Option Contracts

Christos Dimoulas
Harvard University
chrdimo@seas.harvard.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Many languages support behavioral software contracts so that programmers can describe a component's obligations and promises via logical assertions in its interface. The contract system monitors program execution, checks whether the assertions hold, and, if not, blames the guilty component. Pinning down the violator gets the debugging process started in the right direction. Quality contracts impose a serious run-time cost, however, and programmers therefore compromise in many ways. Some turn off contracts for deployment, but then contracts and code quickly get out of sync during maintenance. Others test contracts randomly or probabilistically. In all cases, programmers have to cope with lack of blame information when the program eventually fails.

In response, we propose *option contracts* as an addition to the contract tool box. Our key insight is that in ordinary contract systems, server components impose their contract on client components, giving them no choice whether to trust the server's promises or check them. With option contracts, server components may choose to tag a contract as an option and clients may choose to exercise the option or accept it, in which case they also shoulder some responsibility. We show that option contracts permit programmers to specify flexible checking policies, that their cost is reasonable, and that they satisfy a complete monitoring theorem.

Categories and Subject Descriptors D.2.4 [*Software Verification*]: Programming by contract

Keywords programming language design; behavioral software contracts; random testing; probabilistic spot checking

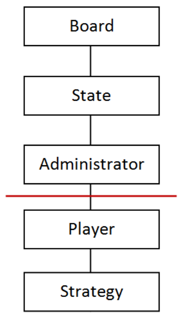
1. The High Costs of Contracts

Large programs consist of many collaborating components. Interfaces describe these collaborations, specifically the promises that each component makes and the obligations that it imposes on its clients for use of its services. The simplest interfaces specify statically checked types. One step up, programmers employ behavioral—also called functional—software contracts to supplement types [2].

Conceptually, a behavioral contract refines the domain and/or range types of a method with logical constraints. Most contract systems allow programmers to express these constraints as boolean-typed expressions in the underlying programming language itself; a few also include additional logical connectors [22]. While some research-oriented combinations of languages and IDEs support a degree of static verification of contracts [1, 13, 32], most systems compile the logical assertions into run-time checks. When these checks discover a contract violation, they raise an exception to stop the program and send along information that explains which component violated which contract and how. This information provides programmer with a starting point for their debugging efforts.

Run-time monitoring means run-time cost. Thus, while contracts allow programmers to express the obligations and promises of a method in as much detail as desired, they also impose a serious cost. To avoid these costs, programmers compromise in two major ways.

One common compromise is to turn off contracts for deployment. Compilers tend to support appropriate switches for this purpose. Unfortunately, when—not if—the program fails eventually, the maintenance programmer will not receive any information from the contract system to narrow down the search for the error. Consider this system fragment:



The `Board` component exports graphs arranged on a grid that satisfy certain conditions. The `State` component includes the grid, together with other game elements, in an internal state representation, which the game `Admininstrator` uses to track a play. `Player` components, contributed by third-party programmers, get to manipulate a part of the game

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509548

state via an appropriate interface, though they probably hand on their arguments to strategy components.

To ensure the integrity of the central game piece, a programmer may impose a contract on the grid. This contract makes sure that the grid preserves its invariants as it flows through `State` and `Administrator` to `Player` and `Strategy`. Turning off this contract for deployment implicates all five components when something goes wrong. Also, when maintenance programmers fix code, they may forget to update the contracts. Alternatively, the programmer moves the contract from `board` to the border between `Admininstrator` and `Player` and keep it around during deployment. This arrangement, however, violates basic design principles—keep the contract with the matching component—and it may harm debugging during development when `State` or `Administrator` accidentally violate the grid's invariants.

Another common compromise is to use random testing in contracts. Consider this contract for a binary search method:

```
<T> Maybe[Integer] bS(T v, T[] d)
 pre  for all i < d.size-1 : d[i] <= d[i+1]
 post @bS.isJust() ==> 0 <= @bS && @bs < d.size()
```

It specifies that the component supplies the `bS` function. Its pre-condition says that the second argument `d` is an array of values sorted by <= (for type `T`). The post-condition promises that the result `@bS` is an optional integer and, if it is an integer, it is a valid index into `d`. The latter suggests `d[@bS] == v`, though the programmer chooses not to promise this fact.

Of course, ensuring that the entire vector `d` is sorted changes the algorithmic complexity of `bS`. In response, a programmer may weaken the contract as follows:

```
<T> Maybe[Integer] bS(T v, T[] d)
 pre  for some random 0 < i && i < d.size()-1:
   d[i-1] <= d[i] && d[i] <= d[i+1]
 post @bS.isJust() ==> 0 <= @bS && @bs < d.size()
```

The randomly checked pre-condition is an algorithmic spot checker [9]. Programmers tend to use such sophisticated algorithms in lieu of an expensive contract to reduce the cost of contract checking and to obtain some assurance that the contract's specification holds.

Like turning off contracts for deployment, weakening assertions via random testing or spot checking poses problems when programs or contracts eventually fail. If component `A` uses a service from component `B` with a randomly checked result and `A` then passes the result to `C`, a contract failure between these two will blame `A` in all existing contract systems—even though `B`'s randomly checked contract does not truly absolve it from its obligations.

2. Reducing Costs with Option Contracts

When Meyer proposed behavioral software contracts [24], he described contracts with analogies to the business world. In a nutshell, a component offers its services together with a contract that makes promises about its services and obliges client components to behave in certain ways if they wish to use these services.

Our linguistic solution to the above problems is to borrow another idea from the business world: *option contracts*. When a server component supplies its services with an option contract, a client component may accept it in two ways. On the one hand, it can *exercise* the option and live up to its conditions. If something goes wrong with the services, the contract system continues to blame the server component as if it had chosen a conventional contract. On the other hand, clients may accept objects monitored by an option contract on an "as is" basis. If a client *transfers* such an object to a third party, the contract system tracks this flow and names the client as a party to any future contract violation concerning this contract. With option contracts, programmers have the infrastructure to mark spot checking in contracts, and they can codify one contract checking policy for development and another one for deployment. Indeed, programmers can develop *dynamic* changes to the contract monitoring policies so that systems can reduce monitoring activities as core components learn to trust some components.

The rest of the paper introduces option contracts, first with an informal specification and then in the context of some non-trivial examples. The fifth section uses experimental setups to demonstrate the performance benefits of option contracts. The sixth section presents a formal specification in the form of a semantic model; the semantics satisfies a completeness theorem [7], the key property of contract systems. The last two sections place our work in context.

3. Exploring Option Contracts

Option contracts extend contract systems in a straightforward manner; they introduce one new mechanism for stating contracts in an interface and several different client-side operations. In this section, we first introduce option contracts abstractly, in a language-neutral manner, and then make this presentation concrete with Racket code snippets; for a precise semantics, see section 6.

3.1 Option Contracts, Abstractly

Adding options to an existing contract system requires at least three changes. The first one allows programmers to annotate a contract for objects—not basic values such as booleans or numbers—in a server component as an option. When an object O flows through such an option contract C, the contract system checks the applicable portions of C and then wraps O in an option-contract object, which includes C.

The second change allows an importing client to *exercise* an option contract. Doing so extracts O and C; it combines these two in a regular contract wrapper that checks every access to O according to C. If the option is not exercised, O is accessed as if it had no wrapper.

The third change concerns the relationship among clients. When a client accepts an object with an option-contract

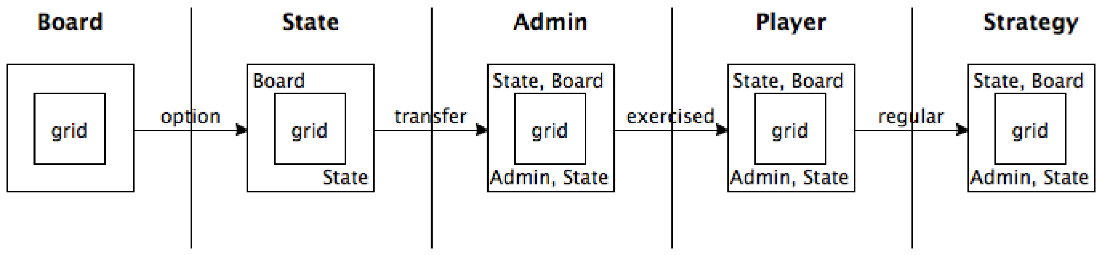


Figure 1: Option contracts by example

wrapper, it may re-export it in two different ways. First, it may *transfer* the wrapped object to its own clients. If it chooses this alternative, it and its client avoid an additional contract wrapper object but they both accept some responsibility for the object's behavior. Second, the client may re-export the object through a regular contract boundary in which case it shoulders no responsibility for the object's behavior but imposes a performance penalty on its clients.

Let us illustrate option contracts with the first example from the introduction. Figure 1 presents the revised view. As discussed, the `Board` component exports a `grid` object. It choose to offer an option contract and `State` transfers `grid` to `Admin`. This arrangement significantly reduces the cost of contract checking for `Board` and `Admin` because `grid` is wrapped in just one contract layer. Since `Player` is "foreign" code, it is natural that the option contract is exercised and passed on to `Strategy` via a regular export. That way `Player` and `Strategy` do not need to accept any blame if the `grid` object misbehaves.

In addition to these operations, our option contract system also provides a mechanism for *stripping*, which extracts the underlying object from an option-contract wrapper. While stripping removes all contract overhead, it also prevents future clients from exercising the option and protecting themselves. In short, stripping chooses raw performance over any form of protection.

3.2 Racket Contracts, a Refresher

The Racket programming language [14] comes with a comprehensive contract system, including contracts for higher-order values [12]. While contracts for first-order values and methods require little support from the programming language or its implementation, higher-order contracts force language designers to think of contract specifications as borderlines between two parties [28].

In this section, we introduce Racket's contract system via a DrRacket component. DrRacket is Racket's IDE, and its code base is already enriched with contracts. DrRacket's text coloring component interactively colors programs as the user edits the program. It differentiates lexemes such as strings and identifiers with different colors and even identifies misspelled words. The implementation uses stream processing functions that accept an input port. Unlike ordinary input ports, these ports are built from the contents of the editor to get efficient and re-usable stream processing. The following function contract specifies how lexers behave:

```
(define plain-lexer/c
  (-> input-port?
      any/c
      (values symbol?
              (maybe/c natural-number/c)
              (maybe/c natural-number/c)
              any/c)))
```

Lexers are functions that accept two inputs: the input port and a "mode" value that the lexer can use as an accumulator to transmit information forward in the stream. The result consists of four `values`: a symbol that describes the token, two numbers that determine the position of the token in the input stream or `#f` if EOF has been reached, and a new mode value, which is passed back into the lexer when processing the next token.

While this contract specifies the basic behavior, lexer functions satisfy a number of additional invariants and the specifications of those require dependent contracts.[1] Racket uses `->i` for dependent contracts; its syntax is similar to that of `->`, except that each place where a contract appears in a `->` expression, an `->i` expression has a name and a contract:

```
(define dep-lexer/c
  (->i ([in input-port?]
        [mode any/c])
       (values
        [tok symbol?]
        [start (tok end)
         (and (not (equal? 'eof tok))
              (and/c natural-number/c (</c end)))]
        [end (tok)
         (and (not (equal? 'eof tok))
              natural-number/c)]
        [new-mode any/c])))
```

[1] This contract is significantly simplified from the actual contract for lexers. The curious reader may want to inspect the full contract in Racket v.5.3.5.

In essence, this contract says that the function still accepts the same two arguments, but now gives them names: `in` and `mode`. Similarly the results are now named `tok`, `start`, `end`, and `new-mode`.

The `->i` combinator specifies a dependency of one part of the contract on other parts by placing the names of the latter in parentheses between the name and the associated contract. Here `end`'s contract depends on the value of `tok` and specifies that returning `#f` is acceptable if `tok` is the symbol `'eof`. Otherwise, `end` is a natural number. Similarly, `start` can be `#f` only if the `tok` symbol is `'eof`, and `start` is additionally constrained to be strictly less than `end`.

The flow of a higher-order value (closure, object) through a contract establishes a boundary between the value and its surroundings. Each one of the two parties, the server and the client, become responsible for some pieces of the contract. For first-order contracts, such as `dep-lexer/c`, the client is responsible for providing arguments that meet the pre-conditions of the contract and the server is responsible for producing results that meet the post-conditions.

Consider the following simple lexer:

```
(define (bogus-lexer in mode)
  (values 'bogus 10 2 #f))
```

We use the `define/contract` construct [27] to attach a contract to `bogus-lexer`:

```
(define/contract bogus-lexer-with-contract
  dep-lexer/c
  bogus-lexer)
```

Evaluating this definition sends `bogus-lexer` through `dep-lexer/c`. The resulting value is associated with the name `bogus-lexer-with-contract`. Its wrapper monitors all flow of values between it and its context, the rest of the module. If the context of `bogus-lexer-with-contract` does not respect the pre-condition, the contract system raises a contract error that assigns blame to the caller of the function:

```
> (bogus-lexer-with-contract "[" #f)
bogus-lexer-with-contract: contract violation
  expected: input-port?
  given: "["
  in: the in argument of
        (->i
         ((in input-port?) (mode any/c))
         (values
          (tok symbol?)
          (start (tok end) ...)
          (end (tok) ...)
          (new-mode any/c)))
  contract from:
        (definition bogus-lexer-with-contract)
  blaming: top-level
```

In case the context respects its obligations, this lexer does not live up to its post-condition. Hence, the contract system points to `bogus-lexer-with-contract` for the violation:

```
> (bogus-lexer-with-contract
   (open-input-string "[")
   #f)
bogus-lexer-with-contract: broke its contract
  promised: (and/c natural-number/c (</c 2))
  produced: 10
  which isn't: (</c 2)
  in: the start result of
        (->i
         ((in input-port?) (mode any/c))
         (values
          (tok symbol?)
          (start (tok end) ...)
          (end (tok) ...)
          (new-mode any/c)))
  contract from:
        (definition bogus-lexer-with-contract)
  blaming: (definition bogus-lexer-with-contract)
```

The view of a contract as an agreement between two parties scales naturally to higher-order functions, borrowing notation for expressing contracts for higher-order functions from types for higher-order functions. For example, here is a higher-order `lexer-tester` with its contract:

```
(define/contract (lexer-tester lexer)
  (-> dep-lexer/c boolean?)
  (lexer (open-input-string "[") #f)
  #t)
```

The tester applies the lexer on a single character stream and if the lexer returns, the tester returns `#t`. However, the contract system wraps the argument lexer with a proxy that checks `dep-lexer/c` for uses of the argument in the body of `lexer-tester`. Thus the tester's contract makes sure that for the lexer to return succesfully, its result must satisfy the post-condition of `dep-lexer/c`.

As far for blame assignment in this case, the contract system passes information about the two responsible parties to the proxy that enforces `dep-lexer/c`. More specifically, it turns the two initial parties for the contract on `lexer-tester` to parties for `dep-lexer/c`. Since the caller of `lexer` is the body of `lexer-tester` and the provider of `lexer` is the caller of `lexer-tester`, the contract system swaps the roles of the original parties; `lexer-tester` becomes the client for the argument and the context of `lexer-tester` the server:

```
> (lexer-tester bogus-lexer)
lexer-tester: contract violation
  expected: (and/c natural-number/c (</c 2))
  given: 10
  which isn't: (</c 2)
```

```
    in: the start result of
        the 1st argument of
        (->
         (->i
          ((in input-port?) (mode any/c))
          (values
           (tok symbol?)
           (start (tok end) ...)
           (end (tok) ...)
           (new-mode any/c)))
         boolean?)
    contract from: (function lexer-tester)
    blaming: top-level
```

In a nutshell, contracts for higher-order functions generalize those for first-order functions; they establish a boundary between two parties and the contract system starts monitoring the values that go through the boundary. When the contract system detects a contract violation, it assigns blame to the party that does not conform with the obligations imposed by the boundary; the client for the negative pieces of the contract and the server for the positive ones.

3.3 Introducing Options

A dependent contract introduces a non-trivial overhead when DrRacket processes an editor as the user types. Hence, it is a natural candidate for an option contract. Here is the code that builds an option contract based on `dep-lexer/c`:

```
(define lexer/c
  (option/c
   dep-lexer/c
   #:tester (λ (l) (try-random-inputs l))))
```

The `option/c` contract combinator accepts an ordinary contract as its first argument and, optionally via the `#:tester` keyword argument, a function that may examine the contracted value in arbitrary ways.

Intuitively, the tester provides a minimum amount of validation of any value that flows through this contract. For DrRacket, the tester ensures that `lexer` is a function of two arguments, and it applies this function to 10 streams of random lowercase characters:

```
(define (try-random-inputs lexer)
  (for ([attempt (in-range 10)])
    (define n (random 50))
    (define s (build-string n lc-letters))
    (lexer (open-input-string s) #f)))


(define (lc-letters _)
  (integer->char (+ 97 (random 26))))
```

When the lexer is passed to the tester, it is wrapped in the contract, so calling it with some random inputs ensures that it obeys the contract for at least a few streams.

A definition that imposes `lexer/c` on `broken-lexer` from above immediately exposes it as a fraud:

```
(define/contract (broken-lexer in mode)
  lexer/c
  (values 'bogus 10 2 #f))
broken-lexer: broke its contract
  promised: (and/c natural-number/c (</c 2))
  produced: 10
  which isn't: (</c 2)
  in: the start result of
      ...
      (option/c
       (->i
        ((in input-port?) (mode any/c))
        (values
         (tok symbol?)
         (start (tok end) ...)
         (end (tok) ...)
         (new-mode any/c)))
       #:tester
       #<procedure>)
  contract from: (function broken-lexer)
  blaming: (function broken-lexer)
```

As the lexer gets tested on a number of random inputs, its contract wrapper discover that its `start` result is too large.

3.4 Exercising and Waiving Options

Once contracted values have passed the tester's examination, the option contract wrapper—technically, a proxy value [28]—no longer checks any properties. Instead, it transparently stores the underlying contract with the value for future use, leaving `dep-lexer/c` inactive for now.

For instance, here is a flawed lexer that passes the tests in the `#:tester` because they supply only lowercase letters:

```
(define/contract (less-broken-lexer in mode)
  lexer/c
  (define c (read-char in))
  (cond [(eof-object? c)
         (values 'eof #f #f #f)]
        [(equal? c #\[)
         (values 'bogus 10 2 #f)]
        [else
         (values 'symbol 1 2 #f)]))
```

This lexer does misbehave if its input port delivers a `#\[` character. Once the `#:tester` function has checked this function on its 100 random lowercase strings, however, even the presence of `#\[` does not lead to a contract violation:

```
> (less-broken-lexer (open-input-string "[") #f)
'bogus
10
2
#f
```

Since the contract on `less-broken-lexer` is still present, however, we can activate it by exercising the option:

```
> ((exercise-option less-broken-lexer)
   (open-input-string "[") #f)
less-broken-lexer: broke its contract
  promised: (and/c natural-number/c (</c 2))
  produced: 10
  which isn't: (</c 2)
  in: the start result of
      the option of
      (option/c
       (->i
        ((in input-port?) (mode any/c))
        (values
         (tok symbol?)
         (start (tok end) ...)
         (end (tok) ...)
         (new-mode any/c)))
       #:tester
       #<procedure>)
  contract from: (function less-broken-lexer)
  blaming: (function less-broken-lexer)
```

The result of `exercise-option` is a function that behaves as if the original contract, `dep-lexer/c`, had been put directly on `less-broken-lexer`. Passing the same arguments to this contracted lexer thus results in a contract violation.

The dual to `exercise-option` is `waive-option`:

```
> ((waive-option less-broken-lexer)
   (open-input-string "[") #f)
'bogus
10
2
#f
```

More specifically, the result of `waive-option` is a function that behaves as if `less-broken-lexer` had been defined without a contract. In particular, `exercise-option` cannot activate `dep-lexer/c` for the result of `waive-option`. At the same time, calling `less-broken-lexer` after waiving the option is cheaper than calling `less-broken-lexer` directly, because it is no longer protected by a proxy.

3.5 Transferring Options

In addition to exercising and waiving an option, a function may decide to shoulder responsibility for the contract without applying the contract again. For example, the following definition returns one of our earlier lexers, but instead of using `lexer/c` for the result contract, it uses `transfer/c`:

```
(define/contract (pick-a-lexer b)
  (-> boolean? transfer/c)
  (if b
      broken-lexer
      less-broken-lexer))
```

When the option is eventually exercised the `pick-a-lexer` function agrees to take on joint responsibility for its result, together with the original option-contract server:

```
> ((exercise-option (pick-a-lexer #f))
   (open-input-string "[")
   #f)
less-broken-lexer: broke its contract
  promised: (and/c natural-number/c (</c 2))
  produced: 10
  which isn't: (</c 2)
  in: the start result of
      the option of
      (option/c
       (->i
        ((in input-port?) (mode any/c))
        (values
         (tok symbol?)
         (start (tok end) ...)
         (end (tok) ...)
         (new-mode any/c)))
       #:tester
       #<procedure>)
  contract from: (function less-broken-lexer)
  blaming multiple parties:
  (function pick-a-lexer)
  (function less-broken-lexer)
```

In contrast, if `pick-a-lexer` were to specify `lexer/c` as its co-domain, only `less-broken-lexer` would have been blamed but at the cost of checking the same contract twice:

```
(define/contract (pick-a-lexer b)
  (-> boolean? lexer/c)
  (if b
      broken-lexer
      less-broken-lexer))
```

```
> ((exercise-option (pick-a-lexer #f))
   (open-input-string "[")
   #f)
pick-a-lexer: broke its contract
  promised: (and/c natural-number/c (</c 2))
  produced: 10
  which isn't: (</c 2)
  in: the start result of
      the option of
      the range of
      (->
       boolean?
       (option/c
        (->i
         ((in input-port?) (mode any/c))
         (values
          (tok symbol?)
```

```
            (start (tok end) ...)
            (end (tok) ...)
            (new-mode any/c)))
          #:tester
          #<procedure>))
    contract from: (function pick-a-lexer)
    blaming: (function pick-a-lexer)
```

3.6 Options and Spot-Checkers

As mentioned in section 1, programmers tend to use random tests and spot checkers, their sophisticated siblings, to replace expensive contracts. Since the contract system does not provide support for substituting contracts with spot checkers, programmers do so in ad-hoc ways such as the contract of `binary-search` from section 1.

Option contracts offer the necessary hooks for systematically using spot checkers and random tests as part of contracts. The following simple syntax re-writing rule, declares a spot checker contract for data structures such as vectors:

```
(define-syntax-rule (spotchecker/c c inv spot)
  (option/c c #:invariant inv #:tester spot))
```

The form `(spotchecker/c c inv spot)` expands into an option contract. The latter uses contract `c` as the cheap contract that each element must satisfy, e.g., that each element is of a specific data type; `inv` as the expensive invariant that should really be monitored; and `spot` as the spot checker that weakens the invariant for the sake of performance.

We can use `spotchecker/c` to express the spot checker of the example of the introduction:

```
(define binary-search/c
  (->i ([k V?]
        [D (spotchecker/c (vectorof V?)
                          (sorted? V<)
                          mostly-sorted?)])
       [index-of-k-in-D (D)
        (maybe/c (</c (vector-length D)))]))
```

Besides providing a concise way to express the contract weakening, the `spotchecker/c` abstraction makes the contract system aware of the fact that a spot checker replaces a precise contract. Hence a client component can use `transfer/c` to share this information with its clients or `exercise-option` to activate the full contract in cases where correctness is more important than performance.

4. Option Contracts in Practice

The key test of a design idea such as option contracts is its application to non-trivial software systems. To demonstrate the value of option contracts, we employed option contracts in three realistic settings: the Typed Racket implementation, DrRacket's text coloring mechanism, and a game called Acquire. The first two are critical parts of the standard distribution of Racket. The third is a semester project for a course on program design at Northeastern University.

4.1 Maintenance of Typed Racket

Typed Racket [30] is a statically typed dialect of Racket. It is designed to accommodate common idioms of Racket programmers [31] and to enable the sound co-operation of typed and untyped Racket components [29]. Both design goals are pivotal for creating a pathway for porting components from the untyped to the typed world, gradually and without significant re-programming effort. The type system is sophisticated and Racket programmers currently pay for this sophistication in the running time of the type checker.

The implementation of Typed Racket defines a series of data structures for storing and propagating type information. Some, such as variants of a type environment, are similar to those of any type checker. Others, such as filters, are specific to features of Typed Racket's type system. The correctness of the type checker relies on the proper use and behavior of functions that access and modify the contents of type representations. For instance, the following function typechecks recursive functions and its correctness depends on the appropriate calls to `with-lexical-env-extend`, which extends the type environment, and `make-arr` and `make-Function` which together construct the type of the recursive function based on the types of its arguments and results:

```
(define (tc/rec-lambda
            formals body name args return)
  (with-lexical-env/extend
   (syntax->list formals) args
   (let∗ ([r (tc-results->values return)]
          [t (make-arr args r)]
          [ft (make-Function (list t))])
     (with-lexical-env/extend
      (list name) (list ft)
      (begin (tc-exprs/check
               (syntax->list body) return)
             (ret ft))))))
```

To enforce the necessary discipline for working in such a stringently structured environment, the developers of Typed Racket attach contracts to the data structures and functions of the type checker libraries. Thus, the (simplified) contract header of `make-arr` looks like this:

```
(define/contract (make-arr args r)
  (-> (listof Type/c) Type/c)
  ...)
```

They also add contracts to the functions of the type checker proper, such as `tc/rec-lambda`, to obtain fine-grained blame information:

```
(define/contract (tc/rec-lambda f b n a r)
 (-> syntax? syntax? syntax? tc-results/c
     tc-results/c)
 ...)
```

The contracts in the implementation of Typed Racket do not check behavioral properties but are limited to structural, type-like properties. Although no contract individually performs expensive computation, the sheer size of the code base and the number of contracts creates a significant overhead due to contract checking. More precisely, even for small Typed Racket programs the time spent in contract checking can dominate type checking. For that reason, the developers of Typed Racket use a configuration module to remove contracts from deployment code and add them back only during development. Alas, it turns out that constantly enabling and disabling contract checking is error-prone and slows down the development cycle. Hence, the developers seldomly reconfigure the code base when extending or fixing problems in the implementation; they reinstate contracts only when debugging broken code without contracts becomes difficult.

Unsurprisingly, software developers do not update disabled contracts as they evolve the code base. Over the course of three months, we spotted numerous changes to the Typed Racket implementation that broke its contracted version. The problems range from syntactic errors in contracts to function arity problems and module dependency omissions.

Our observation suggests that contracts in the Typed Racket implementation should always be checked to some degree. To implement this idea without imposing a performance penalty, we formulated all function contracts in the Typed Racket code base as option contracts, e.g.,

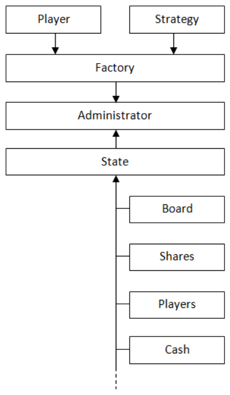
```
(define/contract (make-arr ... )
  (option/c (-> (listof Type/c) Type/c)
            #:tester (λ (x) #t))
            ...)
```

Our options contracts come with a trivial tester that enforces the validation of the first-order properties of the values attached to the contracts but no more contract checking takes place after that. Moreover, we use `waive-option` to remove all the overhead of option contracts. Thus we made the access to the contracted functions as cheap as if contracts had not been applied. As a consequence, type checking with option contracts results in a reduced overhead even for type checking computationally intensive modules (on average less than 15%, see section 5 for details) while performing the syntactic, arity and dependencies checks that would prevent the discrepancies we discovered.

Our study shows that option contracts provide adequate infrastructure for gradually increasing the amount of contract checking in the implementation of Typed Racket without restructuring its code. At the level of our study, option contracts perform basic checks. The developers can now enrich specific testers with deeper behavioral or random and probabilistic tests, increasing their assurance as they see fit.

4.2 Contract Checking Only Where Necessary

Acquire is a market-based board game. Players try to maximize profit from purchasing, trading and expanding hotel franchises. An implementation of Acquire can be conceptually divided into three independent pieces, a `Player`, a `Strategy` and a `Game` component. The `Game` component can be further split into sub-components that implement the `Board` with its pieces, the `State` of the game between rounds and the game `Administrator` that interacts with the `State` in the name of each `Player`; diagrammatically, we have:



In the context of a program design course project, students implement the game, players and strategies. For the final evaluation of the course, players, strategies and game implementations from different students are combined to play a tournament. For this purpose, an additional `Factory` component is inserted between the `Administrator` and the `Player` components that composes strategies with players before linking them to the game `Administrator`.

This set-up naturally gives rise to a world where programmers combine third-party components to obtain working systems. Of course, a successful simulation requires that all components correctly implement agreed-upon interfaces. In addition, for a component to compete successfully in the competition, it must make sure that it protects itself from partners that do not respect their interfaces.

In our implementation, we enforce the components interfaces with contracts. For instance, a simplified version of the contract for the interface of `Strategy` is:

```
(define strategy/c
  (->i ([board (board-well-formed)]
        [player-s-tiles (listof tile?)]
        [cash cash?]
        [available-shares shares?]
        [available-hotels (board)
                          (open-hotels board)])
       (values
        [tile (board player-s-tiles)
         (good-placement board player-s-tiles)]
        [hotel (board available-hotels tile)
         (good-hotel board available-hotels tile)]
        [shares (board cash tile hotel)
                (correct-purchase
                 board cash tile hotel)])))
```

The contract specifies that a strategy function consumes a valid `Board`, the tiles of the `Player`, the `Player`'s cash, and the administrator's hotel Shares, and a list of the hotels that the Player may still found. Given this information, the strategy function returns to the Player the elements of the next move: a valid tile for the Player to place on the board, an available hotel to found or acquire, and a purchase order for shares whose price depends on the State of the game and the tiles placement. The Player component defines players as instances of the class player%. The module also provides the function create, which instantiates player% to obtain instances that execute a given Strategy:

```
(define/contract (create name strategy)
  (-> string? strategy/c (instanceof/c player/c))
  ...)
```

The alert reader may have noticed that both Strategy and Player use strategy/c in the contracts of their interface, and thus the strategy argument of create is subjected to strategy/c twice: once when it is exported from Strategy and once when it is passed to create. This may seem redundant but in fact it is necessary in a world of third-party components. Since a third component, Factory, brings together Player and Strategy, imposing the contract twice is necessary; otherwise, the contract system may blame Factory instead of Strategy when something goes wrong. If strategy/c were missing from Strategy, there is no contract boundary between Factory and Strategy from the perspective of the contract system, and in case of a violation, Factory is identified as the source of the problem because it provided the misbehaving value. Similarly if strategy/c was missing from the pre-condition of create, the contract system may shift blame from Player to Factory. In effect, the two contracts signal to the contract system that Factory is a medium that simply passes values between Player and Strategy and should not get blamed for them. In turn, this helps the programmer to localize the source of a violation of strategy/c. The price for the protection of Factory is the double-checking of the contract.

In addition to the contracts on the interfaces of Player, Strategy and Administrator, our implementation comes with interfaces and contracts for all the components of the game. These contracts enforce internal invariants of the components but they are not critical after the implementation enters a stable phase. Like the Typed Racket contracts, they result in significant slowdown when running games.

Option contracts can help improve the implementation in two different ways. First we employ option contracts to control the cost of checking non-critical contracts. These are all contracts except those between the Player component and, respectively, Strategy and Administrator. Second, with transfer/c and exercise-option, we eliminate the cost of checking strategy/c on the pre-condition of create without completely losing the ability to track misbehaving strategies back to the strategy component.

Hence, strategy/c becomes an option contract:

```
(define strategy/c
  (option/c  (->i ( ... ) ... )))
```

and strategy/c is replaced with transfer/c in the pre-condition of create:

```
(define/contract (create name strategy)
  (-> string? transfer/c (instanceof/c player/c))
  ...)
```

The transfer/c contract recognizes the option contract, adding Factory to the responsible providers and Player to the clients of the strategy function as it flows to create.

Now, with exercise-option we can activate contract checking for strategy/c when instantiating the player% class to obtain a player object:

```
(define (create n strategy)
  (new player%
    [name n] [choice (exercise-option strategy)]))
```

The contract system monitors the exercised strategy function for the exercise/c contract in Player. If it detects a violation of the contract's post-condition it assigns blame both to Strategy and Factory and explains that blame information concerns multiple parties due to the transfer of the strategy function from Factory to Player. Thus the programmer can follow the transfer links back to the actual source of the problem in Strategy.

Our analysis of the Acquire implementation reveals how the features of the options library allow us to turn off contract checking for efficiency and selectively turn it on based on static information about the way components exchange critical values. The next subsection explains how we can enrich these policies with dynamic information.

4.3 Contract Checking Only When Necessary

Since exercise-option is a plain Racket function, Racket programs can call it when a dynamically checked condition holds. DrRacket exploits this ability to selectively exercise option contracts to enforce contracts only for specific lexers that color the contents of the DrRacket editor pane.

Racket is a family of languages, one of which is called racket. Each of these languages comes with its own lexer and DrRacket needs a way to recognize the language of the contents of an editor in order to call the appropriate lexer. Each Racket file, therefore, begins with a #lang specification to indicate which of the Racket-based programming languages the program uses. For example, a file containing code in the Racket language should begin with

```
#lang racket
```

In contrast, document generating programs—such as this subsection—are implemented in the scribble language and their files start with

```
#lang scribble/base
```

To support syntax coloring that is sensitive to the #lang specification, DrRacket comes with a lexer for just this line; its result is the lexer for the language. Once the latter is found, DrRacket dynamically links to the lexer and uses it for the rest of the module.

Lexers for languages of the Racket family implement an interface specified as a contract. We have already discussed a simplified version of this contract in section 2:

```
(define dep-lexer/c
  (->i ([in input-port?]
        [mode any/c])
       (values
        [tok symbol?]
        [start (tok end)
               (if (equal? 'eof tok)
                   #f
                   (and/c natural-number/c
                          (</c end)))]
        [end (tok)
             (if (equal? 'eof tok)
                 #f
                 natural-number/c)]
        [new-mode any/c])))
```

Checking dep-lexer/c causes significant overhead, especially when the lexer tries to color a file as the programmer edits it. Since lexers such as the ones for Racket and Scribble are well-tested and have been found bug-free for a few years, we can consider them "trusted" and use an option contract to silence their contracts. The following option contract around lexer/c also uses a tester that generates a few random inputs to try out the lexer and thus re-establish trust:

```
(define lexer/c
  (option/c
   dep-lexer/c
   #:tester (λ (l) (try-random-streams l))))
```

In contrast to racket and scribble, most of the other lexers distributed with Racket, for example, the lexers that are part of the datalog and algol60 implementations, are not trusted. The predicate trusted-lexer? consumes a lexer and checks if the lexer is in the pre-defined list of trusted lexers of DrRacket. For the lexers that trusted-lexer returns #f, DrRacket should use the dep-lexer/c contract.

Unfortunately, we have now two different contracts for lexers; one with the option for the trusted lexers and one without it for the untrusted lexers. As a result the information about which lexers DrRacket trusts, and which not is spread all over the code base of Racket and requires error-prone and intrusive infrastructure to maintain and update.

One possible solution is to modify the structure of the #lang-line lexer. Instead of having that lexer call directly the language lexers, we could insert a lexer manager function that accesses the list of trusted lexers and attaches to each lexer an appropriate contract. In Racket we can implement this idea easily with with-contract, a construct for creating nested contract regions inside a component [27]:

```
(define (lexer-manager lexer)
  (if (trusted-lexer? lexer)
      (with-contract new-lexer
        #:result lexer/c lexer)
      (with-contract new-lexer
        #:result dep-lexer/c lexer)))
```

Despite its simplicity, the above solution has a shortcoming. Since we separate the lexers and their specification, the contract system does not have enough information to track the error back to the language lexer implementation in case of a contract violation. In particular, from the perspective of the contract system, the contract dep-lexer/c is not between an untrusted lexer and its clients but rather between the contract region labeled new-lexer and its context. Thus, if the post-condition of dep-lexer/c fails, the contract system assigns blame to new-lexer which resides in the module of the #lang-line lexer and not the module that defines the broken lexer. In short, this kind of violation report misleads the programmer because it requires an additional search for the guilty party. With option contracts, we can solve the problem elegantly. First, we use lexer/c as the interface for every lexer, keeping lexers and their specification together. Second, we employ exercise-option and waive-option to activate the contract on untrusted lexers and remove the options related overhead for trusted ones, respectively:

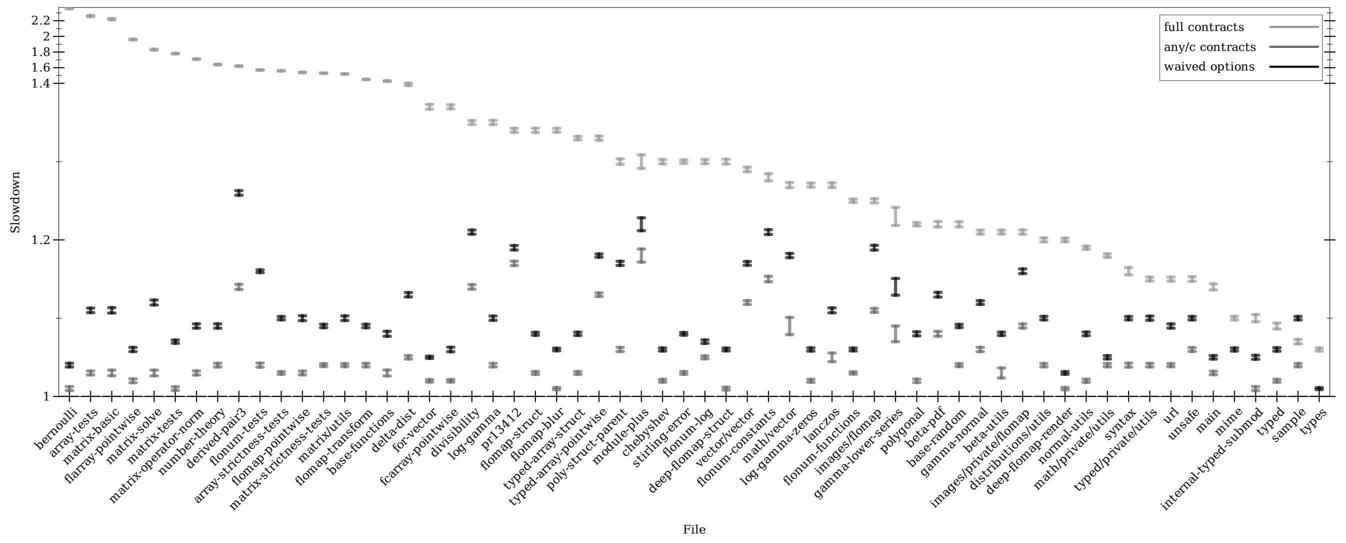
```
(define  (lexer-manager lexer)
  (if (trusted-lexer? lexer)
      (waive-option lexer)
      (exercise-option lexer)))
```

This definition of lexer-manager takes advantage of the option contract associated with the lexer. It activates it only as necessary. If a lexer violates its exercised contract, the contract system can now assign blame to the lexer itself and provide the programmer with precise debugging information.

The options-based solution is part of the latest Racket implementation. Option contracts have made it possible to reduce the cost of coloring text in DrRacket when trusted lexers are used. Conversely option contracts permit to monitor the contract on untrusted lexers and pinpoint faults when contracts are violated.

5. Performance Evaluation

The preceding section shows how option contracts make it straight-forward to implement a variety of contract checking policies. In this section, we provide experimental evidence to support this claim. Specifically, we use the results from measuring three benchmark suites based on our three case studies to make our case. Each measurement compares the execution time of the benchmarks without contracts, with plain contracts and with option contracts. Our measurements



Normalized average of cpu time of 30 runs. The length of the bars denotes the width of the 95% confidence interval for the slowdown. Measurements performed on an Intel Core i5-3550S CPU @ 3.00GHz X 4 with 4 GiB memory running 32-bit Ubuntu 13.04 and Racket 5.3.4.7.

Figure 1: Experimental Results for Typed Racket

confirm that option contracts significantly reduce the slowdown inflicted on programs from contract checking.

5.1 Typed Racket

In the setting of Typed Racket, we measure the cost of typechecking 60 different files from the Racket distribution that are implemented in Typed Racket. These modules range from tests for the Typed Racket implementation to pieces of the `math` and `plot` libraries of Racket. Of the sixty modules, twenty are those with the highest typechecking time, twenty are closest to the median time and the remaining twenty are the most popular, i.e., those that are imported most frequently by other modules in the code base.

Each benchmark consists of a call to the compiler for the corresponding file and a call to the garbage collector. The latter helps us to account for the memory use during compilation. Without it, our benchmark script may terminate before all of the allocated memory has been involved in a collection, unfairly lowering the price of the allocation.

We run our benchmarks in three major modes: without contracts, with the plain contracts that the Typed Racket developers specify, and with all these contracts turned into option contracts as described in sub-section 4.1. In addition to measuring the cost of the option contracts mechanism, we run the benchmarks in the "`any/c`" mode where all contracts are replaced by a trivial, never failing contract. This mode measures the vanilla cost of applying contracts to values.

The chart of files to slowdown in figure 1 displays the results of our measurements. To each file, which defines one of the modules in our benchmarks set, correspond up three error bars. For every file—except two, discussed below—the error bars show up in the same order: the bottom one (gray) shows the slowdown due to contract application, that is the cost of the "`any/c`" mode, the middle one (black) shows the cost of option contracts, and the top one (light gray) shows the cost of plain contracts. All of the numbers are normalized to the no-contracts mode, which serves as our baseline and the width of each bar indicates the length of the 95% confidence interval for the slowdown establishing the significance of our results.[2]

We can read a few general insights from the chart. First, the overall overhead from option contracts fluctuates between 1% and 27%. In all but three benchmarks, it is below 20%, and in all but eight benchmarks, it is below 15%. In all but one case, the overall cost of option contracts is smaller than that of plain contracts by 5% to 127%. Second, since the bars in the chart are sorted by the overhead of plain contract checking, the cost of option contracts does not scale with the cost of plain contract checking. Third, a good proportion of the cost of option contracts comes from contract application (0% – 17%), i.e., the cost of of the "`any/c`" mode.

We have further analyzed the source of the raw cost of option contracts, i.e., the cost of the option contracts mode minus the cost of the "`any/c`" mode. This cost ranges between 1% and 10%. Our measurements show that a small percentage of this cost is due to creating option contracts (1 – 4%, 50ms out of 1100ms in the worst case) while the rest is due to the cost of waiving option contracts together with the cost of the first-order checks that options perform. We omit the details of these measurement here for conciseness.

[2] We follow the methodology of George [15, Section 4.2.4] for the statistical analysis of slowdowns.

Acquire	
contracts	**option contracts**
1.62 (0.02)	1.01 (0.01)

Normalized average of cpu time of 30 runs. The numbers in parenthesis denote the width of the 95% confidence interval for the slowdown. Measurements performed on an Intel Core i5-3550S CPU @ 3.00GHz X 4 with 4 GiB memory running 32-bit Ubuntu 13.04 and Racket 5.3.4.7.

Figure 2: Experimental Results for Acquire

The chart exhibits two anomalies. First, in module `types` (the last column) the "`any/c`" mode is less expensive than the "without contracts" mode by a small amount, 9ms out of 1820ms. We do not have a good explanation for this inversion. Second, module `sample` (second from the right) is the only case where the option contracts mode takes longer that the full contracts mode. Again the absolute difference is small, 34ms out of 1850ms. Type-checking this particular module is trivial in terms of performance, despite the fact that the type-checker constructs 700 contracts. The cost of creating and waiving these contracts is a plausible explanation for the extra cost of the option contracts mode.

We also looked more carefully into the cost that the `any/c` contract imposes relative to the no-contracts baseline and discovered that most of this cost comes in the form of inhibited optimizations, as the code that the contract library generates can confuse the compiler's analysis.

Currently the cost of using option contracts instead of disabling contract checking is prohibitive for integrating option contracts with Typed Racket's typechecker. However, since most of this cost comes from contract application, we anticipate that performance improvements to the contract system of Racket will allow us in the near future to make option contracts the default mechanism for controlling the contract checking overhead in the implementation of Typed Racket.

5.2 Acquire

Our Acquire benchmark runs four complete games with three, four, five and six players respectively. We measure the performance of the simulation in three different modes: without contracts in the implementation of the game, with plain contracts on the interfaces of all the components of the game, and with all these contracts replaced by option contracts. In the latter case, we use `exercise-option` to activate contracts between `Player` and `Administrator`; following the description in section 4.2, we also replace `strategy/c` in the contract of `create` with `transfer/c` and apply `exercise-option` to activate the option contract on the strategy function within `create`.

The table in figure 2 shows the results of our measurements. Contract checking results in 62% slowdown and option contracts manage to bring it down to 1% even though the critical contracts between `Strategy`, `Player` and `Administrator` are checked.

5.3 DrRacket's Lexer

For DrRacket's code coloring infrastructure, we construct two benchmarks. The first loads a 5,000-line `racket` file and measures the time it takes to color the lexemes of the file while treating the lexer of the Racket language as a trusted lexer. The second carries out the same task but considering the lexer as untrusted. Similar to the Acquire benchmarks, we run the two benchmarks without contracts, with plain contracts and with option contracts. For the "with contracts" mode, all the lexers involved (the `#lang`-line lexer and the `racket` lexer) are equipped with the `dep-lexer/c` contract mentioned in section 4.3. For the "with option" mode, we use the `lexer/c` option contract. The tester of this contract tries the given lexer on ten stream inputs of random size between zero and a hundred characters.

The table in figure 3 shows the results of our measurements. Contract checking results in 21% slowdown. In the trusted case, option contracts eliminate this overhead almost entirely (1%), which implies that the cost from the random tests of the option contracts are insignificant. In the untrusted case, we get a 14% slowdown from option contracts. Since the `lexer/c` contract is exercised for the Racket lexer, the contract system monitors this contract while the lexer colors the file. The 7% improvement compared to the plain contracts mode is due to the fact that the `#lang`-line lexer is still considered trusted and its option contract is not exercised.

DrRacket's Lexer		
	contracts	**option contracts**
trusted	1.21 (0.02)	1.01 (0.01)
untrusted	1.21 (0.02)	1.14 (0.01)

Normalized average of cpu time of 30 runs. The numbers in parenthesis denote the width of the 95% confidence interval for the slowdown. Measurements performed on an Intel Core i5-3550S CPU @ 3.00GHz X 4 with 4 GiB memory running 32-bit Ubuntu 13.04 and Racket 5.3.4.7.

Figure 3: Experimental Results for Racket's Lexer

6. Option Contracts in Theory

Like type systems, contract systems should satisfy some basic universal properties so programers can rely on their feedback [7]. Specifically, (1) when a contract system blames a component for a violation, the component must be able to affect the flow of values through the violated contract boundary and (2) contract systems must be able to monitor all the channels of communication between components so that no unchecked value may migrate into a seemingly protected component. As history has shown, it is easy for a language designer to violate these conditions [6, 7]. This section shows that option contracts satisfy these properties.

We specify the semantics of option contracts as an extension to CPCF, called OCPCF. In turn, CPCF [5] extends Plotkin's PCF [26] with generic constructs for formulating and monitoring contracts. Unlike Racket, CPCF is a simply-typed higher-order functional language; we use a typed model to expose the orthogonality of types and contracts and to help the designers of typed languages with importing option contracts into their world.. A contract expresses computable properties of values at any level in the type hierarchy. Monitors interpose contracts between value producers and consumers and, at runtime, unfold into code that enforces the properties.

$$
\begin{array}{llcl}
\textbf{Types} & \tau & = & o \mid \tau \to \tau \mid \mathsf{con}(\tau) \\
 & o & = & \mathsf{I} \mid \mathsf{B} \\
\textbf{Contracts} & c & = & \mathtt{flat}(e) \mid c \overset{d}{\mapsto} (\lambda x.c) \\
 & & \mid & \mathtt{option}(c \overset{d}{\mapsto} (\lambda x.c), e) \\
 & & \mid & \mathtt{transfer} \\
\textbf{Terms} & e & = & v \mid x \mid e\ e \mid \mu x{:}\tau.e \mid e+e \\
 & & \mid & e-e \mid e \wedge e \mid e \vee e \mid \mathtt{zero?}(e) \\
 & & \mid & \mathtt{if}\ e\ e\ e \mid \mathtt{M}^{l,l}_{l}(c,e) \\
 & & \mid & \mathtt{exercise}(e) \mid \mathtt{waive}(e) \\
\textbf{Comp. Values} & v^* & = & \lambda x{:}\tau.e \\
\textbf{Values} & v & = & \mathsf{b} \mid v^* \\
\textbf{Base Values} & \mathsf{b} & = & 0 \mid 1 \mid -1 \mid \ldots \mid \mathtt{tt} \mid \mathtt{ff}
\end{array}
$$

Figure 4: OCPCF: source syntax

Figure 4 shows the source syntax for OCPCF. Like CPCF, OCPCF is a typed language that comes with predicate contracts for primitive values, written $\mathtt{flat}(e)$, and contracts for higher-order functions, written $c_1 \overset{d}{\mapsto} (\lambda x.c_2)$ where the "precondition" c_1 is the contract on the argument and the "postcondition" c_2 is the contract on the result. The latter is a *dependent* function contract where the argument to the function is bound to x making it visible in the post-condition of the contract and allowing for properties of the result to depend on the argument. In addition, OCPCF offers two more kinds of contracts: option contracts and transfer contracts. Concretely, $\mathtt{option}(c,e)$ pairs a function contract c with a tester e that exercises the function; like our Racket implementation, OCPCF syntax restricts option contracts to function contracts. A `transfer` contract tags a value as trusted and never signals a contract violation. If the value comes with an option contract, the consumer is added to its list of responsible client parties and the provider to the responsible server parties.

The programmer can apply a contract c to a component e using $\mathtt{M}^{k,l}_{j}(c,e)$. Such a monitor separates two components: the service provider e and its surrounding context, the client. It corresponds to Racket's `define/contract` and `with/contract` forms. The labels k, l and j serve as identifiers for the exporting component; dubbed the server; the importing component, the client, and the contract itself, respectively. In source code, monitors have only one server and client label, but due to option contracts monitors may accumulate multiple server and client labels during evaluation. These sets of labels represent those components that have endorsed a value with an option contract.

A component can use $\mathtt{exercise}(e)$ to remove the option contract from the value of e (if any) and replace it with the underlying contract. The $\mathtt{waive}(e)$ expression discards the option contract (if any) from the value of e.

Here is an example contract for a derivative operator:

$$
\begin{array}{lcl}
c & = & c_1 \overset{d}{\mapsto} (\lambda f.\mathtt{option}(c_2, \mathit{tester})) \\
c_1 & = & \mathit{positive} \overset{d}{\mapsto} (\lambda x.\mathit{positive}) \\
c_2 & = & \mathit{positive} \overset{d}{\mapsto} (\lambda x.\text{close-to-slope-of-f@x})
\end{array}
$$

The post-condition of c_2 asserts that the slope of f around x is close to the result of *deriv* applied to f at x. The option contract for c_2 performs some random testing on f as specified in the *tester* predicate.

Component e_1, named k_1, imports *deriv* with contract c and applies it to a function f:

$$e_1 = (\mathtt{M}^{s,k_1}_{j_1}(c, \mathit{deriv}))\ f$$

Then e_1 does not exercise the option contract, but transfers its result to a client e_2, named k_2:

$$e_2 = (\mathtt{M}^{k_1,k_2}_{j_2}(\mathtt{transfer}, e_1))\ 0$$

Component e_2 imports the result of e_1, trusts the random testing and opts out of any further checks related to c_2. In other words, e_2 uses the result of e_1 without any contract checking. By importing a `transfer`ed value, client e_2 acknowledges responsibility as a client of the randomly checked result of e_1. Moreover, since e_1 transfers its result to e_2, it deliberately chooses to become a server for this value.

A different client e_3, labeled k_3, imports e_1 and exercises the option:

$$e_3 \quad = \quad (\mathtt{exercise}(\mathtt{M}^{k_1,k_3}_{j_3}(\mathtt{transfer}, e_1)))\ 0$$

Let f' denote the derivative function of f and c_2' and tester' the result of substituting f for x in c_2 and *tester*, respectively, which is necessary due to the dependent nature of c. Client e_3 considers c_2', the contract of f', to be critical and thus decides not to trust the random testing. Since e_3 applies f' to a value that violates the precondition of c_2', the contract system detects the violation and blames the components that imported f', in this case e_1 and e_3 and reports their labels k_1 and k_3, respectively, in the contract error message. If e_3 provided to f' a value that does not violate the precondition of c_2' and the contract system detected a violation of the postcondition of c_2', it would blame all the responsible servers of f' and report their labels, namely s and k_1.

Finally, a client e_4 deems that c_2' is not a critical property to monitor and decides to `waive` the option contract on f'.

The `waive` expression allows e_4 to use the function without paying for the overhead of the option contract monitor:

$$e_4 \quad = \quad \texttt{waive}(\texttt{M}^{k_1,k_4}_{j_4}(\texttt{transfer}, e_1))\, 0$$

The `waive` operator frees the value from the monitor and the application proceeds as if f' had never had a contract.

In order to prove the fundamental soundness theorem for our contract system, we formulate a reduction semantics. Interested readers can find the concrete definition of the semantics in appendix A.1. What we need to know here is that the semantics specifies a reduction relation $\rightarrow$ whose transitive closure $\rightarrow^*$ reduces the program to its final value (if any). The relation requires an additional kind of term syntax—so-called option guards $\texttt{O}^{l,k}_{j}(c,e)$—to express values with option contracts wrapped around them.

To demonstrate the workings of our semantics, we revisit our examples from above. Under the semantics of the model, e_2 reduces as follows:

$$\begin{aligned} e_2 &\rightarrow^* (\texttt{M}^{k_1,k_2}_{j_2}(\texttt{transfer}, \texttt{M}^{s,k_1}_{j_1}(\texttt{option}(c'_2, \mathit{tester}'), f')))\, 0 \\ &\rightarrow^* (\texttt{M}^{k_1,k_2}_{j_2}(\texttt{transfer}, \texttt{O}^{s,k_1}_{j_1}(c'_2, f')))\, 0 \\ &\rightarrow^* (\texttt{O}^{\{k_1,s\},\{k_2,k_1\}}_{j_1}(c'_2, f'))\, 0 \\ &\rightarrow \ f'\, 0 \end{aligned}$$

Since the option contract is not exercised the result of e_1 does not come with a monitor around it and the application proceeds without any contract checking. In short, the use of 0 does not result in a violation of c'_2.

In the case of e_3, the reduction proceeds differently:

$$\begin{aligned} e_3 &\rightarrow^* (\texttt{exercise}(\texttt{M}^{k_1,k_3}_{j_3}(\texttt{transfer}, \texttt{O}^{s,k_1}_{j_1}(c'_2, f'))))\, 0 \\ &\rightarrow^* (\texttt{exercise}(\texttt{O}^{\{k_1,s\},\{k_3,k_1\}}_{j_3}(c'_2, f')))\, 0 \\ &\rightarrow^* (\texttt{M}^{\{k_1,s\},\{k_3,k_1\}}_{j_1}(c'_2, f'))\, 0 \end{aligned}$$

Since e_3 chooses to exercise the option contract, the application involves checking c'_2 and thus results in a contract error. The contract system blames the components labeled k_1 and k_3, meaning e_1 and e_3, the components that imported f':

$$e_3 \ \rightarrow^* \ \texttt{error}^{\{k_3,k_1\}}_{j_1}$$

The last example e_4 shows how `waive` discards an option guard around a value:

$$e_5 \quad \rightarrow^* \quad \texttt{waive}(\texttt{O}^{\{k_1,s\},\{k_4,k_1\}}_{j_1}(c'_2, f'))\, 0 \quad \rightarrow \quad f'\, 0$$

As the examples point out, our model introduces a policy for assigning blame that somewhat deviates from the Findler-Felleisen model. Instead of blaming one component for violating its contractual obligations, our new model assigns blame to potentially many components. Moreover, `transfer` contracts permit values to entirely bypass contract checks, undermining the ability of the contract system to detect contract violations. Considering the difficulties of getting blame assignment correct for the Findler-Felleisen version of contracts [6], these changes call for a formal investigation of the correctness of our contract system. In particular, we must prove that our contract system is able to:

- disallow values to bypass contract checks, unless they are explicitly transferred from one component to another,
- keep track of transferred values, and
- on contract violation, report all the parties that created the value or were involved in a transfer.

These informal criteria can be formalized as a variant of a basic correctness property for contract systems, dubbed complete monitoring [7].

THEOREM 1. *$\rightarrow$ satisfies complete monitoring for OCPCF*

PROOF. A detailed account of the proof technique and the proof itself is provided in appendix A.2 and A.3. ∎

7. Related Work

Eiffel [23, 25] first popularized software contracts and introduced the design-by-contract paradigm. The latter builds on a view of the world of software components as a market where software contracts play the role of business contracts, imposing obligations and making promises about components. Option contracts take this analogy one step further introducing notions that correspond to financial options together with actions such as transfer and exercise.

Since Eiffel introduced contracts, contracts have been used both for extended static checking [1, 3, 10, 16, 33], runtime monitoring of higher-order programs [12, 18], and even a mixture of the two approaches [17, 21]. Nowadays, contracts in one form or another are part of many mainstream languages and libraries. Option contracts live in the world of dynamically enforced contracts and build on a decade of linguistic research on software contracts [12].

The designers of languages with software contracts recognize the performance impact of contract checking and provide compile-time mechanisms that disable contract checking entirely or partially. For example, Ada [20] programmers can use the built-in pragma `Assert` for this purpose. Eiffel [8] programmers can modify the `Assert` options of the Eiffel compiler to enable or disable specific kinds of assertions. For instance, the `"Supplier Precondition"` option addresses interaction with trusted libraries, disabling all assertions for these libraries except pre-conditions. Racket programmers implement such compile time mechanisms with macros. As discussed, the Typed Racket developers achieve a reasonably flexible use of the contracts in their implementation. Unfortunately, all these methods for controlling contract checking permit only *static*, all-or-nothing policies either at the component or the contract level. Option contracts offer another alternative, namely, fine-grained control of contract checking without weakening the precision of blame assignment. Moreover, option contracts can also be used to implement *dynamic* contract-checking policies, as demonstrated in our DrRacket example.

8. Conclusion

Software contracts are notorious for their cost. Given the economic incentives for performance, any given programmer routinely disablesf contracts for product deployment, acting, as Hoare [19] puts it, "[as] a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea." Our work tackles this problem, giving programmers new powers to create new policies of contract checking, avoiding some performance overhead and the code base skew that results from disabled contracts.

With this new expressive power of option contracts, programmers acquire new responsibilities. Client-side programmers must adapt their programming style to option contracts. The creators of server components cannot remove option annotations from contracts in a lighthearted manner because doing so may have serious performance implications. The Racket community has just begun its experimentation with options, and we hope to report our insights in the future.

Acknowledgments Thanks to Amal Ahmed and Amr Sabry for the inspiring exchange that triggered this research. Sam Tobin-Hochstadt coined the phrase "option contracts." AFOSR supported the exploration of contracts for Dimoulas and Felleisen in the past; NSF provides support for Findler.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system. In *CASSIS*, pages 49–69, 2004.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7): 38–45, July 1999.

[3] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 158, Compaq SRC Research Report, 1998.

[4] C. Dimoulas. *Foundations for Behavioral Higher-Order Contracts*. PhD thesis, Northeastern University, 2012.

[5] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, 33(5):16:1 – 16:29, 2011.

[6] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *POPL*, pages 215 – 226, 2011.

[7] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *ESOP*, pages 211 – 230, 2012.

[8] *Standard ECMA-367 Eiffel: Analysis, Design and Programming Language*. Ecma International, 2006.

[9] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. *Journal of Computer and System Sciences*, 60(3):717–751, 200.

[10] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110, 2010.

[11] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[14] M. Flatt and PLT. Reference: Racket. Reference Manual PLT-TR2010-reference-v5.3.3, PLT Design Inc., February 2013. URL `http://racket-lang.org/techreports/`.

[15] A. George. *Three Pitfalls in Java Performance Evaluation*. PhD thesis, Ghent University, 2008.

[16] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL*, pages 353–364, 2010.

[17] J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *TFP*, pages 54–69, 2007.

[18] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *FLOPS*, pages 208–235, 2006.

[19] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, 1973.

[20] *Ada 2012 Language Reference Manual*. International Organization for Standardization, 2012.

[21] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic, 2006. URL `http://sage.soe.ucsc.edu/`.

[22] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. 1999.

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

[24] B. Meyer. Applying design by contract. *IEEE Computer*, 25 (10):40–51, 1992.

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

[26] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[27] T. S. Strickland and M. Felleisen. Nested and dynamic contract boundaries. In *IFL*, pages 141 – 158, 2009.

[28] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators. In *OOPSLA*, pages 943–962, 2012.

[29] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, pages 964–974, 2006.

[30] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–407, 2008.

[31] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, pages 117–128, 2010.

[32] S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In *OOPSLA*, pages 537–554, 2012.

[33] D. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *POPL*, pages 41–52, 2009.

A. Appendix

This appendix presents a proof of complete monitoring [7] for OCPCF, a model of a typed variant of Racket with option contracts. Complete monitoring guarantees that the contract system assigns blame accurately and protects components effectively. A complete monitoring theorem is thus the minimal standard that a contract system should satisfy.

Unfortunately, the OCPCF model does not come with the necessary infrastructure to prove complete monitoring. We deal with this issue via an indirect approach. We first define another model *OCPCF, which has the required hooks for establishing complete monitoring. After that we show that the two models are observably equivalent.

The first section of this appendix introduces a suitable reduction semantics [11] of OCPCF. While the second section presents *OCPCF, the third one proves that it satisfies complete monitoring and bi-simulates OCPCF.

A.1 Reduction Semantics for OCPCF

Figures 5 through 7 present the evaluation syntax of OCPCF as an extention to its source syntax (figure 4) plus its evaluation contexts and reduction rules. The rules that do not involve contracts are the same as those for Plotkin's PCF.

The evaluation syntax of OCPCF comes with two kinds of monitors: $\mathtt{M}_j^{k,l}(c,e)$ and $\mathtt{O}_j^{k,l}(c,e)$. The first correspond to components whose result the contract system monitors for c while the second denote components whose result the contract system does not monitor for c but they contain c and enable the use of `exercise` or `transfer`.

$$\begin{array}{llcl}
\textbf{Terms} & e & = & \ldots \mid \mathtt{check}_l^{\bar{l}}(e,v) \mid \mathtt{try}_l^{\bar{l}}(e,e) \\
 & & \mid & \mathtt{error}_l^{\bar{l}} \mid mg \mid og \\
\textbf{Guards} & mg & = & \mathtt{M}_l^{\bar{l},\bar{l}}(c,e) \\
 & og & = & \mathtt{O}_l^{\bar{\bar{l}},\bar{l}}(c,e) \\
\textbf{Values} & & & \\
MGV: & mgv & = & \mathtt{M}_l^{\bar{l},\bar{l}}(c,v^*) \\
OGV: & ogv & = & \mathtt{O}_l^{\bar{l},\bar{l}}(c,v^*) \\
 & v^* & = & \ldots \mid mgv \mid og
\end{array}$$

Figure 5: OCPCF: evaluation syntax

The monitor expression $\mathtt{M}_j^{k,l}(\mathtt{flat}(e),\mathtt{b})$ reduces to a check that tests whether predicate e holds for $\mathtt{b}$. If the check succeeds, it returns $\mathtt{b}$; else it raises a contract violation $\mathtt{error}_j^k$ that blames component k for violating contract j.

The monitor expression $\mathtt{M}_j^{k,l}(c_1 \overset{d}{\mapsto} (\lambda x.c_2), v)$ uses a monitor guard to protect the function v during future applications. When the guard is applied, the reduction rule decomposes the contract, creates a guard that monitors the argument with c_1 and then applies v to the guard. The responsible reduction rule also constructs a guard that monitors the result of the application with c_2. The monitor of the argument flips the server and client labels while the monitor of the result leaves them as they are. The label swap signals that the precondition becomes the responsibility of the client, which after all provides the argument, while the post-condition remains the responsibility of the server. For the dependent aspect of the contract, the rule implements the *indy* semantics [6], meaning x in c_2 is replaced by the argument monitored by c_1 where j replaces k. The intuition behind the labels selection is that the original client l is responsible for the argument's post-condition but the contract j is responsible for the treatment of the argument inside the contract's code.

Finally, $\mathtt{M}_j^{k,l}(\mathtt{option}(c,e),v)$ employs `try` to test whether a guard for v with c satisfies the tester e. Since e is part of the contract itself, the monitor for the test carries j as its client label. If the test does not lead to a contract error, `try` returns an option guard, $\mathtt{O}_j^{k,l}(c,v)$, which is a value that stores v together with c and the labels of the original monitor.

An option guard represents a partially checked function; it keeps track of the contract so that a component may `exercise` the option. In addition, it accumulates the labels of the components that accept the option guard to accurately assign blame when contract checking detects a contract breach. A component may use an option guard like any other function. In particular, the application $\mathtt{O}_j^{k,l}(c,v_f)\ v$ reduces to an ordinary application $v_f\ v$.

$$\begin{array}{rl}
E = & [\,] \mid E\ e \mid v\ E \mid E+e \mid v+E \mid E-e \mid v-E \\
\mid & E \wedge e \mid v \wedge E \mid E \vee e \mid v \vee E \mid \mathtt{zero?}(E) \\
\mid & \mathtt{if}\ E\ e\ e \mid \mathtt{M}_j^{\bar{k},\bar{l}}(c,E) \mid \mathtt{check}_l^{\bar{k}}(E,v) \mid \mathtt{try}_l^{\bar{l}}(E,v) \\
\mid & \mathtt{exercise}(E) \mid \mathtt{waive}(E)
\end{array}$$

Figure 6: OCPCF: evaluation contexts

When a monitoring expression employs `transfer` as a contract, guarded values experience a change of labels. Specifically, $\mathtt{M}_j^{k,l}(\mathtt{transfer},\mathtt{O}_j^{h,q}(c,v))$ updates the option guard's labels; it adds the server and client labels of the monitor to the server and client labels of the guard to produce $\mathtt{O}_j^{kh,lq}(c,v)$. This labeling shows how the consumer of the guard takes responsibility as a client and the exporting component as a server. Monitors of `transfer` contracts do not affect other kind of values; they simply pass them on.

Exercising an option transforms an option guard $\mathtt{O}_j^{k,l}(c,v)$ into a monitored guard, $\mathtt{M}_j^{k,l}(c,v)$.

Due to `transfer` contracts and `exercise`, monitor guards carry a pair of sets of labels instead of one server and one client label. This implies that in case of a contract failure `check` blames all the labels at the server position on the guard, i.e., it blames all the components that accepted responsibility for the guarded value.

Like `exercise`, `waive` affects only guards. It extracts v from $\mathtt{O}_j^{\bar{k},\bar{l}}(c,v)$ and otherwise passes on all other values.

$$
\begin{array}{llll}
E[\cdots] & \rightarrow & E[\cdots] & \\
\hline
\mathtt{n}_1 + \mathtt{n}_2 & . & \mathtt{n} & \text{where } n_1 + n_2 = n \\
\mathtt{n}_1 - \mathtt{n}_2 & . & \mathtt{n} & \text{where } n_1 - n_2 = n \\
\mathtt{zero?}(\mathtt{0}) & . & \mathtt{tt} & \\
\mathtt{zero?}(\mathtt{n}) & . & \mathtt{ff} & \text{if } n \neq 0 \\
v_1 \wedge v_2 & . & v & \text{where } v_1 \wedge v_2 = v \\
v_1 \vee v_2 & . & v & \text{where } v_1 \vee v_2 = v \\
\mathtt{if}\ \mathtt{tt}\ e_1\ e_2 & . & e_1 & \\
\mathtt{if}\ \mathtt{ff}\ e_1\ e_2 & . & e_2 & \\
\lambda x.e\ v & . & \{v/x\}e & \\
\mu x.e & . & \{\mu x.e/x\}e & \\
\hline
\mathtt{M}_j^{\bar{k},\bar{l}}(\mathtt{flat}(e),\mathtt{b}) & . & \mathtt{check}_j^{\bar{k}}(e\,\mathtt{b},\mathtt{b}) & \\
\mathtt{M}_j^{\bar{k},l\bar{l}}(\mathtt{option}(c,e),v) & & & \\
\quad \mathtt{try}_j^{\bar{k}}(e\,\mathtt{M}_j^{\bar{k},j\bar{l}}(c,v),\mathtt{O}_j^{\bar{k},l\bar{l}}(c,v)) & & & \\
\mathtt{M}_j^{k\bar{k},\bar{l}}(c_1 \overset{d}{\mapsto} (\lambda x.c_2),v_f)\ v & . & & \\
\quad \mathtt{M}_j^{k\bar{k},\bar{l}}(\{\mathtt{M}_j^{\bar{l},j\bar{k}}(c_1,v)/x\}c_2, v_f\ \mathtt{M}_j^{\bar{l},k\bar{k}}(c_1,v)) & & & \\
\mathtt{M}_j^{k,l}(\mathtt{transfer},v) & . & v & \text{if } v \notin OGV \\
\mathtt{M}_{j'}^{\bar{k},\bar{l}}(\mathtt{transfer},\mathtt{O}_j^{\bar{h},\bar{q}}(c,v)) & . & \mathtt{O}_j^{\bar{k}\bar{h},\bar{l}\bar{q}}(c,v) & \\
\mathtt{check}_j^{\bar{k}}(\mathtt{tt},v) & . & v & \\
\mathtt{check}_j^{\bar{k}}(\mathtt{ff},v) & . & \mathtt{error}_j^{\bar{k}} & \\
\mathtt{try}_j^{\bar{k}}(\mathtt{tt},v) & . & v & \\
\mathtt{try}_j^{\bar{k}}(\mathtt{ff},v) & . & \mathtt{error}_j^{\bar{k}} & \\
\hline
\mathtt{exercise}(v) & . & v & \text{if } v \notin OGV \\
\mathtt{exercise}(\mathtt{O}_j^{\bar{h},\bar{q}}(c,v)) & . & \mathtt{M}_j^{k\bar{h},l\bar{q}}(c,v) & \\
\mathtt{waive}(v) & . & v & \text{if } v \notin OGV \\
\mathtt{waive}(\mathtt{O}_j^{\bar{h},\bar{q}}(c,v)) & . & v & \\
\mathtt{O}_j^{\bar{k},\bar{l}}(c,v_f)\ v & . & v_f\ v & \\
\hline
E[\mathtt{error}_j^{\bar{k}}] & \rightarrow & \mathtt{error}_j^{\bar{k}} &
\end{array}
$$

Figure 7: OCPCF: reduction semantics

A.2 *OCPCF: A Theorem-Friendly Semantics

Complete monitoring requires that we decorate OCPCF programs with annotations for *ownership* and *obligations* [7]. Ownership and obligations are tools for reasoning about the behavior of the contract system independently of monitors.

An obligation annotation $\lfloor \mathtt{flat}(e) \rfloor^{\bar{l}}$ denotes that components $\bar{l}$ are responsible for meeting the contract $\mathtt{flat}(e)$. An ownership annotation $|e|^l$ indicates that component l owns e.

Ownership provides a mechanism for tracking the migration history of values and thus helps establish that the contract system offers sufficient protection. A revised semantics propagates ownership annotations. If the ownership annotation of a term is different than the owner of its context and the term is not embedded in an appropriate monitor, then the contract system allows values to leak from one component to another without inspection. Put positively, if a contract system manages to enforce a single-owner policy throughout program execution, i.e., every value has a single owner, it allows preogrammers to protect components completely.

As for blame assignment, a correct contract system should blame a component only if it breaks one of its promises. We therefore mark the pieces of a contract that a component needs to live up to with *obligation* annotations. Equipped with this machinery, we can specify what it means for a contract system to assign blame correctly. A contract system may blame a component only if one of its values violates one of its contractual obligations.

Adding ownership and obligations annotations to OCPCF directly poses a challenge. Monitors for `transfer` contracts allow values to circumvent the contract system as they migrate from one component to another. Thus a naive annotation of the OCPCF semantics with ownership labels violates the single-owner policy and breaks complete monitoring. Fortunately, we can construct an extension of CPCF with annotations [7] that is equivalent to OCPCF and tracks uninspected values. We call the new language *OCPCF.

$$
\begin{array}{llll}
\textbf{Contracts} & c & = & \ldots \mid \lfloor \mathtt{option}(c \overset{d}{\mapsto} (\lambda x.c), e) \rfloor^{\bar{l}} \\
 & & \mid & \mathtt{transfer} \\
\textbf{Terms} & e & = & \ldots \mid \mathtt{exercise}(e) \mid \mathtt{waive}(e)
\end{array}
$$

$$\boxed{\Gamma; l \Vdash e}$$

$$
\frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \mathtt{exercise}(e)} \qquad \frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \mathtt{waive}(e)}
$$

$$\boxed{\Gamma; \bar{k}; \bar{l}; j \rhd c}$$

$$
\frac{}{\Gamma; \bar{k}; \bar{l}; j \rhd \mathtt{transfer}}
$$

$$
\frac{\Gamma; \bar{k}; \bar{l}; j \rhd c \qquad \Gamma; j \Vdash e}{\Gamma; \bar{k}; \bar{l}; j \rhd \lfloor \mathtt{option}(c, |e|^j) \rfloor^{\bar{k}}}
$$

Figure 8: *OCPCF: Well-formed source programs

The syntax of *OCPCF extends annotated CPCF with ownership and obligations annotations; see top of figure 8. The bottom of the figure adds rules for well-formed source programs and contracts to those of CPCF. The most interesting rules are those for contracts. Transfer contracts never fail and thus do not impose obligations and are always well-formed. In contrast, option contracts may raise a contract violation due to a failed test and thus they impose obligations like flat contracts. Also, an option contract must own its tester term.

In addition to `try` and `check`, the evaluation syntax of *OCPCF (figure 9) introduces a small ecosystem of guards. We group guards in three categories: monitor guards M, op-

$$
\begin{array}{llcl}
\textbf{Terms} & e & = & \ldots \mid \texttt{error}_l^{\bar{l}} \mid \texttt{try}_l^{\bar{l}}(e,e) \\
 & & \mid & mg \mid og \mid wg \\
\textbf{Guards} & mg & = & \texttt{M}_l^{l,l}(c,e) \mid {}^*\texttt{M}^{l,l}(e) \mid |mg|^l \\
 & og & = & \texttt{O}_l^{l,l}(c,e) \mid {}^*\texttt{O}^{l,l}(e) \mid |og|^l \\
 & wg & = & \texttt{W}^{l,l}(e) \mid |wg|^l \\
\textbf{Values} & & & \\
MGV: & mgv & = & \texttt{M}_l^{l,l}(c,v^*) \mid {}^*\texttt{M}^{l,l}(v^*) \mid |mgv|^l \\
OGV: & ogv & = & \texttt{O}_l^{l,l}(c,v^*) \mid {}^*\texttt{O}^{l,l}(v^*) \mid |ogv|^l \\
WGV: & wgv & = & \texttt{W}^{l,l}(v^*) \mid |wgv|^l \\
 & v^* & = & \ldots \mid mgv \mid ogv \mid wgv
\end{array}
$$

Figure 9: *OCPCF: Evaluation syntax

tion guards O, and *waived guards* W. We use waived guards to mark an unchecked value as foreign inside a component when the value crosses the component's boundary without the contract system's protection but with its approval. Unlike OCPCF, both monitor and option guards carry only two labels rather than two sets of labels. However, the star versions of the guards, $^*M$ and $^*O$ allow us to record multiple components that have accepted an option guard. For instance a stack of star-option guards with an option guard at the bottom in *OCPCF, ${}^*\texttt{O}^{k_1,l_1}(\ldots{}^*\texttt{O}^{k_n,l_n}(\texttt{O}_j^{h,q}(c,v))\ldots)$, corresponds to a single option guard $\texttt{O}_j^{k_1\ldots k_n h, l_1\ldots l_n q}(c,v)$ in OCPCF. Stacks of guards have the advantage of introducing separate boundaries between components that have accepted an option guard. This allows the guards to accommodate ownership annotations without breaking the single-owner policy.

$$
\begin{array}{rl}
E^l = & \ldots \mid \texttt{check}_l^{\bar{k}}(E^{l_o},v) \mid \texttt{check}_h^{\bar{k}}(E^l,v) \mid \texttt{try}_l^{\bar{k}}(E^{l_o},v) \\
 & \mid \texttt{try}_h^{\bar{k}}(E^l,v) \mid \texttt{exercise}(E^l) \mid \texttt{waive}(E^l) \\
 & \mid \texttt{M}_j^{l,k}(c,E^{l_o}) \mid \texttt{M}_j^{h,k}(c,E^l) \mid {}^*\texttt{M}^{l,k}(E^{l_o}) \mid {}^*\texttt{M}^{h,k}(E^l) \\
 & \mid \texttt{O}_j^{l,k}(c,E^{l_o}) \mid \texttt{O}_j^{h,k}(c,E^l) \mid {}^*\texttt{O}^{l,k}(E^{l_o}) \mid {}^*\texttt{O}^{h,k}(E^l) \\
 & \mid \texttt{W}^{l,k}(E^{l_o}) \mid \texttt{W}^{h,k}(E^l) \\
E^{l_o} = & \ldots \mid \texttt{exercise}(E^{l_o}) \mid \texttt{waive}(E^{l_o})
\end{array}
$$

Figure 10: *OCPCF: Evaluation contexts

The guards, together with the `check` and `try` constructs, introduce points where ownership changes. Hence, they affect the ownership of the hole of evaluation contexts changes. Figure 10 defines the evaluation contexts of *OCPCF as an extension of those of CPCF.

Before we present the semantics of *OCPCF, we need to add another case to the special contract substitution function so that it handles option contracts:

$$
\{e/^c x\}\lfloor \texttt{option}(c,|e'|^k)\rfloor^{\bar{l}} = \lfloor \texttt{option}(\{e/^c x\}c, \{|e|^k/x\}|e'|^k)\rfloor^{\bar{l}}
$$

The reduction semantics for *OCPCF adopts some rules from the semantics of CPCF and adds a few. Figures 11, 12 and 13 show the rules that replace or add to those of CPCF.

Rules for `check` and `try` are similar to those of OCPCF except that now they come with ownership annotations.

When a stack of waived guards holds a basic value, the reduction eliminates the guards and delivers the value after removing all ownership annotations. After all, basic values are safe for the context to absorb.

Applying a function wrapped in a stack of waived guards reduces to an application wrapped with the same guards. The argument is wrapped with a reversed stack of guards plus ownership annotations. The labels on the guards for the argument are swapped just as for monitors for function contracts. An application of a stack of option guards reduces to an application of a homomorphic waived guard. The waived guard marks the function as a foreign value but does not add any contract-related constraints on its use.

Figure 12 displays the reduction rules for `exercise` and `waive`. They are in spirit the same as the corresponding rules of OCPCF with two differences: they operate on stacks of guards rather than a single guard and `waive` does not release the value that resides in a stack of option guards but instead turns the option guards into waived ones. Other than that, the rules transform guards in a way that is analogous to the rules of OCPCF but leave ownership annotations intact.

Figure 13 presents the complex rules for manipulating stacks of monitor guards. These rules also cover the cases where there is just a single monitor around a value instead

$E^l[\cdots]$	$\rightsquigarrow$	$E^l[\cdots]$
$\texttt{check}_j^{\bar{k}}(\|\|\texttt{tt}\|\|^j,v)$	.	v
$\texttt{check}_j^{\bar{k}}(\|\|\texttt{ff}\|\|^j,v)$	.	$\texttt{error}_j^{\bar{k}}$
$\texttt{try}_j^{\bar{k}}(\|\|\texttt{tt}\|\|^j,v)$	.	v
$\texttt{try}_j^{\bar{k}}(\|\|\texttt{ff}\|\|^j,v)$	.	$\texttt{error}_j^{\bar{k}}$

wg . $\texttt{b}$
if $wg = \texttt{W}^{k_1,l_1}(\ldots||\texttt{W}^{k_n,l_n}(||\texttt{W}^{m,p}(||\texttt{b}||^r)||^h)||^{q_n}\ldots)$

$||wgv||^l\ ||v||^l$. $||wg||^l$
if $||wgv||^l = ||\texttt{W}^{k_1,l_1}(\ldots||\texttt{W}^{k_n,l_n}(||\texttt{W}^{m,p}(v')||^h)||^{q_n}\ldots)||^{q_1}$
where
$||wg||^l = ||\texttt{W}^{k_1,l_1}(\ldots||\texttt{W}^{k_n,l_n}(||\texttt{W}^{m,p}(v'\ wgv')||^h)||^{q_n}\ldots)||^{q_1}$
and $wgv' = \texttt{W}^{p,m}(||\texttt{W}^{l_n,k_n}(||\ldots\texttt{W}^{l_1,k_1}(|||v|^l||^{q_1})\ldots||^{q_n})||^h)$

$||ogv||^l\ ||v||^l$. $||wgv||^l\ ||v||^l$
if $||ogv||^l = ||{}^*\texttt{O}^{k_1,l_1}(\ldots||{}^*\texttt{O}^{k_n,l_n}(||\texttt{O}_r^{m,p}(c,v)||^s)||^{q_n}\ldots)||^{q_1}$
and where
$||wgv||^l = ||\texttt{W}^{k_1,l_1}(\ldots||\texttt{W}^{k_n,l_n}(||\texttt{W}^{m,p}(v')||^s)||^{q_n}\ldots)||^{q_1}$

Figure 11: *OCPCF: Annotated semantics (part 1)

$$\begin{array}{lll}
\underline{E^{l}[\cdots]} & \rightsquigarrow & \underline{E^{l}[\cdots]} \\
\mathtt{exercise}(||v||^{l}) & . & ||v||^{l} \quad \text{if } v \notin \mathit{OGV} \\[1em]
\mathtt{exercise}(||\mathit{ogv}||^{l}) & . & \mathit{mgv} \\
\multicolumn{3}{l}{\text{if } ||\mathit{ogv}||^{l} = ||{}^{*}\mathtt{O}^{k_1,l_1}(\ldots||{}^{*}\mathtt{O}^{k_n,l_n}(||\mathtt{O}_{r}^{m,p}(c,v)||^{s})||^{q_n}\ldots)||^{q_1}} \\
\multicolumn{3}{l}{\text{and where}} \\
\multicolumn{3}{l}{\quad ||\mathit{mgv}||^{l} = ||{}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{r}^{m,p}(c,v)||^{s})||^{q_n}\ldots)||^{q_1}} \\[1em]
\mathtt{waive}(||v||^{l}) & . & ||v||^{l} \quad \text{if } v \notin \mathit{OGV} \\[1em]
\mathtt{waive}(||\mathit{ogv}||^{l}) & . & \mathit{wgv} \\
\multicolumn{3}{l}{\text{if } ||\mathit{ogv}||^{l} = ||{}^{*}\mathtt{O}^{k_1,l_1}(\ldots||{}^{*}\mathtt{O}^{k_n,l_n}(||\mathtt{O}_{r}^{m,p}(c,v)||^{s})||^{q_n}\ldots)||^{q_1}} \\
\multicolumn{3}{l}{\text{and where}} \\
\multicolumn{3}{l}{\quad ||\mathit{wgv}||^{l} = ||\mathtt{W}^{k_1,l_1}(\ldots||\mathtt{W}^{k_n,l_n}(||\mathtt{W}^{m,p}(v)||^{s})||^{q_n}\ldots)||^{q_1}}
\end{array}$$

Figure 12: *OCPCF: Annotated semantics (part 2)

of a stack of star-monitors. Thus they subsume the corresponding rules of CPCF.

If the bottom of the stack is a guard for a flat contract on a basic value, the stack reduces to a `check` where the blame labels are all the labels in server position on the stack of the guards. If the check fails, the contract system blames all the components that accepted responsibility for that value.

If the bottom of the stack is a guard for an option contract, the stack reduces to a `try` where the test term applies the tester of the contract to the stack of monitors. To get the correct *indy* semantics, the rule replaces the client label of the top of the stack with j. If the test succeeds, `try` returns an option guard like the original one but without the option around the contract; else it raises a contract error blaming all the labels in server position on the stack of the guards.

An application of a stack of monitor guards depends on the value at the bottom of the stack. If it is a stack of option guards, the reduction activates the option by turning the option guards into corresponding monitors and then performs the application. If not, the application is similar to that of waived guards. The difference is that the rule also decomposes the function contract at the bottom of the stack and uses the pre-condition as the contract at the bottom of the argument stack and the post-condition as the contract at the bottom of the application stack. Furthermore, the stack of guards substituted in the post-condition uses the contract label, namely j, as the client label on the top star-guard.

The rules for stacks of monitors for `transfer` contracts are like those of OCPCF. Accepting a stack of option guards ogv results in a star-option stack of guards around ogv that uses the same labels as the stack of monitors for the `transfer` contract. All other value are passed through.

Before we prove complete monitoring for *OCPCF, we go back to the example e_3 of section 6 and examine how its behavior changes under the semantics of our new model. We omit obligation annotations, because they do not affect computation, and we focus on well-formed ownership annotations and guards. In terms of the syntax of *OCPCF, the example's terms become:

$$\begin{array}{lll}
e_1 & = & |\mathtt{M}_{j_1}^{s,k_1}(c, |\mathit{deriv}|^{s}) f|^{k_1} \\
e_3 & = & |\mathtt{exercise}(\mathtt{M}_{j_2}^{k_1,k_3}(\mathtt{transfer}, e_1))\ 0|^{k_3}
\end{array}$$

$$\begin{array}{lll}
\underline{E^{l}[\cdots]} & \rightsquigarrow & \underline{E^{l}[\cdots]} \\
\mathit{mg} & . & \mathtt{check}_{j}^{k_1\ldots k_n m}(e\ \mathtt{b}, \mathtt{b}) \\
\multicolumn{3}{l}{\text{if } \mathit{mg} =} \\
\multicolumn{3}{l}{\quad {}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(\lfloor \mathtt{flat}(e) \rfloor^{\bar{l}}, ||\mathtt{b}||^{r})||^{k})||^{q_n}\ldots)} \\[1em]
\mathit{mg} & . & \mathtt{try}_{j}^{k_1\ldots k_n m}(e\ \mathit{mg}', \mathit{og}) \\
\multicolumn{3}{l}{\text{if } \mathit{mg} =} \\
\multicolumn{3}{l}{\quad {}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(\lfloor \mathtt{option}(c,e) \rfloor^{\bar{l}}, v)||^{k})||^{q_n}\ldots)} \\
\multicolumn{3}{l}{\text{and where } \mathit{mg}' = {}^{*}\mathtt{M}^{k_1,j}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(c,v)||^{k})||^{q_n}\ldots)} \\
\multicolumn{3}{l}{\text{and } \mathit{og} = {}^{*}\mathtt{O}^{k_1,l_1}(\ldots||{}^{*}\mathtt{O}^{k_n,l_n}(||\mathtt{O}_{j}^{m,p}(c,v)||^{k})||^{q_n}\ldots)} \\[1em]
||\mathit{mgv}||^{l}\ ||v||^{l} & . & ||\mathit{mg}||^{l} \\
\multicolumn{3}{l}{\text{if } ||\mathit{mgv}||^{l} =} \\
\multicolumn{3}{l}{\quad ||{}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(c_1 \overset{d}{\mapsto} (\lambda x.c_2), v')||^{h})||^{q_n}\ldots)||^{q_1}} \\
\multicolumn{3}{l}{\text{and where } ||\mathit{mg}||^{l} =} \\
\multicolumn{3}{l}{||{}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(\{\mathit{mg}''/^{c}x\}c_2, v'\ \mathit{mg}')||^{h})||^{q_n}\ldots)||^{q_1}} \\
\multicolumn{3}{l}{\text{and } \ \mathit{mg}' = {}^{*}\mathtt{M}^{p,m}(||{}^{*}\mathtt{M}^{l_n,k_n}(||\ldots\mathtt{M}_{j}^{l_1,k_1}(c_1, |||v|^{l}||^{q_1})\ldots||^{q_n})||^{h})} \\
\multicolumn{3}{l}{\text{and } \mathit{mg}'' = {}^{*}\mathtt{M}^{p,j}(||{}^{*}\mathtt{M}^{l_n,k_n}(||\ldots\mathtt{M}_{j}^{l_1,k_1}(c_1, |||v|^{l}||^{q_1})\ldots||^{q_n})||^{h})} \\[1em]
\mathit{mg} & . & \mathtt{W}^{k_1,l_1}(\ldots||\mathtt{W}^{k_n,l_n}(||\mathtt{W}^{m,p}(v)||^{k})||^{q_n}\ldots) \\
\multicolumn{3}{l}{\text{if } \mathit{mg} = {}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(\mathtt{transfer}, v)||^{k})||^{q_n}\ldots)} \\
\multicolumn{3}{l}{\text{and } v \notin \mathit{OGV}} \\[1em]
\mathit{mg} & . & \mathit{og}' \\
\multicolumn{3}{l}{\text{if } \mathit{mg} = {}^{*}\mathtt{M}^{k_1,l_1}(\ldots||{}^{*}\mathtt{M}^{k_n,l_n}(||\mathtt{M}_{j}^{m,p}(\mathtt{transfer}, \mathit{ogv})||^{k})||^{q_n}\ldots)} \\
\multicolumn{3}{l}{\text{where } \mathit{ogv} = ||{}^{*}\mathtt{O}^{k_1',l_1'}(\ldots||{}^{*}\mathtt{O}^{k_n',l_n'}(||\mathtt{O}_{j'}^{m',p'}(c,v)||^{k'})||^{q_n'}\ldots)||^{q_1}} \\
\multicolumn{3}{l}{\text{and } \mathit{og}' = {}^{*}\mathtt{O}^{k_1',l_1'}(\ldots||{}^{*}\mathtt{O}^{k_n',l_n'}(||{}^{*}\mathtt{O}^{m',p'}(\mathit{ogv})||^{k})||^{q_n}\ldots)}
\end{array}$$

Figure 13: *OCPCF: Annotated semantics (part 3)

The execution of e_3 leads to an application of a stack of waived guards:

$$\begin{array}{l}
e_3 \rightsquigarrow^{*} |\mathtt{exercise}(\mathtt{M}_{j_2}^{k_1,k_3}(\mathtt{transfer}, |\mathtt{O}_{j_1}^{s,k_1}(c_2', |f'|^{s})|^{k_1}))\ 0|^{k_3} \\
\quad \rightsquigarrow^{*} |\mathtt{exercise}({}^{*}\mathtt{O}^{k_1,k_3}(|\mathtt{O}_{j_1}^{s,k_1}(c_2', |f'|^{s})|^{k_1}))\ 0|^{k_3} \\
\quad \rightsquigarrow^{*} |(|{}^{*}\mathtt{M}^{k_1,k_3}(|\mathtt{M}_{j_1}^{s,k_1}(c_2', |f'|^{s})|^{k_1})|^{k_2})\ 0|^{k_3}
\end{array}$$

As in section 6, e_3 reduces to a contract error $\mathtt{error}_{j_1}^{k_3,k_1}$. Notice how the ownership annotations remain well-formed during evaluation. This is the key insight that we are going to use to prove that *OCPCF is well-formed.

A.3 Complete Monitoring for *OCPCF

The definition of complete monitoring for *OCPCF is similar to that for CPCF except that the case for failures of flat contracts is slightly different and there is an additional case for failed option contract tests.

Definition 2 (Complete Monitoring for *OCPCF). *A reduction relation $\hookrightarrow$ is a* complete monitor *for *OCPCF if for all terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- *$e_0 \hookrightarrow^* v$,*
- *for all e_1 such that $e_0 \hookrightarrow^* e_1$ there exists $e_1 \hookrightarrow e_2$,*
- *$e_0 \hookrightarrow^* e_1 \hookrightarrow^* \texttt{error}^{\bar{h}}_j$ and there is at least an e_1 such that*
  - *$e_1 = E^l[{}^*\texttt{M}^{l_2,l_1}(\ldots||{}^*\texttt{M}^{l_{n+1},l_n}(||\texttt{M}^{k,l_{n+1}}_j(c,v)||^{l_{n+1}})||^{l_n}\ldots)]$*

  and for all such terms e_1, $v = |v_1|^k$, $\bar{h} = l_2 \ldots l_{n+1} k$, $k \in \bar{l}$, and $c = \lfloor \texttt{flat}(e) \rfloor^{\bar{l}}$ or $\lfloor \texttt{option}(c', e) \rfloor^{\bar{l}}$.

First, the definition requires the absence of stuck states due to any violation of the single-owner policy. Second, the cases for contract failures guarantee that if the contract system raises a blame error, then the blamed components are all the components that accepted responsibility for the monitor guards. In addition, the definition requires that the flat or option contract at the bottom of the stack is amongst the obligations of one of the blamed components, specifically of the first component that took responsibility for the value.

The proof of complete monitoring for *OCPCF follows the proof of Dimoulas et al. [7]. We develop a subject that generalizes well-formedness and use a progress-and-preservation subject reduction technique. The additional well-formedness rules cover the terms of the evaluation syntax of *OCPCF. Especially for stacks of guards they make sure that appropriate ownership annotations are present in between each guard layer. Moreover the rules for monitors and stacks of guards use a generalized notion of well-formedness to inspect terms inside the bottom guard of the stack of guards. This extention is needed because beta-reduction introduces terms that temporarily deviate from well-formedness [6]. The generalized well-formedness permits us to handle these temporarily out-of-order terms without disturbing the single-ownership and obligations principles. Some modifications are also necessarry for the rules for well-formed flat and option contracts. Since monitors for `transfer` monitors dynamically change the components that are responsible for meeting a contract, statically determined obligations cannot provide an accurate prediction of the blamable parties. Nevertheless, obligations can still tell us that the initially responsible component for a contract is amongst the components that get blamed if the contract fails. The proof technique and corresponding definitions are similar to that of Dimoulas [4, ch. 6] and we ommit them here due to lack of space.

With the extended definition for well-formedness in hand, we prove that $\rightsquigarrow$ defines a complete monitor.

Theorem 3. *$\rightsquigarrow$ is a complete monitor for *OCPCF.*

A.3.1 Complete Monitoring for OCPCF

The introduction of *OCPCF is a detour to prove complete monitoring for OCPCF. To transfer the result to OCPCF, we prove a bisimulation theorem between the two languages. Figure 14 specifies the relation between OCPCF and *OCPCF terms.

$\boxed{e \sim e'}$

$$\frac{e \sim^{ctx} e'}{e \sim |e'|^l} \qquad \frac{e \sim^{ctx} e'}{e \sim \texttt{W}^{k,l}(e')}$$

$$\frac{e \sim^{ctx} e' \qquad c \sim^{ctx} c'}{\begin{array}{l}\texttt{M}^{k_1 \ldots k_n k, l_1 \ldots l_n l}_j(c,e) \sim \\ \qquad {}^*\texttt{M}^{k_1,l_1}(\ldots||{}^*\texttt{M}^{k_n,l_n}(||\texttt{M}^{k,l}_j(c',e')||^l)||^{l_n}\ldots)\end{array}}$$

$$\frac{e \sim^{ctx} e' \qquad c \sim^{ctx} c'}{\begin{array}{l}\texttt{O}^{k_1 \ldots k_n k, l_1 \ldots l_n l}_j(c,e) \sim \\ \qquad {}^*\texttt{O}^{k_1,l_1}(\ldots||{}^*\texttt{O}^{k_n,l_n}(||\texttt{O}^{k,l}_j(c',e')||^l)||^{l_n}\ldots)\end{array}}$$

$\boxed{c \sim c'}$

$$\frac{c \sim c' \qquad e \sim^{ctx} e' \qquad v \sim^{ctx} v'}{\texttt{option}(c,e) \sim \lfloor \texttt{option}(c',e') \rfloor^{\bar{l}}}$$

$$\frac{e \sim^{ctx} e'}{\texttt{flat}(e) \sim \lfloor \texttt{flat}(e') \rfloor^{\bar{l}}}$$

Figure 14: *OCPCF-*OCPCF* bisimulation

Theorem 4. *Let e a term of OCPCF and e^* a term of *OCPCF. If $\varnothing; l_o \Vdash e^*$ and $e \sim^{ctx} e^*$ then*

- *$e \rightarrow^* v$ iff $e^* \rightsquigarrow^* v^*$ and $v \sim^{ctx} v^*$ and,*
- *$e \rightarrow^* \texttt{error}^{\bar{k}}_j$ iff $e^* \rightsquigarrow^* \texttt{error}^{\bar{k}}_j$.*