sequences, we follow the same general approach, except we cannot compile to a finite state automaton. For example, the constraint

```
not ... call(free,?z) !ret(alloc,z)* call(free,z)
```

requires unbound storage to record all freed locations z. Instead, we compile to a lazily constructed infinite automaton. Each automaton is a function from an event (call or return) to a next automaton—as well as a boolean to encode acceptance. Event patterns are functions that return the epsilon automaton on matching and a null (rejecting) automaton otherwise. In contrast, dependent sequences return a new automaton representing the rest of the trace, and which includes an appropriate binding for the dependent variable z.

Each of the regular grammar operators (sequencing, complement, etc) is implemented as an explicit automaton that simulates the operator using the automaton functions representing its pieces. These implementations correspond precisely to the intuitive explanations of regular operator closure properties found in any textbook on automata theory. Since Kleene star and sequencing may invoke their arguments multiple times, dependent sequences embedded in them will be duplicated for each successful match of the pattern. This approach is similar to regular expression derivatives [9, 38].

7.3 Examples

Section 2 presented a variety of examples of structural/temporal contracts, whose meaning and compilation can now be understood based on the formalism of this section. In particular, the contract combinator "`Pair` $\texttt{S}_1$ $\texttt{S}_2$" abbreviates the contract for Church-encoded pairs "$(\texttt{S}_1 \to \texttt{S}_2 \to \texttt{Any}) \to \texttt{Any}$", and generalizes to a `Record` combinator that supports n-ary tuples.

The specification of temporal properties involves some subtleties, which we illustrate by considering various contracts for a lock object with `acquire` and `release` methods. Our initial contract states that `acquire` and `release` are atomic, and calls must alternate between these functions, with `acquire` being called first.

```
LockContract =
  Record acquire : Unit → Unit
         release : Unit → Unit
  where ( call(acquire,_) ret(acquire,_)
          call(release,_) ret(release,_) )*
```

This contract states the "good" behavior of this module. This kind of specification is not *stable under extension*, since adding and calling other methods from this module would violate this contract.

It is often safer to enumerate "bad" behavior, so that all unmentioned good behavior is allowed—including good behavior not yet implemented. The following contract reads, "Do not call `acquire` twice without an interleaving `release`, and do not call `release` twice without an interleaving `acquire`."

```
where not ... call(acquire,_)
              !call(release,_)* call(acquire,_)
  and not ... call(release,_)
              !call(acquire,_)* call(release,_)
```

We must include the initial "..." in the negations, because otherwise we are only disallowing traces that *start* with the illegal sequence.

Unfortunately, this contract does not specify that `acquire` must be called first. We rephrase the property as "After calling `acquire`, you may not call it again until you call `release`, after which you may not call `release` and so on.":

```
where ( call(acquire,_) !call(acquire,_)*
        call(release,_) !call(release,_)* )*
```

This version is extensible, because our use of event negation matches functions that have yet to be written. Unfortunately, it constrains future versions of the module so that `acquire` must be the first call. If we simply rotate the sequence, we avoid that problem:

```
where ( !call(release,_)*  call(acquire,_)
        !call(acquire,_)*  call(release,_) )*
```

This version reads, "It is illegal to call `release` until you `acquire`, and you may not call `acquire` twice before calling `release`." Note that this specification constrains only the client, not the lock object itself. We conjoin the following atomicity requirement to complete our specification.

```
and not ... call(acquire,_) !ret(acquire,_)
and not ... call(release,_) !ret(release,_)
```

8. Implementation

Our presentation so far addresses an idealized language. To evaluate this approach in practice, we have extended and implemented this design for the Racket programing language [22]. Our implementation includes both the programmatic monitor interface from Section 5 and the declarative HOT contract language of Section 7.

Guard/Monitor Implementation Racket already provide a higher-order contract system with a variety of contract combinators that have been widely used to document Racket's libraries, but which do not support temporal properties directly.

Racket also provides a programmatic `make-contract` interface, which creates a contract from a user-provided *projection function* e_{proj} that is expected to satisfy the requirement $[\![e_{proj}]\!] \subseteq [\![\lambda x.\, x]\!]$ (ignoring blame issues for simplicity). Since `make-contract` does not check that e_{proj} is actually a projection, it cannot guarantee non-interference.

Our implementation uses this `make-contract` interface as a foundation on which to implement our guard/monitor architecture. This design allows monitors to interoperate with the Racket language and to annotate module boundaries, while also guaranteeing non-interference.

Our implementation addresses a number of additional issues not considered in the λ-caculus model. For example, the presence of first-class continuations allows functions to return multiple times, or never return, and so the resulting event traces do not satisfy the well-bracketed property. In order to properly match calls and returns, call events include a unique label, which is then repeated in

each matching return event. The implementation works seamlessly with mutation, allowing monitors to interpose between accesses and updates to mutable structures—which are just differently-labeled events in the trace grammar. The implementation's monitors are safe with respect to concurrent interaction via a kill-safe manager thread [21].

Declarative Contract Implementation Our HOT contract language and implementation are almost exactly as described in section 7.2, although the implementation is more general. In particular, as well as the structural contract forms of section 7.2, it also accepts Racket's previous non-temporal contract language (where these contracts do not produce events for the temporal portion).

We also provide a facility for defining macros to abbreviate common temporal constraints, such as the following atomic and transient macros. A function is *atomic* if it returns immediately after it is called, with no intermediate interactions that are visible to the contract; meaning no interactions to explicitly labeled functions, since we do not do deep-tracing by default:

$$\texttt{atomic(f)} \stackrel{\text{def}}{=} \texttt{not ... call(f,\_) !ret(f,\_)}$$

A higher-order argument `g` is *transient* when passed to a function `f` if `g` can only be called before `f` returns—that is, `f` does not get a permanent capability to call `g`, but only a transient capability.

$$\texttt{transient(g,f)} \stackrel{\text{def}}{=} \texttt{not ... ret(f,\_) ... call(g,\_)}$$

The syntax $R_1 \cap R_2$ is introduced via a macro that expands to $\texttt{not}\,(\texttt{not}\,R_1) \cup (\texttt{not}\,R_2)$. Other macros including optional events, bounded-length sequences, etc.

Performance Evaluation The performance overhead of contracts depends critically on the amount of work performed by the application within a contract boundary. To avoid this application dependence, we compared the performance of temporal contracts with Racket's current contract system using the following micro-benchmark. We applied a contract to the identity function that checks for integer arguments and results. The following results measure the time to call the resulting function 10^6 times, under three different contract implementations:

- 870ms for Racket's current contract system.
- 877ms for the new monitor interface.
- 854ms for the HOT contract language.

The results demonstrate that, when the additional expressiveness of temporal contracts is not used, our proposed contract system provides equivalent performance to Racket's current non-temporal contract implementation. Furthermore, both the programmatic monitor interface and the HOT contract language provide equivalent performance.

We investigated the overhead of temporal constraints by extending the contract to check that the called function is atomic. The resulting performance numbers are:

- 931ms for the new monitor interface when enforcing atomicity.
- 1435ms for the HOT contract language when enforcing atomicity, using an optimized implementation that does not support dependent sequencing.
- 2371ms for the HOT contract language when enforcing atomicity, using a more general implementation that supports dependent sequencing.

These results indicate that the monitor architecture provides an efficient mechanism for enforcing simple temporal properties with little additional overhead.

Our current implementation of temporal automata for the HOT contract language is quite general and so a little slow. Restricting temporal constraints to disallow dependent sequences improves performance significantly, and we conjecture that other plausible optimizations could achieve performance comparable to the monitor-based implementation. In theory, it should be possible to optimize HOT contract patterns to the equivalent hand-coded monitor, at least for common specifications such as atomicity, but this remains for future work.

Monitoring Filesystem Accesses Experience with Racket's current contract system indicate its overhead is quite adequate for monitoring the vast majority of APIs, where the work inside the contract boundary dominates the contract overhead, and so contracts have little impact on performance. We conjecture that many temporal contracts would similarly have no performance impact. To evaluate this hypothesis, we modified a GUI text editor to use a temporal contract to monitor its filesystem accesses. As expected, we found no change in the user experience – in particular, the difference in overall application runtime was unmeasurable in benchmarking runs.

Temporal Contracts for the Standard Library We conducted a study of 600 functions in the Racket standard library to determine how many of these function could benefit from temporal contracts. We found that:

- 519 functions in the standard library are intended to be atomic. For example, the `number?` predicate may be passed a function argument, but it is not permitted to call this argument; instead it should immediately return a boolean result.
- 51 library function are passed *transient* function arguments that the library may call, but only before the library function returns. Examples include `map`, `build-list`, `filter`. None of these library functions are allowed to retain a reference to the argument function after the library function returns.
- 17 library function exhibit the opposite behavior, where they are passed function arguments that should *not* be called *until* the library function returns. These are `in-port`, `in-producer`, `stop-before`, `stop-after`, `make-do-sequence`, `make-custom-hash`, `make-immutable-custom-hash`, `make-weak-custom-hash`, `compose`, `procedure-rename`, `procedure->method`, `procedure-reduce-arity`, `make-keyword-procedure`, `procedure-reduce-keyword-arity`, `negate`, `curry`, `curryr`.
- The remaining 13 library functions have quite unconstrained behavior. Examples include `apply`, `keyword-apply`, and 11 "sequence" functions that may cause additional function calls (because inspecting a sequence can cause lazy code evaluation across a module boundary.)

Note that the issue of *when* a function is called, that is addressed in these temporal specifications, is much more critical in a language with imperative state and reactive behavior than in a purely function language, and our work proposes a means to specify and enforce these temporal aspects of behavior.

Adversarial Defense The contract examples presented so far are useful for identifying and localizing defects in well-intentioned but perhaps buggy code. Contracts are also valuable for enforcing properties at trust boundaries between modules written by different principals.

As an example, we developed an implementation of Tic-tac-toe with adversarial players, where each player provides a function

$$\texttt{turn} : \texttt{Board} \rightarrow \texttt{Board}$$

Here, `Board` is an abstract data type that provides `board-get` and `board-set` methods, and each player's `turn` function is supposed

to update the `Board` by placing an additional mark. The core of the game places complete confidence in the players to follow the rules—namely that they must only take one move and they cannot place their mark over the other player's mark—and this assumption greatly simplifies the implementation of the game core.

Of course, this confidence in each player's code may be misplaced, due to unintentional bugs as well as intentional violations of the game rules. To make each untrusted `turn` function trustworthy, it is monitored by a temporal contract. For example, the following contract states that a turn may not contain two calls to `board-set`.

```
not ... call(board-set,_,_,_,_) !ret(turn,_,_,_)*
        call(board-set,_,_,_,_)
```

Additionally, the board should not observe the same row and column of the board being set twice (by either player) during a game.[6]

```
not ... call(board-set,_,?r,?c,_) !ret(game,_)*
        call(board-set,_, r, c,_)
```

To evaluate this architecture, we wrote a collection of player functions, including textual and graphical interactive players, non-cheating AI players, and cheating AI players. In all cases, the contracts behave as expected—they catch all cheaters, both human and AI.

Note that the game core consists entirely of calls to contracted functions. To evaluate the performance overhead, we measured the time to play a 7 move game between two deterministic non-cheating AI players, both with and without the temporal contracts. The average running time of 100 runs is:

- 247ms for the version without contracts, and
- 255ms for the version with contracts.

This example demonstrates a key software engineering benefit of HOT contracts: the game model can be decoupled from the security policy, simplifying both in the process, while imposing minimal performance overhead in this case study.

9. Related Work

Temporal Constraints Temporal constraints are widely understood to be important aspects on a module's behavior, and prior work has eloquently argued for modeling software modules as players in a formal game [12, 13]. Our event sequences are analogous to the *interface automata* of prior work, and extend those ideas to handle higher-order languages by introducing the notions of variable binding and scope in event sequences. It also addresses dynamic enforcement rather than static verification.

Runtime Verification Much prior work has addressed runtime monitoring of system behaviors. The MOP Framework [33] provides a general runtime monitoring framework that supports multiple languages and logics. Aspect-Oriented Programming [29] is a technique for *weaving* into an existing program additional functionality, including contract checks on module boundaries. AOP has been used in tools that enforce temporal properties specified in LTL, such as Tracematches [39] and J-LO [8]. In other approaches, runtime monitors are synthesized from formal specifications, for example in PathExplorer [25], Eagle [6], and RuleR [7]. Program Trace Query Language (PTQL) [23] expresses temporal properties as SQL-like queries over program traces. Generally speaking, these approaches focus on temporal properties and do not provide explicit support for higher-order functions.

[6] Alternatively, this property could be enforced by a contract that calls `board-get` instead of keeping history information, but that specification does not guarantee non-interference.

Behavioral Contracts Meyer [34, 35, 36] introduced contracts with Eiffel and its contract-oriented design approach. Since then contracts have been used to extend static checking [14, 5] and for runtime monitoring of higher-order programs [18]. These investigations have progressed towards more expressive *dependent contracts* [18, 31, 24, 15]. In contrast, our work moves contract expressiveness in a different—orthogonal—direction where contracts enforce temporal properties.

Our work explicitly enforces that contracts can only observe, but not influence, execution (apart from stopping execution on contract violations). Prior work on dependent contracts has been more lax in this regard, including allowing dependent function contracts to observe the argument value, and even call that value in cases where it is a function [18, 31]. This interpretation of "contracts as code" eventually requires that each contract is considered an additional module in the system, with its own blame label [15].

Our usage of the flexibility of Racket contracts to enforce temporal constraints is unique[7], except in one case: Tov and Pucella [43] implement a contract for affine functions. Of course, affine-ness—allowing at most a single call—is a temporal property that is trivially encoded in our DSL. Their implementation approach is neceassarily similar to ours, although it is clearly limited to a single temporal property.

Game Semantics Much prior work has studied the denotational semantics of higher-order languages, often with the goal of developing fully abstract denotational semantics in which observable equivalence implies denotational equivalence. Game semantics has emerged as an appealing foundation for developing fully abstract denotational models. For example, fully abstract game semantics have been developed for PCF [3, 27] or for languages with features such as call-by-value [4], general references [2], and exceptions [11, 30], to name just a few. Compositional game semantics also facilitate compositional verification [1]. Our work draws deeply on game semantics as a tool to study contract systems and module behavior, but follows a different development that starts with the CSI machine, which provides a connection between operational semantics and game semantics.

10. Discussion and Future Work

We have presented a programmatic monitor architecture for contracts that satisfies the twin goals of trace completeness (all computable, prefix-closed contracts are expressible) and non-interference (contracts cannot influence correct executions). The architecture also provides precise blame assignment, even for complex temporal properties.

Additionally, we presented a declarative contract language that can express a variety of temporal constraints on module behavior that are common in software systems with imperative state, but they are often left undocumented. Our contract language provides a convenient declarative means for explicating these temporal aspects of library interfaces.

We have formalized our ideas in the context of an untyped language that supports rich interactions between modules, including mutual recursion, higher-order functions and callbacks, and both encapsulated and shared imperative state. Extending these ideas to additional language features is an important topic for future work. For compound data structures such as lists or arrays, contracts could be enforced eagerly (as with constants) or lazily (as with functions). In this work, compound data structures are Church-encoded as functions and so checked lazily, but eager checking is an interesting alternative. Other topics for future work include the de-

[7] Based on a search on the publically available Racket code: the core distribution and an online package repository.

velopment of temporal contracts for typed languages and the study of temporal higher-order contracts in a concurrent setting.

Acknowledgements We thank Shriram Krishnamurthi and Michael Greenberg for valuable comments on an earlier draft of this paper, Robby Findler for helpful discussions on temporal contracts, and Matthias Felleisen for exploring temporal contracts for game policy enforcement. We are also grateful to the anonymous reviewers of ICFP 2011 for their constructive feedback. This material is based upon work supported by the National Science Foundation under Grants 1016334 and 0905650.

References

[1] Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. Applying game semantics to compositional software modeling and verification. In *TACAS*, pages 421–435, 2004.

[2] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *LICS*, pages 334–344, 1998.

[3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 1996.

[4] Samson Abramsky and Guy McCusker. Call-by-value games. In *CSL*, pages 1–17, 1997.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, pages 49–69, 2004.

[6] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.

[7] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule Systems for Run-time Monitoring: from EAGLE to RULER. *J Logic Computation*, November 2008.

[8] Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University, November 2005.

[9] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, October 1964.

[10] Manuela Carrillo-Castellon, Jesús García Molina, Ernesto Pimentel, and Israel Repiso. Design by contract in smalltalk. *JOOP*, 9(7):23–28, 1996.

[11] Robert Cartwright, Pierre-Louis Curien, and Matthias Felleisen. Fully abstract semantics for observably sequential languages. *Inf. Comput.*, 111(2):297–401, 1994.

[12] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Foundations of Software Engineering*, pages 109–120, 2001.

[13] Luca de Alfaro and Mariëlle Stoelinga. Interfaces: A game-theoretic framework for reasoning about component-based systems. *Electr. Notes Theor. Comput. Sci.*, 97:3–23, 2004.

[14] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[15] C. Dimoulas, R. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *POPL*, 2011.

[16] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[17] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *POPL*, pages 314–325, 1987.

[18] R. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[19] Robert Findler and Matthias Blume. Contracts as pairs of projections. *Functional and Logic Programming*, pages 226–241, 2006.

[20] Robert Bruce Findler, Matthias Blume, and Matthias Felleisen. An investigation of contracts as projections. Technical report, University of Chicago, 2004.

[21] Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *Programming Language Design and Implementation*, pages 47–58, 2004.

[22] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[23] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. OOPSLA, pages 385–402, 2005.

[24] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *POPL*, 2010.

[25] Klaus Havelund and Grigore Rosu. An overview of the runtime verification tool Java PathExplorer. In *Formal Methods in System Design*, 2003.

[26] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Functional and Logic Programming (FLOPS)*, pages 208–225. Springer-Verlag, 2006.

[27] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[28] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support design by contract, 1998.

[29] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, chapter 10, pages 220–242. 1997.

[30] J. Laird. A fully abstract game semantics of local exceptions. In *Logic in Computer Science*, Washington, DC, USA, 2001.

[31] Blume Matthias and David McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16:375–414, July 2006.

[32] K McFarlane. Design by contract framework.

[33] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 2011. to appear.

[34] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[35] B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice-Hall, 1991.

[36] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[37] Mark Samuel Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[38] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19:173–190, March 2009.

[39] Chris Allan Pavel, Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364. ACM Press, 2005.

[40] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, pages 46–57, 1977.

[41] John C. Reynolds. *The essence of ALGOL*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[42] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21, 1995.

[43] Jesse Tov and Riccardo Pucella. Stateful contracts for affine types. *Programming Languages and Systems*, pages 550–569, 2010.

[44] T. Tuglular, C. A. Muftuoglu, F. Belli, and M. Linschulte. Event-based input validation using design-by-contract patterns. In *Software Reliability Engineering*, pages 195–204, 2009.

[45] Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *POPL*, pages 41–52, 2009.