Complete Monitors for Behavioral Contracts*

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen

Northeastern University, Boston, Massachusetts, USA
{chrdimo,samth,matthias}@ccs.neu.edu

Abstract. A behavioral contract in a higher-order language may invoke methods of unknown objects. Although this expressive power allows programmers to formulate sophisticated contracts, it also poses a problem for language designers. Indeed, two distinct semantics have emerged for such method calls, dubbed *lax* and *picky*. While *lax* fails to protect components in certain scenarios, *picky* may blame an uninvolved party for a contract violation.

In this paper, we present *complete monitoring* as the fundamental correctness criterion for contract systems. It demands correct blame assignment as well as complete monitoring of all channels of communication between components. According to this criterion, *lax* and *picky* are indeed incorrect ways to monitor contracts. A third semantics, dubbed *indy*, emerges as the only correct variant.

Keywords: higher-order programming, behavioral contracts, contract checking

1 Blame Correctness Is Not Enough

Programmers embrace Eiffel-style contracts [7] because they can write them in the language itself and they understand them as executable boolean expressions. Conventionally, programmers use contracts to supplement method signatures with relatively simple conditions: a *non-empty* list expected here; a *positive* number promised there; a field whose value is always a string *of a specific length*. They also understand that the contract system checks these conditions when a method is called and/or when a call returns. If the condition evaluates to false, it is either the method's or the caller's fault.

In a higher-order contract system [4], such as the one for Racket [5], programmers can also specify conditions on functions and objects. Here is an example:

```
;; contract for the derivative function
;; for some natural number n and reals δ, ε:
(->d ([f (0<real<1? . -> . 0<real<1?)])
     (fp (0<real<1? . -> . real?))
     #:post-cond
     (for/and ([i (in-range 0 n)])
       (define x (random-number))
       (define slope (/ (- (f (- x ε)) (f (+ x ε))) (* 2 ε)))
       (<= (abs (- slope (fp x))) δ)))
```

* Supported in part by AFOSR grant FA9550-09-1-0110 and the DARPA CRASH program.

It specifies a computational differentiation operator `d/dx` for functions on the unit interval. The specification promises to map a function `f` to a function `fp` that computes a number close to the slope of `f` at `x`.

Due to Rice's theorem, it is impossible to check such contracts directly. Instead contracted functions are wrapped in a monitor that checks the promised property every time it is used during the remainder of the computation. Since such a use may take place after the function returns in a third-party component, the naive understanding of first-order contracts and blame assignment does not apply here.

Thus higher-order values inject several new elements into the realm of contracts. First, it is now important to explicitly think of components as contract parties. These parties agree on monitoring properties for values that flow back and forth across component boundaries. Second, blame assignment requires tracking of contracts and parties because the producer is not necessarily the last function called. In the above example, `d/dx` returns a higher-order value with the requirement to call it on reals between 0 and 1, but a call involving some negative real may take place much later. Third, contracts are no longer predicates on flat values but may involve calls to unknown functions. For instance, the post-condition for `d/dx` tests whether the result `fp` satisfies the desired "slope property" for `f` on some randomly chosen numbers.

Calls to unknown functions pose a challenge for contract designers. To this day it is unclear how a correct contract system should deal with such calls. Take a second look at the above example. Its contract uses `random-number`, which, as it turns out, may produce complex numbers in Racket. Depending on the semantics of contracts, the example behaves in one of three ways:

1. Findler and Felleisen [4] consider contracts a part of the specification and thus "correct by definition." According to their *lax* semantics, the post condition passes the random number to `f` and `fp`. If these functions handle complex numbers, fine; otherwise, execution fails in an unpredictable manner.
2. Blume and McAllester [1] propose an alternative *picky* semantics. According to their proposal, the contracts for `f` and `fp` prohibits their application to complex numbers, and their reuse catches contract-internal problems [6].
3. In prior work [3], we show, however, that *picky* may blame the wrong party for a contract violations and may thus point programmers in the wrong direction in their search for bugs. A variant of *picky*, dubbed *indy*, is shown to be *blame correct*.

Sadly, blame correctness cannot differentiate between *lax* and *indy*. Since *lax* may trigger crashes in the presence of precise specifications, it is clearly not correct. Worse, blame correctness admits contract systems that ignore contracts completely.

We conjecture that a programmer would like the guarantee that the values produced by their components are never used in violation to the interface specifications and, conversely, that their components are not handed values that do not live up to the promises of the specifications. In response, we present a generalization of blame correctness, called *complete monitoring*. We take the ownership-and-obligation framework of blame correctness and extend it so that a component may not manipulate values that it does not create or that have not been transferred from other components via a—possibly vacuous—contract. In short, a complete contract system monitors *all* value flows across component boundaries.

The next section introduces our technical framework, which we exploit to to present informally *complete monitors* in section 3 and subsequently define them formally in section 4. This latter section also presents our main result, the complete monitoring theorem. Sections 5 and 6 illustrate the additional benefits of complete monitors with two examples. Finally, the last section discusses related work.

2 Beyond Blame Correctness

CPCF [2, 3] extends a conventional, typed and higher-order functional language, with contracts for base values and first-class functions:

Types	$\tau =$	$o \mid \tau \to \tau \mid \mathsf{con}(\tau)$
	$o =$	$\mathsf{I} \mid \mathsf{B}$
Contracts	$\kappa =$	$\mathtt{flat}(e) \mid \kappa \mapsto \kappa \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa)$
Terms	$e =$	$v \mid x \mid e\,e \mid \mu x{:}\tau.e \mid e+e \mid e-e \mid e \wedge e \mid e \vee e$
		$\mid \mathtt{zero?}(e) \mid \mathtt{if}\ e\ e\ e \mid \mathtt{mon}_l^{l,l}(\kappa, e)$
Values	$v =$	$\mathtt{c} \mid \lambda x{:}\tau.e$
Base Values	$\mathtt{c} =$	$0 \mid 1 \mid -1 \mid \ldots \mid \mathtt{tt} \mid \mathtt{ff}$

Contracts for flat values, $\mathtt{flat}(e)$, employ predicates that may use the full expressive power of CPCF. Contracts for functions, $\kappa_1 \mapsto \kappa_2$, consist of a pre-condition contract κ_1 for the argument to the function and a post-condition contract κ_2 for its result. Dependent function contracts, $\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$, bind the argument to the function to x and make it visible in κ_2. They thus express how the result may depend on the argument.

A contract κ can be attached to a term e using the monitor construct $\mathtt{mon}_j^{k,l}(\kappa, e)$. Monitors carry three labels: k, l and j.[1] Labels are identifiers for the high-level components that make up a program. A monitor splits a program into three components, dubbed the *contract parties*: a server named k, a client named l, and a contract named j, which may coincide with k or l in a programming language. Intuitively a monitor makes sure that any interaction between the server module and the client module is in accordance with the contract. In CPCF, e plays the role of the server module and the context of $\mathtt{mon}_j^{k,l}(\kappa, e)$ the role of the client. The contract κ is what they agree on concerning the exchange of values.

Component labels play an important role in case a contract failure is detected during contract checking. They are used to pinpoint the contract violator. CPCF syntax is extended with intermediate terms for contract checking:

$$e = \ldots \mid \mathtt{error}_l^l \mid \mathtt{check}_l^l(e, v)$$

Findler and Felleisen [4] show that the above constructs are sufficient to build a semantics for checking higher-order contracts. However, since the goal of our investigation is to verify that the contract system obliges values to meet their specifications as they flow from one component into another, we add the idea of *ownership* for terms and values to the semantics. We use it to keep track of value migration. Ownership of a term e

[1] The labels correspond to source locations or component names in an implementation.

by a component l is expressed with the ownership annotation $|e|^l$. Ownership captures formally that a component owns a value (term) if it can affect or manipulate its flow. In a reduction semantics, the flow of values is modeled via substitution (β_v). Hence, our semantics must attach a new ownership annotation to every value that is substituted for a variable. In CPCF, this means we must treat every function application as a potential boundary crossing. Thus when an annotated value occurs in a function body, its occurrence signals the presence of a *foreign* value and marks a boundary. In sum, the initial owner of a value is its creator but as the value flows through function application, it accumulates more owners, one for each boundary it crosses with the top-most to be the most recent owner.

In addition to ownership, we ensure correct blame assignment by keeping track not only of the owner of each value but also the responsible party for the specifications that are checked against a value upon a component boundary crossing. The *obligations* of a contract party l are the set of ground-type sub-contracts of a contract κ for which l is responsible. Intuitively, the contract system should not blame a party if the party's obligations are satisfied. For a function contract we know that the client is responsible for the pre-condition and the server responsible for the post-condition. For any flat contract, the server is responsible. Generalizing this approach gives us a way to determine the responsible parties for each flat sub-contract of a given contract using type theory terminology: the server is responsible for all flat contracts in positive positions and the client is responsible for all flat contracts in negative positions. CPCF turns obligations into explicit annotations on flat contracts. Thus $\lfloor \texttt{flat}(e) \rfloor^{\overline{l}}$ denotes that the set of parties $\overline{l}$ is responsible for the given flat contract.

Here is CPCF with the annotations for ownership and obligations:

$$\begin{array}{lll} \textbf{Contracts} & \kappa = \lfloor \texttt{flat}(e) \rfloor^{\overline{l}} \mid \kappa \mapsto \kappa \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa) \\ \textbf{Terms} & e = \ldots \mid |e|^l \\ \textbf{Values} & v = \ldots \mid |v|^l \end{array}$$

With obligations and ownership we reify dynamic boundary crossings via syntactic annotations and the specifications that need to be checked. If this *independent* instrumentation coincides in the source code with the monitors and blame labels that the contract system utilizes and the reductions preserve this property, we know that the contract system monitors all communication between components.

Synchronization of ownership and monitors means that the owner of the context of a monitor is the same as the label at the client position of the monitor and the owner of the guarded term is the same as the label in the server position of the monitor. Due to the presence of run-time terms, there is one more case where the labels of the contract system have to agree with ownership. CPCF uses $\texttt{check}^k_j(e,v)$ to check flat contracts. This implies that the checking code e is owned by the contract party j. We consider this construct as another point where values change components and the owner of e must be the contract party.

Synchronization of obligations and blame labels means that the label at the server position of a monitor is a member of the obligations annotations on positive ground-type subcontracts of the monitor's contract and similarly for the client label and negative ground-type subcontracts.

In principle, ownership and obligations could be just observers of the reduction sequence that do not affect evaluation. However, to prove that the contract system allows values to migrate from one component to another only when they are under its control, we use ownership to impose restrictions on value flows between components. We enforce a *single owner* policy that disallows mixing terms with different owners. Instead our reduction relation ensures that foreign values within a component are wrapped in contract checks or that the contract system has completely verified all (flat) specifications during the absorption of a foreign value into a component.

Reduction Rules

$E^l[\cdots]$	$\xrightarrow{m} E^l[\cdots]$	
$\|\|\mathtt{n}_1\|\|^l + \|\|\mathtt{n}_2\|\|^l$	$\mathtt{n}$	where $n_1 + n_2 = n$
$\|\|\mathtt{n}_1\|\|^l - \|\|\mathtt{n}_2\|\|^l$	$\mathtt{n}$	where $n_1 - n_2 = n$
$\mathtt{zero?}(\|\|\mathtt{0}\|\|^l)$	$\mathtt{tt}$	
$\mathtt{zero?}(\|\|\mathtt{n}\|\|^l)$	$\mathtt{ff}$	if $n \neq 0$
$\|\|v_1\|\|^l \wedge \|\|v_2\|\|^l$	v	where $v_1 \wedge v_2 = v$
$\|\|v_1\|\|^l \vee \|\|v_2\|\|^l$	v	where $v_1 \vee v_2 = v$
$\mathtt{if}\ \|\|\mathtt{tt}\|\|^l\ e_1\ e_2$	e_1	
$\mathtt{if}\ \|\|\mathtt{ff}\|\|^l\ e_1\ e_2$	e_2	
$\|\|\lambda x.e\|\|^l\ \|\|v\|\|^l$	$\|\{\|v\|^l/x\}e\|^l$	
$\mu x.e$	$\{\|\mu x.e\|^l/x\}e$	
$\mathtt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)$	$\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))$	
$\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{l'}, \|\|\mathtt{c}\|\|^{l'})$	$\mathtt{check}_j^k(e\ \mathtt{c}, \mathtt{c})$	
$\mathtt{check}_j^k(\|\|\mathtt{tt}\|\|^j, v)$	v	
$\mathtt{check}_j^k(\|\|\mathtt{ff}\|\|^j, v)$	$\mathtt{error}_j^k$	
$E^l[\mathtt{error}_j^k]$	$\xrightarrow{m} \mathtt{error}_j^k$	

Eval. Contexts

$$
\begin{array}{rcl}
E^l & = & E^l\,e \mid v\,E^l \mid E^l + e \mid v + E^l \mid E^l - e \mid v - E^l \mid E^l \wedge e \\
& \mid & v \wedge E^l \mid E^l \vee e \mid v \vee E^l \mid \mathtt{zero?}(E^l) \mid \mathtt{if}\ E^l\ e\ e \\
& \mid & \mathtt{mon}_j^{l,k}(\kappa, E^{l_o}) \mid \mathtt{mon}_j^{l',k}(\kappa, E^l) \mid |E^{l_o}|^l \mid |E^l|^{l'} \\
& \mid & \mathtt{check}_l^k(E^{l_o}, v) \mid \mathtt{check}_{l'}^k(E^l, v) \\
E^{l_o} & = & [\,] \mid E^{l_o}\,e \mid v\,E^{l_o} \mid E^{l_o} + e \mid v + E^{l_o} \mid E^{l_o} - e \\
& \mid & v - E^{l_o} \mid E^{l_o} \wedge e \mid v \wedge E^{l_o} \mid E^{l_o} \vee e \mid v \vee E^{l_o} \\
& \mid & \mathtt{zero?}(E^{l_o}) \mid \mathtt{if}\ E^{l_o}\ e\ e
\end{array}
$$

Fig. 1. CPCF semantics enforces the single-owner policy

To implement this policy, we require all terms in a *redex* to have a single owner. Put differently, our semantics does not perform operations on values that have ownership annotations with different owners. We use $||e||^l$ to denote that e may have no ownership annotations but if it has one then the owner label is l for all such annotations:

$$||e||^l = |...|e|^l...|^l \qquad \text{where for all labels } k \text{ and terms } e',\ e \neq |e'|^k.$$

The *single owner* policy becomes critical for defining the reduction semantics for CPCF. A component should be able to perform an operation if and only if it is the owner of all the arguments of the redex. This implies that either the arguments inherit their implicit ownership annotation from the context or that they come with an explicit ownership annotation that matches with the owner of the context. We model implicit ownership with *labeled evaluation contexts*; see figure 1.

The reduction relation of figure 1 implements the single owner policy by reducing redexes only if the label of the hole matches the owner of the pieces of the redex. For instance the rule for function application is more restrictive than the original rule for CPCF [3]. The latter allows the function and the argument to have different and multiple owners. In contrast, the new rule fires only if l, the owner of the component, is also the only owner of the function and the argument. The argument is substituted in the body of the function, annotated with the common owner so that it keeps its ownership annotation no matter where it lands in the function body. The context absorbs the body of the function, which thus obtains the context's ownership annotation. Since the function and the context have the same owner, however, the body of the function retains its original owner. This rationale explains all the rules, including the rules for monitors where the client label must be the same as the label of the context. When the reduction rules create new values, as in the case of primitives operators, the context becomes directly responsible for the new value and thus no additional ownership annotation is necessary. Finally, the $\texttt{check}_j^k(e,v)$ rules enable executing checking code e that originates from j inside l, the owner of the hole. Doing so ensures that the `check` term is treated as a component boundary and the result of e must be owned by j in order for `check` to reduce. If the check fails, and an error is raised and blames the initial owner k of v.

Values retain their owner as long as they move inside the same component. They change owner only when flat contract checking succeeds. When the check succeeds, the contract system gives permission to the surrounding component l to absorb $\mathtt{c}$, and $\mathtt{c}$ changes hands between k and l.

The reduction rules concerning monitors for dependent function contracts come in three flavors: $l(ax)$, $p(icky)$ and $i(ndy)$. Here are their formal definitions:

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \overset{l}{\to} E^l[\lambda x.\mathtt{mon}_j^{k,l}(\{x/^c x\}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))]$$

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \overset{p}{\to} E^l[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\{\mathtt{mon}_j^{l,k}(\kappa_1, x)/^c x\}\kappa_2}, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))]$$

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \overset{i}{\to} E^l[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\{\mathtt{mon}_j^{l,j}(\kappa_1, x)/^c x\}\kappa_2}, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))]$$

The intuition behind these rules is explained at the end of the section.

The reductions employ a special function $\{e/^c x\}\kappa_2$ for substituting a term e for x in a post-condition κ_2 of a dependent contract:

$$\begin{aligned}
\{e/^c x\}\lfloor \mathtt{flat}(|e'|^{l'})\rfloor^{\bar{l}} &= \lfloor \mathtt{flat}(\{|e|^{l'}/x\}|e'|^{l'})\rfloor^{\bar{l}} \\
\{e/^c x\}(\kappa_1 \mapsto \kappa_2) &= \{e/^c x\}\kappa_1 \mapsto \{e/^c x\}\kappa_2 \\
\{e/^c x\}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)) &= \{e/^c x\}\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2) \\
\{e/^c x\}(\kappa_1 \overset{d}{\mapsto} (\lambda y.\kappa_2)) &= \{e/^c x\}\kappa_1 \overset{d}{\mapsto} (\lambda y.\{e/^c x\}\kappa_2) \text{ where } x \neq y
\end{aligned}$$

The substitution in the post-condition implements a hidden application of $\lambda x.\kappa_2$ to v. The special substitution function makes sure that the argument is wrapped with an ownership annotation for the owner of $\lambda x.\kappa_2$, which is also the owner of the contract [3].

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

$$\frac{}{\Gamma; l \Vdash \mathtt{c}} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1\ e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2 \quad \Gamma; l \Vdash e_3}{\Gamma; l \Vdash \mathtt{if}\ e_1\ e_2\ e_3}$$

$$\frac{\Gamma; l \Vdash e_1}{\Gamma; l \Vdash \mathtt{zero?}(e_1)} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 + e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 - e_2}$$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \wedge e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \vee e_2} \qquad \frac{\Gamma \uplus \{x : l\}; l \Vdash e}{\Gamma; l \Vdash \lambda x.e}$$

$$\frac{\Gamma \uplus \{x : l\}; l \Vdash e}{\Gamma; l \Vdash \mu x.e} \qquad \frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash |e|^l} \qquad \frac{\Gamma(x) = l}{\Gamma; l \Vdash x} \qquad \frac{\begin{array}{c}\Gamma; k \Vdash e \qquad k \neq l \\ \Gamma; \{k\}; \{l\}; j \rhd \kappa\end{array}}{\Gamma; l \Vdash \mathtt{mon}_j^{k,l}(\kappa, |e|^k)}$$

Fig. 2. Well-formed source programs

As mentioned, ownership annotations and obligations may not appear at arbitrary places in a program. To ensure the correctness of these annotations, we use a static well-formedness judgment, $\Gamma; l \Vdash e$, for source programs e. The interesting cases in source syntax are the ones concerning variables, variable bindings, ownership annotations and contract monitors, and they appear at the bottom of figure 2. The occurrence of a free variable in a term is one of the ways foreign values can flow into a component. The environment Γ keeps track of the origin of values bound to variables. It records the owner of the spot where a binder for a variable is introduced. To ensure that components are free of foreign terms we force the single owner policy, i.e., ownership annotations inside a component must carry the same owner label as the component. We can embed foreign code in a component under the protection of the contract system, that is, a component can contain foreign terms as long as they are wrapped in a monitor annotation and they are explicitly marked as foreign terms with an appropriate ownership annotation. In such cases the client label on the monitor must match the owner of the surrounding component and the server label must coincide with the explicit ownership annotation on the guarded term. Note that this also allows the embedding of free variables as long as they are monitored. Furthermore, the rule forces the client and blame labels on monitors in the source code to be different to emphasize that monitors are used on the boundaries between different components in the source code.

The rule for well-formed monitors requires that the contract is well-formed, i.e., that obligations inside of a contract are properly attributed. Figure 3 shows the rules for well-formed contracts. The judgment $\Gamma; \bar{k}; \bar{l}; j \rhd \kappa$ has three label-related parts. The first, $\bar{k}$, includes the set of parties responsible for the flat contracts on positive positions in κ. The second, $\bar{l}$, corresponds to the parties responsible for the negative positions.

$\boxed{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa}$

$$\frac{\Gamma; j \Vdash e}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{flat}(|e|^j) \rfloor^{\overline{k}}} \qquad \frac{\begin{array}{c}\Gamma; \overline{l}; \overline{k}; j \rhd \kappa_1 \\ \Gamma; k; l; j \rhd \kappa_2\end{array}}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \mapsto \kappa_2} \qquad \frac{\begin{array}{c}\Gamma; \overline{l}; \overline{k} \cup \{j\}; j \rhd \kappa_1 \\ \Gamma \uplus \{x : j\}; k; l; j \rhd \kappa_2\end{array}}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)}$$

Fig. 3. Well-formed contracts

Finally, j is the owner of the contract code. The initial values of these labels are drawn from the monitor expression, and they are propagated by a structural traversal of κ to its pieces. The server label is initially the only member of the labels responsible for the positive pieces of the contracts while the client label is the only member of the parties responsible for the negative pieces. Also the contract label of the monitor is appointed as owner party of the contract's code. In the case of function contracts, the set of responsible parties are reversed in the pre-condition [4] and for dependent function contracts the contract party j is added to the set of labels responsible for the pre-condition. In the latter case, we record in the environment the variable x that binds the argument in the post-condition with the contract party as its owner. After all, x is a binder that belongs to the contract's code. Finally, for flat contracts the rules require that the obligation annotations on the contract coincide with the set of parties responsible for the positive pieces of the contract and that the party j is explicitly marked as the owner of the contract code.

Note: This semantics of CPCF differs from the semantics of our previous work. The main deviation is the introduction of the single owner policy. It helps us prove complete monitoring, a deep notion of correctness for a contract system that subsumes blame correctness.

In our previous result, ownership and obligations are used to verify that whenever a contract error is raised, its witness value is owned by the party that is blamed and that the party failed to satisfy one of its obligations. The *picky* semantics fails to live up to this standard [3]. The problem with *picky* is due to the way the semantics decorates the monitor that protects the argument in the post-condition of a dependent contract on a function f. More specifically, the monitor holds the server of f responsible for invalid uses of the argument inside the contract despite the fact that the server does not have control over the flow of values in the contract.

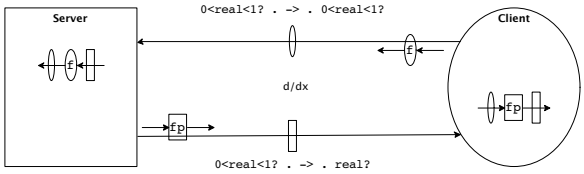
The *indy* semantics eliminates this shortcoming of *picky*. It treats the contract as a separate party that is responsible for the use of values that flow in the contract. Thus the semantics injects monitors that protect the argument and hold the contract itself responsible for any use of the argument in the post-condition of a dependent contract. The obligations of the contract party are the flat contracts in negative position of the pre-conditions of dependent contracts, which also explains why we use sets of labels for the obligation annotations. A flat contract can be part of the obligations of the contract party and, also, of the client or the server.

Our blame correctness criterion, though, is not strong enough to decide whether *lax* is preferable to *indy*, or vice versa, as both of them are blame correct. In fact it ad-

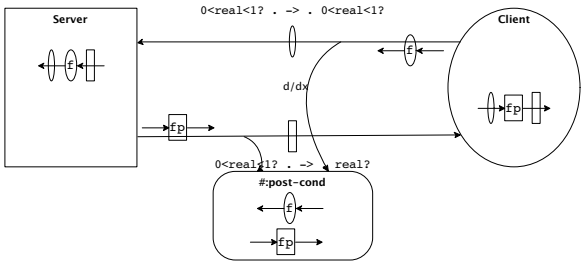
mits contract systems that permit uncontrolled flow of values between components. The problem lies with the way the original semantics of CPCF treats ownership. It allows for components to mix freely and for values to acquire multiple owners as they cross boundaries. For instance a term $|e|^l$ can show up without any restrictions as a sub-term of a term $|e'|^k$. Similarly a value $|\ldots|v|^{l_1}\ldots|^{l_n}$ comes with multiple owners as the annotations keep track of the whole history of migrations from one module to another. These annotations do not affect evaluation, however, because it ignores them and proceeds as if they are not there.

Our new semantics turn ownership into a computational device that is exploited to enforce the single owner policy. This change enables us to state when a contract system is a *complete monitor* for all specified properties. **End note**

Contracts without post-condition:



Lax contracts for post-conditions:



Picky contracts for post-conditions:

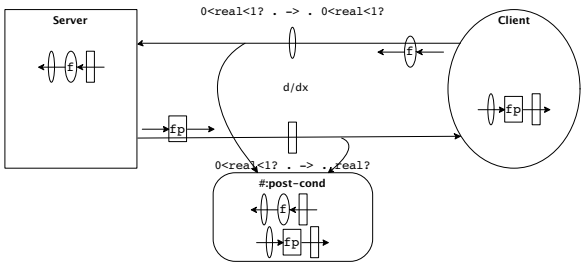


Fig. 4. Monitoring in pictures

3 Complete Monitors with Pictures

While we can use the CPCF model to articulate a formal criterion of monitoring completeness, it can also provide an intuitive understanding of the idea. In this section, we present three pictures of contract monitoring that employ some of the elements of the CPCF model and introduce complete monitors on this basis.

The first picture in figure 4 illustrates how the contract system monitors the contract of `d/dx` without its **`#:post-cond`** clause. The client owns `f` and applies `d/dx` to it. Pictorially, it ships `f` to the server over the `d/dx` channel. The contract system monitors the channel and attaches the appropriate pieces of the contract to `f`. Thus the server component receives a wrapped version of `f`. The wrapper checks that the argument and the result of any application to `f` are real numbers between 0 and 1. The result of the application, `fp`, returns to the client component via a similar channel.

Our pictures use shapes to express *ownership* of values. Thus `f` comes in an *ellipsis* to match the shape of the client and `fp` is in a *rectangle*, like the server component that creates the function. If the client were to pass `fp` back to the server to create the second derivative of `f`, the value would come in an ellipsis around the rectangle and the contract wrapper. In other words, wrapping shapes within shapes illustrates how the semantics uses ownership to keep track of a value's provenance.

Similarly, shapes on the input-output arrows mark *obligations*. In particular, the flat contracts that guard channels have the same shape as the component responsible for satisfying them. The characterization holds for both components and higher-order values that flow back and forth and receive wrappers.

In a graphical form, our picture suggests that if the initial program is *well-formed*, meaning it separates the client and the server component with a properly formed contract boundary, a complete contract system preserves a two-part invariant. The first part dictates that each value has the same shape as its origin and, if the origin differs from the current host component, then the contract system guards the value with contracts. The second part adds that the host component is responsible for meeting the pre-condition for the uses of the foreign value and the origin component is responsible for the post-condition.

Even though the invariant seems easy to maintain, adding back the **`#:post-cond`** clause shows that doing so poses subtle challenges. Concretely, a post condition clause consists of a piece of code and thus introduces a new component. In a real-world language such as Racket, this new component could exist within the server module, the client module, or as a third-party component all by itself [3, §2.3]. No matter where it exists, it hosts both `f` and `fp`, and this co-habitation is the source of all subtleties.

The new component connects to the `d/dx` channels with its own branch channels and can thus absorb the values from these channels. As the second picture of figure 4 shows, the *lax* semantics allows `f` and `fp` to enter the new component before they flow through the monitors—meaning no guards are attached to these new channels. Since `f` and `fp` have different owners, at least one of the values must be considered a foreign value and, as such, inhabits the component without the necessary guard.

In contrast to *lax*, *picky* protects these additional channels of communication, too. Figure 4 explains this idea with forks in the channels *behind* the monitors that protect the channels. Unfortunately, the obligations for the flat contracts in the **`#:post-cond`**

component do not agree with the second part of the completeness invariant. That is, at least one of the two values inhabits the new component as a foreign value but is protected by misshaped contracts.

As the next section shows, the third contract monitoring system, dubbed *indy*, addresses both parts of the invariant across the entire computation. Technically, our framework serves as an independent specification of the contract system and excludes scenarios such as the two above by halting computation when the single-owner policy breaks.

4 Complete Monitors Formally

The CPCF semantics enforces the single owner policy. If a redex does not respect it, the evaluation gets stuck. Since embeddings of foreign terms in a component are wrapped with contract monitors, such stuck states are evidence that a value has leaked from one component to another without the contract system's approval. If a contract system can eliminate all such stuck states and force programs to reduce to a value or to diverge or to raise a contract error, then the contract system insulates the components of the program and regulates exchanges of values between them. We call such a contract system a *complete monitor*.

Definition 1 (Complete Monitors for CPCF). *A contract semantics m specifies a* complete monitor *if for all well typed terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- $e_0 \xrightarrow{m}_* v$ *or,*
- *for all e_1 such that $e_0 \xrightarrow{m}_* e_1$ there exists e_2 such that $e_1 \xrightarrow{m} e_2$ or,*
- $e_0 \xrightarrow{m}_* e_1 \xrightarrow{m}_* \texttt{error}^k_j$ *and there is at least an e_1 of the form $E^l[\texttt{mon}^{k,l}_j(\lfloor \texttt{flat}(e) \rfloor^{\overline{l}}, v)]$ and for all such terms e_1, $v = |v_1|^k$ and $k \in \overline{l}$.*

Complete monitoring takes advantage of the ownership and obligation annotations to verify that well-formed programs do not get stuck. In addition if a contract error is raised indicating contract j failed, then for all checks of flat contracts from j, the owner k of the guarded value is identical to the server label on the contract monitor and the flat contract is part of the obligations of k. Clearly, this gives more guarantees than *blame correctness*.

At first glance *complete monitoring* appears too weak to establish the correctness of a contract system; it simply guarantees that when a value crosses a boundary, the contract system attaches *some* contract to it and that if a contract violation is detected, blame is assigned to the party that contributed the witness value. What complete monitoring does *not* require is that the contract system (1) attaches the *proper* contracts to migrating values and (2) checks flat contracts. Point 1 concerns the decomposition of compound contracts, i.e., a contract system must check the pieces of a compound value (for example, functions) with the proper pieces of their contracts (for example, domain and range). In CPCF, this property comes for free with type soundness which forces the proper distribution of compound contracts over their pieces to retain type safety. As for point 2, it is necessary to ensure that a contract system actually executes the application of the predicate to the witness value. Again, this is obvious in the case of CPCF and it is easy to check in general. In short, complete monitoring is the main ingredient

language designers must check if they wish to implement a correct contract system; the remaining properties can be validated by inspection.

We show now that *indy* is a *complete monitor* while *lax* is not. The proof of complete monitoring follows a subject reduction technique similar to those of type soundness [10]. The first subsection presents the construction of the subject, progress and preservation, and how these results imply completeness for *indy*. The second subsection presents how we construct a counter-example showing the incompleteness of *lax* and *picky*.

4.1 Indy Is a Complete Monitor

The judgments for well-formed terms and well-formed contracts imply both the single owner policy for components and the agreement between monitor labels, ownership and obligation annotations for monitors. They are too weak, however, for the proof because they do not cover all intermediate terms. The semantics of CPCF uses a superset of the source language of CPCF to deal with errors and contract checks. Moreover, evaluation constructs monitor terms that are not well-formed according to our rules. Fortunately this is only temporary; after some reduction steps, the terms become well-formed again.

In order to account for the extra intermediate terms, we generalize well-formedness for terms and contracts. The generalized judgment for well-formed terms in most cases is almost the same as the corresponding source code judgment. Figure 5 shows only the extra/modified rules.

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

$$\frac{}{\Gamma; l \Vdash \mathtt{error}^k_j} \qquad \frac{\Gamma; j \vDash e \quad \Gamma; l \Vdash v}{\Gamma; l \Vdash \mathtt{check}^k_j(e, v)} \qquad \frac{\Gamma; k \vDash e \quad \Gamma; \{k\}; \{l\}; j \rhd \kappa}{\Gamma; l \Vdash \mathtt{mon}^{k,l}_j(\kappa, e)}$$

Fig. 5. Well-formed intermediate terms

According to section 2 a monitor in the source code is well-formed if its negative label matches the owner of the monitor, its contract is well-formed and the guarded term is explicitly annotated as property of the server. For intermediate terms this last condition is too strict. The reduction rules for monitors of function contracts and dependent function contracts result in monitors where the protected term is a variable or an application. Because this happens in a restricted way a fixed number of steps yield monitors where the guarded term comes with the correct ownership annotation.

We capture these cases with the judgment of loosely well-formed terms $\Gamma; l \vDash e$; see figure 6 for the definition. A term with an ownership annotation with label l is both well-formed and loosely well-formed if the owner of the term is l. A variable is loosely well-formed if the environment verifies that the variable shows up in the same component l as its owner. After all, the variable is going to be substituted with a value of shape $|v|^l$. An application is loosely well formed if the operator has form $|e_1|^l$, the

operand is well-formed under l, and l is also the owner of the application. After the application is performed the resulting term is owned by l.

As for $\Gamma; l \Vdash \mathtt{check}^k_j(e, v)$, well-formedness requires that e is loosely well-formed under j and v well-formed under l, the owner of the check. The second part is necessary because v is going to be embedded in the component l as is if the check succeeds. The first is required because the contract check e establishes a component boundary separating l from j. Our approach demands that all such boundaries are indicated explicitly with ownership annotations. However, when the check is first created, e is an application with shape $|e_1|^j\ v$. Again, after, the application the ownership annotations appear at the right place and until then loose well-formedness suffices to admit the term.

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

$$\frac{\Gamma; l \Vdash e}{\Gamma; l \vDash |e|^l} \qquad \frac{\Gamma(x) = l}{\Gamma; l \vDash x} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \vDash |e_1|^l\ e_2}$$

Fig. 6. Loosely well-formed terms

The judgment for well-formed contracts requires a minor change:

$$\frac{\Gamma; j \Vdash e \qquad \overline{k} \subseteq \overline{k'}}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{flat}(|e|^j) \rfloor^{\overline{k'}}}$$

The rule for flat contracts is weakened so that it requires the parties responsible for the positive pieces of the contract to be a subset of the obligations of the flat contract rather than the same set.

Now we are ready to prove *indy* correct.

Theorem 1. *$\xrightarrow{i}$ is a complete monitor.*

The proof is direct consequence of two major lemmas: progress and preservation. A well-formed typed, term reduces to another term unless is a value or a contract error.

Lemma 1. *(Progress) For all e such that $\varnothing; l \Vdash e$, $e = v$ or $e = \mathtt{error}^k_j$ or $e \xrightarrow{i} e_0$.*

If a well-formed term reduces according to *indy*, it reduces to a well-formed term.

Lemma 2. *(Preservation) For all e and e_0 such that $\varnothing; l_o \Vdash e$ and $e \xrightarrow{i} e_0$, $\varnothing; l_o \Vdash e_0$.*

4.2 Neither Lax Nor Picky Is a Complete Monitor

In contrast to *indy*, *lax* is not a complete monitor.

As an example where *lax* does not manage to live up to complete monitoring, consider the following program.

$$\Pi^0_l = \mathtt{mon}^{k,l_o}_l(\kappa_l, |\lambda h_1.h_1\ \lambda x.5\ (\lambda g.g\ 1)|^k)\ (\lambda f.\lambda h_2.h_2\ \lambda x.6)$$

where

$$\begin{aligned}
\kappa_l &= ((\lfloor \mathtt{P?}_l \rfloor^{l_o} \mapsto \lfloor \mathtt{P?}_l \rfloor^{k}) \overset{d}{\mapsto} (\lambda f.\kappa_l^1)) \mapsto \lfloor \mathtt{P?}_l \rfloor^{k} \\
\kappa_l^1 &= ((\lfloor \mathtt{P?}_l \rfloor^{k} \mapsto \lfloor \mathtt{P?}_l \rfloor^{l_o}) \overset{d}{\mapsto} (\lambda g.\kappa_l^2)) \mapsto \lfloor \mathtt{P?}_l \rfloor^{k} \\
\kappa_l^2 &= \lfloor \mathtt{flat}(|\lambda x.\mathtt{zero?}(f\ 1 - g\ 0)|^{l}) \rfloor^{k} \\
\mathtt{P?}_l &= \mathtt{flat}(|\lambda x.x > 0|^{l}).
\end{aligned}$$

For all $l \in \mathbb{L}$ if $k \neq l_o$, then $\varnothing; l_o \Vdash \Pi_l^0$. The constraint on k and l_o comes from the rules for well-formed source terms and captures the intuition that contracts are used as the interface between different components.

Π_l^0 is a not a unique program but rather a schema of programs. Label l can be any label including k and l_o. Also Π_l^0 is not interesting in terms of computation. What makes it an example worth considering is its contract κ_l and more specifically its flat sub-contract κ_l^2. Note that κ_l^2 invokes f on a positive number and g on 0. In addition the value bound to f, $f_v = \lambda x.5$, comes from the client l_o while the value bound to g, $g_v = \lambda x.6$, originates from the server k. We start by showing that l must equal l_o in order to satisfy complete monitoring. The reduction of Π_l^0 eventually applies f_v to 1. After the substitution of f_v for f in κ_l^2 we get

$$\kappa_l^{\dagger} = \lfloor \mathtt{flat}(|\lambda x.\mathtt{zero?}(||f_v|^{l_o}|^{l}\ 1 - g\ 0)|^{l}) \rfloor^{k}.$$

In order for the *lax* system to satisfy the complete monitoring condition, $\kappa_l^{\dagger}$ must remain well formed:

$$\{f : l, g : l\}; \{k\}; \{\}; l \rhd \kappa_l^{\dagger}.$$

This judgment, however, demands that $\{f : l, g : l\}; l \Vdash ||f_v|^{l_o}|^{l}$, which in turn requires that l must be equal to l_o. If so, the contract looks like this:

$$\kappa_{l_o}^{\dagger} = \lfloor \mathtt{flat}(|\lambda x.\mathtt{zero?}(||f_v|^{l_o}|^{l_o}\ 1 - g\ 0)|^{l_o}) \rfloor^{k}.$$

The next few steps of the reduction process produce a state that is inconsistent with complete monitoring. Specifically, κ_l^2 also applies g_v to 0:

$$\kappa_{l_o}^{\dagger\dagger} = \lfloor \mathtt{flat}(|\lambda x.\mathtt{zero?}(||f_v|^{l_o}|^{l_o}\ 1 - ||g_v|^{k}|^{l_o}\ 0)|^{l_o}) \rfloor^{k}.$$

And this last contract disagrees with the subject because k cannot equal l_o. More specifically

$$\{f : l_o, g : l_o\}; \{k\}; \{\}; l_o \not\rhd \kappa_{l_o}^{\dagger\dagger}$$

since $\{f : l_o, g : l_o\}; l_o \nVdash ||g_v|^{k}|^{l_o}$.

Our example shows that independently of the choice of l, Π_l^0 does not respect preservation under the *lax* semantics. As a consequence *lax* can violate the single owner policy. Indeed $\Pi_{l_o}^0 \xrightarrow{l}_{*} E^{l_o}[||g_v|^{k}|^{l_o}\ 0]$. This last state is a stuck state as it involves the application of a function that has multiple owner tags, i.e., it has crossed contract-free boundaries between distinct components.

Theorem 2. *$\xrightarrow{l}$ is not a complete monitor.*

The same example shows that *picky* CPCF is not a complete monitor.

Theorem 3. *$\xrightarrow{p}$ is not a complete monitor.*

5 Mutation Needs Complete Monitors

The principle of complete monitoring provides guidance for the addition of linguistic features to CPCF. Concretely, consider the addition of reference cells, i.e., sharable, mutable data. Doing so requires both a notation for contracts on cells and also a mechanism that monitors all channels of communication between components that exchange cells.

We investigate this setting via CPCF!, an imperative variant of CPCF. The source syntax of CPCF!, figure 7, extends the source syntax of CPCF with the standard operators of a language with mutable cells. CPCF! also comes with contracts for mutable cells, $\texttt{ref/c}(\kappa)$. Intuitively the contract specifies that the protected cell should conform at any point with κ. CPCF!, just like CPCF, is typed. The type system and its soundness impose no challenges, and are omitted.

Types	$\tau = \ldots \mid \mathsf{ref}(\tau)$	**Values**	$v = \ldots \mid \mathit{loc} \mid \gamma$
Contracts	$\kappa = \ldots \mid \texttt{ref/c}(\kappa)$	**Guards**	$\gamma = \texttt{G}\{v\ (\kappa\ l\ l\ l)\}$
Terms	$e = \ldots \mid \texttt{ref}(e) \mid \texttt{get}(e) \mid \texttt{set}(e,e)$		

Fig. 7. CPCF! syntax (left) and intermediate syntax (right)

The additions to the source syntax demand additions to the definitions of well-formed terms and contracts. The first are straightforward requiring that the arguments of the operators related to store are well-formed under the same owner as the operator:

$$\frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \texttt{ref}(e)} \qquad \frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \texttt{get}(e)} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash \texttt{set}(e_1, e_2)}$$

The second addition poses a small challenge. The same component can read from, and write to, a mutable cell. Thus the distinction between clients and servers of the contents of the cells collapses. To reflect this insight, the rule for well-formed contracts on cells merges the parties responsible for the negative and positive pieces of the contract when assigning obligations for the contract that protects the contents of a cell. All parties $\overline{l}$ and $\overline{k}$ have the obligation to treat the contents according to the contract both as clients and servers:

$$\frac{\Gamma; \overline{lk}; \overline{lk}; j \rhd \kappa}{\Gamma; \overline{k}; \overline{l}; j \rhd \texttt{ref/c}(\kappa)}$$

Mutable cells are represented at run-time as memory locations loc. To enforce contract checks on the contents of memory locations we have to delay checking until a component tries to read the location. For that reason we introduce guards $\texttt{G}\{v\ (\kappa\ k\ l\ j)\}$ as intermediate terms. They are contract monitors similar to $\texttt{mon}_j^{k,l}(\kappa, e)$. The difference is that in contrast with monitors, guards are values, and thus they attach themselves permanently around locations when the locations cross component boundaries. Figure 7 shows the intermediate syntax for CPCF! that extends the intermediate syntax of CPCF.

The definition of the reduction relation for CPCF! demands some preparation. Locations require the presence of the addition of a store, which changes the shape of states. They now have two parts: e and σ. The reduction relation describes now transitions between such states: $E^l[e],\sigma \xrightarrow{m} E^l[e'],\sigma'$. Moreover we derive additional evaluation contexts from the new operators just like for the primitive operators in CPCF.

Now we are ready for the reduction relation of CPCF!. The reduction rules for CPCF become also reduction rules for CPCF! after adding the same store on both sides of each rule. The additional operations on mutable data, figure 8, are straightforward when they are performed directly on store locations. They only fire when the context owns the location in order to guarantee that a component can read or write to properly acquired cells. Things become interesting when a component other than the creator and owner of the location tries to access or modify the location's contents. Doing so requires a guard $\mathtt{G}\{v\ (\kappa\ k\ l\ j)\}$. Guards are the result of a monitor of a contract $\mathtt{ref/c}(\kappa)$ on a value v. They contain the guarded value, the contract κ and the labels that decorate the monitor. A $\mathtt{get}(||\gamma||^l)$ opens the guard γ and delegates the $\mathtt{get}$ operation to the value that resides in γ. Moreover it wraps the result with a monitor built out of the contract and the labels from γ. This ensures that the contract is checked when the host component tries to use the value obtained from the location.

Writing a value v' to a mutable cell via a guard γ is also delegated to the value that resides in γ; v' is wrapped with the appropriate contract monitor. However, in this case the semantics must take into account two other factors. First, a $\mathtt{set}$ operation creates a flow of values in the opposite direction than a $\mathtt{get}$ operation. Thus the server for the new content of the location should be the client for the old one and vice versa. Second, the result of $\mathtt{set}$ should be the same guard γ as the one applied on the operation. This ensures that the location remains protected. To achieve this, the reduction rule reverses the labels on the monitor of the term that is written in the location and wraps the whole operation with a monitor that is going to reproduce γ. Finally the rule expands the operation into a function application so that v' becomes explicitly decorated with the label of the host component before written to the location.

$$
\begin{array}{ll}
E^l[\mathtt{ref}(v)],\sigma & \xrightarrow{m} E^l[loc],\sigma' \\
\quad \text{where } loc \notin dom(\sigma) \text{ and } \sigma' = \sigma \uplus \{loc \mapsto v\} & \\
E^l[\mathtt{get}(||loc||^l)],\sigma & \xrightarrow{m} E^l[|v|^l],\sigma \\
\quad \text{where } \sigma = \sigma' \uplus \{loc \mapsto v\} & \\
E^l[\mathtt{set}(||loc||^l,v')],\sigma & \xrightarrow{m} E^l[|loc|^l],\sigma' \\
\quad \text{where } \sigma = \sigma'' \uplus \{loc \mapsto v\} \text{ and } \sigma' = \sigma'' \uplus \{loc \mapsto v'\} & \\
E^l[\mathtt{mon}^{k,l}_j(\mathtt{ref/c}(\kappa),v)],\sigma & \xrightarrow{m} E^l[\mathtt{G}\{v\ (\kappa\ k\ l\ j)\}],\sigma \\
E^l[\mathtt{get}(||\gamma||^l)],\sigma & \xrightarrow{m} E^l[|\mathtt{mon}^{k,l}_j(\kappa,\mathtt{get}(v))|^l],\sigma \\
\quad \text{where } \gamma = \mathtt{G}\{v\ (\kappa\ k\ l\ j)\} & \\
E^l[\mathtt{set}(||\gamma||^l,||v'||^l)],\sigma & \xrightarrow{m} E^l[|(\lambda x.\mathtt{mon}^{k,l}_j(\mathtt{ref/c}(\kappa),\mathtt{set}(v,\mathtt{mon}^{l,k}_j(\kappa,x))))\ v''|^l],\sigma \\
\quad \text{where } \gamma = \mathtt{G}\{v\ (\kappa\ k\ l\ j)\} \text{ and } v'' = |v'|^l &
\end{array}
$$

Fig. 8. Operations on mutable data

Proving that CPCF! is a complete monitor follows the same pattern as for CPCF. We first adapt the definition of complete monitoring to a store semantics.

Definition 2 (Complete Monitors for CPCF!). *A contract semantics m specifies a* complete monitor *if for all well typed terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- $e_0, \varnothing \xrightarrow{m}_* v, \sigma_1$ *or,*
- *for all terms e_1 and stores σ_1 such that $e_0, \varnothing \xrightarrow{m}_* e_1, \sigma_1$ there exists term e_2 and store σ_2 such that $e_1, \sigma_1 \xrightarrow{m} e_2, \sigma_2$ or,*
- $e_0, \varnothing \xrightarrow{m}_* e_1, \sigma_1 \xrightarrow{m}_* \mathtt{error}^k_j, \sigma_2$ *and there is at least an e_1 such that e_1 is of the form $E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l}}, v)]$ and for all such terms e_1, $v = |v_1|^k$ and $k \in \overline{l}$.*

Then we generalize well-formedness for source code and contracts to intermediate terms and prove preservation and progress main lemmas. The subject consists of two new judgments, $\Sigma; \Gamma; l \Vdash e$ and $\Sigma; \Gamma; k; l; j \rhd \kappa$.

The most important modification to the corresponding subject in CPCF is the introduction of store ownership, which establishes that the store is well-formed. The store ownership relates locations and owners. A store is well-formed if its contents are well-formed under the owner store ownership points to.

$$\frac{\text{for all } loc \in dom(\sigma),\ \Sigma; \varnothing; \Sigma(loc) \Vdash \sigma(loc)}{\Sigma \sim \sigma}$$

This is necessary for the same reason that store typing is necessary to prove type soundness for languages with mutable data: it admits circularity in the store.

The generalized judgment for well-formed terms and contracts is almost the same as the corresponding well-formed judgments in CPCF. The differences are the additional rules for store operations and that together with the environment, it propagates the ownership typing. There are also rules for guards and locations:

$$\frac{\Sigma(loc) = l}{\Sigma; \Gamma; l \Vdash loc} \qquad \frac{\Sigma; \Gamma; k \Vdash v \qquad \Sigma; \Gamma; \{k,l\}; \{k,l\}; j \rhd \kappa}{\Sigma; \Gamma; l \Vdash \mathtt{G}\{|v|^k\ (\kappa\ k\ l\ j)\}}$$

A location is well-formed only under the owner that is associated with in the store ownership. Well-formed guards are only those where the guarded value is explicitly annotated as owned by the component with the positive label (k) in the guard. Furthermore, the contract κ must be also well-formed. The last label (j) serves as the owner of the contract's code. Since guards are used to protect locations and locations can be used by components both for writing and reading both the negative label l and positive label k must be responsible for the positive and negative pieces of κ.

Furthermore we need to extend the CPCF rules for loosely well-formed terms with store ownership. We also add two rules due to store related operations:

$$\frac{\Sigma; \Gamma; l \Vdash e}{\Sigma; \Gamma; l \vDash \mathtt{get}(|e|^l)} \qquad \frac{\Sigma; \Gamma; l \Vdash e_1 \qquad \Sigma; \Gamma; l \Vdash e_2}{\Sigma; \Gamma; l \vDash \mathtt{set}(|e_1|^l, e_2)}$$

The `get` and `set` operators are loosely well-formed if the term in the position of the mutable cell is tagged with the owner of the operation. The reduction semantics guarantees that reducing the operation produces a term explicitly owned by l.

We can now show that the *indy* semantics is a complete monitor for CPCF!.

Theorem 4. *$\xrightarrow{i}$ is a complete monitor.*

6 Complete Monitors Enable Typed-Untyped Interaction

Typed Racket [8] enables mixing typed modules with untyped Racket modules. Type-like contracts prevent untyped code from violating the type discipline when interacting with typed code. Tobin-Hochstadt and Felleisen [8] define and prove the soundness of this approach in a multi-lingual setting via a so-called Blame Theorem, a name due to Wadler and Findler [9], which establishes that a program execution can only raise contract violations due to the untyped part.

To prove type soundness for an imperative version of this system we create an untyped sister language of CPCF!, UCPCF!, with a shared term syntax, and prove the corresponding blame theorem exploiting complete monitoring for CPCF!. As CPCF! has only base types, function types, and reference types, it suffices to consider only the corresponding contracts:

$$\kappa = \lfloor \mathtt{I} \rfloor^{\bar{l}} \mid \lfloor \mathtt{B} \rfloor^{\bar{l}} \mid \kappa \mapsto \kappa \mid \mathtt{ref/c}(\kappa)$$

This restriction enables a series of additional simplifications in our framework. First, flat contracts contain only built-in predicates and not arbitrary code. Thus their code is not the property of any specific party. This decision is reflected in simpler rules for well-formed flat contracts:

$$\frac{}{\Gamma;\bar{k};\bar{l};j \rhd \lfloor \mathtt{I} \rfloor^{\bar{k}}} \qquad \frac{}{\Gamma;\bar{k};\bar{l};j \rhd \lfloor \mathtt{B} \rfloor^{\bar{k}}}$$

Second, the omission of dependent function contracts makes the distinction between *lax*, *picky* and *indy* irrelevant. We use $\rightarrow$ without any subscript to denote the reduction relation for UCPCF!.

Third, checking of flat contracts does not require the special check construct:

$$\begin{array}{l}
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{I} \rfloor^{\bar{l'}}, ||\mathtt{n}||^{l'})], \sigma \rightarrow E^l[\mathtt{n}], \sigma \\
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{I} \rfloor^{\bar{l'}}, ||\mathtt{c}||^{l'})], \sigma \rightarrow E^l[\mathtt{error}_j^k], \sigma \text{ if } \mathtt{c} \neq \mathtt{n} \\
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{B} \rfloor^{\bar{l'}}, ||\mathtt{c}||^{l'})], \sigma \rightarrow E^l[\mathtt{c}], \sigma \qquad \text{if } \mathtt{c} \in \{\mathtt{tt}, \mathtt{ff}\} \\
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{B} \rfloor^{\bar{l'}}, ||\mathtt{c}||^{l'})], \sigma \rightarrow E^l[\mathtt{error}_j^k], \sigma \text{ if } \mathtt{c} \notin \{\mathtt{tt}, \mathtt{ff}\}
\end{array}$$

The untyped nature of UCPCF! obliges us to extend the reduction relation of the language. Type soundness for CPCF! allowed us to ignore redexes like $||v_1||^l\ ||v_2||^l$ where v_1 is not a function. In UCPCF! such states *can* occur. We deal with them by introducing dynamic type errors $\mathtt{error}_{\mathcal{T}}^l$ where l is the owner of the hole in which the ill-formed redex occurs.

This change must be propagated to our definition of complete monitoring. The definition of the property includes an extra case for run-time type errors.

Definition 3 (Complete Monitors for UCPCF!). *A contract semantics m specifies a* complete monitor *if for all terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- $e_0, \varnothing \xrightarrow{m}_* v, \sigma$ *or,*
- $e_0, \varnothing \xrightarrow{m}_* \mathtt{error}_{\mathcal{T}}^l, \sigma$ *or*

- *for all terms e_1 and stores σ_1 such that $e_0, \varnothing \xrightarrow{m}_* e_1, \sigma_1$ there exists term e_2 and store σ_2 such that $e_1, \sigma_1 \xrightarrow{m} e_2, \sigma_2$ or,*
- *$e_0, \varnothing \xrightarrow{m}_* e_1, \sigma_1 \xrightarrow{m}_* \mathtt{error}^k_j, \sigma_2$ where $\underline{j \neq \mathcal{T}}$, and there is at least an e_1 such that e_1 is of the form $E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{I} \rfloor^{\bar{l'}}, v)]$ or e_1 is of the form $E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{B} \rfloor^{\bar{l'}}, v)]$ and for all such e_1, $v = |v_1|^k$ and $k \in \bar{l}$.*

The addition of run-time type errors does not eliminate all stuck states. The single owner policy still must hold for a redex to reduce. We can show, though, that these stuck states are not reachable and establish that $\rightarrow$ *is* a complete monitor for UCPCF!.

Since CPCF! and UCPCF! share the same source code syntax, there is a subset of UCPCF! programs that are well-typed under CPCF!'s sound type system. We use $\mathcal{S}, \mathcal{G} \vdash e : \tau$ to express that a term e has type τ given type environment $\mathcal{G}$ and store typing $\mathcal{S}$. For simplicity we assume that there are only two component labels, u for untyped code and t for typed code. We can extend CPCF!'s type system to allow for embedding of untyped UCPCF! code:

$$\frac{\mathcal{S}, \mathcal{G} \vdash e}{\mathcal{S}, \mathcal{G} \vdash \mathtt{mon}^{u,t}_j(\kappa, e) : \mathcal{T}\,[[\kappa]]} \qquad \frac{\mathcal{S}, \mathcal{G} \vdash v}{\mathcal{S}, \mathcal{G} \vdash \mathtt{G}\{v\ (\kappa\ u\ t\ j)\} : \mathcal{T}\,[[\kappa]]}$$

The meta-function $\mathcal{T}$ maps a contract to the corresponding type. For flat contracts, $\mathcal{T}\,[[\lfloor \mathtt{I} \rfloor^{\bar{k}}]] = \mathsf{I}$ and $\mathcal{T}\,[[\lfloor \mathtt{B} \rfloor^{\bar{k}}]] = \mathsf{B}$.

The judgment $\mathcal{S}, \mathcal{G} \vdash e$ denotes that any typed code embedded in untyped code is well-typed. The judgment structurally decomposes e. Things become more interesting when a sub-term is typed:

$$\frac{\mathcal{S}, \mathcal{G} \vdash e : \mathcal{T}\,[[\kappa]]}{\mathcal{S}, \mathcal{G} \vdash \mathtt{mon}^{t,u}_j(\kappa, e)} \qquad \frac{\mathcal{S}, \mathcal{G} \vdash v : \mathcal{T}\,[[\mathtt{ref/c}(\kappa)]]}{\mathcal{S}, \mathcal{G} \vdash \mathtt{G}\{v\ (\kappa\ t\ u\ j)\}}$$

Free variables and locations in typed code can only originate from typed code. This goes hand in hand with the idea that a well-formed term can only refer to variables and locations of the same owner as the term and writing and reading foreign mutable cells can be done only through guards.

We can now state and prove the Blame Theorem.

Theorem 5. *(Blame Theorem) For all UCPCF! terms e_0 such that $\varnothing, \varnothing \vdash e_0$ and $\varnothing; u \Vdash e_0$, $e_0 \not\rightarrow_* \mathtt{error}^t_j$.*

In our setting the proof of the theorem benefits greatly from complete monitoring as it allows us to reduce the space of the proof cases. For instance when typed code retrieves values from the store, complete monitoring guarantees that those are either the property of typed code and thus, from type soundness for CPCF!, they are well-typed, or they come from the untyped code and thus they are wrapped in a contract monitor. This observation reduces the proof cases essentially to only those that create new contract monitors. There we utilize the subject introduced in this section to make sure that the new monitors that contain terms from the typed party are protecting the code with contracts that correspond to their type.

In essence the proof of the Blame Theorem says that typed terms e can only show up inside monitors of the form $\mathtt{mon}_j^{t,u}(\kappa, e)$ and that for some $\mathcal{S}$ and $\mathcal{G}$, $\mathcal{S}, \mathcal{G} \vdash e : \mathcal{T}[[\kappa]]$. Since type safety guarantees that type errors $\mathtt{error}_{\mathcal{T}}^t$ do not emerge in any case, we must only rule out contract errors blaming the typed code. From complete monitoring, this requires a failure of a contract check of the form $\mathtt{mon}_j^{t,u}(\kappa, ||\mathtt{c}||^t)$ where κ is a flat contract. However, this is impossible since $\varnothing, \varnothing \vdash \mathtt{c} : \mathcal{T}[[\kappa]]$ and by the semantics for flat contract monitors and the translation of contracts to types no such check can fail. Thus no error blaming the typed code ever occurs.

7 Related Work

Our results are based on decades-long research in behavioral contract systems and tracking of provenance. A review of and comparison with results in these fields can be found in the related work section of Dimoulas et al. [3].

Here we focus on the critically important work of Zdancewic et al. [11]. They use the idea of principals for proving type abstraction. In their semantics, each component is a different principal that allows other principals to access its data only through abstract operators. If a principal tries to manipulate directly data that it does not own, the evaluation gets stuck. In the type system foreign data is given an abstract type. Thus if the type system is sound all stuck states are unreachable.

While Zdancewic et al. directly inspire our single owner policy, our semantics is unrelated to theirs and we apply the idea to define and prove a novel property of contract systems instead of type systems.

References

1. Blume, M., McAllester, D.: Sound and complete models of contracts. Journal of Functional Programming 16(4-5), 375–414 (2006)
2. Dimoulas, C., Felleisen, M.: On contract satisfaction in a higher-order world. ACM Transactions on Programming Languages and Systems (TOPLAS) 33(5), 16:1 – 16:29 (2011)
3. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: No more scapegoating. In: POPL. pp. 215 – 226 (2011)
4. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP. pp. 48–59 (2002)
5. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/
6. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: POPL. pp. 353–364 (2010)
7. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
8. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: DLS. pp. 964–974 (2006)
9. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: ESOP. pp. 1–16 (2009)
10. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)
11. Zdancewic, S., Grossman, D., Morrisett, G.: Principals in programming languages: A syntactic proof technique. In: ICFP. pp. 197–207 (1999)