Effectful Software Contracts

CAMERON MOY, PLT @ Northeastern University, USA
CHRISTOS DIMOULAS, PLT @ Northwestern University, USA
MATTHIAS FELLEISEN, PLT @ Northeastern University, USA

Software contracts empower programmers to describe functional properties of components. When it comes to constraining effects, though, the literature offers only one-off solutions for various effects. It lacks a universal principle. This paper presents the design of an effectful contract system in the context of effect handlers. A key metatheorem shows that contracts cannot unduly interfere with a program's execution. An implementation of this design, along with an evaluation of its generality, demonstrates that the theory can guide practice.

CCS Concepts: • **Software and its engineering** → **Semantics**.

Additional Key Words and Phrases: effect handlers, software contracts

ACM Reference Format:
Cameron Moy, Christos Dimoulas, and Matthias Felleisen. 2024. Effectful Software Contracts. *Proc. ACM Program. Lang.* 8, POPL, Article 88 (January 2024), 28 pages. https://doi.org/10.1145/3632930

1 CONTRACTS AND EFFECTS: UBIQUITOUS YET IGNORED

For many years, functional programming languages have included constructs for expressing and checking higher-order behavioral contracts [Findler and Felleisen 2002; Keil and Thiemann 2015b; Xu 2014; Xu et al. 2009]. With such contracts, programmers state function specifications and the language's runtime checks them.[1] Concretely, a contract describes the promises a library makes about exported values, and the expectations it imposes on uses [Meyer 1988, 1992]. Put differently, contracts represent agreements between modules about values that flow from one to the other.

Although these contract systems deal with a wide range of functional properties, none can systematically express properties concerning effects. For example, a library that parallelizes `map` computations [Dean and Ghemawat 2008] should enforce—but often does not—that the function argument to `map` is pure. Similarly, when a module exports a function that mutates a hash table, its interface should promise client modules—but often cannot—that it modifies only the given table.

The literature is teeming with *ad hoc* solutions: affine contracts [Tov and Pucella 2010] to interoperate with substructural type systems; framing contracts [Shinnar 2011] to limit mutation; temporal contracts [Disney et al. 2011] to monitor protocols; authorization contracts [Moore et al. 2016] to enforce access control; size-change contracts [Nguyễn et al. 2019] to guarantee termination; trace contracts [Moy and Felleisen 2023] to check properties across multiple calls. All of these systems use effects in *contracts* to constrain effects in *code*. No existing work supplies a unified approach for doing so, however.

[1]The presentation here focuses on run-time checks, but some tools [Nguyễn et al. 2018; Xu 2012] can partially verify higher-order contracts at compile time and generate residual run-time checks for unverified properties.

Authors' addresses: Cameron Moy, PLT @ Northeastern University, Boston, Massachusetts, USA, camoy@ccs.neu.edu; Christos Dimoulas, PLT @ Northwestern University, Evanston, Illinois, USA, chrdimo@northwestern.edu; Matthias Felleisen, PLT @ Northeastern University, Boston, Massachusetts, USA, matthias@ccs.neu.edu.

ACM 2475-1421/2024/1-ART88
https://doi.org/10.1145/3632930

This paper presents *effect-handler contracts*, a universal mechanism for expressing and monitoring properties of effectful code (Section 2). Its central contribution is a formal semantics of effectful software contracts (Section 3). The model consists of a language where effectful operations are expressed in terms of effect requests and handlers [Plotkin and Pretnar 2009], not as primitive operations; in the context of such a language, effect-handler contracts suffice to check a broad class of constraints. An extension to the model (Section 4) formalizes dependent variants of these contract forms. The model is carefully constructed to satisfy an *erasure* property (Section 5), meaning that contracts cannot interfere with a program's computation, other than signaling an error and stopping the world. It also satisfies *blame correctness*, meaning contracts correctly identify components serving values that break the contract assertion.

A secondary contribution is an implementation based on this design. The implementation is a standalone language within the Racket ecosystem [Felleisen et al. 2018] that has both effect handlers and effect-handler contracts (Section 6). A thorough literature survey confirms that effect-handler contracts subsume many existing constructs from prior work (Section 7).

2 EFFECT-HANDLER CONTRACTS, INFORMALLY

This section presents an effect-handler language that extends a functional core with constructs for requesting effects, interpreting effects, and contracts governing effects. While adding ordinary higher-order functional contracts to such a language is straightforward, extending it with contracts on effects requires careful language design.

The first subsection presents the syntax of the model, while the second subsection illustrates the semantics informally, using a series of code snippets that add up to a complete example.

2.1 Syntax and Informal Semantics

The model's syntax will be presented in three steps: the untyped by-value λ-calculus [Plotkin 1975]; an extension with functional contracts [Dimoulas and Felleisen 2011]; and an extension with effect handlers [Plotkin and Pretnar 2009] that also includes syntax for contracts governing effects.

CORE

$$
\begin{aligned}
e \in \mathsf{Expr} &= x \mid b \mid f \mid \langle e, e \rangle \mid \texttt{if}\ e\ e\ e \mid e\ e \\
b \in \mathsf{Bool} &= \texttt{true} \mid \texttt{false} \\
f \in \mathsf{Fun} &= o \mid \lambda x.e \\
o \in \mathsf{Op} &= \texttt{fst} \mid \texttt{snd} \\
x, y, z &\in \mathsf{Var}
\end{aligned}
$$

The functional CORE language comes with three built-in data types: Booleans, functions, and pairs. They are eliminated by conditionals, application, and projections, respectively.

CONTRACTS extends CORE

$$
\begin{aligned}
e \in \mathsf{Expr} &= \ldots \mid \kappa \mid \texttt{mon}_j^{k,l}\ e\ e \\
\kappa \in \mathsf{Con} &= b \mid f \mid \langle e, e \rangle \mid e \longrightarrow e \\
j, k, l &\in \mathsf{Lab}
\end{aligned}
$$

The CONTRACTS extension reinterprets the base data types as contracts. As a contract, `true` and `false` accept and reject all values, respectively. Functions, when used as a contract, are predicates that describe *flat* [Findler and Felleisen 2002] first-order constraints. A contract pair $\langle e_1, e_2 \rangle$ checks the first component of a value pair with e_1 and the second component with e_2. A function contract $e_1 \longrightarrow e_2$ protects functions by checking that arguments satisfy e_1 and results satisfy e_2. A monitor $\texttt{mon}_j^{k,l}\ e_1\ e_2$ attaches the contract e_1 to the value of e_2 (the *carrier*). Labels j, k, and l name the

contract-defining, carrier-providing, and carrier-consuming parties, respectively. These labels are used in error messages to *blame* the party responsible for a violation [Findler and Felleisen 2002].

EFFECTS extends CONTRACTS
$e \in \mathsf{Expr} = \ldots \mid \texttt{handle}^m\ e\ \texttt{with}\ e \mid \texttt{do}\, e$
$\kappa \in \mathsf{Con} = \ldots \mid e \triangleright e \mid \Diamond e$
$m \in \mathsf{Mode} = \triangleright \mid \Diamond$

The EFFECTS extension introduces two new pieces of syntax related to effect handlers: `handle` and `do`. Evaluating $\texttt{do}\, v$ requests the effect described by v. The evaluation of a request proceeds by searching for the matching handler in the enclosing evaluation context and supplying it with v. Handlers come in two flavors:

- $\texttt{handle}^{\triangleright}\ e\ \texttt{with}\ e_h$ is a *main-effect handler*. It interprets only effects performed by *ordinary code* in the body expression e using the handler e_h.
- $\texttt{handle}^{\Diamond}\ e\ \texttt{with}\ e_h$ is a *contract-effect handler*. It interprets only effects performed by *contract-checking code* in the body expression e using the handler e_h.

Note. Two handler forms are needed to eliminate effect interference. If $\texttt{handle}^{\triangleright}$ were to interpret effects at the contract level, a contract could use this channel of communication to change the outcome of a program. By interpreting effects at different levels with different handlers, contract code cannot affect the result of a program. Thus, if a flat contract requests an effect via $\texttt{do}\, v$, it is not interpreted by a $\texttt{handle}^{\triangleright}$ form, even if it is the nearest enclosing handler. ▪

Symmetrically, *effect-handler contracts* also demand two constructs, one per level. Both of these forms monitor a function f that may request effects:

- $e_1 \triangleright e_2$ is a *main-effect contract*. It ensures that effects performed during the application of f satisfy e_1 and values received from the handler satisfy e_2.
- $\Diamond\, e$ is a *contract-handler contract*. It handles, using e, effect requests during the application of f that occur during the dynamic extent of a contract check.

2.2 Examples, Informally

The model suffices to establish essential metaproperties, but illustrating the ideas with such a spartan syntax is too cumbersome. Hence, this section uses ML-like syntactic sugar to present simple examples that illustrate the informal semantics of contracts, effect handlers, and effect-handler contracts. For interesting examples, rather than synthetic ones, see Section 7.1.

Higher-Order Contracts. The RSA cryptographic algorithm is widely used for secure communication [Rivest et al. 1978]. Crucially, RSA relies on the difficulty of factoring prime numbers. Here is the sketch of an RSA-key-generating function, using first-class contracts on a higher-order function to describe the primality constraint:

```
let p_gen_c = is_unit ⟶ ⟨is_prime, is_prime⟩
let k_gen_c = is_unit ⟶ ⟨is_key, is_key⟩
let rsa_c = p_gen_c ⟶ k_gen_c
let rsa : rsa_c = ── elided ──
```

The contract on `rsa`, attached with a colon, tells the reader that `rsa` is a function that accepts a pair-of-primes-generating thunk and returns a key-pair-generating thunk. Contracts are first-class values and can be defined using `let`. The $\cdot \longrightarrow \cdot$ combinator protects functions by composing an argument contract and a result contract. Furthermore, unlike a type for such a function, contracts can employ user-defined predicates, e.g., `is_prime`, to check the validity of arguments and results.

If the runtime discovers a contract violation—possibly in a distant client module—an error is signaled identifying the violated contract and blaming the responsible party. Given an invalid p_gen function—say, one that does not generate primes—the contract system identifies the source of the violation like this:

```
> let bad_p_gen () = ⟨3, 4⟩
> rsa bad_p_gen ()
rsa: contract violation
  expected: is_prime
  given: 4
  blaming: bad_p_gen
  (assuming the contract is correct)
```

Note. Contracts on their own can enforce only safety properties; they do not suffice to establish security properties. Abstractions that enforce security can be built on top of contracts though, as shown in Section 6.3. ▪

Main-Effect Handlers. A key requirement of RSA is that the generated prime numbers are random. To generate random primes, there must be some way to generate ordinary random numbers. A pseudorandom number generator (PRNG) is a deterministic algorithm for generating numbers with properties similar to truly random numbers. The interface to most PRNGs is effectful: generating a random number causes the PRNG's internal state to change.

In a language with effect handlers, a PRNG function collaborates with an effect handler via effect-request messages to realize state changes [Pretnar 2015]:

```
data gen = Gen
let rand () = do Gen

let prng_h req kont =
  match req with
  | Gen → λs.kont (prng_extract s) (prng_next s)
  | _   → λs.kont (do req) s

let run_with_prng thk seed =
  (handle▷ (let r = thk () in λ_.r) with prng_h) (prng_init seed)
```

Consider the run_with_prng function. Given a thunk and a random seed, it runs the thunk in a context that makes random-number generation available. To provide this service, run_with_prng applies the thunk inside handle▷ with prng_h, an effect handler that interprets requests for generating a random number. This handler function takes two arguments: the requested effect and a continuation that reifies the computation between the origin of the effect request and the handler.[2]

When thk needs a random number, it applies rand, which in turn, issues a do Gen effect request. The handler of a request packages up the request (Gen) and the delimited continuation to give the handler function. Once the handler function receives these values, it constructs a λ that, when given the PRNG state, invokes the continuation (kont) with the next random number. The context then applies this function to the PRNG state. If some other effect is requested, the handler propagates the request to an outer handler. Propagation works by applying the continuation to a

[2]Effect handlers come in two flavors: *deep* [Cartwright and Felleisen 1994; Plotkin and Pretnar 2009] and *shallow* [Hillerström and Lindley 2018]. In the deep setting, the delimited continuation includes the handler itself; in the shallow one, it does not. The handle▷ form uses the deep flavor.

renewed request. Since `do req` is not a value, the call-by-value semantics ensures that the request is handled before the continuation is resumed.

As a reminder, effect composition is a key benefit of using an effect-handler-based language instead of a language with primitive effects. Since an effect-handler language expresses effects *uniformly*, it is straightforward to reinterpret them, too. In particular, a programmer can replace or supplement the default PRNG provided by `run_with_prng`, without changing the computation (`thk`) at all. For example, here is a handler that biases random numbers toward extreme values by squaring them:

```
let bias_h req kont =
  let bias x = if req = Gen then x * x else x in
  kont (bias (do req))

let run_with_bias thk = handle▷ thk () with bias_h
```

Assuming the original generator produces reals in $[0, 1]$, this new handler can be composed with the original PRNG to yield a biased generator:

```
run_with_prng (λ_.run_with_bias (λ_.rand ())) 0
```

Main-Effect Contracts. In the presence of I/O effects, the contract for `rsa` does not suffice. A program may accidentally (or intentionally) use a prime-generating function that reveals more information than desired:

```
let bad_p_gen_v2 () =
  let ⟨p, q⟩ = —— elided —— in
  do (Write "secret.txt" p); ⟨p, q⟩
```

In this snippet, the prime-generating function writes the secret prime p to a file and thus compromises the RSA key. A contract for `rsa` should prohibit the use of effectful arguments such as `bad_p_gen_v2`.

With main-effect contracts, expressing this restriction is straightforward:

```
let p_gen_c_v2 = p_gen_c ⊓ (is_gen ▷ is_real)
```

This revised contract is a conjunction; the ⊓ combinator applies each of the two conjuncts, one after another. Consequently, the prime-generating function must satisfy both. While the first conjunct is the original `p_gen_c` contract, the second one describes a constraint on effects. In this example, `is_gen ▷ is_real` ensures that effect requests satisfy `is_gen`, and that the handler passes only values to the continuation if they satisfy `is_real`. Since `is_gen` returns `true` only for `Gen`, but not `Write`, a use of `bad_p_gen_v2` signals the desired contract violation.[3]

Main-effect contracts are active only during the dynamic extent of the protected function, and not at any other point. Consider the following handler:

```
let printer_h req kont =
  match req with
  | Gen -> let res = do Gen in
           do (Write "secret.txt" res);
           kont res
  | _   -> kont (do req)

let run_with_printer thk = handle▷ thk () with printer_h
```

[3]This example assumes that `data` generates a tag-checking predicate, such as `is_gen`, for each variant.

This handler intercepts all random-number requests and writes them to the filesystem. In the same way as bad_p_gen_v2, this handler can be used to expose information:

```
run_with_prng (λ_.run_with_printer (λ_.rsa p_gen)) 0
```

However, this program does not result in a contract violation even when the contract of the prime-generating function is p_gen_c_v2. When the prime-generating function requests a random number, evaluation moves to the body of the handler printer_h, which is outside the prime generator's dynamic extent. Therefore is_gen ▷ is_real is no longer active when printer_h writes to the filesystem.

The above behavior is by design; it is critical for assigning correct blame. Recall that a contract establishes an agreement between a client and server module. According to p_gen_c_v2, the p_gen function is responsible only for ensuring that its code does not perform forbidden effects directly, or indirectly by calling other functions. Client code, including the code that calls p_gen and the code that handles the legitimate effects p_gen performs, is not restricted by this part of the contract. In other words, blaming p_gen_c_v2 would be wrong even though printer_h writes to the filesystem; it would violate the blame correctness property (Section 5.3).

Contract-Effect Handlers. The p_gen_c_v2 contract guarantees that the thunk always receives a real number from the PRNG handler in response to its requests, but gives no assurance that these real numbers are even somewhat random. A PRNG function that always returns $\frac{1}{2}$ does not cause an error, but yields a useless prime generator. Statistical tests exist to detect faulty PRNGs [Bassham et al. 2010]; a contract can employ such tests to detect obviously bad PRNG implementations.

Consider the simple-minded test that ensures two consecutive random numbers are different:

```
data check = Check is_real

let rec diff_h prev req =
  match req with
  | Check cur ⟶ ⟨prev = cur, diff_h cur⟩
  | _         ⟶ ⟨do req, diff_h prev⟩

let run_with_diff_check thk = handle◊ thk () with (diff_h -1)
```

When executed via run_with_diff_check, contracts can use this test to determine whether the generated random number differs from the previously generated one. The handle◊ form is a restricted handler that interprets effects requested in the dynamic extent of a contract check. It does not get to directly invoke the delimited continuation of the effect request; instead, the handler function is expected to return a pair of values: the effect result and a new handler to replace the current one. Critically, handle◊ affects only contract-checking code because it can transfer values only to contract code.

Note. This restriction is similar to that of a runner [Ahman and Bauer 2020] where, informally, a handler *may* invoke the continuation at most once in tail position. Here, the handler *must* invoke the continuation exactly once in tail position.

Direct access to the delimited continuation would permit tampering with the program's result and would thus allow interference between program code and contract code. For example, a handler could ignore the continuation completely and return an arbitrary value. ▪

Despite its limitations, handle◊ is still quite useful. Adapting p_gen_c_v2 yields a contract with the desired test:

```
let diff_real x = is_real x && do (Check x)
let p_gen_c_v3 = p_gen_c ⊓ (is_gen ▷ diff_real)
```

Here, `diff_real` requests an effect whose purpose is to check whether the latest argument to a function differs from the most recent one. With this contract, and its corresponding effect handler installed, a PRNG that always returns $\frac{1}{2}$ signals a contract error.

The `p_gen_c_v3` example illustrates why contracts themselves may need to perform effects. Moreover, these effects cannot be locally encapsulated within the contract. In this example, state should persist across multiple calls to the prime-generating function. If the state was locally contained to `p_gen_c_v3`, then subsequent invocations of the prime-generating function would reset the state. This approach would allow more faulty PRNGs to pass the contract. More broadly, locally encapsulated effects do not suffice to express many of the systems described in Section 7.

Contract-Handler Contracts. Suppose, for unit testing, the author of `rsa` wants to use a predetermined pool of numbers for random generation, instead of a PRNG. As such, it is important that the number of times a program requests a random number does not exceed the length of the pool. Thus, the contract needs to keep track of this information:

```
data remaining = Remaining

let rec rem_h k req =
  match req with
  | Remaining ⟶ ⟨k, rem_h (k - 1)⟩
  | _         ⟶ ⟨do req, rem_h k⟩

let has_rem req = not (is_gen req) || (do Remaining) > 0
let pool_c k = (is_unit ⟶⟶ is_any) ⊓ (has_rem ▷ is_real) ⊓ ◊(rem_h k)
let run_with_pool (xs : is_list) (thk : pool_c (length xs)) = ── elided ──
```

Like `diff_h` in the previous example, `rem_h` is a contract-effect-handler function. It stores the number of values remaining in the random number pool. Instead of being installed directly using `handle`$^{\Diamond}$, it is installed by `pool_c`. Specifically, the contract-handler contract `◊(rem_h k)` installs the function `rem_h k` using `handle`$^{\Diamond}$. As such, `run_with_pool` executes `thk` in a context where the `Remaining` effect is interpreted by `rem_h` initialized with the size of the pool.

On its own, a contract-handler contract cannot signal a violation. Rather, it supports other contracts that can. Here, that check happens in the ▷ conjunct of `pool_c`. When the `thk` requests a random number, `has_rem` checks if there are still numbers left in the pool. If so, the request is forwarded. Otherwise, an error is raised.

Note. The order of conjuncts in `pool_c` is relevant. Since `has_rem` requires that the `rem_h` handler is installed, it must come earlier in the list of conjuncts than `◊(rem_h k)`. The ⊓ combinator applies contracts left-to-right. Thus, the right-most conjunct creates the outermost wrapper. ▪

3 A FORMAL MODEL OF EFFECT HANDLER CONTRACTS

Defining a semantics amounts to defining an evaluation function that maps programs to answers. Specifying such a function with a reduction relation provides an easy way to prove metatheorems. Following tradition, this section starts with an extension of the model's syntax to an evaluation syntax (Section 3.1). Next, the reduction relation is defined piecemeal across four subsections (Sections 3.2 to 3.5). The reduction rules for contracts differ a bit from conventional definitions—namely, flat contracts have *cascading* behavior where, instead of just a Boolean, they can return any arbitrary contract that is then applied. This behavior, and its purpose in the context of effectful contracts, is examined in the last subsection.

3.1 Evaluation Syntax

The evaluation syntax extends the grammar of expressions and defines the set of values:

EFFECTS (EVAL) extends EFFECTS

$$e \in \mathsf{Expr} = \ldots \mid \mathtt{mark}_j^{k,l}\, v\, e \mid \mathtt{err}_j^k$$
$$v \in \mathsf{Val} = b \mid f \mid \langle v, v \rangle \mid v \longrightarrow v \mid v \rhd v \mid \Diamond v$$

Expressions include marks and errors, which can arise during evaluation but cannot be expressed in written programs. The expression $\mathtt{mark}_j^{k,l}\, v_\kappa\, e$ states that effects requested by e, and their fulfillment, must satisfy contract v_κ. In other words, effect requests "passing through" the mark must satisfy the contract. These marks are installed by main-effect contracts.

Next comes the grammar of evaluation contexts. The reduction relation requires three different kinds of evaluation context, each with a different role:

- $E^\rhd$ is the set of *main-executing contexts* containing regular code that is handled with $\mathtt{handle}^\rhd$.
- $E^\Diamond$ is the set of *contract-executing contexts* describing the dynamic extent of contract code that is handled with $\mathtt{handle}^\Diamond$.
- E is the set of *unrestricted contexts*, which is the union of the (disjoint) sets $E^\rhd$ and $E^\Diamond$.

Here are the elements of the grammar that are shared between each kind of evaluation context:

EFFECTS (EVAL) extends EFFECTS

$$E \in \mathsf{Ctx} = \langle E, e \rangle \mid \langle v, E \rangle \mid \mathtt{if}\, E\, e\, e \mid E\, e \mid v\, E \mid \mathtt{handle}^m\, E\, \mathtt{with}\, v \mid \mathtt{do}\, E \mid E \longrightarrow e \mid v \longrightarrow E \mid E \rhd e \mid v \rhd E \mid \Diamond E \mid \mathtt{mon}_j^{k,l}\, v\, E \mid \mathtt{mark}_j^{k,l}\, v\, E$$
$$E^\rhd \in \mathsf{Ctx}^\rhd = \text{—— } \textit{the above mutatis mutandis} \text{ ——}$$
$$E^\Diamond \in \mathsf{Ctx}^\Diamond = \text{—— } \textit{the above mutatis mutandis} \text{ ——}$$

And here are the elements that differ between the evaluation contexts:

EFFECTS (EVAL) extends EFFECTS

$$E \in \mathsf{Ctx} = \ldots \mid \square \mid \mathtt{mon}_j^{k,l}\, E\, e \mid \mathtt{handle}^m\, e\, \mathtt{with}\, E$$
$$E^\rhd \in \mathsf{Ctx}^\rhd = \ldots \mid \square \mid \mathtt{handle}^\rhd\, e\, \mathtt{with}\, E^\rhd$$
$$E^\Diamond \in \mathsf{Ctx}^\Diamond = \ldots \mid \mathtt{mon}_j^{k,l}\, E\, e \mid \mathtt{handle}^\Diamond\, e\, \mathtt{with}\, E \mid \mathtt{handle}^\rhd\, e\, \mathtt{with}\, E^\Diamond$$

Contract code executes in two syntactic positions: e_κ in $\mathtt{mon}_j^{k,l}\, e_\kappa\, e$, and e_h in $\mathtt{handle}^\Diamond\, e\ \mathtt{with}\ e_h$. While the former is clear, the latter might be a surprise. Recall the purpose of $\mathtt{handle}^\Diamond$: it interprets effect requests that originate in contract code. By implication, e_h may receive and execute higher-order values originating from contract code. Therefore, it *must* be considered contract code.

The definition of evaluation contexts reflects this reasoning. In particular, $E^\rhd$ omits productions of the shape $\mathtt{mon}_j^{k,l}\, E^\rhd\, e$ and $\mathtt{handle}^\Diamond\, e\, \mathtt{with}\, E^\rhd$, while $E^\Diamond$ omits the production for $\square$. This restriction on $E^\Diamond$ ensures that fully formed $E^\Diamond$ contexts contain either $\mathtt{mon}_j^{k,l}\, E\, e$ or $\mathtt{handle}^\Diamond\, e\, \mathtt{with}\, E$.

Finally, formulating the reduction rules and the evaluator requires two more definitions:

EFFECTS (EVAL) extends EFFECTS

$$U \in \mathsf{unhandled} = \{E_1 \mid E_1 \neq E_2[\mathtt{handle}^m\, E_3^m\, \mathtt{with}\, v_h]\}$$
$$s \in \mathsf{stuck} = \{E[v_f\, v] \mid v_f \notin \mathsf{Fun}\} \cup \{E[o\, v] \mid \delta(o, v) \text{ is undefined}\} \cup \{E[\mathtt{do}\, v] \mid E \in \mathsf{unhandled}\} \cup \{E[\mathtt{handle}^\Diamond\, e\, \mathtt{with}\, v_h] \mid v_h \neq \langle v_a, v_b \rangle, v_h \neq f\}$$

An unhandled evaluation context lacks a handler for any effect requests that may originate from an expression plugged into the hole. The set of stuck expressions describes those to which no reduction rule applies, i.e., they are not in the domain of the reduction relation. Examples are the application of non-functions to values or an effect request in the hole of an unhandled context. Instead of dealing with stuck expressions in the reduction relation, the evaluator is defined to produce a sensible error when the reduction relation (transitively) reduces to a stuck expression.

The next four subsections present the one-step reduction relation for complete programs using the evaluation contexts [Felleisen et al. 2009; Felleisen and Hieb 1992]. The evaluator is defined by the reflexive-transitive closure of the union of these relations.

3.2 Core Reduction Rules

$$
\begin{array}{rr}
\textsc{if-true} & E[\texttt{if}\, v\, e_1\, e_2] \longmapsto E[e_1] \,\text{if}\, v \neq \texttt{false} \\
\textsc{if-false} & E[\texttt{if false}\, e_1\, e_2] \longmapsto E[e_2] \\
\textsc{app-lambda} & E[(\lambda x.e)\, v] \longmapsto E[e[v/x]] \\
\textsc{app-op} & E[o\, v] \longmapsto E[\delta(o, v)]
\end{array}
$$

$$
\delta(o, v) = \begin{cases} v_1 & \text{if } o = \texttt{fst}, v = \langle v_1, v_2 \rangle \\ v_2 & \text{if } o = \texttt{snd}, v = \langle v_1, v_2 \rangle \end{cases}
$$

Fig. 1. Core Reduction Rules

Figure 1 displays the core reduction rules, which are entirely standard [Plotkin 1975]. Conditionals and applications of λ are reduced in the expected manner. A δ metafunction interprets primitive operations [Barendregt 1981]; this choice renders the reduction relation easily extensible.

3.3 Contract Reduction Rules

$$
\begin{array}{rr}
\textsc{mon-true} & E[\texttt{mon}_j^{k,l}\, \texttt{true}\, v] \longmapsto E[v] \\
\textsc{mon-false} & E[\texttt{mon}_j^{k,l}\, \texttt{false}\, v] \longmapsto E[\texttt{err}_j^k] \\
\textsc{mon-flat} & E[\texttt{mon}_j^{k,l}\, f\, v] \longmapsto E[\texttt{mon}_j^{k,l}\, (f\, v)\, v\,] \\
\textsc{mon-pair} & E[\texttt{mon}_j^{k,l}\, \langle v_1, v_2 \rangle\, v] \longmapsto E[\texttt{err}_j^k] \,\text{if}\, v \neq \langle v_3, v_4 \rangle \\
\textsc{grd-pair} & E[\texttt{mon}_j^{k,l}\, \langle v_1, v_2 \rangle\, \langle v_3, v_4 \rangle] \longmapsto E[\langle \texttt{mon}_j^{k,l}\, v_1\, v_3, \texttt{mon}_j^{k,l}\, v_2\, v_4 \rangle] \\
\textsc{mon-fun} & E[\texttt{mon}_j^{k,l}\, (v_1 \longrightarrow v_2)\, v] \longmapsto E[\texttt{err}_j^k] \,\text{if}\, v \notin \texttt{Fun} \\
\textsc{grd-fun} & E[\texttt{mon}_j^{k,l}\, (v_1 \longrightarrow v_2)\, f] \longmapsto E[\lambda x.\texttt{mon}_j^{k,l}\, v_2\, (f\, (\texttt{mon}_j^{l,k}\, v_1\, x))]
\end{array}
$$

Fig. 2. Contract Reduction Rules

Figure 2 presents the rules governing contract monitors. Following Dimoulas and Felleisen [2011], $\texttt{mon}_j^{k,l}\, e_\kappa\, e$ monitors the value of e with the contract expression e_κ. The reduction of many contract

expressions is specified via two related rules, prefixed with MON and GRD, respectively. A MON rule checks a first-order property of the carrier. In this model, checking first-order properties amounts to checking whether the top-level *shape* of the carrier value is the expected one. For example, MON-FUN checks whether the carrier is a function. A GRD rule assumes the shape check is satisfied and constructs a wrapper to check either higher-order (as in GRD-FUN) or nested properties (as in GRD-PAIR).

The MON-TRUE and MON-FALSE rules immediately succeed and fail, respectively. When the contract is a predicate f, MON-FLAT applies the predicate to the carrier and uses the result as a contract. Since `true` and `false` double as contracts, this cascading of checks works as expected. It is possible to return non-Boolean contracts as well; Section 3.6 explains why this matters.

The GRD rules cover values that need "deep" checking. A pair of contracts distributes over a pair of values. A function contract yields a wrapper value that checks the argument and result contract when the wrapper is applied. The blame labels on the argument monitor are swapped since the argument position of a function contract is contravariant [Findler and Felleisen 2002].

3.4 Effect-Handler Reduction Rules

$$
\begin{array}{lrl}
\textsc{handle} & E[\texttt{handle}^m\ v\ \texttt{with}\ v_h] & \longmapsto E[v] \\
\textsc{do}^{\triangleright} & E[\texttt{handle}^{\triangleright}\ E^{\triangleright}[\texttt{do}\ v]\ \texttt{with}\ v_h] & \longmapsto E[v_h\ e_v\ (\lambda x.\texttt{handle}^{\triangleright}\ E^{\triangleright}[e_x]\ \texttt{with}\ v_h)] \\
 & & \quad \text{if } E^{\triangleright} \in \text{unhandled} \\
 & & \quad \text{where } e_v = (\uparrow E^{\triangleright})[v], e_x = (\downarrow E^{\triangleright})[x] \\
\textsc{do-pair}^{\Diamond} & E[\texttt{handle}^{\Diamond}\ E^{\Diamond}[\texttt{do}\ v]\ \texttt{with}\ \langle v_1, v_2\rangle] & \longmapsto E[\texttt{handle}^{\Diamond}\ E^{\Diamond}[v_1]\ \texttt{with}\ v_2] \\
 & & \quad \text{if } E^{\Diamond} \in \text{unhandled} \\
\textsc{do-fun}^{\Diamond} & E[\texttt{handle}^{\Diamond}\ E^{\Diamond}[\texttt{do}\ v]\ \texttt{with}\ f] & \longmapsto E[\texttt{handle}^{\Diamond}\ E^{\Diamond}[\texttt{do}\ v]\ \texttt{with}\ (f\ v)] \\
 & & \quad \text{if } E^{\Diamond} \in \text{unhandled}
\end{array}
$$

$$
\boxed{\uparrow : \text{Ctx} \to \text{Ctx}}
\qquad
\begin{array}{rcl}
\uparrow\Box & = & \Box \\
\uparrow\langle E, e\rangle & = & \uparrow E \\
\uparrow\langle v, E\rangle & = & \uparrow E \\
 & \cdots & \\
\uparrow(\texttt{mark}_j^{k,l}\ (v_1 \rhd v_2)\ E) & = & \texttt{mon}_j^{k,l}\ v_1\ (\uparrow E)
\end{array}
$$

$$
\boxed{\downarrow : \text{Ctx} \to \text{Ctx}}
\qquad
\begin{array}{rcl}
\downarrow\Box & = & \Box \\
\downarrow\langle E, e\rangle & = & \downarrow E \\
\downarrow\langle v, E\rangle & = & \downarrow E \\
 & \cdots & \\
\downarrow(\texttt{mark}_j^{k,l}\ (v_1 \rhd v_2)\ E) & = & (\downarrow E)[\texttt{mon}_j^{l,k}\ v_2\ \Box]
\end{array}
$$

Fig. 3. Effect-Handler Reduction Rules

Figure 3 presents the reduction rules for effect handlers. When the body of any handler is a value, the effect computation has run its course and the handler is eliminated. Otherwise, one of the DO rules may apply. The unhandled side condition in all of these rules ensures that only the innermost handler is matched with an effect request. Both $\textsc{do}^{\triangleright}$ and $\textsc{do}^{\Diamond}$ use the special evaluation contexts from Section 3.1 to ensure that the requested effect ($\texttt{do}\ v$) originates from either main-program code or contract code.

The $\textsc{do}^{\triangleright}$ rule specifies main-program handlers as deep. Concretely, the handler is applied to the effect request and a delimited continuation that includes the handler itself. The evaluation context

$E^{\triangleright}$ may contain marks deposited by main-effect contracts. Two metafunctions, $\uparrow$ and $\downarrow$, collect the contracts for main-effect requests and their fulfillment, respectively. Plugging the raw effect request v into the context created by $\uparrow$ produces an expression that performs all of the necessary contract checks. The same goes for x and $\downarrow$.

Note that the blame labels flip for $\downarrow$. The return value, given to the continuation, comes from a handler, which exists in the context of an effect request. As such, swapping the labels is necessary so that blame assignment points to the party that violated the contract [Dimoulas et al. 2011].

By contrast, the DO-PAIR$^{\Diamond}$ and DO-FUN$^{\Diamond}$ rules specify handlers that have no control over the continuation. Furthermore, two rules are needed to distinguish the two contract cases, analogous to the rules for Boolean contracts and predicate contracts. Specifically, the expression e_h in $\texttt{handle}^{\Diamond}\ e\ \texttt{with}\ e_h$ can reduce to either a function or a pair:

- In DO-PAIR$^{\Diamond}$, the first component is plugged into the evaluation context, which is the continuation of the effect request, and the second component becomes the next handler.
- In DO-FUN$^{\Diamond}$, the function is applied to the effect request with the expectation that this new contract expression eventually reduces to a pair. Like MON-FLAT, this rule ensures that contract code is always executed in one syntactic position.

3.5 Effect-Handler Contract Reduction Rules

$$
\begin{array}{rr}
\textsc{mon-handle}^{\triangleright} & E[\texttt{mon}_j^{k,l}\ (v_1 \triangleright v_2)\ v] \longmapsto E[\texttt{err}_j^{k}] \text{ if } v \notin \texttt{Fun} \\
\textsc{grd-handle}^{\triangleright} & E[\texttt{mon}_j^{k,l}\ (v_1 \triangleright v_2)\ f] \longmapsto E[\lambda x.\texttt{mark}_j^{k,l}\ (v_1 \triangleright v_2)\ (f\ x)] \\
\textsc{mark} & E[\texttt{mark}_j^{k,l}\ v_\kappa\ v] \longmapsto E[v] \\
\textsc{mon-handle}^{\Diamond} & E[\texttt{mon}_j^{k,l}\ (\Diamond\, v_h)\ v] \longmapsto E[\texttt{err}_j^{k}] \text{ if } v \notin \texttt{Fun} \\
\textsc{grd-handle}^{\Diamond} & E[\texttt{mon}_j^{k,l}\ (\Diamond\, v_h)\ f] \longmapsto E[\lambda x.\texttt{handle}^{\Diamond}\ (f\ x)\ \texttt{with}\ v_h]
\end{array}
$$

Fig. 4. Effect-Handler Contract Reduction Rules

Finally, Figure 4 presents the reduction rules governing both kinds of effect-handler contract. The MON rules ensure that the carriers are functions; if not, they signal a violation. If the carriers are functions, the contracts act in a higher-order manner via GRD-HANDLE$^{\triangleright}$ and GRD-HANDLE$^{\Diamond}$.

The GRD-HANDLE$^{\triangleright}$ rule simply installs a mark that constrains effects performed in f. Actually checking these contracts is delegated to DO$^{\triangleright}$. Once the dynamic extent of a mark expression is over, the mark itself can be eliminated via the MARK rule.

The GRD-HANDLE$^{\Diamond}$ rule wraps the carrier in a contract-effect handler, where v_h becomes the handler function. As such, v_h also becomes contract-checking code.

3.6 On the Importance of Cascading Contracts

Flat contracts in this model generalize the ones from the literature to allow cascading. In particular, a flat contract can return any contract, not just a Boolean. Generalizing flat contracts in this manner is highly useful. Take affine contracts [Tov and Pucella 2010] as an example. An affine contract guarantees that a function is called at most once by keeping track of how many times the function has previously been called. It does so with mutable state.

```
let x ⟵[n]⟶ y =
  λ_.let r = do (Ref n) in
     ((unused/c r) ⊓ x) ⟶ y

let unused/c r =
  λ_.if is_zero (do (Get r)) then
       false
     else
       do (Update r (λn.n - 1)); true

let run_with_mut_refs thk = handle◊ thk () with ── elided ──
```

Fig. 5. Affine Contract via Cascading

Figure 5 shows the code for a contract that allows a function to be called at most n times. The `run_with_mut_refs` function grants contract code the ability to create, read from, and write to mutable references. Accordingly, the $x \overset{n}{\multimap} y$ contract specifies a function $x \longrightarrow y$ that can be called at most n times. This property is maintained by allocating a reference containing the remaining number of calls permitted. Each time the function is applied, the number contained inside this reference is decremented.

The $x \overset{n}{\multimap} y$ contract itself is flat; it is not a function contract. When applied to a function, $x \overset{n}{\multimap} y$ ignores its argument (the function itself) and allocates a cell initialized with n; *then* it returns a function contract. Due to the cascading behavior, this allocation happens exactly once for each carrier whose monitor enforces the "call at most n times" constraint. Without cascading, this kind of contract is not expressible [Felleisen 1991] in terms of existing contract forms.

4 DEPENDENT CONTRACTS

The contract forms considered thus far cannot deal with dependencies. For example, the result part of a function contract might have to depend on the actual argument.

This section extends the model with dependency: both traditional dependent function contracts, written as $e_1 \Longrightarrow e_2$, and new dependent main-effect contracts, written as $e_1 \blacktriangleright e_2$. Formally, the syntax is extended as follows:

$\boxed{\textsc{dependent (eval)}}$ extends EFFECTS (EVAL)

$$
\begin{aligned}
\kappa \in \mathsf{Con} &= \ldots \mid e \Longrightarrow e \mid e \blacktriangleright e \\
v \in \mathsf{Val} &= \ldots \mid v \Longrightarrow v \mid v \blacktriangleright v \\
E \in \mathsf{Ctx} &= \ldots \mid E \Longrightarrow e \mid v \Longrightarrow E \mid E \blacktriangleright e \mid v \blacktriangleright E \\
E^{\triangleright} \in \mathsf{Ctx}^{\triangleright} &= \text{── } \textit{the above mutatis mutandis} \text{ ──} \\
E^{\Diamond} \in \mathsf{Ctx}^{\Diamond} &= \text{── } \textit{the above mutatis mutandis} \text{ ──}
\end{aligned}
$$

4.1 Dependent Function Contracts

Recall the `run_with_pool` function from Section 2.2. This function takes two arguments: a list of numbers (`xs`) and a thunk (`thk`). The contract on `thk` is `pool_c (length xs)`, which *depends* on the first argument. As is, the model cannot express this dependency because GRD-FUN does not communicate the argument value to the result contract.

Dependent contracts have an extensive history in the literature [Blume and McAllester 2006; Findler and Blume 2006; Greenberg et al. 2010]. The "indy" semantics, due to Dimoulas et al. [2011], is now accepted as standard:

$$\textsc{mon-dep-fun} \quad E[\mathsf{mon}_j^{k,l}\,(v_1 \Longrightarrow v_2)\,v] \longmapsto E[\mathsf{err}_j^k] \text{ if } v \notin \mathsf{Fun}$$

$$\textsc{grd-dep-fun} \quad E[\mathsf{mon}_j^{k,l}\,(v_1 \Longrightarrow v_2)\,f] \longmapsto E[\lambda x.\mathtt{let}\,x_j = \mathsf{mon}_j^{l,j}\,v_1\,x\,\mathtt{in}\;\mathtt{let}\,x_k = \mathsf{mon}_j^{l,k}\,v_1\,x\,\mathtt{in}\;\mathsf{mon}_j^{k,l}\,(v_2\,x_j)\,(f\,x_k)]$$

Instead of being a result contract, as in a normal function contract, v_2 is a function that produces a result contract when given the argument. In the contractum, v_2 is applied not directly to the argument x. Doing so would be the "lax" semantics [Findler and Felleisen 2002]. For indy, v_2 is applied to x protected by the argument contract. This is because v_2 itself may violate the contract. To reflect this possiblity, the client blame label on x_j is j, the contract-defining party. Otherwise, this rule is the same as GRD-FUN.

Note. Moy and Felleisen [2023] observe that, under certain circumstances, dependent function contracts can duplicate effects. They present a solution to this problem that stages contract effects. Since the purpose of this section is to convey the essence of dependent contracts, the model here does not include the complexity of staged contract effects. However, the solution is orthogonal to this formalism and could be readily adopted. ▪

4.2 Dependent Main-Effect Contracts

Dependency can also arise in main-effect contracts. Consider a random-number-generating effect `Gen k` that yields a random integer between 0 and k inclusive. Guaranteeing that the random number is within bounds requires dependency:

```
data gen = Gen is_integer

let is_in_range req =
  match req with
  | Gen upper → λres.(is_int res) && (0 <= res) && (res <= upper)
  | _         → λ_.true

let rand_c = is_any ▶ is_in_range
```

In this example, `is_in_range` matches on the effect request itself to determine the greatest valid random number. This number, given the name `upper`, is used to construct a predicate that ensures the generated number is within bounds.

Formalizing dependent main-effect contracts requires a few adjustments to the original semantics. First, two additional rules are needed to reduce monitors containing dependent main-effect contracts. These are analogous to the ones for ordinary main-effect contracts:

$$\textsc{mon-handle}^{\blacktriangleright} \quad E[\mathsf{mon}_j^{k,l}\,(v_1 \blacktriangleright v_2)\,v] \longmapsto E[\mathsf{err}_j^k] \text{ if } v \notin \mathsf{Fun}$$

$$\textsc{grd-handle}^{\blacktriangleright} \quad E[\mathsf{mon}_j^{k,l}\,(v_1 \blacktriangleright v_2)\,f] \longmapsto E[\lambda x.\mathsf{mark}_j^{k,l}\,(v_1 \blacktriangleright v_2)\,(f\,x)]$$

Second, the $\downarrow$ metafunction must be extended to permit dependencies:

$\boxed{\downarrow : \mathsf{Val} \times \mathsf{Ctx} \to \mathsf{Ctx}}$

$$
\begin{aligned}
v \downarrow \square &= \square \\
v \downarrow \langle E, e \rangle &= v \downarrow E \\
v \downarrow \langle v_1, E \rangle &= v \downarrow E \\
&\cdots \\
v \downarrow (\mathsf{mark}_j^{k,l}\ (v_1 \rhd v_2)\ E) &= (v \downarrow E)[\mathsf{mon}_j^{l,k}\ v_2\ \square] \\
v \downarrow (\mathsf{mark}_j^{k,l}\ (v_1 \blacktriangleright v_2)\ E) &= (v \downarrow E)[\mathsf{mon}_j^{l,k}\ (v_2\ e)\ \square] \\
&\quad \text{where } e = \mathsf{mon}_j^{k,j}\ v_1\ ((\uparrow E)[v])
\end{aligned}
$$

With this revision, $\downarrow$ has access to the raw effect request v. When a mark contains a dependent contract, it must generate the wrapper needed for the effect response. To do so, it applies v_2 to e, where e is the protected effect request. In a lax semantics, $e = (\uparrow E)[v]$. For indy, e must also protect v with v_1 where the client label is the contract-defining party j.

Finally, the DO$^\rhd$ rule must be adjusted to use the newly adapted metafunction:

$$
\begin{aligned}
\textsc{do}^\rhd \qquad E[\mathsf{handle}^\rhd\ E^\rhd[\mathsf{do}\ v]\ \mathtt{with}\ v_h] &\longmapsto E[v_h\ e_v\ (\lambda x.\mathsf{handle}^\rhd\ E^\rhd[e_x]\ \mathtt{with}\ v_h)] \\
&\quad \text{if } E^\rhd \in \mathsf{unhandled} \\
&\quad \text{where } e_v = (\uparrow E^\rhd)[v], e_x = \ (v \downarrow E^\rhd)\ [x]
\end{aligned}
$$

Here, e_x uses the updated metafunction (highlighted) with the raw effect request v supplied.

5 SEMANTIC PROPERTIES

At this point, defining a partial evaluation function, also known as an *evaluator*, is straightforward:

$\boxed{\textsc{dependent (proof)}}$ extends DEPENDENT (EVAL)

$p \in \mathsf{Prog} = \{e \mid e \text{ is closed}\}$

$a \in \mathsf{Ans} = b \mid \mathsf{opaque} \mid \mathsf{err}_j^k \mid \mathsf{err}_\bullet^\circ$

$\boxed{\mathsf{eval} : \mathsf{Prog} \to \mathsf{Ans}}$

$$
\mathsf{eval}(e) = \begin{cases} b & \text{if } e \longmapsto^* b \\ \mathsf{opaque} & \text{if } e \longmapsto^* v, v \notin \mathsf{Bool} \\ \mathsf{err}_j^k & \text{if } e \longmapsto^* E[\mathsf{err}_j^k] \\ \mathsf{err}_\bullet^\circ & \text{if } e \longmapsto^* s \end{cases}
$$

Programs, i.e. closed expressions, are the input to the evaluator. Answers are the output of the evaluator. If a program reduces to a Boolean, the answer is the same Boolean. All other values yield the opaque token.[4] This behavior matches that of most REPLs where function values are printed as an opaque symbol. Two kinds of error can occur during execution: contract errors, which produce err_j^k, and language errors,[5] which produce $\mathsf{err}_\bullet^\circ$.

[4]For simplicity, the function turns pairs into opaque, too.

[5]In essence, such errors are violations of the runtime system's contracts.

5.1 Well-Definedness

Following convention, the first theorem states two properties that ensure the sanity of the reduction relation. Specifically, eval is a partial function because the reduction relation relates each program to at most one answer. Programs where eval is undefined are exactly those with unbounded reduction sequences.

Theorem 5.1 (Functional Evaluation). Two facts about the evaluator hold:

(1) The eval relation is a partial function.
(2) If e is a program, then either (i) $\mathsf{eval}(e)$ is defined or (ii) the reduction sequence starting with e is unbounded.

Proof. See Appendix B. □

5.2 Erasure

The key property of interest for the model is *contract erasure*. Contracts serve one purpose, namely, to detect violations of specifications. Therefore, the output of a correct program should not depend on the presence or absence of contracts. In short, contracts must not interfere with program execution—other than possibly signaling an error. Non-interference in the presence of effects is critical for modular reasoning [Oliveira et al. 2012].

Stating the erasure theorem requires defining an erasure function $\mathscr{E}$ for contract monitors:

$$\boxed{\mathscr{E} : \mathsf{Expr} \to \mathsf{Expr}}$$

$$\mathscr{E}(b) = b$$
$$\mathscr{E}(x) = x$$
$$\mathscr{E}(\lambda x.e) = \lambda x.\mathscr{E}(e)$$
$$\cdots$$
$$\mathscr{E}(\mathsf{mon}^{k,l}_j\ e_\kappa\ e) = \mathscr{E}(e)$$
$$\mathscr{E}(\mathtt{handle}^{\Diamond}\ e\ \mathtt{with}\ e_h) = \mathtt{handle}^{\Diamond}\ \mathscr{E}(e)\ \mathtt{with}\ \mathscr{E}(e_h)$$

$$\boxed{\mathscr{E}^+ : \mathsf{Expr} \to \mathsf{Expr}}$$

$$\mathscr{E}^+(b) = b$$
$$\mathscr{E}^+(x) = x$$
$$\mathscr{E}^+(\lambda x.e) = \lambda x.\mathscr{E}^+(e)$$
$$\cdots$$
$$\mathscr{E}^+(\mathsf{mon}^{k,l}_j\ e_\kappa\ e) = \mathscr{E}^+(e)$$
$$\mathscr{E}^+(\mathtt{handle}^{\Diamond}\ e\ \mathtt{with}\ e_h) = \mathscr{E}^+(e)$$

Theorem 5.2 (Erasure). If $\mathsf{eval}(e) = b$ then $\mathsf{eval}(\mathscr{E}(e)) = b$.

Proof. The proof of erasure proceeds by a simulation argument with the following simulation:

$$\lambda x.f\ x \sim \widetilde{f}$$
$$\mathtt{handle}^{\Diamond}\ e\ \mathtt{with}\ e_h \sim \mathtt{handle}^{\Diamond}\ \widetilde{e}\ \mathtt{with}\ \widetilde{e_h}$$
$$\mathtt{handle}^{\Diamond}\ e\ \mathtt{with}\ e_h \sim \widetilde{e}$$
$$\mathsf{mon}^{k,l}_j\ e_\kappa\ e \sim \widetilde{e}$$
$$\mathsf{mark}^{k,l}_j\ v\ e \sim \widetilde{e}$$
$$\cdots$$

By convention, a metavariable with a tilde such as $\widetilde{e}$ is in simulation with its plain counterpart e. See Appendix C for details. □

Technically, non-termination is the one contract effect that can affect a program's behavior. So long as contracts contain code in a Turing-complete language, this effect is unavoidable. As stated, Theorem 5.2 holds because the antecedent rules out non-terminating contracts.

While the syntax design already clarifies that there are two separate, disjoint levels of effect handling, the proof for Theorem 5.2 confirms this claim: main code cannot be serviced by contract-effect handlers. A small adjustment to the erasure function, defined above as $\mathscr{E}^+$, makes it possible to state the claim formally.

Corollary 5.3 (No Effect Interference). If $\mathsf{eval}(e) = b$ then $\mathsf{eval}(\mathscr{E}^+(e)) = b$.

Proof. Follows directly from the proof of Theorem 5.2. □

Establishing the erasure theorem is straightforward in a pure setting, yet difficult to achieve in a language with effects. Ensuring erasure means contract code must not interfere with the main program directly or indirectly via effects. A language with effect-handler contracts poses the additional problem of having to grant contract code the right to interact with effects, while also imposing constrains on such interactions.

A physicist may describe the model as being in an "unstable equilibrium;" a programming-languages researcher may use the word "brittle" and compare the design to Hindley-Milner type inference. Directly put, designing a language semantics that satisfies contract erasure demands balancing expressive power with preventing interference. The model presented here achieves this delicate balance, as the theorem and Section 7 show. Limiting the expressive power any further makes programming inconvenient and would neglect existing use cases. However, experiments adding more power to the model show that many extensions violate erasure.

For example, consider a naive design where the reduction relation for handlers merges the two levels of effect handling:

$$E[\texttt{handle}\ E_k[\texttt{do}\ v]\ \texttt{with}\ v_h] \longmapsto E[v_h\ v\ (\lambda x.\texttt{handle}\ E_k[x]\ \texttt{with}\ v_h)]\ \text{if}\ E_k \in \texttt{unhandled}$$

Instead of restricting the evaluation context in the body of the handler, this rule uses the unrestricted context E_k. Such a rule violates contract erasure as the following program demonstrates:

$$\texttt{handle}\ (\texttt{mon}_j^{k,l}\ (\texttt{do}\ \texttt{false})\ \texttt{true})\ \texttt{with}\ \lambda x.\lambda y_k.x$$

The original program evaluates to `false`, but erasing the contract yields a variant whose value is `true`. Similarly, modifying DO$^\triangleright$ to use E, or modifying DO-FUN$^\Diamond$ to give direct access to the continuation, both result in erasure violations as they produce a rule equivalent to the one above.

Introducing "main-handler contracts" like so

$$E[\texttt{mon}_j^{k,l}\ (\triangleright v_h)\ f] \longmapsto E[\lambda x.\texttt{handle}^\triangleright\ (f\ x)\ \texttt{with}\ v_h]$$

also violates erasure. Here is a counterexample:

$$\texttt{handle}^\triangleright\ ((\texttt{mon}_j^{k,l}\ (\triangleright(\lambda y.\lambda z_k.\texttt{false}))\ (\lambda x.\texttt{do}\ x))\ \texttt{true})\ \texttt{with}\ \lambda y.\lambda z_k.y$$

Again, the original program evaluates to `false`, while its erased variant yields `true`. In short, GRD-HANDLE$^\Diamond$ cannot be generalized.

5.3 Blame Correctness

The final property to consider is blame correctness, that is, whether a failing monitor assigns blame to the component that serves a faulty value. In the context of the model, the DO$^\triangleright$ reduction deserves particular attention. Like the rule for monitoring first-class functions, the reduction for main-effect handling switches the order of blame labels as it pushes the relevant contracts down the handler's continuation ($v \downarrow E^\triangleright$). The question is—as it was for the original work on higher-order (dependent) function contracts [Findler and Felleisen 2002]—whether this switch is correct. As Dimoulas et al. [2011] show, the answer is a blame correctness theorem.

By now, the strategy for proving blame correctness is reasonably standard. The first step is to introduce ownership labels on expressions, values, and evaluation contexts. Intuitively, an expression $|e|^l$ denotes that the owner of e is the component labeled l.

The second step is to adjust the reduction relation so that ownership changes when a value crosses from one component to another. Crossing may either add or drop a label from a value. The reduction drops a label when the crossing involves a contract check, meaning the value is vetted and "absorbed" by a new host component. A blame label is added when the crossing does not involve a check, meaning the value becomes co-owned by several distinct components. It is critical that the ownership labels do not affect the semantics proper.

The third and final step is to show that when a monitor is about to check a value, the latest ownership label of the value is the same one that the monitor uses to assign blame.

Theorem 5.4 (Blame Correctness). For all e, if $l_o; \emptyset \vdash e$ and $e \longmapsto_o^* E[\mathtt{mon}_j^{k,l}\, v_\kappa\, v]$, then $v = |v'|^k$.

Proof. The proof uses the standard subject-reduction technique [Curry and Feys 1958; Wright and Felleisen 1994] and a consistency judgment for the ownership annotations. The judgment $l; \Gamma \vdash e$ says that e is well-formed if its owner is l, given an environment Γ that maps variables to their owners. Importantly, if a program is well-formed under the default owner l_o, then for any monitors it contains, the owner of the carrier matches the server label of the monitor. Subject reduction shows that this consistency is preserved across reduction sequences, and hence, if a monitor check fails, blame is assigned to the correct component. See Appendix D for the full proof. □

The labeled reduction semantics is indeed equivalent to the unlabeled one after erasing ownership labels ($\mathcal{O}(\cdot)$) from the first one.

Proposition 5.5 (Ownership Erasure). For all labeled e, $e \longmapsto_o^* e'$ if and only if $\mathcal{O}(e) \longmapsto^* \mathcal{O}(e')$.

Note. The stronger *complete monitoring* property states that all channels of communication between components can be monitored using contracts [Dimoulas et al. 2012]. The presented model does not satisfy complete monitoring. As Section 7 explains, the intent of effect-handler contracts is to be a low-level mechanism for implementing other constructs. Complete monitoring is more relevant to prove for these higher-level contract systems, not the low-level target. ▪

6 EFFECT RACKET

Rapidly moving from a model to a full-fledged programming language calls for (1) a programmable production-level language with (2) linguistic constructs for realizing effect handlers easily and (3) a well-developed higher-order contract system. Racket is such a language [Felleisen et al. 2018; Findler and Felleisen 2002; Flatt and PLT 2010; Flatt et al. 2007]. This section presents `effect/racket`, a language with effect handlers and a full contract system (Section 6.1). Following the precedent of `typed/racket`, the language is implemented as a library [Tobin-Hochstadt et al. 2011] (Section 6.2). The language implementation validates that the model can be realized. Therefore, it may help guide implementers of other effect-handler languages.

6.1 The Language, By Example

This section is organized like Section 2.2, but uses different examples to keep things interesting.

Main-Effect Handlers. As an introductory example, consider implementing ML's first-class mutable references using effect handlers. References come with a `ref` constructor and two elimination forms: `ref-get` and `ref-set`. In `effect/racket`, each form demands the declaration of a corresponding effect: one for allocating a reference cell, one for getting its value, and yet another for

```
#lang effect/racket

(effect ref (v))
(effect ref-get (r))
(effect ref-set (r v))

;; Store → Service
(define (ref-service store)
  (handler
    ;; Creates a reference
    [(ref init)
     (define-values (r new-store) (store-allocate store init))
     (with ((ref-service new-store))
       (continue* r))]

    ;; Returns a reference's value
    [(ref-get r)
     (continue (store-get store r))]

    ;; Sets a reference's value
    [(ref-set r v)
     (with ((ref-service (store-set store r v)))
       (continue* (void)))]))
```

(a) Program

```
> (with ((ref-service empty-store))
    (define r (ref 0))
    (ref-set r (add1 (ref-get r)))
    (ref-get r))
1
```

(b) REPL

Fig. 6. Mutable References with `effect/racket`

assigning to a cell. Declaring an effect makes the effect name available both for requesting the effect and, within a handler, interpreting the effect.

Figure 6a displays the code for both the effect declarations and the effect handler. The handler, dubbed a *service* for references, comes with three clauses, one per declared effect; all other effects are propagated automatically. Furthermore, the `handler` form binds two identifiers to delimited continuations: `continue`, for resuming in a deep manner; and `continue*`, for resuming in a shallow manner. Otherwise, the handler uses standard techniques for implementing a store in this setting [Cartwright and Felleisen 1994; Pretnar 2015].

Any language in the Racket ecosystem, including `effect/racket`, is easily equipped with a read-eval-print loop (REPL). By running the `effect/racket` program, the definitions of effects and services become available for interactive experimentation. Figure 6b shows how to install the handler function using the `with` form. In the context of this `with` expression, it is now possible to allocate a numeric reference cell, to increase its value by `1`, and then to retrieve this value.

Main-Effect Contracts. Suppose a programmer wishes to write a library function that guarantees a frame condition. To make this concrete, the function guarantees that it manipulates only a specific, given reference cell during the dynamic extent of any call. A good name for this contract would be `mutates-only/c`, and here is how the library's interface would state that guarantee:

```
(provide
  (contract-out
   [ref-restore
    ;; Runs (f r), restores the content of r, and
    ;; returns the value of r that f stores there.
    (->i ([r reference?]
          [f (r) (and/c (mutates-only/c r)
                        (-> reference? any/c))])
         [result any/c])]))
```

The function contract is a standard indy dependent contract [Dimoulas et al. 2011] that governs two arguments—`r` and `f`—and promises nothing about its result. The new part is the contract for `f`, which says that (1) `f` is a function from a reference cell to any value and (2) it may mutate only `r`.

The frame contract is a rather straightforward instance of a main-effect contract:

```
(define (mutates-only/c r-ok)
  (define (effect-ok? e)
    (match e
      [(ref-set r _) (equal? r r-ok)]
      [_ true]))
  (->e effect-ok? any/c))
```

The `mutates-only/c` function takes a reference cell as an argument and returns a main-effect contract that permits only writing to the given cell and no other one. The two-part `->e` contract (i.e., $\cdot \triangleright \cdot$) tells a reader that requested effects must satisfy the `effect-ok?` predicate and that values returned by the handler can be anything. According to `effect-ok?`, any write effect must be to a reference cell that is equal to `r-ok`. All other effects are permitted.

Contract-Effect Handlers. Equipped with reference cells, it is now possible to transliterate the affine-function contract from Section 3.6 into running code. Figure 7 shows the implementation of —∘ as a contract exported from a library.

Since the contract relies on reference cells at the contract level, it is mandatory to lift the service from the main level to the contract level; see Figure 7 (lines 26–34). The `contract-handler` form does not make the delimited continuation available; instead, each arm must return a pair of values: the value to be supplied to the delimited continuation, and a new handler to be installed around the continuation.

Using `ref-contract-service`, both —∘ and `unused?` can be defined using Racket's existing contract library, with reference effects performed as needed; see Figure 7 (lines 12–24). As in Section 3.6, —∘ must use cascading to allocate a reference for affine functions at the right time; the presented code realizes this constraint using the `self/c` combinator, which when protecting a carrier `v`, applies a function to `v` and uses the result to protect `v`—just like flat contracts in the model. Here, the function given to `self/c` returns the expected function contract. For —∘, the value `v` is not needed and is discarded.

```
#lang effect/racket

(provide
 ;; Store → Service
 ;; Service to be installed for uses of ⟜.
 ref-contract-service

 ;; Natural Contract Contract → Contract
 ;; Returns a contract for a function that is called at most n times.
 ⟜)

(define (⟜ n dom cod)
  (self/c
   (λ _
     (define r (ref n))
     (->i ([x dom])
          #:pre () (unused? r)
          [result cod]))))

(define (unused? r)
  (define m (ref-get r))
  (cond
    [(zero? m) false]
    [else (ref-set r (sub1 m)) true]))

(define (ref-contract-service store)
  (contract-handler
    [(ref init)
     (define-values (r new-store) (store-allocate store init))
     (values r (ref-contract-service new-store))]
    [(ref-get r)
     (values (store-get store r) (ref-contract-service store))]
    [(ref-set r v)
     (values (void) (ref-contract-service (store-set store r v)))]))
```

Fig. 7. Affine-Function Contracts with `effect/racket`

Contract-Handler Contracts. A function is *reentrant* if it can call itself recursively, directly or indirectly. A contract-handler contract can check for non-reentrancy by prohibiting recursive calls during a function's dynamic extent. Implementing such a constraint requires both a `contract-handler` to mark the dynamic extent of a function call and a contract-handler contract:

```
(effect non-reentrant? ())

(define non-reentrant-service
  (contract-handler
   [(non-reentrant?)
    (values false non-reentrant-service)]))
```

The contract for a non-reentrant function `f` installs this handler, as such:

```
1  (provide
2    (contract-out
3     [f (and/c
4          (with/c non-reentrant-service)
5          (->i ([x any/c])
6               #:pre () (non-reentrant? #:fail true)
7               [result any/c]))]))
```

When a client applies `f`, the precondition requests a `non-reentrant?` effect (line 6). If this returns `false`, the function may already be running; otherwise, the `#:fail` option, which provides a default value if no matching handler is installed, returns `true`. Once the precondition check passes, the second wrapper sets up a contract-handler contract (line 4) using `with/c` (i.e., $\Diamond\cdot$). Thus, if `f` were to call itself, the contract prohibits it because the installed `non-reentrant-service` supplies `false`.

6.2 The Implementation, An Overview

The implementation of `effect/racket` consists of about 1,100 lines of code. Most of these lines compose elements from existing libraries. For example, effect handlers themselves are implemented as thin wrappers around Racket's existing library of delimited control operators [Flatt et al. 2007]. Other pieces of the implementation ensure that Racket's effectful primitive operations are inaccessible to programs in `effect/racket`. After all, a main-effect contract would be meaningless if certain primitive effects cannot be reinterpreted.

One critical aspect of the implementation concerns the key assumption of the model in Section 3, which demands that handlers can detect whether an effect request originates from within main code or contract code. Formally, this idea is encoded via special evaluation contexts; see Section 3.1. As it turns out, Racket's contract system already provides a mechanism for determining whether code is executing inside a contract [Andersen et al. 2018]. Specifically, contracts set up continuation marks [Clements et al. 2001] that delineate contract-specific code from user code. Thus, the effect handler forms inspect the delimited continuation and look for this mark to determine whether the effect should be handled. As a result, `effect/racket` does not require any modifications to Racket's contract system.

As a language in Racket's ecosystem, `effect/racket` inherits the module system too, which raises the interoperability issue. Indeed, the preceding examples already rely on the module system, showing that `effect/racket` modules can export functions with effectful software contracts. In addition to full interoperability with other `effect/racket` modules, the language has a shallow form of interoperability with plain Racket modules. Following the terminology of Matthews and Findler [2007], the interoperability uses a first-order natural, higher-order lump-embedding; first-order values can freely flow from an `effect/racket` module to a foreign module and back; in contrast, higher-order values are wrapped in an opaque structure so they become unusable.

In summary, the implementation effort reveals that the addition of effectful software contracts to an effect-handler language is rather straightforward, with the exception of effect stratification. Assuming the erasure property is desirable, an implementer must add a mechanism that demarcates the dynamic extent of contract-checking code.

6.3 Restricting Handlers

As presented, handlers have unlimited access to interpose and reinterpret all effects. This means library authors have no guarantee about how their effects are interpreted. Others have recognized

this lack of abstraction safety and have proposed solutions, especially in typed settings [Biernacki et al. 2017; Brachthäuser et al. 2022; Leijen 2013; Xie et al. 2020; Zhang and Myers 2019].

An alternative design can rectify this problem easily. Racket gives programmers the ability to attach metadata to continuations via continuation marks [Clements and Felleisen 2004; Flatt and Dybvig 2020]. To prevent other parties from arbitrarily tampering with this information, the language only permits access to continuation marks via *keys*. Racket also uses this mechanism to limit how much of the continuation a program can capture and abort [Felleisen 1988; Flatt et al. 2007; Sitaram and Felleisen 1990]. These keys are first-class unforgeable values. If a module does not export its key, then no other party can view or update the information associated with that key. To prevent interception, a module can just keep a key internal. Effectively, a key is a capability [Dennis and Van Horn 1966].

Instead of an arbitrary match pattern, a handler could be restricted to explicitly provide a set of "effect keys" that it can interpret. Similarly, main-effect contracts would have to include a set of effect keys instead of an arbitrary predicate over all effect requests.

There is a downside to this approach; it eliminates a useful class of contracts such as those for purity. A contract that guarantees purity must, by definition, be able to interpose on *all* effects. This includes effects that are kept hidden. Unsurprisingly, there is a trade-off between security and expressiveness.

If desired, though, this kind of restriction can be built on top of `effect/racket`; the language is flexible and can serve as a foundation upon which other abstractions can be constructed. Thus, language implementors can choose the design that fits their situation.

7 EVALUATION AND RELATED WORK

The introduction of this paper claims that effect-handler contracts are a universal mechanism. An evaluation of such a claim must show that the model and its full-scale implementation cover all existing work.[6] Additionally, such related research must be analyzed and systematically compared. As such, this section consists of two pieces: (1) an evaluation of effect-handler contracts with respect to a survey of existing literature; and (2) a summary of each piece of related research and how it compares to this paper.

7.1 Analysis

Table 1 presents an overview of the existing literature. It explicates the many overlapping problems that various papers address. Rows correspond to existing pieces of literature, and columns correspond to properties that at least one system can express.

A concise description of these properties follows:

ALLOW CALL A function may be called only during the dynamic extent of another function.
EXCEPTIONS Only specified exceptions may be raised during a function call. This property is the dynamic analogue to Java's checked exceptions.
FRAMING Mutations are restricted to specified memory locations.
GHOST STATE Values are associated with a mutable reference that is used to check conformance with a protocol.
MUST CALL A function *must* be called during the dynamic extent of a call to another function.
NON-REENTRANT A function must not call itself recursively.
PURE No effects—other than non-termination and error signals—are permitted.

[6] Effect-handler contracts subsume only the low-level contract aspects of existing work—nothing more. All of the papers surveyed here build sophisticated systems on top of the low-level constructs. These contributions are orthogonal to, and not subsumed by, effect-handler contracts.

Table 1. Detailed Comparison Matrix

	ALLOW CALL	EXCEPTIONS	FRAMING	GHOST STATE	MUST CALL	NON-REENTRANT	PURE	RESTRICTED EFFECT	TERMINATION	UNION CONTRACTS
Chalin et al. [2006]		•	•	•			•		•	
Tov and Pucella [2010]				•						
Shinnar [2011]		•	•	•			•			
Disney et al. [2011]	•		•	•	•	•				
Keil and Thiemann [2015a]										•
Scholliers et al. [2015]	•			•	•					
Moore et al. [2016]	•			•	•	•				
Dimoulas et al. [2016]	•			•	•					
Bañados Schwerter [2016]		•					•	•		
Williams et al. [2018]										•
Nguyễn et al. [2019]									•	
Moy and Felleisen [2023]	•		•	•	•	•				
Effect-Handler Contracts	•	•	•	•	•	•	•	•	•	•

RESTRICTED EFFECT Effects are restricted at a fine-grained level.

TERMINATION A function call must terminate. Specifically, a call graph keeps track of changes to the size of arguments.

UNION CONTRACTS Given a set of contracts, the protected value satisfies at least one of the given contracts. Checking the union of flat contracts is easy, but checking the union of higher-order contracts relies on state to keep track of violations and assign blame.

Most papers illustrate these properties with a plethora of examples, all of which can be implemented in `effect/racket`.

The cells of the table have the following rough meaning. A • indicates that the presented contract framework supports this property. Note that the number of • entries in a row does not indicate anything about the "power" of the presented system. It merely means that a paper with fewer • entries may focus on a narrower set of properties. Also, these papers differ in other, significant ways that are not communicated by the • markings.

The `effect/racket` language can faithfully express all but one existing contract. This exception is the computational contract [Scholliers et al. 2015] that prohibits a function from being called during the dynamic extent of a call to another function. Effect-handler contracts can achieve this behavior as long as the excluded function comes with a contract that enables the system to monitor it; if not, this contract is impossible to realize without invasive monitoring techniques. For details, see the next subsection.

7.2 Related Work

The Java Modeling Language (JML) [Chalin et al. 2006] is a specification language for stating and verifying properties of objects in Java. It encompasses a broad range of features including assertions, class invariant statements, frame conditions, purity constraints, termination constraints, and

ghost state declarations—just to name a few. Property checking takes place in one of two modes: static deductive verification (DV) or dynamic runtime-assertion checking (RAC). Some properties, such as termination, can be checked only using DV. JML differs from higher-order contract systems in three major ways. First, properties are described using a restrictive set of "well-defined" terms, a limitation compared to contracts written with ordinary constructs. Second, JML supports only first-order properties. Finally, JML lacks a blame assignment component, meaning developers are on their own when a contract check fails.

Tov and Pucella [2010]'s research on interoperability between a language with a substructural type system and one with a plain structural type system relies on an affinity check for function arguments. Specifically, the boundary employs a run-time check to ensure that a function argument is *affine*, meaning it can be applied at most once. This check uses a mutable Boolean field associated with each function value, i.e. ghost state, which indicates whether a function has been applied. Dimoulas et al. [2016] also use ghost state. They define a general-purpose DSL that uses ghost state to check protocol conformance. As described in Section 3.6, contract-effect handlers can easily introduce and manipulate ghost state.

For interoperability, a language with a sound gradual type-and-effect system relies on a run-time enforcement mechanism to restrict the effects performed by untyped code. The contracts for such a language are formulated in terms of two operations [Bañados Schwerter et al. 2014]: `has` (for checking the privileges granted by the current context) and `restrict` (for restricting the privileges of an expression). In the effect-handler language, these primitives are just main-effect contracts.

Shinnar [2011] takes some of the constructs from JML, in particular framing contracts, and adapts them to Haskell. The implementation uses delimited checkpointing to keep track of state. A delimited checkpoint is a snapshot of memory captured using software transactional memory (STM). Framing contracts can detect and restrict writes to transactional references by comparing memory snapshots. Shinnar proves erasure for a limited model of Haskell with delimited checkpoints. This work is similar to those pieces of research [Findler and Felleisen 2001; Strickland et al. 2012] that consider erasure for only a few restricted effects.

Disney et al. [2011]'s higher-order temporal (HOT) contracts and Moy and Felleisen [2023]'s trace contracts check properties of sequences of argument and return values for functions and methods. While the two differ in many respects, from the perspective of effectful software contracts they fall into the same class of extended higher-order contracts. Describing constraints over sequences amounts to a writing a predicate that "folds over" the sequence incrementally, storing intermediate state in a mutable reference. As such, contract-effect handlers can supply the needed mutable references to such contracts. Indeed, Disney et al. [2011] present some examples that are *more directly* expressed using effect-handler contracts than HOT contracts. For example, their HOT contract for non-reentrancy does not suffice in the presence of control effects, whereas an effect-handler-contract implementation of the same property is robust.

Scholliers et al. [2015]'s computational contracts instantiate aspect-oriented programming for the contract world. Critically, such contracts can prohibit or enforce that a function f is called in a particular dynamic extent. Due to the intrusiveness of aspect-oriented programming, computational contracts do not require that f is aware of the contract. Indeed, without aspect-oriented programming or a similarly invasive mechanism, there is no way to interpose on function applications in a dynamic extent, which is why the effect-handler language cannot fully realize this form of checking.

Moore et al. [2016]'s authorization contracts enforce access control with contracts about granted privileges. Specifically, authorization contracts can capture, check, and restore access privileges via an authority environment that records such privileges. Moore et al. [2016]'s model is essentially a variant of contract-handler contracts topped off with a DSL for authorization management.

Effectful contracts alone do not implement any of the security aspects of the system. However, authorization contracts could be built on top of contract-handler contracts given the secure design described in Section 6.3.

Nguyễn et al. [2019] provide a run-time check for termination by monitoring the size-change property (SCP) of functions dynamically. Any diverging function must exhibit an SCP violation, causing a contract violation. They turn this run-time check into a static one, using existing contract verification techniques [Nguyễn et al. 2018]. To guarantee termination, they use continuation marks to store size-change information on the stack. Contract-handler contracts can be used to store the same information.

While the literature on higher-order contracts tends to mention intersection and union contracts, implementing these in general is a serious challenge. Indeed, Racket rejects `or/c` contracts if the disjuncts are not "first-order distinguishable." Several researchers [Freund et al. 2021; Keil and Thiemann 2015a; Williams et al. 2018] have studied this problem, and all come to the conclusion that effects are needed. For example, Williams et al. [2018] use a mutable blame state to keep track of contract violations. A contract-effect handler can be used to implement this blame state. Moreover, erasure guarantees that such an implementation does not have adverse effects on a program's result. This property is critically important because even benign-looking contract effects can have unintended consequences. Such a phenomenon has been observed in practice [Lazarek et al. 2020, Section 6.1].

8 IGNORED NO LONGER

In the real world, developers use contracts with effects; in papers, researchers study how to employ effects in contracts. What has been lacking is a general framework for combining contracts and effects. As a result, existing extensions solve specific problems and do not generalize.

This paper offers the first general model of effectful software contracts. As such, it synthesizes a model of effect handlers with a model of contracts. In this combination, contracts can check effects, contracts can request effects, and contracts can handle contract-requested effects. Yet, since contracts should not affect the main program—other than signaling violations—the model is designed to avoid interference between contract-level effects and main-level code. Hence, in addition to well-definedness and blame correctness, the model satisfies an erasure theorem.

Beyond theoretical explorations, the formalism also provides guidance for implementation efforts. A fully faithful implementation, `effect/racket`, exists as a standalone language within the Racket ecosystem. This language demonstrates that the design can be realized. It is an open question how to modify an existing contract system to support all of the model's expressive power in a backwards compatible manner. Still, the theory can serve as a roadmap for others who wish to combine effects and contracts in a principled way. And, as effect handlers go mainstream [Chandrasekaran et al. 2018], the theory may find many more practical uses.

ACKNOWLEDGMENTS

This work was supported by NSF grant SHF 2116372. The authors would like to thank: Robby Findler, for help with Racket's contract system; participants of the NII Shonan Meeting 203 on effect handlers, for their insightful questions and discussions; and the anonymous POPL reviewers, for their comments and suggestions.

DATA AVAILABILITY STATEMENT

An up-to-date version of the software and appendices associated with this paper can be found at: https://doi.org/10.5281/zenodo.10129039. An archived version is also available [Moy et al. 2023].

REFERENCES

Danel Ahman and Andrej Bauer. 2020. Runners in Action. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-030-44914-8_2

Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. 2018. Feature-Specific Profiling. *Transactions on Programming Languages and Systems (TOPLAS)*. https://doi.org/10.1145/3275519

Felipe Bañados Schwerter. 2016. Side Effects Take the Blame. In *Software Language Engineering (SLE)*. https://doi.org/10.1145/2997364.2997381

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2692915.2628149

Hendrik Pieter Barendregt. 1981. *The Lambda Calculus*. North-Holland Publishing Co.

Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. 2010. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Technical Report. National Institute of Standards and Technology. https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3158096

Matthias Blume and David McAllester. 2006. Sound and Complete Models of Contracts. *Journal of Functional Programming (JFP)*. https://doi.org/10.1017/S0956796806005971

Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, Capabilities, and Boxes: From Scope-Based Reasoning to Type-Based Reasoning and Back. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/3527320

Robert Cartwright and Matthias Felleisen. 1994. Extensible Denotational Language Specifications. In *Theoretical Aspects of Computer Software (TACS)*. https://doi.org/10.1007/3-540-57887-0_99

Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2006. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects*. https://doi.org/10.1007/11804192_16

Sivaramakrishnan Krishnamoorthy Chandrasekaran, Daan Leijen, Matija Pretnar, and Tom Schrijvers. 2018. Algebraic Effect Handlers go Mainstream (Dagstuhl Seminar 18172). *Dagstuhl Reports*. https://doi.org/10.4230/DagRep.8.4.104

John Clements and Matthias Felleisen. 2004. A Tail-Recursive Machine with Stack Inspection. In *Transactions on Programming Languages and Systems (TOPLAS)*. https://doi.org/10.1145/1034774.1034778

John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an Algebraic Stepper. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/3-540-45309-1_21

H.B. Curry and R. Feys. 1958. *Combinatory Logic, Volume I*. North-Holland, Amsterdam.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM (CACM)*. https://doi.org/10.1145/1327452.1327492

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM (CACM)*. https://doi.org/10.1145/365230.365252

Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-Order World. *Transactions on Programming Languages and Systems (TOPLAS)*. https://doi.org/10.1145/2039346.2039348

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1926385.1926410

Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2951913.2951930

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-28869-2_11

Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal Higher-Order Contracts. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2034773.2034800

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/73560.73576

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming*. https://doi.org/10.1016/0167-6423(91)90036-W

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Communications of the ACM (CACM)*. https://doi.org/10.1145/3127323

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. In *Theoretical Computer Science*. https://doi.org/10.1016/0304-3975(92)90014-7

Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *Functional and Logic Programming (FLP)*. https://doi.org/10.1007/11737414_16

Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for Object-Oriented Languages. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/504311.504283

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/581478.581484

Matthew Flatt and R. Kent Dybvig. 2020. Compiler and Runtime Support for Continuation Marks. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385981

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1291151.1291178

Teodoro Freund, Yann Hamdaoui, and Arnaud Spiwack. 2021. Union and Intersection Contracts Are Hard, Actually. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/3486602.3486767

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1706299.1706341

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Asian Symposium on Programming Languages and Systems (APLAS)*. https://doi.org/10.1007/978-3-030-02768-1_22

Matthias Keil and Peter Thiemann. 2015a. Blame Assignment for Higher-Order Contracts with Intersection and Union. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2784731.2784737

Matthias Keil and Peter Thiemann. 2015b. TreatJS: Higher-Order Contracts for JavaScripts. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.28

Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. 2020. Does Blame Shifting Work?. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3371133

Daan Leijen. 2013. *Koka: Programming with Row-Polymorphic Effect Types.* Technical Report MSR-TR-2013-79. Microsoft Research. https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types/.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1498926.1498930

Bertrand Meyer. 1988. *Object-Oriented Software Construction.* Prentice Hall.

Bertrand Meyer. 1992. Applying "Design by Contract". *Computer*. https://doi.org/10.1109/2.161279

Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible Access Control with Authorization Contracts. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2983990.2984021

Cameron Moy, Christos Dimoulas, and Matthias Felleisen. 2023. Artifact: Effectful Software Contracts. https://doi.org/10.5281/zenodo.10151333

Cameron Moy and Matthias Felleisen. 2023. Trace Contracts. *Journal of Functional Programming (JFP)*. https://doi.org/10.1017/S0956796823000096

Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft Contract Verification for Higher-Order Stateful Programs. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3158139

Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-Change Termination as a Contract. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3325984

Bruno C. D. S. Oliveira, Tom Schrijvers, and William R. Cook. 2012. MRI: Modular Reasoning About Interference in Incremental Programming. *Journal of Functional Programming (JFP)*. https://doi.org/10.1017/S0956796812000354

Gordon Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*. https://doi.org/10.1016/0304-3975(75)90017-1

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-00590-9_7

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. In *Mathematical Foundations of Programming Semantics (MFPS)*. https://doi.org/10.1016/j.entcs.2015.12.003

R. L. Rivest, A. Shamir, and A. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In *Communications of the ACM (CACM)*. https://doi.org/10.1145/359340.359342

Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2015. Computational Contracts. *Science of Computer Programming*. https://doi.org/10.1016/j.scico.2013.09.005

Avraham Ever Shinnar. 2011. *Safe and Effective Contracts.* Ph. D. Dissertation. Harvard University.

Dorai Sitaram and Matthias Felleisen. 1990. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation.* https://doi.org/10.1007/BF01806126

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* https://doi.org/10.1145/2384616.2384685

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/1993316.1993514

Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *European Symposium on Programming (ESOP).* https://doi.org/10.1007/978-3-642-11957-6_29

Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. The Root Cause of Blame: Contracts for Intersection and Union Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* https://doi.org/10.1145/3276504

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation.* https://doi.org/10.1006/inco.1994.1093

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. In *International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/3408981

Dana N. Xu. 2012. Hybrid Contract Checking via Symbolic Simplification. In *Partial Evaluation and Program Manipulation (PEPM).* https://doi.org/10.1145/2103746.2103767

Dana N. Xu. 2014. Dynamic Contract Checking for OCaml. http://gallium.inria.fr/~naxu/research/camlcontract.pdf.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell. In *Principles of Programming Languages (POPL).* https://doi.org/10.1145/1480881.1480889

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. In *Principles of Programming Languages (POPL).* https://doi.org/10.1145/3290318

Received 2023-07-11; accepted 2023-11-07