JFP, 47 pages, 2023.
doi:10.1017/xxxxx

Trace contracts

CAMERON MOY and MATTHIAS FELLEISEN

Northeastern University, Boston, MA, USA
(e-mails: camoy@ccs.neu.edu, matthias@ccs.neu.edu*)*

Abstract

Behavioral software contracts allow programmers to strengthen the obligations and promises that they express with conventional types. They lack expressive power, though, when it comes to invariants that hold across several function calls. Trace contracts narrow this expressiveness gap. A trace contract is a predicate over the sequence of values that flow through function calls and returns. This paper presents a principled design, an implementation, and an evaluation of trace contracts.

1 Multi-call constraints for APIs

Conventional type systems lack the power to express all the obligations and promises that an API imposes on, or promises to, client modules. Some language designers cover this expressiveness gap with contracts (Meyer, 1988, 1992), dubbed *behavioral contracts* in the literature. Simply put, a contract is a Boolean-valued assertion that governs some aspect of an API. Say a programmer wishes to narrow the set of acceptable inputs to a function from integers to primes. A type combined with a contract, say `{p:Int | isPrime p}`, expresses this concisely. A proof assistant might discharge this assertion at compile time or a run-time check might monitor it during execution.

While contracts can easily express logical constraints on function signatures, other constraints pose challenges. Temporal properties in particular are difficult to express. Due to this expressiveness gap, APIs come with sequence diagrams, protocol descriptions, and other informal specifications. The Unix I/O API is a standard example: “open a file before reading from it.” A framework for specifying static-analysis passes may state that the given transfer functions must be monotone. A GUI framework may allow the registration of callback objects and promise to call them back in the order of registration.

This paper presents *trace contracts*, an extension of contract systems that permits the functional specification of constraints across multiple function and method calls. A *trace* reifies the sequence of values that flow through certain interception points of a contract system (Dimoulas et al., 2016), say, function calls. A *trace contract* inspects this reified trace with a predicate that decides whether a property holds.

Concretely, this paper reports two contributions. The first is a principled blueprint of trace contracts (section 4), including the design of a compiler to ordinary contracts with a correctness theorem (section 6). Working through the blueprint points to the central challenge of extending existing systems with trace contracts: on the one hand, specifications

should remain functional, while on the other hand, collecting a trace of values necessarily involves mutable state. Managing this state while maintaining ordinary contract composition is key. Our insight is to separate value-interception time from the point when a value crosses from one component to another.

The second contribution is a practical and efficient implementation of the blueprint in Racket, which could be ported to any other language that satisfies some basic requirements (section 7). The implementation supports both predicates over full traces (as streams) as well as the use of efficient, bespoke data structures. For example, the creator of a static-analysis pass could state the monotonicity obligation as a predicate either across a full trace of all input-output pairs or a special-purpose, tree-based data structure. A performance evaluation shows that the fixed-cost overhead of trace contracts is between 1% and 17% on average (section 8).

2 Pedagogic trace-contract examples

Constraints on sequences of function calls are common. Sometimes these constraints cover just one function, but more commonly they involve several. In a functional language such as Racket, they also govern higher-order functions. This section introduces the Racket implementation of trace contracts with pedagogic examples of such constraints. It demonstrates how the integration of trace contracts with Racket's higher-order contract system facilitates authoring maintainable specifications.

2.1 A naive look at trace contracts

In 2020, a developer reported a bug to Racket's mailing list about the `current-memory-use` function.[1] The documentation states that the function "returns an estimate of the total number of bytes allocated since start up, including bytes that have since been reclaimed by garbage collection" (Flatt and PLT, 2010). Given this description, one might expect that the series of return values from `current-memory-use` would increase over time. However, a memory-consumption plot for a long-running system showed periodic dips.

In a language with a conventional type system, such as Java, this function would have the following signature:

```
// Returns the number of bytes allocated since start up,
// including those deallocated during garbage collection.
int currentMemoryUse();
```

The comment mentions two unchecked constraints. First, the function's result cannot be negative, so `int` is imprecise. In Racket, the API author could improve on this type with a run-time-checked contract such as `(-> natural?)`. This notation denotes the signature of a function that takes no arguments and returns natural numbers. Second, the documentation

[1] https://groups.google.com/g/racket-users/c/xqOY8uevGzE/m/mBtHeq2jAwAJ

implies that every call returns a number that is greater than or equal to the result of all previous calls. Existing contract systems cannot express this constraint easily.

With trace contracts, it is possible to express this second constraint directly:

```
(provide
  (contract-out
   [current-memory-use
    (trace/c ([y natural?])1
      (-> y)2
      (full (y) sorted?)3)]))
```

This contract captures both of the constraints that conventional type systems could not express. As the highlighting and subscripts indicate, a trace contract consists of three parts: (1) a sequence of *trace variable declarations*, including one behavioral contract for each; (2) a contract expression, dubbed the *body contract*; and (3) a sequence of *predicate clauses*, in this example introduced with `full`.

Here, there is a single trace variable, `y`, associated with `natural?`. The body contract is `(-> y)`, which specifies ordinary, single-call constraints placed on values protected by the trace contract. When a client module calls `current-memory-use`, the contract system ensures that the returned value is a natural number and, if so, collects the value in a data structure associated with `y`. This data structure is called a *trace*. Additionally, the trace contract specifies a `full` predicate clause that depends on `y`. For `full`, the trace data structure is a stream. Every time the contract system collects a value in the `y` trace, it applies the function specified in the predicate clause—`sorted?`—to the stream of values. The trace contract fails if `sorted?` returns `false`, indicating a dip in the sequence.

Note that `sorted?` is a pure function in the host language, just like ordinary first-order behavioral contracts. One immediate advantage is that a developer can test contracts like any other piece of code—an important property considering that all code, including specification code, may have bugs. Testing builds confidence in the correctness of the specification itself.

With this contract in place, violations are detected as soon as they occur. Moreover, the trace contract blames the appropriate party for the violation:

```
> (current-memory-use)
100
> (current-memory-use)
200
> (current-memory-use)
; current-memory-use: broke its own contract
;   produced: 0
;   ...
;   blaming: current-memory-use
```

In this interaction, `current-memory-use` returns increasing values for the first two calls. On the third call it produces `0`, causing a contract error. Since the problematic value was

collected from the module that defined `current-memory-use`, the function itself is to blame. Developers confronted with this error message can immediately report a bug in the run-time library, knowing with confidence that their code is not responsible for the fault.

2.2 A less naive look: tolerable performance

In its current form, the `current-memory-use` contract comes with a steep performance cost. While any contract can slow down a program, naive trace contracts can be especially expensive because they execute code every time a value is added to a trace. Programmers should be mindful of this expense. In particular, `sorted?` iterates through the entirety of the `y` trace every time a new value is collected. Thus, checking this trace contract is quadratic in the number of calls to `current-memory-use`. To reduce this overhead, a trace-contract system must hand developers fine-grained control over the trace data structure.

Fine-grained control means that developers can choose a custom representation of the trace instead of the naive, stream data structure. When choosing, a developer must: (1) decide on a data structure, (2) pick an initial value, and (3) supply an operation that incorporates a value into the existing trace representation or signals a failure. This kind of predicate clause is introduced with `accumulate` and the data structure is referred to as the *accumulator*. Note that the function given to `accumulate` is no longer a predicate. Instead, it receives two values: the current accumulator and the newly collected values. It returns the new accumulator on success or a designated failure value otherwise.

For the running example, it suffices to use a single number as the accumulator. A simple comparison between any collected value and the accumulator is enough to enforce the promised behavior:

```
(trace/c ([y natural?])
  (-> y)
  (accumulate 0
    [(y) (λ (acc cur)
           (if (<= acc cur) cur (fail)))]))
```

The `accumulate` clause specifies an initial accumulator value of `0` and an accumulating function. When `y` receives a new value, the latter is applied to the current accumulator and the latest value. If the current accumulator is smaller than the new value, then the new value is returned and becomes the next accumulator.[2] Otherwise, the function's result is `(fail)`, the designated failure value.

Every trace contract can be expressed with `accumulate` instead of `full`. In fact, `full` is just syntactic sugar over an `accumulate` clause with a stream accumulator. While `full` is a useful tool to understand trace contracts conceptually, in practice programmers should almost always use `accumulate` combined with an efficient trace data structure.

[2] If `current-memory-use` were to return a non-numeric result, an error would be raised even without the `natural?` check on `y` because `<=` expects two numbers. The error message, however, would blame the contract itself for violating the precondition of `<=`, instead of `current-memory-use`. Thus, to generate practical error messages, the `natural?` check must remain.

2.3 Checking all calls to one function

Consider a compiler pass that computes a live-variables analysis via fixed-point iteration. The interface to such an analysis, using ordinary contracts, might look like this:

```
(provide
  (contract-out
   ;; The transfer function must be monotonically increasing.
   [live-vars (-> (-> set? set?) label? set?)]))
```

Given a monotonically increasing transfer function and a program label, `live-vars` returns the set of live variables at that label (Nielson et al., 2005). Unlike the simplistic example from the preceding section, this constraint involves a higher-order function. A comment describes the constraint, but it is not enforced. Since an incorrectly computed least fixed point can lead to a silent failure, this problem may be especially difficult to debug.

A trace contract can replace the informal comment, enforcing monotonicity:

```
(provide
  (contract-out
   [live-vars (-> (monotone/c set? set? subset?) label? set?)]))

;; Contract Contract (Set Set -> Boolean) -> Contract
(define (monotone/c dom/c cod/c leq?)
  (trace/c ([x dom/c] [y cod/c])
    (-> x y)
    (accumulate (red-black-tree leq?)
      [(x y) (monotone-func leq?)])))
```

The `monotone/c` function consumes two contracts and a comparison function; it returns a function contract that checks monotonicity with respect to the given comparison function. When a client module imports `live-vars` and invokes it, the highlighted contract is attached to the supplied transfer function. This contract stipulates that the transfer function takes and returns sets and is monotone with respect to set inclusion. During fixed-point iteration, the trace contract observes all input-output pairs of the transfer function and builds an extensional representation of the function. Violations are detected by ensuring that no two input-output pairs fail monotonicity.

While a stream containing all input-output pairs would work, it would be inefficient. An order-aware data representation can reduce the time needed to determine whether monotonicity holds from $O(n^3)$ to $O(n \log n)$, where n is the number of calls to the transfer function. One possible choice is a red-black tree as it can quickly determine the immediate predecessor and successor of an ordered element.[3]

[3] Ordinarily this works only for a total order, not a partial order such as set inclusion. However, since fixed-point iteration always explores comparable elements, a red-black tree is acceptable. A general-purpose contract for monotonicity that supports partial orders would require a different data structure. Assuming that fixed-point iteration climbs the lattice *in order*, as it usually does, a contract like the one from section 2.2 would also work.

Every time the trace contract monitors a new value, it initializes a new accumulator. If `live-vars` is invoked twice, two separate accumulators are created, one for each given transfer function. This policy allows trace contracts to compose sensibly with other contract combinators.

Here is the (curried) function that finishes the definition of `monotone/c`:

```
;; Acc = [Ordered-Dict Set Set]
;; Comparator -> (Acc Set Set -> [Or Acc Fail])
(define ((monotone-func leq?) acc x y)
  (cond
    [(dict-has-key? acc x)
     (if (equal? y (dict-ref acc x)) acc (fail))]
    [else
     (define pred-y (dict-pred acc x))
     (define succ-y (dict-succ acc x))
     (if (and (=> pred-y (leq? pred-y y))
              (=> succ-y (leq? y succ-y)))
         (dict-set acc x y)
         (fail))]))
```

When the transfer function returns, `monotone-func` is applied to the current accumulator `acc`, the latest input `x`, and the latest output `y`. It determines the transfer function's predecessor and successor results for `x` and, if they exist, checks that they properly relate to the current output `y`. Just two comparisons suffice: by transitivity there are no other monotonicity violations. If successful, `monotone-func` returns the next accumulator, relating the new input-output pair in the augmented red-black tree.

2.4 Global initialization of traces

The following warning from Racket's documentation tells developers about an essential constraint that the language does not enforce:

> *"If a key in an equal?-based hash table is mutated (e.g., a key string is modified with string-set!), then the hash table's behavior for insertion and lookup operations becomes unpredictable."*

Time and again, however, programmers—especially novices—fail to heed this warning, experience arbitrary program behavior, and have a difficult time debugging such mistakes. Trace contracts can enforce such constraints:

```
(provide
  (contract-out
   [hash-set hash-set/c]
   [string-set! (-> mutable/c natural? char? void?)]))
```

```
(define-values (hash-set/c mutable/c)
  (trace/c ([t any/c])
    #:global
    (values (-> hash? (list/t 'set t) any/c void?)
            (list/t 'mut t))
    (full (t) not-interfere?)))
```

This trace contract makes use of a few features. First, the body contract produces two values using Racket's `values` function, which allows an expression to return multiple values (Ashley and Dybvig, 1994). Because the property relates different functions, i.e. `hash-set` and `string-set!`, their contracts need to be created within the same `trace/c`. Second, the `#:global` option causes the state of the trace contract to be initialized at *definition* time, not the usual *attachment* time. Without `#:global`, the `hash-set/c` and `mutable/c` contracts would be initialized separately and could never interact. Finally, the `list/t` function alters the given collector to tag incoming values with a symbol. Here, the symbol is used to indicate the operation.

The `not-interfere?` predicate ensures that no key is modified after it becomes a key in a hash table:

```
(define/match (not-interfere? xs)
  [((stream))
   true]
  [((stream* `(mut ,x) xt))
   (not-interfere? xt)]
  [((stream* `(set ,x) xt))
   (and (not (stream-member? xt `(mut ,x)))
        (not-interfere? xt))])
```

2.5 The full grammar of trace contracts

In summary, the trace contract library extends Racket's grammar with a `trace/c` form that constructs trace contracts. Figure 1 displays the extension to Racket's grammar. As the preceding examples motivate, each piece of the trace contract (trace variable declarations, the body contract expression, and predicate clauses) come with enhancements that make the system practical:

Trace Variable Declarations The trace-variable declarations $[x\, e_t]$ determine how many traces the contract creates. Each declaration comes with a contract e_t that governs newly collected values.

Body-Contract Expression When a trace contract is attached to a value, the body-contract expression e_b is evaluated in an environment where trace variables are bound to *collectors*. A *collector* is a contract that gathers values that flow through the corresponding points in the body contract. These points are called *interception points*, e.g., argument or return positions. Once collected, values are added to all dependent trace data structures.

$$
\begin{array}{rcl}
e \in \mathsf{Expr} & = & \ldots \mid (\texttt{trace/c}\ ([x\ e_t]\ \ldots^+)\ o\ e_b\ c\ \ldots^+) \\
o \in \mathsf{Opt} & = & \texttt{\#:global} \mid \varepsilon \\
c \in \mathsf{Clause} & = & (\texttt{accumulate}\ e\ [(x\ \ldots^+)\ e_a]\ \ldots^+) \\
 & \mid & (\texttt{full}\ (x\ \ldots^+)\ e_p) \\
 & \mid & (\texttt{track}\ e\ c\ \ldots^+) \\
e_t, e_b \in \mathsf{Expr}_\kappa & = & \{e \mid e \textit{ evaluates to a contract}\} \\
e_a \in \mathsf{Expr}_a & = & \{e \mid e \textit{ evaluates to an accumulating function}\} \\
e_p \in \mathsf{Expr}_p & = & \{e \mid e \textit{ evaluates to a predicate}\} \\
x \in \mathsf{Var} & &
\end{array}
$$

Fig. 1. The Extended Racket Grammar for Trace Contracts

If `trace/c` comes with the `#:global` option, then the collectors are initialized only once, namely, when the contract is created. The default behavior, as demonstrated in section 2.3, initializes collectors each time the trace contract is attached to a value.

The body-contract expression may produce multiple values, which is useful in conjunction with `#:global`. Programmers should use the `#:global` option when more than one contract must share a trace or multiple traces, as seen in section 2.4.

Predicate Clauses A predicate clause c is responsible for determining how the trace should be updated when a new value is collected and whether the contract is violated. The implementation supports three types: `accumulate`, `full`, and `track`.

The `accumulate` clause consists of several subclauses that determine how the accumulator is updated when a new value is collected. A subclause consists of a dependency specification and an expression e_a, which must evaluate to a function. When a subclause depends on more than one collector, the contract system waits until all values have been collected before applying the function. If a collector receives more than one value before the other collectors are ready, then all but the last are discarded.[4] The corresponding accumulating function must return either an updated accumulator or a value indicating failure.

The `full` clause evaluates the expression e_p to a predicate and applies this predicate to a time-ordered stream of collected values. Instead of triggering when *all* the dependent collectors have new values, the predicate is applied when *any* of the dependent collectors have new values.

The `track` clause augments the error message of other clauses with information about all the parties that contributed values to the trace. Section 7.1 describes this feature in detail.

3 Real-world trace-contract examples

This section provides two real-world examples of trace contracts. The first comes from Racket's drawing library and the second comes from code written as part of the grading infrastructure for an undergraduate course.

[4] Alternative choices are expressible by having multiple `accumulate` subclauses with one dependency each. The accumulator would store collected values and then the accumulating function would determine the policy.

3.1 Reusing trace contracts

Racket comes with a built-in library, `racket/draw`, for drawing images. The library provides a thin wrapper around a low-level graphics API written in C. As such, the wrapper must protect against client behavior that would induce undefined behavior at the C level. One instance of undefined behavior occurs with drawing context (DC) objects.

To produce an image with `racket/draw`, a developer must first choose a DC representing the desired output device. There are many such contexts, but they all share a common interface. Part of this interface is a collection of methods that manages the pages of a document: `start-doc`, `start-page`, `end-page`, `end-doc`. Clients must call these methods in a particular order. It does not make sense to call, e.g., `end-doc` before `start-doc`. Moreover, all drawing commands must occur within a page.

Here is a regular expression that describes a valid *complete* sequence of method calls:

$$\texttt{start-doc}, (\texttt{start-page}, \texttt{draw}^\star, \texttt{end-page})^\star, \texttt{end-doc}$$

This regular expression is not suitable for trace-contract monitoring. A trace contract also checks every *incomplete* sequence of method calls, not just the complete sequence. So, this regular expression has to be adapted to accept any prefix of the complete sequence.

Here is an adapted version of the regular expression above, described using Racket's `automata` library (McCarthy, 2011):

```
(define SINGLE-PAGE
  (re (seq/close 'start-page (star 'draw) 'end-page)))
(define DC-RE
  (re (seq/close 'start-doc (star ,SINGLE-PAGE) 'end-doc)))
```

The `re` form compiles a finite-state automaton that accepts the given regular expression. Within `re`, `seq/close` denotes a regular expression that accepts not just the given sequence, but any prefix of that sequence.

The following trace contract enforces the protocol using `DC-RE`:

```
(provide
  (contract-out [make-ps-dc (-> (dc/c DC-RE))]))

(define (dc/c aut)
  (trace/c ([s symbol?])
    (object/c
      [start-doc  (apply/c [s 'start-doc])]
      [start-page (apply/c [s 'start-page])]
      [draw-point (apply/c [s 'draw])]
      [end-page   (apply/c [s 'end-page])]
      [end-doc    (apply/c [s 'end-doc])])
    (accumulate aut
      [(s) (λ (acc x)
             (define acc* (acc x))
             (if (machine-accepting? acc*) acc* (fail)))])))
```

Given a finite-state automaton, `dc/c` produces a contract for a DC where the method call sequence is governed by the regular expression. In the body of `dc/c`, a trace contract is wrapped around an object contract specifying each of the DC methods. There is only a single collector, `s`, that collects symbols corresponding to the method calls. The `apply/c` combinator provides the collector with a constant value each time a protected method is called. To check the protocol, the trace predicate uses the state of the automaton as the accumulator. So long as the automaton is accepting, the contract is satisfied. The trace contract is then used in the codomain of `make-ps-dc`, which produces PostScript (PS) drawing contexts.

As mentioned before, there is more than one kind of DC. In particular, an Encapsulated PostScript (EPS) drawing context has a slightly different constraint than an ordinary PS context. Since an EPS file is intended to be embedded in a larger document, it can only have a single page. Supporting EPS is easy since `dc/c` abstracts over the regular expression. Checking a different protocol requires only passing in a different regular expression to `dc/c`:

```
(provide (contract-out [make-eps-dc (-> (dc/c EPS-RE))]))

(define EPS-RE
  (re (seq/close 'start-doc ,SINGLE-PAGE 'end-doc)))
```

3.2 Protocols for many methods

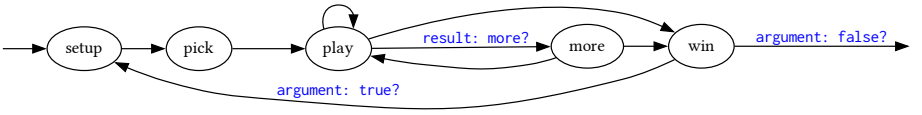
Imagine a board-game framework that pits AI player components against one another. In a typical board game, players (1) receive their game pieces; (2) take turns, which may consist of several interactions with the board; and (3) determine which ones won and lost. Winners of a game can move on to the next round of a tournament while losers are left behind.

A natural implementation of an AI player is as an object with methods that correspond to these game stages. Each player expects that these methods are called in a certain order, which may depend on the state of the game. In short, the methods relate to each other according to a value-dependent, multi-function, temporal property.

Programmers often use state-transition diagrams to document such multi-function protocols. Figure 2 displays a diagram for an AI board-game player (top), together with a matching trace-contract specification (bottom). States in this diagram indicate which method the referee component must call next. Labeled edges represent transitions that depend on either an argument value or a return value. Unlabeled edges represent independent transitions. Since there are several possible transitions for some states, this is a non-deterministic automaton.

Specifically, this diagram dictates that players must implement five methods:

1. A `setup` method that delivers the game pieces.
2. A `pick` method that asks a player to choose some game objectives.
3. A `play` method that grants a player the right to take a turn. The result is either a request to perform an *action* on the game state or a request for *more* game pieces.




```
(provide (contract-out [player-factory (-> strategy/c player/c)]))

(define PLAYER-NFA
  (nfa (setup) (setup pick play more win done)
    [setup ([`(setup ,_)          (pick)])]
    [pick  ([`(pick  ,_)          (play)])]
    [play  ([`(play ,(? action?)) (play win)]
            [`(play ,(? more?))   (play more win)])]
    [more  ([`(more ,_)           (play win)])]
    [win   ([`(win ,true)         (setup)]
            [`(win ,false)        (done)])]
    [done  ()]))

(define player/c
  (trace/c ([x any/c])
    (object/c
      [setup (->m game-map?       (list/t 'setup x))]
      [pick  (->m set?            (list/t 'pick x))]
      [play  (->m state?          (list/t 'play x))]
      [more  (->m list?           (list/t 'more x))]
      [win   (->m (list/t 'win x) any/c)])
    (accumulate PLAYER-NFA
      [(x) (λ (acc x)
             (define acc* (acc x))
             (if (machine-accepting? acc*) acc* (fail)))])))
```

Fig. 2. The state-machine contract for AI players, with a transition diagram

4. If the referee gets this second kind of request in response to `play`, it may invoke the player's `more` method. But, it may also skip this call, depending on the game state.
5. The player is granted turns and more pieces until the referee discovers an end-game condition and then informs the player whether it won or lost. The player may participate in the next game only if `win` is called with `true`.

In this particular software system, a factory function creates AI players from a strategy and returns player objects that implement the above five methods. The contract on this factory method attaches a trace contract to each player object. As a result, every instance of the player class must obey the order of method calls specified in the sequence diagram. Otherwise, the system raises an error with a blame-assignment message that informs the

developer of the player that was mistreated; once again, the trace-contract design greatly benefits from a tight integration with higher-order behavioral contracts.

This protocol is a language over the alphabet containing the names of methods, along with the specific arguments or return values of two of them: `play` and `win`. For example, the following sequence of method calls is correct so long as `play` returned a value satisfying the `more?` predicate: `setup`, `pick`, `play`, `more`. If `play` returned a value satisfying `action?` instead, then that sequence of method calls is invalid, and a contract error should be raised on the call to `more`.

To check this protocol, the trace contract once again simulates the finite-state machine with `accumulate`. Unlike the automaton in section 3.2, this machine inspects pieces of data. For `setup`, `pick`, and `more`, the transition is independent of run-time values. However, `play` and `win` have value-dependent transitions. For example, `play` uses the `action?` and `more?` predicates to determine the next set of states. It does so using Racket's `(? p)` match pattern, which matches a value if the predicate `p` holds.

3.3 Contracts are better than ad-hoc checks

As mentioned previously, these two examples come from real-world projects. In the original code, both contained ad-hoc protocol checks instead of trace contracts. Given that, it is worth reviewing why contracts are preferable to such handwritten checks:

1. Contracts cleanly separate specification code and implementation code—with ad-hoc checks the two are intertwined. This makes programs difficult to read, and thus hard to maintain (Meyer, 1988, 1992). Additionally, the code needed to correctly check a specification is often repetitive and tedious. Getting it wrong is inevitable.
2. As a direct consequence of separating specification and implementation, contracts enable static and dynamic analyses. For example, the contract library supports profiling (Andersen et al., 2018) to determine which contracts are slowing down a program. Static techniques (Nguyễn et al., 2018) can verify whether a program satisfies a contract. These kinds of tools are impossible with ad-hoc checks.
3. The contract library automatically supports detailed error messages with blame that points to the module that violated the contract. This information is exceptionally useful for debugging (Lazarek et al., 2020).
4. Programmers have fine-grained control over the scope of a contract, i.e., which modules get checks and which ones do not. Trusted modules may not need checks. Thus, the balance between correctness and performance can be tuned precisely. This also allows tools to automatically bypass contracts in certain cases, for instance, when they are statically proven to be unnecessary (Moy et al., 2021).
5. Finally, contracts permit specification reuse. In section 3.1, repetitive blocks of ad-hoc checking code are replaced with `make-ps-dc` and `make-eps-dc`; abstracting over the contract eliminates duplicate code.

Λ Surface Syntax

$$
\begin{array}{rcl}
e \in \mathsf{Expr} & = & b \mid x \mid f \mid o\,e \mid \texttt{if}\,e\,e\,e \\
& & \mid e\,e \mid \texttt{queue} \mid \texttt{add!}\,e\,e \\
b \in \mathsf{Bool} & = & \texttt{true} \mid \texttt{false} \\
f \in \mathsf{Fun} & = & \lambda x.e \\
o \in \mathsf{Op} & = & \texttt{null?} \mid \texttt{head} \mid \texttt{tail} \\
x, y, z \in \mathsf{Var} & &
\end{array}
$$

Λ Evaluation Syntax

$$
\begin{array}{rcl}
\varsigma \in \mathsf{Conf} & = & \langle e, \sigma \rangle \\
e \in \mathsf{Expr} & = & \ldots \mid \alpha \mid \texttt{err}_j^k \\
v \in \mathsf{Val} & = & b \mid f \mid \alpha \\
E \in \mathsf{Ctx} & = & \square \mid o\,E \mid \texttt{if}\,E\,e\,e \mid E\,e \\
& & \mid v\,E \mid \texttt{add!}\,E\,e \mid \texttt{add!}\,v\,E \\
u \in \mathsf{SVal} & = & \texttt{null} \mid \texttt{cons}\,v\,\alpha \\
\sigma \in \mathsf{Store} & = & \mathsf{Addr} \rightarrow_{\text{fin}} \mathsf{SVal} \\
\alpha \in \mathsf{Addr} & & \\
j, k, l \in \mathsf{Lab} & &
\end{array}
$$

Fig. 3. Surface and Evaluation Syntax of Λ

4 A model of trace contracts

A design requires a rigorous blueprint so that implementors of other languages can understand the idea and adapt it. This section presents a model of the λ-calculus extended with trace contracts. To keep the formalism accessible, the model is developed and explained incrementally using five languages: Λ, Λ_B, Λ_C, Λ_T, Λ_U. Additionally, some of the pragmatic features of section 2 have been omitted to reduce the complexity of the final model. Section 5 presents formal properties of these models.

4.1 A functional base

Figure 3 (left) defines the surface syntax of Λ, the call-by-value λ-calculus (Plotkin, 1975) extended with Booleans and mutable queues. The final model represents traces using queues. The nullary constructor `queue` builds a new instance and `add!` puts an element into a queue. Primitive operations allow functions to walk over queues similar to immutable lists. All the remaining syntax is standard.

Figure 3 (right) defines the evaluation syntax of Λ. Along with a grammar of values and evaluation contexts, the syntax contains errors and queue-specific stores.

Errors come with two labels: j names the party that specified the violated contract and k names the party that violated the contract. There are two special labels: ○ refers to the language runtime itself and † refers to the read-eval-print-loop (REPL). Since Λ does not have user-defined contracts, the only possible error is $\texttt{err}_{\circ}^{\dagger}$.

Stores map addresses to either an empty queue (`null`) or a cons cells that combines a head value with an address containing the remaining elements. This choice facilitates functional iteration over queues.

Next, figure 4 defines the reduction relation for Λ with the supporting metafunctions provided in figure 5. Conditionals and application are standard. For functional primitive operations, the δ metafunction (Barendregt, 1981) is used to compute the result. Constructing a new queue uses the next free address in the store and sets it to the empty queue. Adding to an existing queue updates the store, replacing the empty queue at the end

Λ Reduction Relation

$$\langle E[\texttt{if}\ v\ e_1\ e_2], \sigma\rangle \longmapsto \langle E[e_1], \sigma\rangle \text{ if } v \neq \texttt{false} \qquad \textsc{If-True}$$

$$\langle E[\texttt{if false}\ e_1\ e_2], \sigma\rangle \longmapsto \langle E[e_2], \sigma\rangle \qquad \textsc{If-False}$$

$$\langle E[(\lambda x.e)\ v], \sigma\rangle \longmapsto \langle E[e[v/x]], \sigma\rangle \qquad \textsc{App}$$

$$\langle E[o\ v], \sigma\rangle \longmapsto \langle E[\delta(o, v, \sigma)], \sigma\rangle \qquad \textsc{Prim}$$

$$\langle E[\texttt{queue}], \sigma\rangle \longmapsto \langle E[\alpha], \sigma[\alpha \mapsto \texttt{null}]\rangle \text{ if } \alpha = \mathsf{next}(\sigma) \qquad \textsc{Queue}$$

$$\langle E[\texttt{add!}\ \alpha\ v], \sigma\rangle \longmapsto \langle E[\alpha], \mathsf{add}(\sigma, \alpha, v)\rangle \qquad \textsc{Add!}$$

$$\langle E[v_f\ v], \sigma\rangle \longmapsto \langle E[\texttt{err}^{\dagger}_{\circ}], \sigma\rangle \text{ if } v_f \notin \mathsf{Fun} \qquad \textsc{Err-App}$$

$$\langle E[\texttt{add!}\ v_q\ v], \sigma\rangle \longmapsto \langle E[\texttt{err}^{\dagger}_{\circ}], \sigma\rangle \text{ if } v_q \notin \mathsf{Addr} \qquad \textsc{Err-Add!}$$

$$\langle E[\texttt{err}^{k}_{j}], \sigma\rangle \longmapsto \langle \texttt{err}^{k}_{j}, \sigma\rangle \text{ if } E \neq \square \qquad \textsc{Err-Unwind}$$

Fig. 4. Reduction Relation of Λ

$$\mathsf{add}(\sigma, \alpha, v) = \begin{cases} \sigma[\alpha \mapsto \texttt{cons}\ v\ \alpha'][\alpha' \mapsto \texttt{null}] & \text{if } \alpha' = \mathsf{next}(\sigma), \sigma(\alpha) = \texttt{null} \\ \mathsf{add}(\sigma, \alpha', v) & \text{if } \sigma(\alpha) = \texttt{cons}\ v_h\ \alpha' \end{cases}$$

$$\mathsf{next}(\sigma) = \mathsf{max}(\mathsf{dom}(\sigma)) + 1$$

$$\delta(o, v, \sigma) = \begin{cases} \texttt{true} & \text{if } o = \texttt{null?}, \sigma(v) = \texttt{null} \\ \texttt{false} & \text{if } o = \texttt{null?}, \sigma(v) = \texttt{cons}\ v\ \alpha' \\ \texttt{err}^{\dagger}_{\circ} & \text{if } o = \texttt{head}, \sigma(v) = \texttt{null} \\ v & \text{if } o = \texttt{head}, \sigma(v) = \texttt{cons}\ v\ \alpha' \\ \texttt{err}^{\dagger}_{\circ} & \text{if } o = \texttt{tail}, \sigma(v) = \texttt{null} \\ \alpha' & \text{if } o = \texttt{tail}, \sigma(v) = \texttt{cons}\ v\ \alpha' \\ \texttt{err}^{\dagger}_{\circ} & \text{if } v \notin \mathsf{Addr} \end{cases}$$

Fig. 5. Metafunctions of Λ

with a cons cell containing the new value. The last three rules deal with error conditions. Errors to do with primitive operations are handled by δ itself.

4.2 The classic contract model

Figure 6 defines the surface and evaluation syntax for Λ_B, a model of higher-order contracts based on that of Dimoulas and Felleisen (2011) and Dimoulas et al. (2011). The surface syntax extends Λ with two new elements: dependent function contracts $e_d \to_i e_c$ and monitors $\texttt{mon}^{k,l}_{j}\ e_\kappa\ e_c$. A dependent function contract can describe properties of functions where the codomain contract depends on the argument to the protected function.[5] A monitor is then used to attach a contract to a value. So, $\texttt{mon}^{k,l}_{j}\ e_\kappa\ e_c$ attaches e_κ to e_c. The value of e_c

[5] This paper uses the abbreviation $e_d \to e_c$ to stand for an independent function contract, i.e., $e_d \to_i (\lambda\_.e_c)$.

Λ_B Surface Syntax extends Λ

$$e \in \mathsf{Expr} = \ldots \mid e \to_i e \mid \mathtt{mon}_j^{k,l}\, e\, e$$
$$j, k, l \in \mathsf{Lab}$$

Λ_B Evaluation Syntax extends Λ

$$v \in \mathsf{Val} = \ldots \mid \kappa$$
$$\kappa \in \mathsf{Con} = b \mid \lambda x.e \mid v \to_i v$$
$$E \in \mathsf{Ctx} = \ldots \mid E \to_i e \mid v \to_i E \mid \mathtt{mon}_j^{k,l}\, E\, e \mid \mathtt{mon}_j^{k,l}\, v\, E$$

Fig. 6. Surface and Evaluation Syntax of Λ_B

is dubbed the *carrier* of the contract. Monitors also come with labels naming the parties that agreed to the contract: the contract-defining module j, the server module k, and the client module l.

In addition to dependent function contacts, the evaluation syntax reveals that Booleans and functions can be used as contracts. When used as a contract, `true` permits any value and `false` forbids all values. These correspond to Racket's `any/c` and `none/c` contracts, respectively. When used as a contract, a function checks first-order properties of the carrier. This corresponds to Racket's flat contracts.

Here is an example program with a contract:[6]

$$\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib,main}}\ (\mathtt{true} \to_i (\lambda x.\lambda y.x = y))\ (\lambda z.z) \tag{4.1}$$

This example contains a contract fully specifying the behavior of the identity function. Since the domain contract is `true`, every argument is accepted. When the function returns, the output value is checked against the codomain contract $\lambda y.x = y$, ensuring that it is equal to the input value.

Figure 7 shows the reduction relation for Λ_B. The first four rules describe the checks performed by each kind of contract. For `true` and `false`, the check immediately succeeds or immediately fails, respectively. For a flat contract $\lambda x.e$, the result of applying this function to the carrier is then used as the new contract. Thus, if $\lambda x.e$ is a predicate, this corresponds exactly to a first-order check because `true` and `false` are themselves contracts.

While $\lambda x.e$ may return a Boolean, there is nothing in the semantics that forces it to be one. In particular, it could return a function contract. This can be used to create *cascading contracts* that combine arbitrary first-order checks with higher-order contracts.

Consider this example:

$$\lambda f.\mathtt{if}\ (\mathtt{arity}\ f = 1)\ (\mathtt{int?} \to \mathtt{int?})\ \mathtt{false}$$

Assuming an arity primitive, this cascading contract checks a first-order constraint, namely that the carrier has arity one. If successful, the higher-order contract $\mathtt{int?} \to \mathtt{int?}$ protects the carrier. Otherwise, the contract fails.

In Racket, function contracts perform arity checks eagerly, exactly in this manner. The model from Dimoulas and Felleisen (2011) cannot encode this behavior. Cascading contracts are essential for defining the compiler in section 6.

[6] These example programs are intended to illustrate a point, and therefore may use language features that are not formally defined. The meaning should always be clear from context.

Λ_B Reduction Relation extends Λ

$$\langle E[\mathtt{mon}_j^{k,l}\ \mathtt{true}\ v], \sigma\rangle \longmapsto \langle E[v], \sigma\rangle \qquad \textsc{Mon-True}$$

$$\langle E[\mathtt{mon}_j^{k,l}\ \mathtt{false}\ v], \sigma\rangle \longmapsto \langle E[\mathtt{err}_j^k], \sigma\rangle \qquad \textsc{Mon-False}$$

$$\langle E[\mathtt{mon}_j^{k,l}\ (\lambda x.e)\ v], \sigma\rangle \longmapsto \langle E[\mathtt{mon}_j^{k,l}\ ((\lambda x.e)\ v)\ ], \sigma\rangle \qquad \textsc{Mon-Flat}$$

$$\langle E[\mathtt{mon}_j^{k,l}\ (v_d \to_i v_c)\ v], \sigma\rangle \longmapsto \langle E[\lambda x.\mathtt{let}\ x_j = \mathtt{mon}_j^{l,j}\ v_d\ x\ \mathtt{in} \qquad \textsc{Mon-Fun}$$
$$\mathtt{let}\ x_k = \mathtt{mon}_j^{l,k}\ v_d\ x\ \mathtt{in}$$
$$\mathtt{mon}_j^{k,l}\ (v_c\ x_j)\ (v\ x_k)], \sigma\rangle$$

$$\langle E[\mathtt{mon}_j^{k,l}\ v_\kappa\ v], \sigma\rangle \longmapsto \langle E[\mathtt{err}_\circ^\dagger], \sigma\rangle \text{ if } v_\kappa \notin \mathsf{Con} \qquad \textsc{Err-Mon}$$

Fig. 7. Reduction Relation of Λ_B

Finally, Mon-Fun describes the indy semantics of dependent function contracts (Dimoulas et al., 2011). The key insight of indy is that the contract itself can be inconsistent, and therefore must be subject to checks.

Here is an example that illustrates this point:

$$(\mathtt{bool?} \to \mathtt{bool?}) \to_i (\lambda f.f\ 42)$$

While the domain contract states that the input is a function over Booleans, generating the codomain contract violates that assumption by applying f to a number. In this case, indy raises an error blaming the contract itself.

4.3 A revised contract model

As is, Λ_B cannot accommodate contracts with effects, such as trace contracts. When used as the domain of a function, a contract's effects are erroneously duplicated.

Take the following variation on program (4.1):

$$\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ (\ (\lambda x.\mathtt{print}\ x\ ;\mathtt{true})\ \to_i (\lambda x.\lambda y.x = y))\ (\lambda z.z)$$

The only difference is the presence of an effect in the domain contract. As the following reduction sequence demonstrates, `print` is executed twice:

$$\langle(\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ ((\lambda x.\mathtt{print}\ x\ ;\mathtt{true}) \to_i (\lambda x.\lambda y.x = y))\ (\lambda z.z))\ 42, \emptyset\rangle$$

By Mon-Fun, the monitor produces a wrapper function that checks the arguments against the domain contract and the return value against the codomain contract.

$$\longmapsto \langle(\lambda x.\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ ((\lambda x.\lambda y.x = y)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{ctc}}\ (\lambda y.\mathtt{print}\ y\ ;\mathtt{true})\ x))$$
$$((\lambda z.z)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{lib}}\ (\lambda y.\mathtt{print}\ y\ ;\mathtt{true})\ x)))\ 42, \emptyset\rangle$$

The wrapper function is applied to 42.

$$\longmapsto \langle\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ ((\lambda x.\lambda y.x = y)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{ctc}}\ (\lambda y.\mathtt{print}\ y\ ;\mathtt{true})\ 42))$$
$$((\lambda z.z)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{lib}}\ (\lambda y.\mathtt{print}\ y\ ;\mathtt{true})\ 42)), \emptyset\rangle$$

To produce the codomain contract, the argument is first checked against the domain contract with the contract-defining party (`ctc`) as the client label. This prints 42.

$$\longmapsto^{+} \langle \mathtt{mon}^{\mathtt{lib},\mathtt{main}}_{\mathtt{ctc}}\ ((\lambda x.\lambda y.x = y)\ 42)\ ((\lambda z.z)\ (\mathtt{mon}^{\mathtt{main},\mathtt{lib}}_{\mathtt{ctc}}\ (\lambda y.\mathtt{print}\ y\ ;\mathtt{true})\ 42)), \emptyset \rangle$$

Once the argument is checked, the codomain contract can be created.

$$\longmapsto \langle \mathtt{mon}^{\mathtt{lib},\mathtt{main}}_{\mathtt{ctc}}\ (\lambda y.42 = y)\ ((\lambda z.z)\ (\mathtt{mon}^{\mathtt{main},\mathtt{lib}}_{\mathtt{ctc}}\ (\lambda y.\mathtt{print}\ y\ ;\mathtt{true})\ 42)), \emptyset \rangle$$

The argument has to be checked against the domain contract once more. This time the client label is `lib`. Again, 42 is printed.

$$\longmapsto^{+} \langle \mathtt{mon}^{\mathtt{lib},\mathtt{main}}_{\mathtt{ctc}}\ (\lambda y.42 = y)\ ((\lambda z.z)\ 42), \emptyset \rangle$$

Now the carrier is applied to 42. Since the carrier is the identity function, it returns 42.

$$\longmapsto^{+} \langle \mathtt{mon}^{\mathtt{lib},\mathtt{main}}_{\mathtt{ctc}}\ (\lambda y.42 = y)\ 42, \emptyset \rangle$$

The returned value is checked against the generated codomain contract. In this case, the contract is satisfied and is discharged.

$$\longmapsto \langle 42, \emptyset \rangle$$

Effect duplication is a major problem for trace contracts. If a collector is used as the domain of a function, then it will collect duplicate values.

To understand the source of the problem, consider the contractum of MON-FUN. It contains two v_d monitors that differ only in their client label: one uses j and the other uses k. A simple `let` binding cannot be used to eliminate the duplicated effect since each of the monitors may produce wrappers that contain different labels.

The conclusion to draw is that Λ_B conflates *interception time* and *crossing time*. Interception time occurs when the contract system intercepts a value from the monitored program, i.e., when a value flows through an interception point. Crossing time occurs when an intercepted value moves to another component.

Consider a wrapper for the contract $v_d \to v_c$. Every time the wrapper is applied, it must perform two tasks related to the argument. First, v_d must be used to check first-order properties of the argument. Second, if v_d is a higher-order contract, wrappers must be created for every client of the argument. In the case of indy, there are two such clients, by convention labeled j and l. Interception time corresponds to when task one occurs and crossing time corresponds to when task two occurs.[7] Since Λ_B has only one `mon` form, both tasks are its responsibility.

Splitting the three-labeled monitor into two forms separates these responsibilities. Figure 8 defines the syntax of Λ_C, a revised contract language. While the surface syntax

[7] Often, interception-time coincides with first-order checks and crossing-time coincides with higher-order wrapping. There are exceptions, however. For example, in Racket the `unconstrained-domain->` contract makes no demand on function arguments. Because such a contract is guaranteed never to blame clients, its wrapper can be constructed at interception time. For simplicity, though, this paper blurs the distinction.

$\boxed{\Lambda_C \text{ Surface Syntax}}$ extends Λ

$$e \in \mathsf{Expr} = \ldots \mid e \to_i e \mid \mathtt{mon}_j^{k,l}\, e\, e$$
$$j, k, l \in \mathsf{Lab}$$

$\boxed{\Lambda_C \text{ Evaluation Syntax}}$ extends Λ

$$\begin{aligned}
e \in \mathsf{Expr} &= \ldots \mid \mathtt{mon}_j^k\, e\, e \mid \mathtt{grd}_j^k\, \omega\, v \\
&\mid e \cdot l \\
v \in \mathsf{Val} &= \ldots \mid \kappa \mid \mathtt{grd}_j^k\, \omega\, v \\
\kappa \in \mathsf{Con} &= b \mid \lambda x.e \mid v \to_i v \\
\omega \in \mathsf{Wrap} &= \mathtt{true} \mid v \to_i v \\
E \in \mathsf{Ctx} &= \ldots \mid E \to_i e \mid v \to_i E \\
&\mid \mathtt{mon}_j^k\, E\, e \mid \mathtt{mon}_j^k\, v\, E \mid E \cdot l
\end{aligned}$$

Fig. 8. Surface and Evaluation Syntax of Λ_C

is the same as Λ_B, the evaluation syntax has a few differences (highlighted): two-labeled monitors $\mathtt{mon}_j^k\, e_\kappa\, e_c$, guarded values $\mathtt{grd}_j^k\, \omega\, v$, and label applications $e_g \cdot l$. Reduction of $\mathtt{mon}_j^k\, e_\kappa\, e_c$ corresponds to interception time, when first-order properties of the carrier are checked. Reduction of $(\mathtt{grd}_j^k\, \omega\, v) \cdot l$ corresponds to crossing time and produces a wrapper for client l.

Figure 9 displays the reduction relation for Λ_C. The first rule, MON-APPLY, decomposes the surface-level monitor into a two-labeled monitor applied to the client label. If successful, the two-labeled monitor produces a guarded value. The next four rules are responsible for the first-order checks of each contract. In the case of MON-TRUE and MON-FALSE, the first-order check is all that needs to occur.

Below the monitor rules, there are two rules for guarded values: GRD-TRUE and GRD-FUN. For `true` there is no wrapper needed so the carrier is produced directly. A wrapper is needed for function contracts, though. The wrapper in the contractum of GRD-FUN exploits the two-stage process. Instead of two v_d monitors, there is now only one, with its result bound to x_g. Effects caused by checking v_d occur only once while binding x_g. In the scope of this `let` binding, two wrappers are produced by applying x_g to the two client labels. Constructing these wrappers is not effectful.

4.4 The trace contract model

Finally, figure 10 defines the trace contract model Λ_T that extends Λ_C. The surface syntax contains only one new form: $\mathtt{tr}\, e_\kappa\, e_p$. This represents a trace contract with body-contract constructor e_κ and trace predicate e_p. A body-contract constructor is a function that, when provided with a collector, returns the body contract. The evaluation syntax contains one new form: $\mathtt{co}\, \alpha\, v_p$. This represents a collector with trace address α and trace predicate v_p.

The reduction relation for Λ_T is presented in figure 11. MON-TRACE performs two tasks. First, it allocates a queue for storing the trace. Second, it creates a collector and provides it to the body-contract constructor. MON-COL produces code that adds a new value to the trace and checks it using the trace predicate.

$\boxed{\Lambda_C \text{ Reduction Relation}}$ extends Λ

$$\langle E[\mathtt{mon}^{k,l}_j\ e_\kappa\ e], \sigma\rangle \longmapsto \langle E[(\mathtt{mon}^{k}_j\ e_\kappa\ e)\cdot l], \sigma\rangle \qquad \textsc{Mon-Apply}$$

$$\langle E[\mathtt{mon}^{k}_j\ \mathtt{true}\ v], \sigma\rangle \longmapsto \langle E[\mathtt{grd}^{k}_j\ \mathtt{true}\ v], \sigma\rangle \qquad \textsc{Mon-True}$$

$$\langle E[\mathtt{mon}^{k}_j\ \mathtt{false}\ v], \sigma\rangle \longmapsto \langle E[\mathtt{err}^{k}_j], \sigma\rangle \qquad \textsc{Mon-False}$$

$$\langle E[\mathtt{mon}^{k}_j\ (\lambda x.e)\ v], \sigma\rangle \longmapsto \langle E[\mathtt{mon}^{k}_j\ ((\lambda x.e)\ v)\ v\,], \sigma\rangle \qquad \textsc{Mon-Flat}$$

$$\langle E[\mathtt{mon}^{k}_j\ (v_d \to_i v_c)\ v], \sigma\rangle \longmapsto \langle E[\mathtt{grd}^{k}_j\ (v_d \to_i v_c)\ v], \sigma\rangle \qquad \textsc{Mon-Fun}$$

$$\langle E[(\mathtt{grd}^{k}_j\ \mathtt{true}\ v)\cdot l], \sigma\rangle \longmapsto \langle E[v], \sigma\rangle \qquad \textsc{Grd-True}$$

$$\langle E[(\mathtt{grd}^{k}_j\ (v_d \to_i v_c)\ v)\cdot l], \sigma\rangle \longmapsto \langle E[\lambda x.\mathtt{let}\ x_g = \mathtt{mon}^{l}_j\ v_d\ x\ \mathtt{in} \qquad \textsc{Grd-Fun}$$
$$\mathtt{let}\ x_j = x_g\cdot j\ \mathtt{in}$$
$$\mathtt{let}\ x_k = x_g\cdot k\ \mathtt{in}$$
$$\mathtt{mon}^{k,l}_j\ (v_c\ x_j)\ (v\ x_k)], \sigma\rangle$$

$$\langle E[\mathtt{mon}^{k}_j\ v_\kappa\ v], \sigma\rangle \longmapsto \langle E[\mathtt{err}^{\dagger}_{\circ}], \sigma\rangle \text{ if } v_\kappa \notin \mathsf{Con} \qquad \textsc{Err-Mon}$$

Fig. 9. Reduction Relation of Λ_C

$\boxed{\Lambda_T \text{ Surface Syntax}}$ extends Λ_C

$$e \in \mathsf{Expr} = \ldots \mid \mathtt{tr}\ e\ e$$

$\boxed{\Lambda_T \text{ Evaluation Syntax}}$ extends Λ_C

$$e \in \mathsf{Expr} = \ldots \mid \mathtt{co}\ \alpha\ v$$
$$\kappa \in \mathsf{Con} = \ldots \mid \mathtt{tr}\ v\ v \mid \mathtt{co}\ \alpha\ v$$
$$E \in \mathsf{Ctx} = \ldots \mid \mathtt{tr}\ E\ e \mid \mathtt{tr}\ v\ E$$

Fig. 10. Surface and Evaluation Syntax of Λ_T

$\boxed{\Lambda_T \text{ Reduction Relation}}$ extends Λ_C

$$\langle E[\mathtt{mon}^{k}_j\ (\mathtt{tr}\ v_b\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\mathtt{mon}^{k}_j\ (v_b\ (\mathtt{co}\ \alpha\ v_p))\ v], \sigma[\alpha \mapsto \mathtt{null}]\rangle \qquad \textsc{Mon-Trace}$$
$$\text{if } \alpha = \mathsf{next}(\sigma)$$

$$\langle E[\mathtt{mon}^{k}_j\ (\mathtt{co}\ \alpha\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\mathtt{mon}^{k}_j\ (v_p\ (\mathtt{add!}\ \alpha\ v))\ v], \sigma\rangle \qquad \textsc{Mon-Col}$$

Fig. 11. Reduction Relation of Λ_T

Here is a translation of the `current-memory-use` example from section 2.1 into this model:

$$\mathtt{tr}\ (\lambda y.\mathtt{true} \to y)\ \mathtt{sorted?} \tag{4.2}$$

As mentioned earlier, the body-contract constructor consumes a collector k and returns a contract: $\mathtt{true} \to k$. That is, the generated contract does not impose any precondition on

the argument of the carrier; the collector itself serves as the function's codomain contract. The trace predicate `sorted?` consumes and inspects a queue to ensure that it is sorted.[8]

Here is an example reduction sequence generated by protecting a function with this contract and applying it to `false`:

$$\langle \texttt{let } f = \texttt{mon}^{\texttt{lib,main}}_{\texttt{lib}}\ (\texttt{tr}\ (\lambda y.\texttt{true} \to y)\ \texttt{sorted?})\ (\lambda x.\ \texttt{--})\ \texttt{in}\ f\ \texttt{false}, \emptyset \rangle$$

A three-labeled `mon` becomes a two-labeled `mon` that is immediately applied to the client label. All other monitor reductions are defined only on the two-labeled form.

$$\longmapsto \quad \langle \texttt{let } f = (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{tr}\ (\lambda y.\texttt{true} \to y)\ \texttt{sorted?})\ (\lambda x.\ \texttt{--}))\cdot \texttt{main}\ \texttt{in}\ f\ \texttt{false}, \emptyset \rangle$$

MON-TRACE allocates a fresh queue for the trace and constructs a collector to give to the body-contract constructor.

$$\longmapsto \quad \langle \texttt{let } f = (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ ((\lambda y.\texttt{true} \to y)\ (\texttt{co}\ \alpha_0\ \texttt{sorted?}))\ (\lambda x.\ \texttt{--}))\cdot \texttt{main}\ \texttt{in}\ f\ \texttt{false}, [\alpha_0 \mapsto \texttt{null}] \rangle$$

In this step, the first argument to the trace contract produces the body contract—filling in the appropriate spot with the collector.

$$\longmapsto \quad \langle \texttt{let } f = (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{true} \to (\texttt{co}\ \alpha_0\ \texttt{sorted?}))\ (\lambda x.\ \texttt{--}))\cdot \texttt{main}\ \texttt{in}\ f\ \texttt{false}, [\alpha_0 \mapsto \texttt{null}] \rangle$$

The monitor contains a function contract, so the first-order check succeeds and produces a guarded value by MON-FUN.

$$\longmapsto \quad \langle \texttt{let } f = (\texttt{grd}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{true} \to (\texttt{co}\ \alpha_0\ \texttt{sorted?}))\ (\lambda x.\ \texttt{--}))\cdot \texttt{main}\ \texttt{in}\ f\ \texttt{false}, [\alpha_0 \mapsto \texttt{null}] \rangle$$

After several `let`-based steps, the elided function is applied to `false`.

$$\longmapsto^+ \quad \langle (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{co}\ \alpha_0\ \texttt{sorted?})\ ((\lambda x.\ \texttt{--})\ \texttt{false}))\cdot \texttt{main}, [\alpha_0 \mapsto \texttt{null}] \rangle$$

Assume that the elided function produces 42.

$$\longmapsto^+ \quad \langle (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{co}\ \alpha_0\ \texttt{sorted?})\ 42)\cdot \texttt{main}, [\alpha_0 \mapsto \texttt{null}] \rangle$$

MON-COL appends the newly received value, 42, to the trace. It then arranges for the trace predicate to be checked.

$$\longmapsto^+ \quad \langle (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{sorted?}\ \alpha_0)\ 42)\cdot \texttt{main}, [\alpha_0 \mapsto \texttt{cons}\ 42\ \alpha_1, \alpha_1 \mapsto \texttt{null}] \rangle$$

Since the singleton queue containing just 42 is sorted, the predicate succeeds.

$$\longmapsto^+ \quad \langle (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ \texttt{true}\ 42)\cdot \texttt{main}, [\alpha_0 \mapsto \texttt{cons}\ 42\ \alpha_1, \alpha_1 \mapsto \texttt{null}] \rangle$$

The result is just the return value of the function.

$$\longmapsto^+ \quad \langle 42, [\alpha_0 \mapsto \texttt{cons}\ 42\ \alpha_1, \alpha_1 \mapsto \texttt{null}] \rangle$$

[8] This model's syntax does not support trace variable declarations, so the `natural?` constraint from section 2.1 is missing. Section 4.5 demonstrates how to add this feature to the model.

$\boxed{\Lambda_U \text{ Surface Syntax}}$ extends Λ_C

$$e \in \mathsf{Expr} = \ldots \mid \texttt{tr}\ e\ e\ e$$

$\boxed{\Lambda_U \text{ Evaluation Syntax}}$ extends Λ_C

$$\begin{aligned} e \in \mathsf{Expr} &= \ldots \mid \texttt{co}\ v\ \alpha\ v \\ \kappa \in \mathsf{Con} &= \ldots \mid \texttt{tr}\ v\ v\ v \mid \texttt{co}\ v\ \alpha\ v \\ E \in \mathsf{Ctx} &= \ldots \mid \texttt{tr}\ E\ e\ e \mid \texttt{tr}\ v\ E\ e \\ &\quad \mid \texttt{tr}\ v\ v\ E \end{aligned}$$

Fig. 12. Surface and Evaluation Syntax of Λ_U

$\boxed{\Lambda_U \text{ Reduction Relation}}$ extends Λ_C

$$\langle E[\texttt{mon}^k_j\ (\texttt{tr}\ v_\kappa\ v_b\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\texttt{mon}^k_j\ (v_b\ (\texttt{co}\ v_\kappa\ \alpha\ v_p))\ v], \sigma[\alpha \mapsto \texttt{null}]\rangle \quad \textsc{Mon-Trace}$$
$$\text{if } \alpha = \mathsf{next}(\sigma)$$

$$\langle E[\texttt{mon}^k_j\ (\texttt{co}\ v_\kappa\ \alpha\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\texttt{let}\ x_v = \texttt{mon}^k_j\ v_\kappa\ v\ \texttt{in} \quad \textsc{Mon-Col}$$
$$\texttt{let}\ x_j = x_v \cdot j\ \texttt{in}$$
$$\texttt{add!}\ \alpha\ x_j\ ; \texttt{mon}^k_j\ (v_p\ \alpha)\ v\ ; x_v], \sigma\rangle$$

Fig. 13. Reduction Relation of Λ_U

4.5 Extending the model

While the Racket implementation pairs each trace variable with a contract that governs collected values, the model omits this capability. To illustrate the versatility of the model, this subsection shows how to add this feature. To do so is relatively simple: one tweak to the syntax and another to Mon-Col. Other adaptations to the model—making it more faithful to the implementation—are similarly straightforward.

The revised surface syntax, shown in figure 12, adds contracts to the body-contract constructor; an analogous change augments collectors with contracts to protect collected values. Figure 13 shows the modified reduction relation. The Mon-Trace rule is just adapted for the new argument, while the revised Mon-Col reduction has some new behavior. In the contractum, a `let` expression binds x_v to the collected value v, monitored with contract v_κ. The second binding, for x_j, applies the monitored value x_v to j because the consumer of the trace is the *contract-defining party*. At this point, the value is added to the trace, and the trace is tested with the predicate. If the predicate succeeds, the monitored value x_v becomes the result of the `let` expression.

This variant of Mon-Col demands careful construction. First, it requires the proper management of blame parties. Monitoring the to-be-collected value is the responsibility of the contract-defining party, but using the value remains the responsibility of the client, which is the context. Second, the right-hand side may not duplicate the monitoring expression because a contract may have effects—after all, it could be another collector. So, like Grd-Fun, this rule is arranged such that the effects of v_κ are performed only once.

5 Semantic properties

Here is an evaluation function that can be used for all the languages defined in section 4:

$$\mathsf{eval}_{\mathscr{L}} : \mathsf{Prog} \to \mathsf{Ans}$$

$$\mathsf{eval}_{\mathscr{L}}(e) = \begin{cases} b & \text{if } \langle e, \emptyset \rangle \longmapsto^{\star} \langle b, \sigma \rangle \\ \texttt{opaque} & \text{if } \langle e, \emptyset \rangle \longmapsto^{\star} \langle v, \sigma \rangle, v \notin \mathsf{Bool} \\ \texttt{err}^{k}_{j} & \text{if } \langle e, \emptyset \rangle \longmapsto^{\star} \langle \texttt{err}^{k}_{j}, \sigma \rangle \end{cases}$$

The $\mathsf{eval}_{\mathscr{L}}$ function takes *programs* as input. A program is a closed surface expression. If the reduction relation connects the program to a Boolean, then $\mathsf{eval}_{\mathscr{L}}$ produces the same Boolean. If the reduction relation connects the program to any other value, then $\mathsf{eval}_{\mathscr{L}}$ produces `opaque`, just like the REPL does for a λ expression. Finally, the evaluator produces an error token with two labels when the reduction relation does too.

The $\mathsf{eval}_{\mathscr{L}}$ relation is a partial function. Therefore, a deterministic interpreter can be defined.

Theorem 5.1 (Functional Evaluator). $\mathsf{eval}_{\mathscr{L}}$ is a partial function.

Proof. See appendix B. □

Moreover, the only time $\mathsf{eval}_{\mathscr{L}}$ is undefined is when it diverges.

Theorem 5.2 (Uniform Evaluator). Either $\mathsf{eval}_{\mathscr{L}}(e)$ is defined or the reduction sequence starting with $\langle e, \emptyset \rangle$ is unbounded.

Proof. See appendix C. □

Finally, the revised contract semantics is equivalent to the original model in the absence of mutations.

Definition (Mutation Free). An expression e is *mutation free* if for all e' such that $\langle e, \emptyset \rangle \longmapsto^{\star} \langle e', \sigma \rangle$ it must be that $\sigma = \emptyset$.

Theorem 5.3 (Evaluator Equivalence). If e is mutation free, then $\mathsf{eval}_{\Lambda_B}(e) = \mathsf{eval}_{\Lambda_C}(e)$.

Proof. See appendix D. □

6 Implementation in principle

The semantics of section 4 suggests a macro-style compilation of trace contracts into a mix of plain contracts and queue manipulations. Such a translation requires the timely initialization of traces, strict control of effects (i.e., queue manipulation), the injection of run-time checks, and proper blame assignment. Compiler correctness follows from a theorem like the one Findler and Felleisen (2002) prove for plain contracts.

6.1 Theoretical compiler

Consider the following compiler that translates a Λ_T program into a Λ_C program:

$$\mathscr{C}(\texttt{tr}\ e_b\ e_p) = \begin{cases} \texttt{let}\ x_b = \mathscr{C}(e_b)\ \texttt{in} \\ \texttt{let}\ x_p = \mathscr{C}(e_p)\ \texttt{in} \\ \lambda\_.\texttt{let}\ x_\alpha = \texttt{queue}\ \texttt{in} \\ \quad x_b\ (\lambda y.x_p\ (\texttt{add!}\ x_\alpha\ y)) \end{cases}$$

Since there is only one construct related to trace contracts in the surface syntax, $\mathscr{C}$ has only one interesting case and is otherwise a homomorphism.

For a trace contract, the compiler sets up two bindings in a `let` expression: x_b and x_p. These stand for the compilations of the body-contract constructor and the trace predicate, respectively. The body of the `let` expression is a flat contract. Like MON-TRACE, it creates a fresh queue, and then an instance of the body contract by applying x_b to (the compilation of) a collector. The flat contract is used as a mechanism to initialize the queue at attachment time. Similarly, the compilation of the collector yields a flat contract that simulates MON-COL. Specifically, it adds the given element to the queue and then passes the extended queue to the trace predicate.

Here is the compilation of program (4.2):

$$\begin{array}{l} \texttt{let}\ x_b = \lambda y.\texttt{true} \to y\ \texttt{in} \\ \texttt{let}\ x_p = \texttt{sorted?}\ \texttt{in} \\ \lambda\_.\texttt{let}\ x_\alpha = \texttt{queue}\ \texttt{in} \\ \quad x_b\ (\lambda y.x_p\ (\texttt{add!}\ x_\alpha\ y)) \end{array} \tag{6.1}$$

6.2 Compiler correctness

Compare the reduction sequences for program (4.2) with that of program (6.1):

$$\begin{array}{ll} & \langle \texttt{let}\, f = \texttt{mon}_{\texttt{lib}}^{\texttt{lib,main}}\ (\texttt{let}\ x_b = \lambda y.\texttt{true} \to y\ \texttt{in} \\ & \qquad \texttt{let}\ x_p = \texttt{sorted?}\ \texttt{in} \\ & \qquad \lambda\_.\texttt{let}\ x_b = \texttt{queue}\ \texttt{in} \\ & \qquad\quad x_\kappa\ (\lambda y.x_p\ (\texttt{add!}\ x_\alpha\ y))) \\ & \qquad (\lambda x.\ \_\_)\ \texttt{in}\ f\ \texttt{false}, \emptyset\rangle \end{array}$$

Following left-to-right evaluation, the compilation uses a sequence of `let` expressions to evaluate the arguments of the trace contract.

$$\begin{array}{ll} \longmapsto^+ & \langle \texttt{let}\, f = \texttt{mon}_{\texttt{lib}}^{\texttt{lib,main}}\ (\lambda\_.\texttt{let}\ x_\alpha = \texttt{queue}\ \texttt{in} \\ & \qquad\quad (\lambda y.\texttt{true} \to y)\ (\lambda y.\texttt{sorted?}\ (\texttt{add!}\ x_\alpha\ y))) \\ & \qquad (\lambda x.\ \_\_)\ \texttt{in}\ f\ \texttt{false}, \emptyset\rangle \end{array}$$

The three-labeled `mon` becomes a two-labeled `mon` applied to the client label.

$$\begin{array}{ll} \longmapsto & \langle \texttt{let}\, f = (\texttt{mon}_{\texttt{lib}}^{\texttt{lib}}\ (\lambda\_.\texttt{let}\ x_\alpha = \texttt{queue}\ \texttt{in} \\ & \qquad\quad (\lambda y.\texttt{true} \to y)\ (\lambda y.\texttt{sorted?}\ (\texttt{add!}\ x_\alpha\ y))) \\ & \qquad (\lambda x.\ \_\_))\cdot\texttt{main}\ \texttt{in}\ f\ \texttt{false}, \emptyset\rangle \end{array}$$

The flat contract constructs a new queue and then produces an application of the body-contract constructor to the compiled collector.

$$\longmapsto^{+} \langle \texttt{let} f = (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ ((\lambda y.\texttt{true} \to y)\ (\lambda y.\texttt{sorted?}\ (\texttt{add!}\ \alpha_0\ y)))\ (\lambda x.\ \_\_))\cdot \texttt{main in}\ f\ \texttt{false}, [\alpha_0 \mapsto \texttt{null}]\rangle$$

Substituting gives a function contract with the compiled collector in the codomain position.

$$\longmapsto \langle \texttt{let} f = (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ \texttt{true} \to (\lambda y.\texttt{sorted?}\ (\texttt{add!}\ \alpha_0\ y)))\ (\lambda x.\ \_\_))\cdot \texttt{main in}\ f\ \texttt{false}, [\alpha_0 \mapsto \texttt{null}]\rangle$$

After a few steps, the elided function produces 42 by assumption. This must be checked against the compiled collector.

$$\longmapsto^{+} \langle (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\lambda y.\texttt{sorted?}\ (\texttt{add!}\ \alpha_0\ y))\ 42)\cdot \texttt{main}, [\alpha_0 \mapsto \texttt{null}]\rangle$$

The compiled collector adds the given value to the associated trace.

$$\longmapsto \langle (\texttt{mon}^{\texttt{lib}}_{\texttt{lib}}\ (\texttt{sorted?}\ \alpha_0)\ 42)\cdot \texttt{main}, [\alpha_0 \mapsto \texttt{cons}\ 42\ \alpha_1, \alpha_1 \mapsto \texttt{null}]\rangle$$

Finally, the trace predicate is run to ensure that the trace is sorted. Since it is, the final value is the result of the function: 42.

$$\longmapsto \langle 42, [\alpha_0 \mapsto \texttt{cons}\ 42\ \alpha_1, \alpha_1 \mapsto \texttt{null}]\rangle$$

This comparison suggests a proof that the compiled trace contract simulates the original behavior. Indeed, evaluating the compiled code always yields the same answer as the uncompiled source code, including divergence and errors.

Theorem 6.1 (Compiler Correctness). $\mathsf{eval}_{\Lambda_T} = \mathsf{eval}_{\Lambda_C} \circ \mathscr{C}$

Proof. See appendix E. □

7 Implementation in practice

A principled design (section 4) specifies when traces are initialized, when they are updated, and when a predicate evaluates their validity. The design gives rise to a principled implementation (section 6), which clarifies how to translate key features into a kernel language. But, developers do not live by principles alone; pragmatics matter just as much.

One pragmatic concern is contract blame. Contracts help enforce basic correctness claims, and contract failures alert developers to problems. Findler and Felleisen (2002) insist on precise blame assignment in failure messages. The design of the trace contract system carefully reuses the blame assignment mechanism from the underlying contract system. Experience suggests that for trace contracts, developers may need additional information beyond what standard blame provides (section 7.1).

Another concern is the availability of contract combinators. Working with the trace contract system pointed to limitations in the existing behavioral contract system. In particular, additional combinators are needed to support the specification of interception points

relevant to trace contracts. Fortunately, these pragmatically important combinators are orthogonal additions to the base system (section 7.2).

Finally, an implementation effort also informs designers of what is needed in a target host language to add a new feature. While the use of Racket's macro system greatly facilitates the addition of macro-expressible features, it should not be much more effort to extend existing compilers directly with support for trace contracts, provided the target language supports certain features (section 7.3).

7.1 Blame and suspects

When a contract system discovers a contract violation, it raises an exception, including a witness value and a pointer to the responsible component. This is dubbed *blame assignment*. Section 2.1 illustrates this point with an example of a violated trace contract.

As Lazarek et al. (2020) show in the context of behavioral contracts, blame assignment comes with enough information to almost always locate the actual source of the bug. They simulate tens of thousands of buggy programs by introducing a targeted fault via mutation. In most cases, following blame assignment leads to the source of the bug. For the few hundred cases where blame fails to identify the bug, Lazarek et al. (2020) reduce the failure to a lack of multi-call contracts. One of their examples is the DUNGEON program. As section 8 explains, strengthening the behavioral contract to a trace contract for DUNGEON provides exactly the needed blame information.

Trace contracts also complicate the situation, however. By default, blame goes to the party that added a value to the trace just before the predicate fails. Since all prefixes of the trace satisfied the predicate, this blame assignment seems to make sense. Yet, debugging real scenarios suggests that neither the blame correctness property (Dimoulas and Felleisen, 2011) nor the complete monitoring property (Dimoulas et al., 2012) are as useful for trace contracts as they are for behavioral ones.

Imagine a scenario with five components (A, B, C, D, E), where each contributes a number to a trace in increasing order ($\leq$). Here is an execution:

Component	A	B	C	E	D
Contribution	1.41	2.71	3.14	5.00	4.67

The model blames D because it contributes 4.67, causing the $\leq$ relation to fail. But, E might have made a call to the API out of order, and blaming just D does not even indicate a suspicion that some other component could be at fault. It is often useful to know the source of *all* values in a trace. After all, the idea behind traces is to subject multi-function interactions to contractual obligations.

A careful reader may argue that the problem is not with the blame assignment system, but with the predicate. Perhaps $\leq$ does not capture the specification to a sufficient degree. This claim is already true about behavioral contracts because a predicate may always be weaker than the intended property. And if the predicate is weaker than the intended property, the contract system may blame the wrong party.

This argument, however, overlooks the key premise of contract-system design: blame assignment must help developers narrow the search space for bugs, *regardless of the*

strength of the predicate. To explain this idea rigorously, Lazarek et al. (2020) turn folk wisdom into two properties: *blame trail* and *search progress*. The blame trail property states that either (i) blame is assigned to the buggy component or (ii) blame can be shifted to another component by strengthening contracts. The search progress property states that blame shifting always points to a component closer to the bug than before the modification.

For trace contracts, both properties can be violated in practice. In the example, strengthening contracts on D is unlikely to shift the blame, meaning the blame trail property is violated. When strengthening a trace predicate, the violating trace may decrease in length, but there is no reason to think *a priori* that the last contributor to a trace is always closest to the source of a bug, violating the search progress property. In short, the current blame assignment scheme points to the broken contract, but more information is needed to help developers identify the fault.

To address this problem, the implementation comes with three different ways of expressing blame assignments. Let a *suspect* be any party that contributes to a trace. Here are the three mechanisms used to express blame:

1. By default, the `trace/c` implementation does not report suspects. Instead, the error message merely mentions the violated contract and its parties.
2. The `setof-suspect` option forces the trace-contract system to track the set of all suspects and report that information when assigning blame. Frequently, there are just two parties to a contract. Without `#:global`, a two-party contract has a suspect set with at most two elements.
3. The `listof-suspect` option causes the trace-contract system to report the exact sequence of suspects, one per value in the trace. This option supplies the most comprehensive information, but it requires a large amount of memory and makes for large error messages.

Whether all of these strategies are useful in practice, only some of them, or some in certain circumstances and some in other circumstances, is left as an open research question.

7.2 Supporting functionality

The trace contract library comes with additional functions for manipulating interception points, resetting state explicitly, transforming collectors, and augmenting error messages with additional information.

Unlike behavioral contracts, trace contracts occasionally need to note events even in the absence of an informative value flow. For example, when a function receives no arguments, there is no natural interception point. The trace contract library supplies some combinators to create interception points for such situations (e.g., `apply/c`, `return/c`). See section 3.1 for sample uses.

Collector transformers wrap a collector and compute the value to be added to a trace from the given one. An example is `list/t`, which allows a programmer to tag values before they go into a trace. Typically, this tag adds information about the interception point. See section 3.2 for an example. Another one is `map/t`, which applies a given function to the captured value before adding it to a trace.

In practical situations, the `fail` function may have to perform more tasks than just inform the contract system of a failure. A software system may have to recover from a contract failure, and in those cases, a failure should reset accumulators to certain values. The author of a trace contract may also wish to add information about the rationale behind a failure. To this end, the trace-contract system supports augmenting error messages.

7.3 Implementing trace contracts in general

While the implementation is based on Racket's contract system, the design is language independent. An implementor of another programming language may thus wonder what it takes to add trace contracts. Our experience suggests a few criteria.

A trace is a data structure representing the sequence of values collected from various interception points. In the context of a functional language, function calls and returns are obvious interception points. Similarly, in an object-oriented language, this same role is played by methods. Generally speaking, an implementor's first business is to decide where to intercept and how to monitor the flow of values. The rest of this section assumes that call-and-return points suffice.

7.3.1 Monitoring higher-order values

In a higher-order language, functions, objects, modules, and classes may be first-class values. This implies that a contract system cannot determine statically where a particular call or return takes place. It is the task of the target language's runtime to support the monitoring of value flows. The Racket implementation employs proxy values (Strickland et al., 2012)—invisible wrappers—for interception. With such wrappers, it is straightforward to intercept values even in the presence of higher-order values.

Wrappers are not the only option. For instance, the weaving mechanism from aspect-oriented programming (Kiczales et al., 1997) could be used for a similar purpose. Roughly speaking, weaving injects code into the program at specifiable program points. Although weaving is powerful, it is not clear whether weaving can efficiently intercept values in a higher-order language, as needed by the proposed design.

7.3.2 Mutation within contracts

Trace-contract checking is effectful. When a collector receives a value, it mutably adds this value to a trace. Even though, as some of the examples in section 2 show, the component itself can be purely functional. Hence, the underlying language must allow side effects in contracts, even though trace predicates themselves are pure functions.[9]

Formally, section 6 validates that trace contracts are expressible as shorthand in an underlying language with higher-order contracts and a mutable data structure. In the terminology of Felleisen (1991), the new feature is macro expressible. Theorem 6.1 shows that this translation completely preserves the specified behavior. Though, Felleisen (1991) also

[9] Since collectors mutate traces, checking a collector is not idempotent. While idempotence is sometimes considered an important property of contract systems (Degen et al., 2009; Findler and Blume, 2006), it often fails to hold for other reasons. For example, Owens (2012) and Hinze et al. (2006) observe violations of idempotence in several useful contexts.

shows that imperative assignment increases the expressive power of a pure host language. By implication, trace contracts are *not* expressible in such a setting.

7.3.3 Interception and crossing times

As mentioned in section 4.3, a trace-contract system assumes that crossing and interception time in the target contract system are separate. As it turns out, the implementation of trace contracts exposed the lack of this separation in Racket's contract system. Racket fails to separate the two points in one combinator: the depended-upon argument contract in `->i` (Dimoulas et al., 2013). A change to Racket's contract system allows trace contracts to distinguish these boundary crossings, meaning that a collector may ignore arguments passing through a boundary that has an indy (third) party.[10] This is sufficient to eliminate the duplicate-collection problem.

7.3.4 Macros not needed

An implementor can easily add trace contracts to a language with a rich macro system, such as a Racket. Including all the practical features mentioned in section 2 makes this macro rather large and complex. While macros are a convenient implementation mechanism for trace contracts, they are not a requirement. The implementor of a functional language such as SML, which elaborates surface syntax into a small kernel, can add trace contracts with a similar addition to the front-end elaborator.

8 Usability and performance evaluation

Usability questions concern the ease with which programmers can write trace-contract properties for their programs and what performance penalty the system imposes.

Section 8.2 gives a qualitative assessment of our experience writing trace contracts. This assessment suggests two opposite insights. On the one hand, trace contracts enable developers to use the entire underlying programming language. Hence, developing a trace-contract property is just like developing an ordinary predicate in an ordinary language, using all available tools—especially unit and property-testing frameworks. On the other hand, as experience with ordinary higher-order contracts shows, contracts are a special-purpose domain. Such domains call for specific, tailor-made notations to eliminate boilerplate code. Developing such notations remains future work.

As for performance, the only relevant question is what kind of *fixed cost* the mechanism itself imposes on programs, not the *variable cost* of the programmer-defined predicates.[11] Trace initialization, trace updates, and calls to predicates are all included in this fixed cost. The results of measuring the performance of trace contracts, presented in section 8.3, are quite encouraging.

[10] Thanks to Robby Findler for help with this change to Racket's contract system.

[11] The performance evaluation cannot answer questions concerning the *variable cost* of trace predicates. Trace contracts are *property agnostic*, so the variable cost of a trace contract depends largely on the property being checked. In other words, this cost is solely under the purview of the programmer, not the trace-contract system.

8.1 Benchmark programs

The selected benchmarks represent real-world uses of Racket that offer opportunities for adding trace contracts. MEMORY turns the example from section 2.1 into a pathological stress test. FUTURE is a large existing Racket library equipped with trace contracts, plus an application that stresses the functionality. Four of the benchmark programs (DUNGEON, JPEG, LNM, TETRIS) are variants on programs from the standard gradual typing benchmark suite (Greenman et al., 2019). Three (DATAFLOW, FISH, TICKET) are programs developed for use in university courses. All the benchmarks have been adapted so that they do not measure I/O operations.

DATAFLOW Computes a constant propagation analysis for a simple imperative language. A trace contract, similar to the one from section 2.3, checks the monotonicity of a transfer function during fixed-point iteration.

DUNGEON Generates the specification of a maze. A trace contract on the random-number generator ensures that it does not exhaust a fixed pool of random numbers. In the original program, resizing the random number pool caused a contract violation that failed to provide helpful blame information (Lazarek et al., 2020, sec. 5.1). With a trace contract, this same bug produces an error message with a blame assignment that directly points to the problem. The contract must keep track of how many times the `random` function is called, so its accumulator is just a natural number and the check is cheap.

FISH Runs a "That's My Fish" board game tournament. There are two trace contracts: a referee contract and a player contract.

The referee contract ensures that the referee calls back players in the specified order unless the game state does not permit the player to take a turn. The contract is a promise made by the referee to all the players. To enforce this promise, the contract is placed on the referee's list of player objects. A collector receives a new value every time the referee calls the `take-turn` method on any player. The trace contract then checks that this is in accordance with the promised callback order on the *players*, including skipping over players that are momentarily prohibited from taking a turn.

The player contract enforces a sequence property on its method calls. In other words, the player components ensure that their individual *methods* are called in the specified order. This contract is similar to the value-dependent temporal protocol example from section 3.2. It is independent of, and orthogonal to, the referee contract.

FUTURE Visualizes the performance of a futures benchmark. Futures are a run-time mechanism for incrementally adding parallelism to programs (Swaine et al., 2010). The future visualizer (Swaine et al., 2012) uses Racket's drawing library, which has been equipped with trace contracts to enforce multi-call properties. A full list of these properties is enumerated in appendix F. Some of the properties were monitored by the drawing library using ad-hoc checks and others were not checked at all.

JPEG Parses a JPEG input stream and writes it to an output stream. A trace contract guarantees that operations on the output stream occur in the correct order. Like the example in section 3.2, it checks every stream-related function call against a finite

Benchmark	SLOC	Protects	Checks	Disabled	Enabled	Predicate	Overhead
DATAFLOW	502	1	584	83 ± 3	87 ± 2	274 ± 3	5%
DUNGEON	589	0	538,000	2441 ± 38	2715 ± 46	2713 ± 33	11%
FISH	1,452	2,698	63,175	7780 ± 70	8340 ± 82	8366 ± 80	7%
FUTURE	1,721	16,360	234,444	6075 ± 54	7083 ± 83	7502 ± 86	17%
JPEG	1,481	0	54,556	276 ± 5	303 ± 6	316 ± 6	10%
LNM	564	168	3,248	522 ± 8	532 ± 9	534 ± 9	2%
MEMORY	59	0	10,000	141 ± 4	164 ± 4	164 ± 4	16%
TETRIS	334	6,807	125,570	3040 ± 24	3566 ± 36	3927 ± 43	17%
TICKET	1,427	384	15,794	13062 ± 149	13186 ± 170	13199 ± 182	1%

Table 4. Basic Metrics and Performance Measurements

automaton. Formulating the trace contract involves creating several contracts that share the same accumulator, the state of the finite automaton.

LNM Draws plots of the performance measurements of a gradual type system. Like FUTURE, this benchmark uses a variant of Racket's drawing library equipped with trace contracts.

MEMORY Reports memory use, including garbage-collected blocks. The trace contract from section 2.1 ensures that `current-memory-use` returns increasing numbers over time; it is called 10,000 times in a tight loop, the results of which are graphed on a line chart using Racket's plot (Toronto and Harsányi, 2011) library.

TETRIS Simulates and displays a recording of the game of Tetris. This benchmark also uses a variant of Racket's drawing library equipped with trace contracts.

TICKET Runs a "Ticket to Ride" board game tournament. Like FISH, TICKET has both a referee and a player contract. The referee contract enforces a promise that the referee calls back players in the specified order. This trace contract is significantly simpler than the one for FISH, because every player can execute an action in every game state. The player-side trace contract enforces the correct sequence of method calls. The example presented in section 3.2 is a simplified version of this contract.

8.2 Benchmark summary

Table 4 first lists the number of essential lines of source code (SLOC) for each program, including the trace contract and its auxiliary functions.

None of the trace contracts require much code. FISH and TICKET contain the most complex ones, but the others are relatively simple. Even the most complex trace contracts are concise. Indeed, the contract for TICKET is shown nearly verbatim in section 3.2. Since predicates are ordinary code, they can make use of existing data structure libraries, and those libraries serve as workhorses in many cases. For example, JPEG uses an existing FSM package that renders its temporal constraint predicate practically a one-liner.

Tight integration with the existing contract system makes writing many trace contracts natural. Since the trace contract mechanism manages state behind the scenes, contract composition and contract abstraction work as expected. Developers can write trace contracts as ordinary code, compose them as usual, and even abstract over them.

Programming trace contracts for these benchmark programs also points to limitations. For example, placing collector contracts can be awkward and repetitive. Consider the trace contracts in sections 3.1 and 3.2, both of which contain several nearly identical lines. A macro can eliminate the repetition in each case individually, but it is not obvious if there is a general-purpose DSL that could reduce such repetitive code across many cases.

8.3 Performance measurements

The performance measurements on the right side of table 4 were recorded on a dedicated Linux machine with an Intel Xeon E3 processor running at 3.10 GHz with 32 GB of RAM and with Racket 8.6 CS. Each benchmark configuration was repeated 100 times with a maximum timeout of two minutes.

The Protects column reports the number of times a trace contract protects a new value during the steady state of a program's execution. Each time, there is some overhead due to allocating references for accumulators and creating collector contracts. Some benchmarks have a zero entry because all of the trace contracts are initialized before the main body of the program begins, for example, when dependencies are being loaded.

The Checks column states the number of times each trace predicate is checked. As mentioned, this evaluation is concerned with the *fixed cost* of trace contracts. Therefore, each trace predicate is replaced with the trivial predicate that always returns `true`. Benchmarks were executed at two levels: Disabled where trace contracts are disabled, and Enabled where they are enabled. These measurements are the mean number of milliseconds it takes to run each benchmark, averaged over 100 samples, along with the standard deviation. The Predicate column lists the performance numbers where trace contracts are enabled and the predicate actually checks the desired property. Despite it not being the primary means of evaluation, these numbers are provided for context. Such predicates are straightforward implementations and are not heavily optimized. Finally, the Overhead column shows the percent overhead of Enabled compared to Disabled.

The overhead of the trace-contract mechanism is relatively low, somewhere between 1% and 17%. As is, the setups basically simulate worst-case scenarios. For example, MEMORY just calls a simple function in a tight loop, so contract checking takes up a large portion of total execution time. By contrast, benchmarks that are closer to real-world programs, such as TICKET, incur a low overhead. Thus, the evidence suggests that the trace-contract mechanism itself does not exhibit any performance pathology.

These measurements do not exercise an industrial-strength implementation of trace contracts, but rather a direct translation of the design. This implementation serves as a vehicle for exploration. With some performance engineering, it is likely to perform significantly better. While this evaluation can provide some first impression of the performance of trace contracts, it is not enough to generalize to other settings or languages.

9 Related work

Prior work is in the tradition of software contracts or runtime verification (RV). Specifically, this paper leverages the development of higher-order dependent contracts (Findler and Felleisen, 2002; Blume and McAllester, 2006; Findler and Blume, 2006; Greenberg et al., 2010; Dimoulas et al., 2012); the temporal contract system of Disney et al. (2011) is the most directly comparable piece of work from this area. Within the runtime verification area, the most similar approach is the monitor-oriented programming framework (Meredith et al., 2011; Chen and Roşu, 2007; Chen et al., 2005).

These two bodies of research have distinct philosophies about expressing and checking properties. Trace contracts borrow the notion of traces from RV to extend a higher-order behavioral contract system. They seek to bridge the gap between the two areas. Eventually this bridge should make many results from RV available to contract programmers, and it may inject new ideas into RV.

9.1 Runtime verification, generally

Traditional contract systems and RV systems differ along several dimensions. Most importantly, as Meyer (1992) observes, contracts are a design tool for the developer; in contrast, RV is a tool for the quality assurance stage of the development process.

9.1.1 Scope

Contracts are *modular*. A programmer attaches contracts to the interface of a "server" component. When a "client" component imports a server component, it is forced to agree to the contract. Similarly, a client component may impose a contract on imported pieces of functionality to protect itself from a misbehaving service component. In the first case, clients do not need to be adapted to the service contract, and in the second case, service components remain unaware of the client's protective contract. Put differently, it is possible to compile these components in either order or, even better, to link pre-compiled binary objects.

RV is *whole program*. A programmer specifies events of interest and properties about event traces. The RV system converts this specification into an executable monitor and weaves interception code into the host program to communicate first-order data about events to a separate monitor process (Bartocci et al., 2018).

Monitoring higher-order values is possible with RV, but the encoding uses a complex protocol between the server and the client module; it requires source modification to both components. Implementing the protocol on a modular basis is either impossible, which precludes the binary-linking approach available with contracts, or requires complex extensions (Xiang et al., 2015).

9.1.2 Language

Contracts are linguistic elements that are *inside* the language. The programmer uses the same language—and the exact same tools—for writing code and contracts. Extending the notation for contracts in a domain-specific manner (via macros in Racket) is useful; the

-> abbreviation for function contracts is one example. Racket also treats contracts as first-class objects, meaning they can be put into lists, passed and returned from functions, and composed at run time.

RV is extra-linguistic; that is, RV systems exist *outside* the language. Specifications are usually written in a distinct, external logic language and tend to make temporal statements about sequences of first-order data (Havelund et al., 2018). While this language may contain fragments of host-language code, it is only loosely connected with the host language and its tool chain.

9.1.3 Violations

As a consequence of the differences along the linguistic axis, contracts and RV differ in two ways concerning the violation of specifications: recovery and error-location information.

When a contract system discovers a violation of an assertion, it raises an exception that includes information about the parties that agreed to the contract and which of them violated it—blame information. By raising an exception at the very point where a contract violation is discovered, the contract system gives the program a chance to recover immediately and with a response targeted to the problem. In a language with resumable exceptions, such as Common Lisp (Steele, 1990), a program may even resume its execution at the exact place where the violation occurred.

The *precise* error information in violation messages enables the developer to understand the cause of a violation. Lazarek et al. (2020) show that this blame information is effective at narrowing the search space during debugging. It is also a well-founded concept; Dimoulas et al. (2012) provide a framework for proving that blame information points to the component which supplies a value that does not meet the specification.

Traditionally, RV systems report violations of specifications with delay and do not contain blame information (Swords, 2019). The delay is due to the underlying process-communication arrangement between the program proper and its monitor. This poses a problem for tracking the provenance of values and for assigning blame. Hence, RV makes it difficult to restart programs with a problem-specific, localized response, unless an additional "diagnosis layer" is supplied (Leucker and Schallhart, 2009).

9.1.4 Properties

Contracts are property *agnostic*. Any predicate, including one that tries to decide a recursively-enumerable property, can be used as a contract. This is maximally expressive but can be computationally expensive.

RV is property *sensitive*. Much of RV research focuses on the development of specification languages that can express properties of interest concisely and that can be compiled into efficient monitoring code (Leucker and Schallhart, 2009). Often these are variants of temporal logic. These specialized logics can provide hard guarantees about time and space efficiency, at the cost of expressive power.

9.2 Runtime verification, specifically

Within the landscape of RV tools, JavaMOP is the best point for comparison. It is the most versatile implementation in the family of monitor-oriented programming (MOP) systems (Meredith et al., 2011). A selling feature of JavaMOP is that it is generic; the programmer can choose the events of interest, specification logic, and violation handler code. Chen and Roşu (2007) argue that there is no logic suitable to express all properties, and thus JavaMOP developers must engineer external logic "plugins" (Chen et al., 2005).

Trace contracts, by contrast, allow programmers to take full advantage of the host language. If this host language comes with expressive meta-programming facilities, such as the macros of Racket (Ballantyne et al., 2020; Felleisen et al., 2018; Flatt, 2002), developers can easily add a custom notation for trace contracts. Consider section 3.2 which uses Racket's automata package (McCarthy, 2011) and significantly improves the readability of the trace predicate without external tooling. With the visual-interactive syntax of Andersen et al. (2020), a developer could even edit and view the NFA graphically.

For an example of cross-pollination, consider trace slicing. This idea is due to the RV community (Chen and Roşu, 2007). In the RV world, this operation is *not* exposed to users of RV systems; rather, an efficient slicing algorithm is derived from data quantifiers in the specification logic. The trace contract library supports trace slicing via tagging and ordinary stream functions. In keeping with the philosophy of contract-system design, the power is handed to programmers.

9.3 Higher-order contracts, specifically

While higher-order contracts are typically *independent* of state, trace contracts manage state behind the scenes to support a mostly functional view of specifications. Others show that contracts could occasionally benefit from a modicum of state (Tov and Pucella, 2010; Moore et al., 2016; Waye et al., 2017), though these systems do not come with the expressiveness of trace contracts.

The higher-order temporal contracts of Disney et al. (2011) are the closest prior work to trace contracts. Their research focuses on two aspects: an operational theory of temporal event sequences and the specification of properties. On the theory side, the work introduces a novel approach to operational semantics that formalizes the meaning of modules as automata that create trees of observable events, similar to game-based denotational semantics. The semantics satisfies a non-interference theorem, meaning that streams of values are kept separate. On the practical side, the work focuses on specifying properties of event sequences as regular expressions *without* giving programmers access to a data representation of traces. Trace contracts come with more expressive power, yet do not necessarily sacrifice efficiency.

At first glance, computational contracts (Scholliers et al., 2015) look similar to higher-order temporal contracts. But, computational contracts go far beyond any classical contract classification scheme (Beugnard et al., 1999, 2010), providing unprecedented power and imposing a similarly high cost. A computational contract system empowers programmers

to impose arbitrary restrictions on components from the outside and in a post-hoc manner. Thus, computational contracts depart from the idea that contracts are assertions at the boundary between black-box components, instead turning components into glass boxes.

9.4 Typestate and type systems

Researchers often try to move from dynamically checked contracts to statically checked types, because discovering general mistakes during compile time is safer than discovering specific mistakes at run time, perhaps even after a program has been deployed. This subsection deals with two distantly related ideas from the world of static checking.

The work of Strom and Yemini (1986) on typestate systems, recently resumed in various forms (Jaspan and Aldrich, 2009; Pucella and Tov, 2008; Wolff et al., 2011), directly addresses simple but common affinity restrictions in APIs. For example, typestate systems can check constraints such as "method `m` may be called at most once" and even "method `m` must be called before method `n`." These constraints are restricted to regular properties, i.e., those that can be expressed using a finite-state machine.

Honda et al. (1998)'s notion of session type is a closely related idea. Recently this field has experienced rapid growth. Roughly speaking, session types for objects come with the same expressive power as typestate (Gay et al., 2010).

Effect systems are also capable, in a limited way, of constraining the order in which effects can be performed. Ordinary effect systems do not consider the order of effects, but sequential effect systems (Tate, 2013; Koskinen and Terauchi, 2014) can. Further extensions can statically verify some temporal logic propositions (Gordon, 2017).

No existing static technique can express all of the trace-contract examples. By combining traces with plain code, a programmer can formulate arbitrary predicates and check value-dependent constraints on traces. Trace predicates can look for specific values or use specific values to express a constraint, which is impossible with these type systems. Dependent session types (Toninho et al., 2011) may be able to do better, but are still limited to statically decidable properties. Trace contracts, by monitoring programs at run time, are able to take advantage of the precision that run-time checking offers. A combination of session types and contracts (Bocchi et al., 2010) can refine the content of messages passed between parties, but the structure of the protocol remains fixed. This approach also does not naturally extend to contracts on higher-order values.

10 Trace contracts for rich specifications

Engineering complex software requires mechanisms for expressing and enforcing component specifications. Types, contracts, run-time verification—each has been successful in its own way, but major expressiveness gaps remain.

This paper introduces trace contracts as a novel, practical, and well-founded element of this spectrum. Specifically, trace contracts enable developers to protect the elements of their API across multiple function and method calls. The trace contract system provides traces of argument and result values as a first-class piece of data. Hence, trace contracts can express protocols that are ubiquitous in practice, but are usually specified informally.

In addition to a principled design, this paper describes an implementation of trace contracts, along with an evaluation. The implementation addresses a good number of pragmatic concerns, especially those of performance. On the question of blame assignment, the implementation supports several natural strategies with different precision and memory consumption trade-offs.

Critically, the trace-contract design separates the concept of a value trace from the language of enforced properties. In other words, trace contracts separate the low-level collection mechanism from the high-level property formulation. Hence, the design enables an investigation of trace-collection performance, independent of an exploration of problem-specific notations for expressing the properties of traces. Racket, with its powerful tools for creating embedded and extensible DSLs (Ballantyne et al., 2020), is a convenient platform for this kind of research.

Plenty of work remains. Section 7.1 proposes three blame strategies but gives no theoretical or empirical justification for any of them. What are the tradeoffs between these approaches with regard to theory (blame correctness), implementation (memory use), and pragmatics (debugging violations)? Protocols are common in concurrent progams but are often informally described. Can trace contracts be adapted to monitor protocols in concurrent applications? Techniques exist to statically verify functional contracts in Racket (Nguyễn et al., 2018). Is static verification practical for trace contracts? Section 9 compares trace contracts to other research results. How many of these systems can be implemented on top of trace contracts? If they can, what are the benefits of doing so? If they cannot, how can trace contracts be extended to accomodate such systems?

Even though future work is needed to turn trace contracts into a truly practical technology, hopefully the foundation put forth in this paper is sufficient to advance the practice of software specification in Racket and beyond.

Acknowledgements

This work was supported by National Science Foundation grant SHF 2116372. The authors thank anonymous POPL and JFP reviewers for their comments.

Artifact

The implementation of trace contracts has been released as an open-source library. Right now, Racket developers can use trace contracts to fortify their programs.

Conflicts of Interest

None.

References

Andersen, L., Ballantyne, M. & Felleisen, M. (2020) Adding Interactive Visual Syntax to Textual

Code. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).
Andersen, L., St-Amour, V., Vitek, J. & Felleisen, M. (2018) Feature-Specific Profiling. *Transactions on Programming Languages and Systems (TOPLAS)*.
Ashley, J. M. & Dybvig, R. K. (1994) An Efficient Implementation of Multiple Return Values in Scheme. *LISP and Functional Programming (LFP)*.
Ballantyne, M., King, A. & Felleisen, M. (2020) Macros for Domain-Specific Languages. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).
Barendregt, H. P. (1981) *The Lambda Calculus*. North-Holland Publishing Co.
Bartocci, E., Falcone, Y., Francalanza, A. & Reger, G. (2018) Introduction to Runtime Verification. In *Lectures on Runtime Verification*. Springer.
Beugnard, A., Jézéquel, J.-M. & Plouzeau, N. (2010) Contract Aware Components, 10 Years After. International Workshop on Component and Service Interoperability (WCSI).
Beugnard, A., Jézéquel, J.-M., Plouzeau, N. & Watkins, D. (1999) Making Components Contract Aware. *Computer*.
Blume, M. & McAllester, D. (2006) Sound and Complete Models of Contracts. *Journal of Functional Programming (JFP)*.
Bocchi, L., Honda, K., Tuosto, E. & Yoshida, N. (2010) A Theory of Design-by-Contract for Distributed Multiparty Interactions. International Conference on Concurrency Theory.
Chen, F., d'Amorim, M. & Roşu, G. (2005) Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP. Workshop on Runtime Verification (RV).
Chen, F. & Roşu, G. (2007) MOP: An Efficient and Generic Runtime Verification Framework. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).
Degen, M., Thiemann, P. & Wehr, S. (2009) True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). Arbeitstagung Programmiersprachen (ATPS).
Dimoulas, C. & Felleisen, M. (2011) On Contract Satisfaction in a Higher-Order World. *Transactions on Programming Languages and Systems (TOPLAS)*.
Dimoulas, C., Findler, R. B. & Felleisen, M. (2013) Option Contracts. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).
Dimoulas, C., Findler, R. B., Flanagan, C. & Felleisen, M. (2011) Correct Blame for Contracts: No More Scapegoating. Principles of Programming Languages (POPL).
Dimoulas, C., New, M. S., Findler, R. B. & Felleisen, M. (2016) Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). International Conference on Functional Programming (ICFP).
Dimoulas, C., Tobin-Hochstadt, S. & Felleisen, M. (2012) Complete Monitors for Behavioral Contracts. European Symposium on Programming (ESOP).
Disney, T., Flanagan, C. & McCarthy, J. (2011) Temporal Higher-Order Contracts. International Conference on Functional Programming (ICFP).
Felleisen, M. (1991) On the Expressive Power of Programming Languages. *Science of Computer Programming*.
Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J. & Tobin-Hochstadt, S. (2018) A Programmable Programming Language. *Communications of the ACM (CACM)*.
Findler, R. B. & Blume, M. (2006) Contracts as Pairs of Projections. Functional and Logic Programming (FLP).
Findler, R. B. & Felleisen, M. (2002) Contracts for Higher-Order Functions. International Conference on Functional Programming (ICFP).
Flatt, M. (2002) Composable and Compilable Macros: You Want it When? International Conference on Functional Programming (ICFP).
Flatt, M. & PLT. (2010) Reference: Racket. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.
Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N. & Caldeira, A. Z. (2010) Modular Session Types for Distributed Object-Oriented Programming. Principles of Programming Languages (POPL).

Gordon, C. S. (2017) A Generic Approach to Flow-Sensitive Polymorphic Effects. European Conference on Object-Oriented Programming (ECOOP).
Greenberg, M., Pierce, B. C. & Weirich, S. (2010) Contracts Made Manifest. Principles of Programming Languages (POPL).
Greenman, B., Takikawa, A., New, M. S., Feltey, D., Findler, R. B., Vitek, J. & Felleisen, M. (2019) How to Evaluate the Performance of Gradual Typing Systems. *Journal of Functional Programming (JFP)*.
Havelund, K., Reger, G., Thoma, D. & Zălinescu, E. (2018) Monitoring Events that Carry Data. In *Lectures on Runtime Verification*. Springer.
Hinze, R., Jeuring, J. & Löh, A. (2006) Typed Contracts for Functional Programming. Functional and Logic Programming (FLP).
Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language Primitives and Type Discipline for Structured Communication-Based Programming. European Symposium on Programming (ESOP).
Jaspan, C. & Aldrich, J. (2009) Checking Framework Interactions with Relationships. European Conference on Object-Oriented Programming (ECOOP).
Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997) Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP).
Koskinen, E. & Terauchi, T. (2014) Local Temporal Reasoning. Logic in Computer Science (LICS).
Lazarek, L., King, A., Sundar, S., Findler, R. B. & Dimoulas, C. (2020) Does Blame Shifting Work? Principles of Programming Languages (POPL).
Leucker, M. & Schallhart, C. (2009) A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*.
McCarthy, J. (2011) Automata: Compiling State Machines. https://docs.racket-lang.org/automata/index.html.
Meredith, P. O., Jin, D., Griffith, D., Chen, F. & Roşu, G. (2011) An Overview of the MOP Runtime Verification Framework. *International Journal on Software Tools for Technology Transfer*.
Meyer, B. (1988) *Object-Oriented Software Construction*. Prentice Hall.
Meyer, B. (1992) Applying "Design by Contract". *Computer*.
Moore, S., Dimoulas, C., Findler, R. B., Flatt, M. & Chong, S. (2016) Extensible Access Control with Authorization Contracts. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).
Moy, C., Nguyễn, P. C., Tobin-Hochstadt, S. & Van Horn, D. (2021) Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification. Principles of Programming Languages (POPL).
Nguyễn, P. C., Gilray, T., Tobin-Hochstadt, S. & Van Horn, D. (2018) Soft Contract Verification for Higher-Order Stateful Programs. Principles of Programming Languages (POPL).
Nielson, F., Nielson, H. R. & Hankin, C. (2005) *Principles of Program Analysis*. Springer Verlag.
Owens, Z. (2012) Contract Monitoring as an Effect. Higher-Order Programming with Effects (HOPE).
Plotkin, G. (1975) Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*.
Pucella, R. & Tov, J. A. (2008) Haskell Session Types with (Almost) No Class. Haskell Symposium.
Scholliers, C., Tanter, E. & De Meuter, W. (2015) Computational Contracts. *Science of Computer Programming*.
Steele, G. L. (1990) *Common Lisp the Language*. Digital Press.
Strickland, T. S., Tobin-Hochstadt, S., Findler, R. B. & Flatt, M. (2012) Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).
Strom, R. E. & Yemini, S. (1986) Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*.
Swaine, J., Fetscher, B., St-Amour, V., Findler, R. B. & Flatt, M. (2012) Seeing the Futures: Profiling Shared-Memory Parallel Racket. Functional High-Performance Computing (FHPC).
Swaine, J., Tew, K., Dinda, P. A., Findler, R. B. & Flatt, M. (2010) Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems. Object-Oriented Programming, Systems,

Languages and Applications (OOPSLA).
Swords, C. (2019) *A Unified Characterization of Runtime Verification Systems as Patterns of Communication*. Ph.D. thesis. Indiana University.
Tate, R. (2013) The Sequential Semantics of Producer Effect Systems. Principles of Programming Languages (POPL).
Toninho, B., Caires, L. & Pfenning, F. (2011) Dependent Session Types via Intuitionistic Linear Type Theory. Principles and Practice of Declarative Programming (PPDP).
Toronto, N. & Harsányi, A. (2011) Plot: Graph Plotting. https://docs.racket-lang.org/plot/index.html.
Tov, J. A. & Pucella, R. (2010) Stateful Contracts for Affine Types. European Symposium on Programming (ESOP).
Waye, L., Chong, S. & Dimoulas, C. (2017) Whip: Higher-Order Contracts for Modern Services. International Conference on Functional Programming (ICFP).
Wolff, R., Garcia, R., Tanter, E. & Aldrich, J. (2011) Gradual Typestate. European Conference on Object-Oriented Programming (ECOOP).
Xiang, C., Qi, Z. & Binder, W. (2015) Flexible and Extensible Runtime Verification for Java. *International Journal of Software Engineering and Knowledge Engineering*.

A Proof syntax and judgments

The proofs in the sections that follow require some additional syntax and judgments. In particular, certain sets of expressions that exist implicitly in the semantics must be named explicitly. Additionally, a judgment identifying valid expressions is needed.

$\boxed{\Lambda \text{ Proof Syntax}}$

$$
\begin{aligned}
a \in \mathsf{Ans} &= t \mid \texttt{opaque} \\
t \in \mathsf{Ter} &= v \mid \texttt{err}_j^k \\
r \in \mathsf{Redex} &= \texttt{if}\ v\ e\ e \mid v\ v \mid o\ v \mid \texttt{queue} \mid \texttt{add!}\ v\ v \mid \texttt{err}_j^k
\end{aligned}
$$

$\boxed{\Lambda_B \text{ Proof Syntax}}$ extends Λ

$$r \in \mathsf{Redex} = \ldots \mid \texttt{mon}_j^{k,l}\ e\ e$$

$\boxed{\Lambda_C \text{ Proof Syntax}}$ extends Λ

$$r \in \mathsf{Redex} = \ldots \mid \texttt{mon}_j^{k,l}\ e\ e \mid \texttt{mon}_j^k\ v\ v \mid (\texttt{grd}_j^k\ \omega\ v) \cdot l$$

Fig. 14. Proof Syntax of Λ, Λ_B, and Λ_C

Figure 14 defines three sets of terms. An answer is the result of $\mathsf{eval}_{\mathscr{L}}$ (for a language $\mathscr{L}$) and is either a terminal expression or the `opaque` token. A terminal expression is either a value or an error token. Finally, a reducible expression (redex) is an expression that inhabits the hole of the evaluation context on the left-hand side of a reduction rule.

$$
\frac{\mathsf{fv}(e) = \emptyset \qquad \forall \alpha \in \mathsf{addrs}(e)[\sigma \Vdash \sigma(\alpha)]}{\sigma \vdash e}
\qquad
\frac{}{\sigma \Vdash \texttt{null}}
\qquad
\frac{\sigma \vdash v \qquad \sigma \Vdash \sigma(\alpha)}{\sigma \Vdash \texttt{cons}\ v\ \alpha}
$$

Fig. 15. Valid Expression Judgment

Figure 15 defines a judgment that identifies valid expressions from the too-liberal grammar for the evaluation syntax. A valid expression is closed and contains only addresses that map to valid queues. A valid queue contains only valid values.

B Functional evaluator proof

The theorems in this section hold for all languages presented in section 4.

Theorem 5.1 (Functional Evaluator). $\mathsf{eval}_{\mathscr{L}}$ is a partial function.

Proof. A straightforward consequence of lemma B.1. □

Lemma B.1 (Deterministic Evaluator). If $\langle e, \sigma\rangle \longmapsto^{\star} \langle t_1, \sigma_1\rangle$ and $\langle e, \sigma\rangle \longmapsto^{\star} \langle t_2, \sigma_2\rangle$, then $t_1 = t_2$ and $\sigma_1 = \sigma_2$.

Proof. By lemma B.2, every expression can be decomposed into a unique evaluation context and a unique redex. For each redex, there is only one reduction rule that could apply. Thus evaluation is deterministic. □

Lemma B.2 (Unique Decomposition). For all $e \in \mathsf{Expr}$ either $e \in \mathsf{Ter}$ or there exists a unique evaluation context E and unique redex r such that $e = E[r]$.

Proof. By induction on the structure of e.

Case $e = t$**.**
Trivial.

Case $e = o\ e_a$**.**
Applying the inductive hypothesis to e_a, it follows that either (1) $e_a \in \mathsf{Ter}$ or (2) there exists unique E_a and r such that $e_a = E_a[r]$. For (1), e_a could be a value, in which case $E = \square, r = o\ e_a$. Otherwise, $e_a = \mathtt{err}^k_j$, in which case $E = o\ \square, r = \mathtt{err}^k_j$. For (2), $E = o\ E_a$ since $E[r] = (o\ E_a)[r] = o\ E_a[r] = o\ e_a = e$. This decomposition is unique since E_a is unique.

Case $e = \mathtt{mon}^k_j\ e_\kappa\ e_v$**.**
Apply induction to e_κ. Either (1) $e_\kappa \in \mathsf{Ter}$ or (2) there exists a unique E_κ and r such that $e_\kappa = E_\kappa[r]$. For (1), there are two subcases.

Case $e_\kappa = v_\kappa$**.**
Apply induction to e_v. If $e_v = v$ then $E = \square, r = \mathtt{mon}^k_j\ v_\kappa\ v = e$. If $e_v = E_v[r]$ then $E = \mathtt{mon}^k_j\ v_\kappa\ E_v$.

Case $e_\kappa = \mathtt{err}^k_j$**.**
$E = \mathtt{mon}^k_j\ \square\ e_v, r = \mathtt{err}^k_j$.

For (2), $E = \mathtt{mon}^k_j\ E_\kappa\ e_v$.

Otherwise.
The remaining cases are similar to one of the above. □

C Uniform evaluator proof

The proofs in this section hold for all languages presented in section 4.

Theorem 5.2 (Uniform Evaluator). Either $\mathsf{eval}_{\mathscr{L}}(e)$ is defined or the reduction sequence starting with $\langle e, \emptyset\rangle$ is unbounded.

Proof. By interleaved application of lemma C.1 and lemma C.2. □

Lemma C.1 (Progress). If $\sigma \vdash e$ then either $e \in \mathsf{Ter}$ or $\langle e, \sigma\rangle \longmapsto \langle e', \sigma'\rangle$.

Proof. By lemma B.2 either $e \in \mathsf{Ter}$ or $e = E[r]$. By cases on r.

Case $r = \texttt{if}\ v\ e_t\ e_f$**.**
Either IF-TRUE or IF-FALSE apply.

Case $r = \texttt{add!}\ v_\alpha\ v_a$**.**
Suppose v_α is an address. Since $\sigma \vdash e$, $\mathsf{add}(\sigma, v_\alpha, v)$ is defined, so ADD! applies. If v_α is not an address then ERR-ADD! applies.

Case $r = \texttt{mon}^k_j\ v_\kappa\ v$**.**
By cases on v_κ.

Case $v_\kappa \notin \mathsf{Con}$**.**
ERR-MON applies.

Case $v_\kappa = b$**.**
Either MON-TRUE or MON-FALSE applies.

Case $v_\kappa = \lambda x.e$**.**
MON-FLAT applies.

Case $v_\kappa = v_d \to_i v_c$**.**
MON-FUN applies.

Case $v_\kappa = \texttt{tr}\ v_b\ v_p$**.**
MON-TRACE applies.

Case $v_\kappa = \texttt{co}\ \alpha\ v_p$**.**
MON-COL applies.

Otherwise.
The remaining cases are similar to one of the above. □

Lemma C.2 (Preservation). If $\sigma \vdash e$ and $\langle e, \sigma\rangle \longmapsto \langle e', \sigma'\rangle$ then $\sigma' \vdash e'$.

Proof. By cases on the reduction relation.

Case $\langle E[\texttt{if}\ v\ e_t\ e_f], \sigma\rangle \longmapsto \langle E[e_t], \sigma\rangle, v \neq \texttt{false}$**.**
Since $\sigma \vdash \texttt{if}\ v\ e_t\ e_f$ we know $\sigma \vdash e_t$.

Case $\langle E[(\lambda x.e_b)\ v], \sigma\rangle \longmapsto \langle E[e_b[v/x]], \sigma\rangle$**.**
This follows from lemma C.3.

Case $\langle E[\texttt{add!}\ \alpha\ v], \sigma\rangle \longmapsto \langle E[\alpha], \mathsf{add}(\sigma, \alpha, v)\rangle$**.**
This follows from lemma C.4.

Case $\langle E[\texttt{mon}^k_j\ (\texttt{tr}\ v_b\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\texttt{mon}^k_j\ (v_b\ (\texttt{co}\ \alpha\ v_p))\ v], \sigma[\alpha \mapsto \texttt{null}]\rangle$**.**
The contractum is closed since no new variables are introduced. A new address α is introduced. For the expression to remain valid, $\sigma \Vdash \sigma(\alpha)$ must hold which it does since $\sigma(\alpha) = \texttt{null}$.

Case $\langle E[\texttt{mon}^k_j\ (\texttt{co}\ \alpha\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\texttt{mon}^k_j\ (v_p\ (\texttt{add!}\ \alpha\ v))\ v], \sigma\rangle$**.**
No variables are introduced, no addresses are introduced, and the store is maintained. Therefore, the contractum remains closed with addresses still to valid queues.

Otherwise.
The remaining cases are similar to one of the above. □

$$\lambda x.\texttt{let}\ x_j = \texttt{mon}_j^{l,j}\ v_d\ x\ \texttt{in}\ \texttt{let}\ x_k = \texttt{mon}_j^{l,k}\ v_d\ x\ \texttt{in}\ \texttt{mon}_j^{k,l}\ (v_c\ x_j)\ (v\ x_k) \sim \lambda x.\texttt{let}\ x_g = \texttt{mon}_j^{l}\ \widetilde{v_d}\ x\ \texttt{in}\ \texttt{let}\ x_j = x_g \cdot j\ \texttt{in}\ \texttt{let}\ x_k = x_g \cdot kn\ \texttt{in}\ \texttt{mon}_j^{k,l}\ (\widetilde{v_c}\ x_j)\ (\widetilde{v}\ x_k)$$

$$\square \sim \square$$

$$\texttt{mon}_j^{k,l}\ E\ e \sim (\texttt{mon}_j^{k}\ \widetilde{E}\ \widetilde{e}) \cdot l$$

$$\texttt{mon}_j^{k,l}\ v\ E \sim (\texttt{mon}_j^{k}\ \widetilde{v}\ \widetilde{E}) \cdot l$$

$$\ldots$$

$$\texttt{let}\ x_j = e_j\ \texttt{in}\ \texttt{let}\ x_k = e_k\ \texttt{in}\ \texttt{mon}_j^{k,l}\ (v_c\ x_j)\ (v\ x_k) \sim \texttt{let}\ x_j = \widetilde{e_j}\ \texttt{in}\ \texttt{let}\ x_k = \widetilde{e_k}\ \texttt{in}\ \texttt{mon}_j^{k,l}\ (\widetilde{v_c}\ x_j)\ (\widetilde{v}\ x_k)$$

$$\texttt{let}\ x_k = e_k\ \texttt{in}\ \texttt{mon}_j^{k,l}\ (v_c\ v_j)\ (v\ x_k) \sim \texttt{let}\ x_k = \widetilde{e_k}\ \texttt{in}\ \texttt{mon}_j^{k,l}\ (\widetilde{v_c}\ \widetilde{v_j})\ (\widetilde{v}\ x_k)$$

$$\texttt{mon}_j^{k,l}\ e_\kappa\ e \sim (\texttt{mon}_j^{k}\ \widetilde{e_\kappa}\ \widetilde{e}) \cdot l$$

$$\ldots$$

Fig. 16. Expression and Evaluation Context Simulation Relation

Lemma C.3 (Substitution Preservation). If $\sigma \vdash \lambda x.e_b$ and $\sigma \vdash v$ then $\sigma \vdash e_b[v/x]$.

Proof. By induction on e_b. □

Lemma C.4 (Store Preservation). If $\sigma \vdash \alpha$ and $\sigma \vdash v$ then $\mathsf{add}(\sigma, \alpha, v) \vdash \alpha$.

Proof. By induction on $|\mathsf{dom}(\sigma)| - \alpha$. □

D Evaluator equivalence proof

This section shows the equivalence of Λ_B and Λ_C in the absence of queue mutations. Because no mutation occurs, the store is irrelevant to reduction calculations and is thus omitted. The proof proceeds by a simulation argument. Figure 16 relates Λ_B expressions and evaluation contexts to equivalent ones in Λ_C.

Lemma D.1 (Mutation Freedom). If expression e contains no `queue` subexpression, then it is mutation free.

Proof. Assume to the contrary that $\langle e, \emptyset\rangle \longmapsto^\star \langle e', \emptyset\rangle \longmapsto \langle e'', \sigma\rangle$ for $\sigma \neq \emptyset$. The latter reduction must be QUEUE because the only other store-manipulating rule, ADD!, presupposes a non-empty store. However, this is a contradiction since QUEUE only applies if the initial program e contains a `queue` subexpression. □

Theorem 5.3 (Evaluator Equivalence). If e is mutation free, then $\mathsf{eval}_{\Lambda_B}(e) = \mathsf{eval}_{\Lambda_C}(e)$.

Note. By design, trace contracts use mutation and the existing behavior of dependent function contracts is inappropriate for this case. Conversely, queue-mutating programs are excluded because it is the purpose of Λ_C to specify a behavior for $\rightarrow_i$ that is appropriate when contracts perform mutation.

Proof. There are two directions to prove. First, that $\mathsf{eval}_{\Lambda_B} \subseteq \mathsf{eval}_{\Lambda_C}$ on the restricted domain of mutation-free expressions. By cases on $\mathsf{eval}_{\Lambda_B}(e)$.

Case $\mathsf{eval}_{\Lambda_B}(e) = b$**.**

Thus $e \longmapsto^{\star}_{\Lambda_B} b$. Because $e \sim e$, lemma D.2 yields $e \longmapsto^{\star}_{\Lambda_C} b_f$ and there exists $\widetilde{b}$ such that $b_f \simeq_{\text{obs}} \widetilde{b}$ and $b \sim \widetilde{b}$. Observational equivalence and the simulation both preserve Booleans, therefore $b_f = \widetilde{b} = b$. Hence, $e \longmapsto^{\star}_{\Lambda_C} b$ and $\mathsf{eval}_{\Lambda_C}(e) = b$.

Case $\mathsf{eval}_{\Lambda_B}(e) = \texttt{opaque}$**.**

Similar to the prior case since preserving Booleans also implies preserving non-Booleaness.

The inverse direction states that $\mathsf{eval}_{\Lambda_C} \subseteq \mathsf{eval}_{\Lambda_B}$. There is only one interesting case, namely showing that the situation where $e \longmapsto^{\star}_{\Lambda_C} t$ but $\mathsf{eval}_{\Lambda_B}(e)$ is undefined is impossible.

Assume the contrary. Using lemma D.3 yields a contradiction. By theorem 5.2, the reduction sequence in Λ_B is unbounded. Let $e \longmapsto^{\star}_{\Lambda_B} e'$ and $e \longmapsto^{\star}_{\Lambda_C} \widetilde{e'}$ where $e' \sim \widetilde{e'}$ are the last pair of expressions related under $\sim$. This choice is possible since the reduction sequence in Λ_C is finite. Because e' can take a step, lemma D.3 applies and generates a later pair of related expressions, contradicting the choice of $e' \sim \widetilde{e'}$. □

Lemma D.2 (Transitive Simulation). Let e be mutation free. If $e \longmapsto^{\star}_{\Lambda_B} t$ and $e \sim \widetilde{e}$, then there exists t_f and $\widetilde{t}$ such that $\widetilde{e} \longmapsto^{\star}_{\Lambda_C} t_f$, $t_f \simeq_{\text{obs}} \widetilde{t}$, and $t \sim \widetilde{t}$.

Proof. By induction on the number of steps n in $e \longmapsto^{\star}_{\Lambda_B} t$.

Case $n = 0$**.**

Trivial.

Case $n > 0$**.**

By lemma D.3, $e \longmapsto^{+}_{\Lambda_B} e''$, $\widetilde{e} \longmapsto^{+}_{\Lambda_C} e_i$, $e_i \simeq_{\text{obs}} \widetilde{e''}$, and $e'' \sim \widetilde{e''}$. From lemma B.1, $e'' \longmapsto^{\star}_{\Lambda_B} t$. Applying the inductive hypothesis yields $\widetilde{e''} \longmapsto^{\star}_{\Lambda_C} t_i$ where $t_i \simeq_{\text{obs}} \widetilde{t}$ and $t \sim \widetilde{t}$. In summary, $\widetilde{e} \longmapsto^{+}_{\Lambda_C} e_i \simeq_{\text{obs}} \widetilde{e''} \longmapsto^{\star}_{\Lambda_C} t_i \simeq_{\text{obs}} \widetilde{t}$, which suffices.

□

Lemma D.3 (Simulation). Let e be mutation free and $e \sim \widetilde{e}$. If $e \longmapsto_{\Lambda_B} e'$, then there exists e'', e_i, $\widetilde{e''}$ such that $e \longmapsto^{+}_{\Lambda_B} e''$, $\widetilde{e} \longmapsto^{+}_{\Lambda_C} e_i$, $e_i \simeq_{\text{obs}} \widetilde{e''}$, and $e'' \sim \widetilde{e''}$.

Proof. By cases on $e \longmapsto_{\Lambda_B} e'$. Each case relies on lemma D.4 followed by lemma D.5.

Case $E[\texttt{if}\ v\ e_t\ e_f] \longmapsto E[e_t], v \neq \texttt{false}$**.**

Let $\widetilde{e} = \widetilde{E}[\texttt{if}\ \widetilde{v}\ \widetilde{e_t}\ \widetilde{e_f}]$. The simulation preserves non-Booleans, so $\widetilde{v} \neq \texttt{false}$. Thus, $\widetilde{E}[\texttt{if}\ \widetilde{v}\ \widetilde{e_t}\ \widetilde{e_f}] \longmapsto \widetilde{E}[\widetilde{e_t}]$.

Case $E[(\lambda x.\texttt{let}\ x_j = \_\_)\ v] \longmapsto E[\texttt{let}\ x_j = \_\_]$**.**

This reduction implies that $\widetilde{E}[(\lambda x.\texttt{let}\ x_g = \_\_)\ \widetilde{v}] \longmapsto \widetilde{E}[e_i]$ where

$$
\begin{aligned}
e_i = \ & \texttt{let}\ x_g = \texttt{mon}^{l}_{j}\ \widetilde{v_d}\ \widetilde{v}\ \texttt{in} \\
& \texttt{let}\ x_j = x_g \cdot j\ \texttt{in} \\
& \texttt{let}\ x_k = x_g \cdot k\ \texttt{in} \\
& \texttt{mon}^{k,l}_{j}\ (\widetilde{v_c}\ x_j)\ (\widetilde{v}\ x_k).
\end{aligned}
$$

Because e is mutation free, $e_i \simeq_{\text{obs}} \widetilde{e'}$ where

$$\widetilde{e'} = \texttt{let}\, x_j = (\texttt{mon}_j^l\ \widetilde{v_d}\ \widetilde{v}) \cdot j\ \texttt{in}$$
$$\texttt{let}\, x_k = (\texttt{mon}_j^l\ \widetilde{v_d}\ \widetilde{v}) \cdot k\ \texttt{in}$$
$$\texttt{mon}_j^{k,l}\ (\widetilde{v_c}\ x_j)\ (\widetilde{v}\ x_k).$$

Thus, $\widetilde{E}[e_i] \simeq_{\text{obs}} \widetilde{E}[\widetilde{e'}]$. Note that $e' \sim \widetilde{e'}$, therefore $E[\texttt{let}\, x_j = \_\_] \sim \widetilde{E}[\widetilde{e'}]$.

Case $E[\texttt{let}\, x_j = v_j\ \texttt{in}\ \_\_] \longmapsto E[\texttt{let}\, x_k = e_k\ \texttt{in}\ \_\_]$**.**

$$\widetilde{E}[\texttt{let}\, x_j = \widetilde{v_j}\ \texttt{in}\ \_\_] \longmapsto \widetilde{E}[\texttt{let}\, x_k = \widetilde{e_k}\ \texttt{in}\ \_\_]$$

Case $E[\texttt{let}\, x_k = v_k\ \texttt{in}\ \_\_] \longmapsto E[\texttt{mon}_j^{k,l}\ (v_c\ v_j)\ (v\ v_k)]$**.**

$$\widetilde{E}[\texttt{let}\, x_k = \widetilde{v_k}\ \texttt{in}\ \_\_] \longmapsto \widetilde{E}[\texttt{mon}_j^{k,l}\ (\widetilde{v_c}\ \widetilde{v_j})\ (\widetilde{v}\ \widetilde{v_k})]$$

Case $E[\texttt{mon}_j^{k,l}\ \texttt{true}\ v] \longmapsto E[v]$**.**

$$\widetilde{E}[(\texttt{mon}_j^{k,l}\ \texttt{true}\ \widetilde{v}) \cdot l] \longmapsto \widetilde{E}[(\texttt{grd}_j^{k,l}\ \texttt{true}\ \widetilde{v}) \cdot l] \longmapsto \widetilde{E}[\widetilde{v}]$$

Case $E[\texttt{mon}_j^{k,l}\ \texttt{false}\ v] \longmapsto E[\texttt{err}_j^k]$**.**

$$\widetilde{E}[(\texttt{mon}_j^{k,l}\ \texttt{false}\ \widetilde{v}) \cdot l] \longmapsto \widetilde{E}[\texttt{err}_j^k \cdot l] \simeq_{\text{obs}} \widetilde{E}[\texttt{err}_j^k]$$

Case $E[\texttt{mon}_j^{k,l}\ (\lambda x.e)\ v] \longmapsto E[\texttt{mon}_j^{k,l}\ ((\lambda x.e)\ v)\ v]$**.**

$$\widetilde{E}[(\texttt{mon}_j^{k,l}\ (\lambda x.\widetilde{e})\ \widetilde{v}) \cdot l] \longmapsto \widetilde{E}[(\texttt{mon}_j^{k,l}\ ((\lambda x.\widetilde{e})\ \widetilde{v})\ \widetilde{v}) \cdot l]$$

Case $E[\texttt{mon}_j^{k,l}\ (v_d \to_i v_c)\ v] \longmapsto E[\lambda x.\texttt{let}\, x_j = \_\_]$**.**

$$\widetilde{E}[(\texttt{mon}_j^{k,l}\ (\widetilde{v_d} \to_i \widetilde{v_c})\ \widetilde{v}) \cdot l] \longmapsto \widetilde{E}[(\texttt{grd}_j^{k,l}\ (\widetilde{v_d} \to_i \widetilde{v_c})\ \widetilde{v}) \cdot l] \longmapsto \widetilde{E}[\lambda x.\texttt{let}\, x_g = \_\_]$$

Otherwise.

The remaining cases are similar to one of the above or are standard. □

Lemma D.4 (Simulation Decomposition). If $e \sim \widetilde{e}$ and $e = E[e_s]$, then exists $\widetilde{E}$ and $\widetilde{e_s}$ such that $\widetilde{e} = \widetilde{E}[\widetilde{e_s}]$ where $E \sim \widetilde{E}$ and $e_s \sim \widetilde{e_s}$.

Proof. By induction on $e \sim \widetilde{e}$. □

Lemma D.5 (Simulation Composition). If $E \sim \widetilde{E}$ and $e \sim \widetilde{e}$, then $E[e] \sim \widetilde{E}[\widetilde{e}]$.

Proof. By induction on $E \sim \widetilde{E}$. □

E Compiler correctness proof

This section proves that the compiler is correct. Like appendix D, the proof follows from a simulation argument. However, the simulation relation is the compiler function $\mathscr{C}$ itself extended to the evaluation syntax. Since the evaluation syntax contains collectors, $\mathscr{C}$ defines the compilation of collectors following the description in section 6.1. Figure 17 defines the relevant extension of $\mathscr{C}$.

$$\mathscr{C}(e \cdot l) = \mathscr{C}(e) \cdot l \qquad \mathscr{C}(\square) = \square$$

$$\mathscr{C}(\texttt{tr}\ e_b\ e_p) = \begin{cases} \texttt{let}\ x_b = \mathscr{C}(e_b)\ \texttt{in} \\ \texttt{let}\ x_p = \mathscr{C}(e_p)\ \texttt{in} \\ \lambda\_.\texttt{let}\ x_\alpha = \texttt{queue}\ \texttt{in} \\ \quad x_b\ \mathscr{C}(\texttt{co}\ x_\alpha\ x_p) \end{cases} \qquad \begin{aligned} \mathscr{C}(\texttt{tr}\ E\ e) &= \texttt{tr}\ \mathscr{C}(E)\ \mathscr{C}(e) \\ \mathscr{C}(\texttt{tr}\ v\ E) &= \texttt{tr}\ \mathscr{C}(v)\ \mathscr{C}(E) \\ &\ldots \end{aligned}$$

$$\mathscr{C}(\texttt{co}\ \alpha\ v_p) = \lambda y.v_p\ (\texttt{add!}\ \alpha\ y)$$

$$\ldots$$

Fig. 17. Expression and Evaluation Context Compiler

Theorem 6.1 (Compiler Correctness). $\mathsf{eval}_{\Lambda_T} = \mathsf{eval}_{\Lambda_C} \circ \mathscr{C}$

Proof. Similar to the proof of theorem 5.3. Let $e \in \Lambda_T$. It suffices to show that if $\sigma \vdash e$ and $\langle e, \sigma\rangle \longmapsto \langle e', \sigma'\rangle$, then there exists e'' and σ'' such that $\langle e, \sigma\rangle \longmapsto^\star \langle e'', \sigma''\rangle$ and $\langle \mathscr{C}(e), \mathscr{C} \circ \sigma\rangle \longmapsto^\star \langle \mathscr{C}(e''), \mathscr{C} \circ \sigma''\rangle$. By cases on $\langle e, \sigma\rangle \longmapsto \langle e', \sigma'\rangle$.

Case $\langle E[\texttt{mon}^k_j\ (\texttt{tr}\ v_b\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\texttt{mon}^k_j\ (v_b\ (\texttt{co}\ \alpha\ v_p))\ v], \sigma[\alpha \mapsto \texttt{null}]\rangle$**.**
The compiled reduction sequence mirrors this step:

$$\begin{aligned}
& \langle \mathscr{C}(E[\texttt{mon}^k_j\ (\texttt{tr}\ v_b\ v_p)\ v]), \mathscr{C} \circ \sigma\rangle \\
= \quad & \langle \mathscr{C}(E)[\texttt{mon}^k_j\ \mathscr{C}(\texttt{tr}\ v_b\ v_p)\ \mathscr{C}(v)], \mathscr{C} \circ \sigma\rangle \\
= \quad & \langle \mathscr{C}(E)[\texttt{mon}^k_j\ (\texttt{let}\ x_b = \mathscr{C}(v_b)\ \texttt{in} \\
& \qquad\qquad \texttt{let}\ x_p = \mathscr{C}(v_p)\ \texttt{in} \\
& \qquad\qquad \lambda_.\texttt{let}\ x_\alpha = __\ )\ \mathscr{C}(v)], \mathscr{C} \circ \sigma\rangle \\
\longmapsto^+ \quad & \langle \mathscr{C}(E)[\texttt{mon}^k_j\ (\texttt{let}\ x_\alpha = \texttt{queue}\ \texttt{in} \\
& \qquad\qquad \mathscr{C}(v_b)\ \mathscr{C}(\texttt{co}\ \mathscr{C}(v_p)\ x_\alpha))\ \mathscr{C}(v)], \mathscr{C} \circ \sigma\rangle \\
\longmapsto^+ \quad & \langle \texttt{mon}^k_j\ \mathscr{C}(v_b)\ \mathscr{C}(\texttt{co}\ \alpha\ \mathscr{C}(v_p))\ \mathscr{C}(v), \mathscr{C} \circ \sigma'\rangle
\end{aligned}$$

Case $\langle E[\texttt{mon}^k_j\ (\texttt{co}\ \alpha\ v_p)\ v], \sigma\rangle \longmapsto \langle E[\texttt{mon}^k_j\ (v_p\ (\texttt{add!}\ \alpha\ v))\ v], \sigma\rangle$**.**

$$\begin{aligned}
& \langle \mathscr{C}(E[\texttt{mon}^k_j\ (\texttt{co}\ \alpha\ v_p)\ v]), \mathscr{C} \circ \sigma\rangle \\
= \quad & \langle \mathscr{C}(E)[\texttt{mon}^k_j\ \mathscr{C}(\texttt{co}\ \alpha\ v_p)\ \mathscr{C}(v)], \mathscr{C} \circ \sigma\rangle \\
= \quad & \langle \mathscr{C}(E)[\texttt{mon}^k_j\ (\lambda y.\ __\ )\ \mathscr{C}(v)], \mathscr{C} \circ \sigma\rangle \\
\longmapsto^+ \quad & \langle \mathscr{C}(E)[\texttt{mon}^k_j\ (\mathscr{C}(v_p)\ (\texttt{add!}\ \alpha\ \mathscr{C}(v)))\ \mathscr{C}(v)], \mathscr{C} \circ \sigma\rangle
\end{aligned}$$

Otherwise.
The remaining cases are straightforward.

The inverse direction follows from an argument similar to the one made in the proof of theorem 5.3. □

Lemma E.1 (Simulation Decomposition). $\mathscr{C}(E[e]) = \mathscr{C}(E)[\mathscr{C}(e)]$

Proof. By induction on E. □

F Trace contracts for `racket/draw`

The following items describe the properties that `racket/draw`[12] either maintains through defensive-programming checks or documents but does not check:

1. A call to `get-data-from-file` must return `false` unless the bitmap is created with `save-data-from-file` and the image is loaded successfully.
2. The `load-file` method of `bitmap%` cannot be called with bitmaps created by `make-platform-bitmap`, `make-screen-bitmap`, or `make-bitmap` in `canvas%`.
3. The methods `get-text-extent`, `get-char-height`, and `get-char-width` can be called before a bitmap is installed. All others must be called after a bitmap is installed.
4. The method `set-argb-pixels` cannot be called if the given bitmap is produced by `make-screen-bitmap` or `make-bitmap` in `canvas%`.
5. A bitmap can be installed into at most one bitmap drawing context and only when it is not used by a control (as a label), a `pen%`, or a `brush%`.
6. A brush cannot be modified while it is installed into a drawing context.
7. A brush cannot be modified if it is obtained from a `brush-list%`.
8. A color cannot be modified if it is created by passing a string to `make-object` or by retrieving a color from the color database.
9. The methods `start-doc`, `start-page`, `end-page`, and `end-doc` from `dc<%>` must be called in the correct order.
10. Some methods of `dc-path%` extend an open sub-path, some close an open sub-path, and some add closed sub-paths to an existing path. Those must all be kept consistent, e.g. if a method can only extend an open sub-path, then it cannot be called on an object where no sub-path is open.
11. A pen cannot be modified if it is obtained from a `pen-list%`.
12. A pen cannot be modified while it is installed into a drawing context.
13. If `as-eps` is set in a `post-script-dc%` object, then only one page can be created.
14. The `is-empty?` method of `region%` can only be called when associated with a drawing context.
15. There are no restrictions on the sequence of `start-doc`, `start-page`, `end-page`, and `end-doc` for `record-dc%`.

The revision of `racket/draw` enforces all of these properties with trace contracts.

[12] https://docs.racket-lang.org/draw/