Correct Blame for Contracts [*]

No More Scapegoating

Christos Dimoulas
Northeastern University

Robert Bruce Findler
Northwestern University

Cormac Flanagan
University of California, Santa Cruz

Matthias Felleisen
Northeastern University

Abstract

Behavioral software contracts supplement interface information with logical assertions. A rigorous enforcement of contracts provides useful feedback to developers if it signals contract violations as soon as they occur and if it assigns blame to violators with precise explanations. Correct blame assignment gets programmers started with the debugging process and can significantly decrease the time needed to discover and fix bugs.

Sadly the literature on contracts lacks a framework for making statements about the correctness of blame assignment and for validating such statements. This paper fills the gap and uses the framework to demonstrate how one of the proposed semantics for higher-order contracts satisfies this criteria and another semantics occasionally assigns blame to the wrong module.

Concretely, the paper applies the framework to the *lax* enforcement of dependent higher-order contracts and the *picky* one. A higher-order dependent contract specifies constraints for the domain and range of higher-order functions and also relates arguments and results in auxiliary assertions. The picky semantics ensures that the use of arguments in the auxiliary assertion satisfies the domain contracts and the lax one does not. While the picky semantics discovers more contract violations than the lax one, it occasionally blames the wrong module. Hence the paper also introduces a third semantics, dubbed *indy*, which fixes the problems of the picky semantics without giving up its advantages.

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

General Terms Languages, Design, Reliability

Keywords Higher-order Programming, Behavioral Contracts, Blame Assignment

[*] Dimoulas and Felleisen were partly supported by AFOSR (FA9550-09-1-0110) and NSF (CRI 0855140 & CCF 0540818), Findler by NSF (0846012) and Flanagan by NSF (CNS-0905650).

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$5.00

1. Dependent Contracts: Lax or Picky?

Software engineers embrace behavioral[1] software contracts for two reasons. On one hand, contracts help explain and protect the interface of components, e.g., modules, classes, procedures, functions. On the other hand, programmers can use the familiar programming language to specify contracts, which makes it easy to read, write, and interpret them.

While both arguments obviously apply to contracts for first-order languages, Findler and Felleisen (2002)'s introduction of contracts for higher-order functions raises subtle, yet practically interesting questions. One particular question concerns dependent higher-order contracts—that is, contracts that can state assertions relating the potentially higher-order argument to the potentially higher-order result. Such contracts come with two distinct semantics in the literature. The first is the so-called *lax* semantics of Findler and Felleisen, which uses the argument in the assertion without monitoring the argument contract. In contrast, the second, *picky* semantics of Blume and McAllester (2006) monitors the argument contract during the evaluation of the component *and* during the evaluation of the assertion.

```
;; for some natural number n and real δ
(->d ([f (-> real? real?)][ε real?])
     (fp (-> real? 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)))
```

Figure 1. A higher-order, dependent contract

To make this discussion concrete, consider the dependent function contract in figure 1. This Racket fragment (formerly known as PLT Scheme) (Flatt and PLT 2010) specifies a function that maps a real-valued function `f` (and a real number ε) to a real-valued function `fp`. The post-condition adds that for some number `n` of numbers `x`, the slope of `f` at `x` is within δ of the value of `fp` at `x`. A *picky* interpretation enforces that `f` and `fp` are applied to real numbers and produce such numbers during the evaluation of the post-condition; a *lax* interpretation does not check these specifications.

Greenberg et al. (2010) compare these two forms of dependent contracts (and relate contracts to Flanagan (2006)'s hybrid types). They come to the conclusion that *picky* contracts signal the same violations as *lax* contracts and possibly more. For example, `random-number` may actually produce a complex number and

[1] We use Beugnard et al. (1999)'s terminology.

$$
\begin{array}{llll}
\textbf{Types} & \tau & ::= & o \mid \tau \to \tau \\
 & o & ::= & \mathit{num} \mid \mathit{bool} \\
\textbf{Terms} & e & ::= & v \mid x \mid e\ e \mid \mu x{:}\tau.e \mid e + e \\
 & & \mid & e - e \mid e \wedge e \mid e \vee e \mid \mathtt{zero?}(e) \\
 & & \mid & \mathtt{if}\ e\ e\ e \\
\textbf{Values} & v & ::= & 0 \mid 1 \mid -1 \mid \ldots \mid \lambda x{:}\tau.e \\
 & & \mid & \mathtt{tt} \mid \mathtt{ff} \\
\textbf{E.Contexts} & E & ::= & [\ ] \mid E\ e \mid v\ E \mid E + e \mid v + E \\
 & & \mid & E - e \mid v - E \mid E \wedge e \mid v \wedge E \\
 & & \mid & E \vee e \mid v \vee E \mid \mathtt{zero?}(E) \\
 & & \mid & \mathtt{if}\ E\ e\ e
\end{array}
$$

Figure 2. PCF syntax

the contract may thus misapply `f`; the *picky* interpretation catches this potential problem, while the *lax* one doesn't. Our experience shows, however, that Greenberg et al. (2010)'s result doesn't truly settle the issue. When a *picky* contract signals a contract violation, it may blame the wrong party.

In this paper, we develop a third notion of contract monitoring and demonstrate that it satisfies an intuitive correctness criterion. We start from the observation that Greenberg et al. are correct in that a *picky* interpretation is important for dependent contracts. If the "dependency assertion" violates a contract, the computation may go wrong in all kinds of ways. The question is which party the monitoring system should blame for such a problem. The *picky* interpretation blames either the server or the client. Our new interpretation, dubbed *indy*, treats the contract as an independent party and blames it for problems where *picky* blames the wrong party.

To compare the three possible interpretations, we develop a unified semantic framework, based on a reduction semantics (Felleisen et al. 2009) for a PCF-like language with contracts. The three interpretations are expressed as three different, one-rule extensions that specify the semantics of dependent contracts. We then enrich the framework with the necessary information to track code ownership and contract obligations, two novel technical notions that might prove useful in other contexts. This enriched framework is used to formalize the following correctness criterion: a contract system should only blame a party if *the party controls the flow or return of values into the particular contract check that failed.*

We can prove that our new *indy* interpretation satisfies this criterion while *picky* fails to live up to it. Inspired by our theoretical result, we equip Racket with an *indy* dependent contract combinator, `->i`, in addition with the existing *lax* combinator, `->d`.

Finally, we explain how to use the framework to implement a tool that explains the responsibilities that contracts imposes. Specifically, the tool teases out contract obligations from complicated contracts and highlights them with colors. We have implemented the tool and include some screenshots to illustrate its usefulness.

2. Contract PCF

PCF (Plotkin 1977) is the starting point for our model; figure 2 summarizes the well-known syntactic domains. In this setting, a program is a closed term. Also, we equip the language with a standard type system and a call-by-value reduction semantics, though for lack of space, we omit the details (Plotkin 1975; Felleisen et al. 2009). Similarly, we use type annotations only when needed.

2.1 Adding Higher-Order Contracts

Adding plain higher-order contracts (Findler and Felleisen 2002) to PCF is straightforward: see figure 3. First, PCF is enriched with contracts, a contract type, and new terms for attaching contracts to terms and raising contract violations. Second, we extend our type system with rules for the extra terms. The resulting language is CPCF, PCF with contracts.

CPCF is equipped with two kinds of contracts: flat contracts, $\mathtt{flat}(e)$, and higher-order contracts, $\kappa_1 \mapsto \kappa_2$. The former are predicates on base values. The latter combine a contract, κ_1, on the arguments of a function with a contract, κ_2, on the result of the function.

$$
\begin{array}{llll}
\textbf{Contracts} & \kappa & ::= & \mathtt{flat}(e) \mid \kappa \mapsto \kappa \\
\textbf{Types} & \tau & ::= & \ldots \mid \mathit{con}(\tau) \\
\textbf{Terms} & e & ::= & \ldots \mid \mathtt{mon}_l^{l,l}(\kappa, e) \mid \mathtt{error}^l
\end{array}
$$

$$
\frac{\Gamma \vdash e : o \to \mathit{bool}}{\Gamma \vdash \mathtt{flat}(e) : \mathit{con}(o)}
$$

$$
\frac{\Gamma \vdash \kappa_1 : \mathit{con}(\tau_1) \qquad \Gamma \vdash \kappa_2 : \mathit{con}(\tau_2)}{\Gamma \vdash \kappa_1 \mapsto \kappa_2 : \mathit{con}(\tau_1 \to \tau_2)}
$$

$$
\frac{\Gamma \vdash \kappa : \mathit{con}(\tau) \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{mon}_j^{k,l}(\kappa, e) : \tau}
\qquad
\frac{}{\Gamma \vdash \mathtt{error}^l : \tau}
$$

Figure 3. CPCF: syntax and types

The most important new construct is the monitoring construct $\mathtt{mon}_j^{k,l}(\kappa, e)$, which places a contract κ between a term e (the server) and its context (the client). It demands that any value that flows between e and its context is monitored for conformance with the contract. For a flat contract, the predicate is applied to the value; for a higher-order contract, the pieces of the contract are attached to the argument and range position of a wrapper function and the contract is monitored as the function flows through the program.

A monitor comes with three labels:[2] a pair of distinct blame labels k and l for the two parties to the contract and a contract label j, for the origin of the contract. In source code, the contract label j usually differs from k and l but under some circumstances it may be equal to either of the two. Labels are drawn from the enumerable set $\mathbb{L}$. The label l_o is used as the label of the whole program; $\bar{l}$ denotes a subset of $\mathbb{L}$. When a contract fails a contract error, $\mathtt{error}^l$, is raised where l denotes the party responsible for the violation.

$$
\begin{array}{llll}
\textbf{E.Contexts} & E & ::= & \ldots \mid \mathtt{mon}_l^{l,l}(\kappa, E)
\end{array}
$$

$$
\begin{array}{rcl}
E\left[\mathtt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)\right] & \longmapsto & E\left[\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\ \mathtt{mon}_j^{l,k}(\kappa_1, x))\right] \\
E\left[\mathtt{mon}_j^{k,l}(\mathtt{flat}(e), v)\right] & \longmapsto & E\left[\mathtt{if}\ (e\ v)\ v\ \mathtt{error}^k\right] \\
E\left[\mathtt{error}^l\right] & \longmapsto & \mathtt{error}^l
\end{array}
$$

Figure 4. CPCF: semantics

The introduction of contracts requires small changes to the reduction semantics. Figure 4 spells out the details, starting with the slight modification of the set of evaluation contexts. The bottom half shows the reduction rules for contract checking and blame assignment, adapted from Findler and Felleisen (2002)'s original semantics. A higher-order monitor is split into two parts:

1. a monitor for the argument with reversed blame labels;
2. a monitor for the result with the original blame labels.

[2] In an implementation, these labels are synthesized from the program text.

A first-order monitor is transformed to an `if` statement that checks whether the guarded value satisfies the contract's predicate. If the predicate is satisfied, the value is returned; otherwise a contract error is signaled using the first blame label to pinpoint the guilty party. Finally, when a contract error is raised the evaluation is aborted and the contract error is returned as the final result.

2.2 Adding Dependent Contracts

In contrast to conventional contracts for first-order functions, the higher-order contracts of the preceding section cannot express dependencies between arguments and results. Therefore Findler and Felleisen (2002) equip CPCF with a functional contract form that parameterizes the result contract over the argument:

$$\textbf{Contracts} \quad \kappa \quad ::= \quad \ldots \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa)$$

Findler and Felleisen (2002)'s reduction rule for these *dependent contracts* captures this intention:

$$\begin{array}{ll} E\,[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \quad \longmapsto_l & (\mathit{lax}) \\ \qquad E\,[\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))] & \end{array}$$

A dependent monitor acts like a higher-order monitor. The subtle difference is that the parameter x of the proxy function captures the free occurrences of x in the contract's postcondition κ_2. As a result, any argument to the proxy function is substituted for x in κ_2 and is then used in the argument position, suitably wrapped with an argument monitor.

Blume and McAllester (2006) observe that the precondition κ_1 is not enforced during the evaluation of postcondition κ_2. This gap opens the door for potential abuses of the argument in κ_2, i.e., uses that don't conform to κ_1. They rightly consider this a problem and, in turn, they propose the following change to the rule:[3]

$$\begin{array}{ll} E\,[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \quad \longmapsto_p & (\mathit{picky}) \\ \qquad E\,[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\{\mathtt{mon}_j^{l,k}(\kappa_1, x)/x\}}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))] & \end{array}$$

Specifically, every free occurrence of x in κ_2 is replaced with $\mathtt{mon}_j^{l,k}(\kappa_1, x)$. Thus, any argument to the function remains protected by κ_1 even inside κ_2. Note how the injected monitor carries the same blame labels as the monitor for the argument in the body of the function.[4]

Greenberg et al. (2010) compare the *lax* and *picky* contract systems and conclude that the former signals strictly fewer contract errors than a *picky* contract system. More precisely, for any program, the following statements hold:

- neither contract system signals a contract error;
- both raise an error and blame the same party; or
- the *picky* contract system discovers a contract violation and the *lax* system does not raise a contract error.

Their results characterizes two different philosophies of contract code. On the one hand, a *lax* contract system treats contracts as trusted code. Both parties have agreed to the contract and have presumably ensured that its evaluation doesn't violate any invariants. On the other hand, a *picky* contract system considers contracts to contain potentially faulty code. To enforce the contracts within this code, a *picky* system protects values that flow into the contracts.

The problem with the *picky* system is that it may blame the server or the client for violations of a contract κ when neither of them can control the flow of values into the responsible monitor. To illustrate this point and to provide an alternative, we introduce a third contract monitoring system that considers contracts as independent entities. When a dependent contract abuses a value according to κ_1, this revised system blames the contract for the violation.

Here is the reduction rule:

$$\begin{array}{ll} E\,[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \quad \longmapsto_i & (\mathit{indy}) \\ \quad E\,[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\{\mathtt{mon}_j^{l,j}(\kappa_1, x)/x\}}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))] & \end{array}$$

The rule makes the contract responsible if it supplies an inappropriate value to a function argument during the evaluation of the "dependency." It accomplishes this switch of responsibility with the creation of a new monitoring expression for the argument with the contract label as the negative blame label. This new argument expression is substituted into the range part of the contract.[5]

For an example, consider this monitor expression:

$$\Pi^0 = \mathtt{mon}_j^{k,l}(\kappa, \lambda x.(\text{—}0\text{—}))\ \lambda x.(\text{—}1\text{—})$$

where

$$\begin{array}{rcl} \kappa & = & \kappa_1 \overset{d}{\mapsto} (\lambda f.\mathtt{flat}(\lambda x.f\ (\lambda x.(\text{—}2\text{—})) > 0)) \\ \kappa_1 & = & (\mathtt{P?} \mapsto \mathtt{P?}) \overset{d}{\mapsto} (\lambda g.\mathtt{flat}(\lambda x.g\ 3 > 0)) \end{array}$$

Here the server is the function $\lambda x.(\text{—}0\text{—})$, while the client is the context $[\ ]\ \lambda x.(\text{—}1\text{—})$. The mediating contract κ is a higher-order, dependent contract where `P?` checks for positive numbers, i.e., $\mathtt{P?} = \mathtt{flat}(\lambda x.x > 0)$ and $>$ has the standard recursive definition.

The argument $\lambda x.(\text{—}1\text{—})$ flows to the postcondition of κ and replaces f. To protect it from potentially misbehaving contract code, it is wrapped with a monitor that enforces κ_1:

$$\mathtt{mon}_j^{l,j}(\kappa_1, \lambda x.(\text{—}1\text{—}))\ .$$

Since κ_1 is a dependent contract, too, the story continues. When the postcondition is eventually checked, this proxy function for $\lambda x.(\text{—}1\text{—})$ is applied to $\lambda x.(\text{—}2\text{—})$. In that case, the latter flows to the postcondition of κ_1 and replaces g with another monitored domain contract.

Each of the three rules gives rise to a semantics for CPCF. In principal, we extend $\longmapsto$ with $\longmapsto_m$ where $m \in \{l, p, i\}$ to get the three complete reduction relations. Since there is no danger of ambiguity, we overload the symbol $\longmapsto_m$ and use it for the complete semantics.

2.3 Two More Flavors

The treatment of contracts as independent parties is compatible with some practical uses in our Racket implementation. First, contracts for Racket's unit system are given as part of the signature. Strickland and Felleisen (2009) show that linking such units may necessitate blaming the signature itself. Our framework finally provides a semantic explanation for this phenomenon.

Second, in Racket's first-order module system, contracts are specified via `provide/contract`, i.e., in the export interface of modules. This form combines identifiers with contracts and attaches contracts to these values as they flow across the module boundary. When things go wrong with the dependencies in such

[3] In principle, this reduction rule should use a `let` to preserve a strict call-by-value regime. But, due to the restrictions on our grammar for contracts, a straight substitution is technically correct and superior.

[4] An alternative definition for the *picky* rule is to *not* switch the blame labels on the internal monitor. Doing so does not affect our results.

[5] Morally, the monitor should not apply when the client is already labeled with the contract label. In that case, the value flow is entirely within the contract party and should strictly speaking not be monitored. To model this behavior, we would have to add the side condition $l \neq j$ and add a second rule:

$$\begin{array}{lll} E\,[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] & \longmapsto_i & E\,[\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))] \\ & & \text{if } l = j \end{array}$$

Both variants of *indy* satisfy the main theorem, which is why this paper focuses on the theoretically simpler approach.

contracts, the monitoring system considers the contract a part of the server module and blames the server module. We can express this idea—dubbed $+$*indy*—in our framework with the small change of using the module name as the contract label.

Finally, Typed Racket (Tobin-Hochstadt and Felleisen 2010, 2008) protects the interaction of typed and untyped modules with contracts derived from types. Since one of the basic assumptions of Typed Racket is that untyped modules stay unchanged, it implements this protection mechanism with `require/contract`. This contract form guarantees that values from an untyped module satisfy the specified contract. Put differently, the form protects the import boundary. If a programmer attached dependencies to these contracts, the code would have to be considered as a part of the client module. We can capture this semantics, dubbed $-$*indy*, by using the importing module's name as the contract label.

2.4 Comparing Contract Systems

Equipped with three additional contract monitoring systems, we can now explore their relationship. Consider this example:

$$\Pi^1 \quad = \quad \mathtt{mon}_*^{k,l}(\kappa, \lambda f.f\,42)\;\lambda x.x$$
$$\text{where } \kappa = (\mathtt{P?} \mapsto \mathtt{P?}) \overset{d}{\mapsto} (\lambda f.\mathtt{flat}(\lambda x.f\,0 > -1))$$

The example uses the placeholder $*$ for the contract label so that we can include $+$*indy* and $-$*indy*—the two additional flavors of *indy*—in our comparisons. As needed, we replace $*$ with k for $+$*indy*, with l for $-$*indy*, and a distinct label j for *indy*. Recall that the reduction rules for the *lax* and *picky* contract monitoring systems do not employ the contract label.

The evaluation of Π^1 yields the following results for the five different contract monitoring systems:

program	*	monitoring system	result
Π^1	—	*lax*	42
Π^1	—	*picky*	$\mathtt{error}^k$
Π^1	j	*indy*	$\mathtt{error}^j$
Π^1	k	$+$*indy*	$\mathtt{error}^k$
Π^1	l	$-$*indy*	$\mathtt{error}^l$

The table demonstrates several points. First, when a program yields a plain value according to the *lax* system, the *picky* system may still find a fault during contract checking and signal a violation. Second, the *picky* system here blames party k, the server component, for a contract violation. The specific violation is that f is applied to 0 in the dependency assertion, even though the domain contract promises that the function is only applied to positive numbers. Third, the *indy* system blames the contract itself, rather than the server. Fourth, the system based on the $+$*indy* rule agrees with the *picky* system, because it considers all code in a contract as part of the server. Finally, the $-$*indy* system blames party l; after all, the misapplication of f is internal to the client, which chooses to defy the restrictions on the domain of f.

Another example shows that *picky* can also blame the client when things go wrong with the contract:

$$\Pi^2 \quad = \quad \mathtt{mon}_*^{k,l}(\kappa, \lambda f.f\;\lambda x.x)\;\lambda g.g\,42$$
$$\text{where } \kappa = ((\mathtt{P?} \mapsto \mathtt{P?}) \overset{d}{\mapsto} (\lambda f.\mathtt{flat}(\lambda x.f\,0 > -1))) \mapsto \mathtt{P?}$$

Specifically, evaluating Π^2 yields the following results:

program	*	monitoring system	result
Π^2	—	*lax*	42
Π^2	—	*picky*	$\mathtt{error}^l$
Π^2	j	*indy*	$\mathtt{error}^j$
Π^2	k	$+$*indy*	$\mathtt{error}^k$
Π^2	l	$-$*indy*	$\mathtt{error}^l$

Again the *lax* system does not signal contract violations, while the other four report one. Here the *indy* system blames the contract itself rather than the client, which is blamed by the *picky* and $-$*indy* systems. The $+$*indy* system blames the server.

Together the two examples demonstrate that none of our new monitoring rules are logically related to *picky* if we take blame into account. In short, the introduction of contracts as independent parties calls for a comparison that takes into account why a contract violation is detected and why the accused party is blamed.

Note, however, that the *indy* contract system signals an error when the *picky* system signals an error and vice versa, though the errors aren't necessarily labeled with the same party.

PROPOSITION 1. $e \overset{*}{\longmapsto}_i \mathtt{error}^k$ *iff* $e \overset{*}{\longmapsto}_p \mathtt{error}^{k'}$

PROOF IDEA. By a straightforward bi-simulation argument. The bi-simulation used for the proof relates two expressions that are structurally identical except that their labels can differ. ∎

3. Tracking Ownership and Obligations

While the preceding section illuminates the problems of the *picky* contract system and the difficulty of comparing contract systems in general, it also implies a new way of thinking about contract violations. The first major insight is that the *picky* system may blame either the server module or the client module when, in fact, the contract itself is flawed. From here, it is obvious to inspect what a contract monitoring system would do if contracts were a part of the server, a part of the client, or a third party. Doing so produces the second major insight, namely, that none of these alternatives agrees with the *picky* semantics.

Putting the two insights together implies that we need a semantics that (1) for each party, keeps track of its contract obligations and (2) for each value, accounts for its origin. Once a semantics provides this additional information, we can check whether a contract system ever blames a party for violating an obligation if the party has no control over the value's flow into the contract. In this section, we equip CPCF's semantics with ownership and obligation information, which is maintained across reductions. In the next section, we use this information to state a contract correctness property and to measure how the various monitoring systems fare with respect to this property.

3.1 Ownership . . .

To model an ownership relationship between parties and code, we extend CPCF with a new construct that relates terms and values to parties:[6]

$$\begin{array}{llll} \textbf{Terms} & e & ::= & \ldots \mid \|e\|^l \\ \textbf{Values} & v & ::= & \ldots \mid \|v\|^l \end{array}$$

During reductions, terms and values come with a stack of owners, reflecting transfers from one party to another. The notations $\|e\|^{\overrightarrow{l_n}}$ and $\|e\|^{\overleftarrow{l_n}}$ are short-hands for such stacks, abbreviating $\|\ldots\|e\|^{l_1}\ldots\|^{l_n}$ and $\|\ldots\|e\|^{l_n}\ldots\|^{l_1}$, respectively. Ownership l for an expression means its result is attributed to l. In turn, a value with an ownership l originates from component l or is affected by a traversal through component l.

3.2 . . . and Obligations

CPCF contracts consist of trees with flat contracts at the leafs. Exploiting the analogy with function types, Findler and Felleisen (2002) implicitly decompose these trees into obligations for servers

[6] The inspiration of *ownership* comes from the work of Zdancewic et al. (1999) on *principals*. For a comparison, see section 6.

(positive positions) and clients (negative positions). Their semantics tracks this connection via labels; errors use them to pinpoint contract violators.

Dimoulas and Felleisen use this idea for a static decomposition of contracts into server and client obligations. They define two functions from contracts to contracts that tease out the respective obligations. The one for teasing out server obligations replaces flat contracts in negative positions with $\top = \texttt{flat}(\lambda x.\texttt{tt})$ and then reconstructs the overall contract; analogously, the decomposition map for teasing out client obligations replaces flat contracts in positive positions with $\top$. For instance the contract of the Π^2 example in section 2 yields these decompositions:

	$((\texttt{P?} \mapsto \texttt{P?}) \overset{d}{\mapsto} (\lambda f.\texttt{flat}(\lambda x.f\ 0 > -1))) \mapsto \texttt{P?}$
server	$((\top \mapsto \texttt{P?}) \overset{d}{\mapsto} (\lambda f.\top)) \mapsto \texttt{P?}$
client	$((\texttt{P?} \mapsto \top) \overset{d}{\mapsto} (\lambda f.\texttt{flat}(\lambda x.f\ 0 > -1))) \mapsto \top$

Decomposition implies that each flat contract imposes obligations on a specific component, i.e., party to a contract. Since one and the same server may connect with many different clients and since *indy* systems may use the contract itself as a component, it is not just one party that is associated with a flat contract but many. Hence we modify the syntax of contracts to statically associate flat contracts and owners:

$$\textbf{Contracts} \quad \kappa \quad ::= \lfloor \texttt{flat}(\|e\|^{l}) \rfloor^{\overline{l}} \mid \kappa \mapsto \kappa \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa)$$

In contrast with ownership, obligations come as sets of labels $\overline{l}$, not vectors. After all, there is no need to order obligations or to change them during an evaluation. Of course, a static attribute about a dynamic obligation calls for a way to determine whether such annotations are well-formed.

3.3 Well-formed Ownership and Obligations

Only some annotations make sense for a source program. Both ownership and contract monitors specify boundaries and, at the source level, these boundaries should coincide. We therefore introduce a well-formedness judgment to enforce these conditions for source programs. Before doing so, we present the simple typing rules for the two new constructs:

$$\frac{\Gamma \vdash e : o \to \mathit{bool}}{\Gamma \vdash \lfloor \texttt{flat}(e) \rfloor^{\overline{l}} : \mathit{con}(o)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \|e\|^{l} : \tau}$$

Concerning ownership annotations, a CPCF source program may contain those at only two places: in contract monitors and in flat contracts. Since contract monitors establish a boundary between the client component and the server component, we demand an ownership annotation on the server component and that a match of these annotation with the positive label of the monitor. Conversely, the context of such an expression must belong to the client. Finally, flat contracts must come with ownership labels consistent with the surrounding monitors because they are turned into plain code during the evaluation, and the semantics must track where they originated from.

We express this constraint with the well-formedness relation $l \vdash e$, which says that l "owns" e and checks that e is well-formed. Equivalently, l is the owner for the context of e. A closed expression e is a well-formed program if $l_o \vdash e$ where l_o is the label reserved for the owner of the program.

Figure 5 defines most of this well-formedness judgment. For terms that do not involve monitors and contracts the definition is a structural judgment, and base values and variables are well-formed under any owner. The actual key is the one for contract monitor, which we present separately. According to our informal description, a well-formed contract monitor is a boundary between

$$\boxed{l \vdash e}$$

$$\frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1\ e_2} \qquad \frac{l \vdash e}{l \vdash \lambda x.e} \qquad \frac{l \vdash e}{l \vdash \mu x.e}$$

$$\frac{l \vdash e_1 \quad l \vdash e_2 \quad l \vdash e_3}{l \vdash \texttt{if}\ e_1\ e_2\ e_3}$$

$$\frac{l \vdash e_1}{l \vdash \texttt{zero?}(e_1)}$$

$$\frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 + e_2} \qquad \frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 - e_2}$$

$$\frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 \wedge e_2} \qquad \frac{l \vdash e_1 \quad l \vdash e_2}{l \vdash e_1 \vee e_2}$$

$$\frac{}{l \vdash \texttt{n}} \qquad \frac{}{l \vdash \texttt{tt}} \qquad \frac{}{l \vdash \texttt{ff}} \qquad \frac{}{l \vdash x}$$

Figure 5. Ownership coincides with contract monitors

a client and a server, implying this shape for the judgment:

$$l \vdash \texttt{mon}^{k,l}_{j}(\kappa, \|e\|^{k})$$

It says that if l owns the context and k is the blame label for the server, then the blame label for the client should be l *and* the wrapped expression e should come with an ownership annotation that connects it to k. Next, e must be well-formed with respect to its owner k, because it may contain additional contract monitors. But even with this antecedent, the well-formedness judgment is incomplete. After all, the contract κ that governs the flow of values between the server and the client contains code and this code must be inspected. Furthermore, we must ensure that all flat contracts within κ are obligations of the appropriate parties including the contract monitor itself, which is represented by the contract label j.

Putting everything together, we get this well-formedness rule for contract monitors in source programs:

$$\frac{k \vdash e \quad \{k\};\{l\};j \rhd \kappa}{l \vdash \texttt{mon}^{k,l}_{j}(\kappa, \|e\|^{k})}$$

It relies on a secondary well-formedness judgment for contracts, to which we turn next.

$$\frac{\overline{l};\overline{k};j \rhd \kappa_1 \quad \overline{k};\overline{l};j \rhd \kappa_2}{\overline{k};\overline{l};j \rhd \kappa_1 \mapsto \kappa_2}$$

$$\frac{\overline{l};\overline{k} \cup \{j\};j \rhd \kappa_1 \quad \overline{k};\overline{l};j \rhd \kappa_2}{\overline{k};\overline{l};j \rhd \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)}$$

$$\frac{j \vdash e}{\overline{k};\overline{l};j \rhd \lfloor \texttt{flat}(\|e\|^{j}) \rfloor^{\overline{k}}}$$

Figure 6. Obligations coincide with labels on monitors

Roughly speaking, $\overline{k};\overline{l};j \rhd \kappa$ says that contract κ is well-formed for sets of positive and negative obligation labels $\overline{k}$ and $\overline{l}$, respectively, and the owner j of the contract monitor that attaches the contract to a boundary. As the definition in figure 6 shows, the two sets are swapped for the antecedents of higher-order (dependent) contracts. For the negative positions in the precondition of dependent contract label j is added to indicate that these are also obligations of the owner of the contract monitor. For flat contracts the positive obligation labels must coincide with the obligation labels of the contract.

Note that the well-formedness of flat contracts also enforces an ownership annotation. Specifically, the owner of the context—which, by assumption, is a contract monitoring construct labeled with j—is also the owner of the predicate in the flat contract. The antecedent of the rule recursively uses the well-formedness judgments for ownership to ensure that e itself is well-formed.

3.4 Ownership and Obligations Semantics

The final change to the CPCF model concerns the reduction semantics. Specifically, we change the reduction relations so that each reduction step keeps track of ownership rights and obligations. While ownership and obligations do not affect the semantics per se, the information is critical for characterizing the behavior of contract monitoring systems, as we show in the next section.

Our first step is to equip the grammar of evaluation contexts with a parameter that accounts for the owner of the hole. In the parameterized grammar, E^l, of figure 7 the parameter l points to the ownership annotations that is closest to the hole of the context.

$$
\begin{array}{llll}
\textbf{E. Contexts} & E^l & ::= & G^l \\
 & G^l & ::= & G^l\ e \mid v\ G^l \mid G^l + e \mid v + G^l \\
 & & \mid & G^l - e \mid v - G^l \mid G^l \wedge e \mid v \wedge G^l \\
 & & \mid & G^l \vee e \mid v \vee G^l \mid \texttt{zero?}(G^l) \\
 & & \mid & \texttt{if}\ G^l\ e\ e \mid \texttt{mon}_j^{k,l}(\kappa, G^l) \\
 & & \mid & \|F\|^l \mid \|G^l\|^{l'} \\
 & F & ::= & [\,] \mid F\ e \mid v\ F \mid F + e \mid v + F \\
 & & \mid & F - e \mid v - F \mid F \wedge e \mid v \wedge F \\
 & & \mid & F \vee e \mid v \vee F \mid \texttt{zero?}(F) \\
 & & \mid & \texttt{if}\ F\ e\ e \mid \texttt{mon}_j^{k,l}(\kappa, F)
\end{array}
$$

Figure 7. Parameterized evaluation contexts

Evaluation contexts are labeled with the label l_o—the label reserved for the whole program—if they do not contain an ownership constructs on the path from the hole to the root: $E^{l_o} ::= F$.

From now on, all reduction relations assume labeled evaluation contexts. This implies that newly created values are always assigned an owner. For the reduction relations concerning primitive operators and conditionals, the changes are straightforward and summarized in the top part of figure 8. For the rules concerning monitors with flat and plain higher-order contracts and their blame assignments, specified in the lower part of the same figure, we also know that they do not need to manipulate any ownership annotations. These reduction rules remain unchanged, modulo the labeled evaluation contexts. The obligation annotation on flat contracts is ignored. For details, see the bottom part of figure 8. We add one last simple rule separately:

$$E^l[\texttt{error}^k] \longmapsto_m \texttt{error}^k$$

Since the act of signaling errors erases the surrounding evaluation context, the format of this rule doesn't fit the table. Note that the context on the right is $[\,]^{l_o}$ and l may not equal l_o.

The reduction of β_v redexes typically demands several re-alignments with respect to ownership. To start with, the function and the argument may belong to different parties. Furthermore, the context brings together the operand and operator, and the semantics should keep track of this responsibility. Together, the two observations suggest the following relation:

$$E^l[\|\lambda x.e\|^{\vec{l_n}}\ v] \quad \longmapsto_m \quad E^l[\|\{\|v\|^{l\overleftarrow{l_n}}/x\}e\|^{\vec{l_n}}]$$

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

Figure 8. Ownership and obligation propagation

The relation says that after tagging the value with the ownership label l of the context, the value moves under the ownership annotations of the function. The result is a value whose innermost owner is l and whose outermost owner is l_1 of $\vec{l_n}$:

$$\|v\|^{l\overleftarrow{l_n}}$$

The properly annotated value is then substituted into the body e of the function for its parameter x. The result itself is owned by the same owner as the function.

Put differently, it is best to view function application as a form of communication between two components: the function and its context. The context picks the argument, declares itself its owner, and then passes it to the function. The function accepts the argument, adjusts its ownership, and integrates the result into its body.

Recursion is treated as a special form of function application:

$$E^l[\mu x.e] \quad \longmapsto_m \quad E^l[\{\|\mu x.e\|^l/x\}e]$$

The owner of the context l and user of the recursive function declares itself owner of $\mu x.e$ before substituting it in the body of the recursive function.

All the complexity of tracking ownership is due to dependent function contracts. Consider the simplest variant, *lax*:

$$
\begin{array}{lll}
E[\texttt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] & \longmapsto_l & (lax) \\
\qquad E[\lambda y.\texttt{mon}_j^{k,l}(\underline{\{y/x\}\kappa_2}, v\ \texttt{mon}_j^{l,k}(\kappa_1, y))] & &
\end{array}
$$

For emphasis, this version of the reduction rule uses y as the parameter of the proxy function on the right hand side. The use of y as parameter demands that we also replace all occurrences of x in κ_2 with y so that when the proxy function is applied, the actual argument is substituted into the dependent range contract; without the substitution, the reduction would create free variables.[7]

Rewriting the *lax* rule in this way reveals that it encodes a masked function application. The problem is that, as discussed above, a function application must add the label of the responsible owner at the bottom of the stack, and this label is not available here. Instead, it is found at the flat leafs of the contract, which—according to the static semantics of the preceding subsection—must come with an ownership annotation. The solution is to introduce the substitution function $\{e/^c x\}\kappa$, which copies the ownership label from flat contracts to the substituted term.

[7] The value v is unaffected by this change of parameters, because we assume the usual hygiene condition (Barendregt 1984) for metavariables.

With this substitution function in place, it is easy to specify the three variants for the reduction of dependent functional contracts:

$$
\begin{array}{lr}
\multicolumn{2}{c}{E^l[\cdots] \quad \ldots \quad E^l[\cdots]} \\
\hline
\mathtt{mon}^{k,l}_{j}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) \quad \longmapsto_l & \\
\quad \lambda x.\mathtt{mon}^{k,l}_{j}(\{x/^c x\}\kappa_2, v\, \mathtt{mon}^{l,k}_{j}(\kappa_1, x)) & (lax) \\
\mathtt{mon}^{k,l}_{j}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) \quad \longmapsto_p & \\
\lambda x.\mathtt{mon}^{k,l}_{j}(\underline{\{\mathtt{mon}^{l,k}_{j}(\kappa_1, x)/^c x\}\kappa_2}, v\, \mathtt{mon}^{l,k}_{j}(\kappa_1, x)) & (picky) \\
\mathtt{mon}^{k,l}_{j}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) \quad \longmapsto_c & \\
\lambda x.\mathtt{mon}^{k,l}_{j}(\underline{\{\mathtt{mon}^{l,j}_{j}(\kappa_1, x)/^c x\}\kappa_2}, v\, \mathtt{mon}^{l,k}_{j}(\kappa_1, x)) & (indy)
\end{array}
$$

We conclude this section with the definition of the auxiliary substitution function:

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

The definition is total. The redefinition of the contract syntax enforces that flat contracts always have the annotations expected by the domain of the substitution function. This also guarantees that the reduction relations $\longmapsto_m$ are well-defined.

4. Correct Blame

Using ownership and obligation annotations, we can formulate what it means for a contract system to correctly blame a violator. After all, the tracking of ownership and obligations is entirely independent of the contract checking, and it is thus appropriate to use tracking as an independent specification of contract monitoring.

Values should be originating from one of the parties of the contract are checked only against flat pieces of the contract for which the party is responsible. Now we can phrase this property in terms of ownership and obligations: when the evaluation reaches a redex that checks a flat contract on a value, then the owner of the value must be the same as the positive party of the monitor and in addition the positive party is included in the obligations of the contract.

DEFINITION 2 (Blame Correctness). *A contract system m is* blame correct *if for all terms e_0 such that $l_o \vdash e_0$, and*

$$e_0 \overset{*}{\longmapsto}_m E^{\dagger}[\mathtt{mon}^{k,\dagger}_{\dagger}(\lfloor \mathtt{flat}(e_1) \rfloor^{\bar{l}}, v)]$$

$v = \|v_1\|^k$ and $k \in \bar{l}$. The identity of the † labels is irrelevant.

The definition says that when the reduction of a well-formed program reaches a state in which it checks a flat contract, then the server (positive) label of the monitor and the ownership label on the value must coincide and, furthermore, the set of obligations for the flat contract must contain this label. Conversely, if these obligations are not met, the monitor may blame k for a contract failure even though the party had no control over the flow of v into this monitor.

We can prove that the *indy* contract system and even the *lax* system are blame correct, while *picky* isn't. The proof of the positive theorem directly follows from a subject reduction theorem for ownership annotations. This latter theorem requires a complex proof, which is the subject of the second subsection. The third subsection explains how to prove the two main theorems. To start with, however, we clarify that ownership annotations and obligations do are orthogonal to semantics.

4.1 It is all about Information Propagation

The addition of ownership and obligation annotations does not affect the behavior of any programs. Our revised semantics simply propagates this information so that it can be used to characterize execution states. In order to formulate this statement, we use the symbol $\overset{*}{\longmapsto}_m{}^{cpcf}$ for the transitive-reflexive reduction relations of section 2 and $\overset{*}{\longmapsto}_m{}^{anno}$ for the relations of section 3.

PROPOSITION 3. *The following statements hold for $m \in \{l, p, c\}$.*

1. *Let e be a well-formed CPCF program: $l_o \vdash e$. Let $\bar{e}$ be the plain CPCF expression that is like e without annotations. If $e \overset{*}{\longmapsto}_m{}^{anno} e'$, then $\bar{e} \overset{*}{\longmapsto}_m{}^{cpcf} \bar{e}'$.*
2. *Let $\bar{e}$ be a plain CPCF program. There exists some labeled CPCF program e such that $l_o \vdash e$. Furthermore, if $\bar{e} \overset{*}{\longmapsto}_m{}^{cpcf} \bar{e}'$, then $e \overset{*}{\longmapsto}_m{}^{anno} e'$.*

PROOF IDEA. By a straightforward bi-simulation argument. ∎

One consequence of this proposition is that *picky* and *indy*signal still the same number of contract violations (proposition 1).

4.2 Subject Reduction

While $l \vdash e$ specifies when source programs are well-formed, the reduction semantics creates many expressions that do not satisfy these narrow constraints. For example, a well-formed program contains only monitor terms of the form $\mathtt{mon}^{k,l}_{j}(\kappa, \|e_0\|^k)$. A reduction sequence may contain programs with differently shaped monitors, however. In particular due to the reductions of higher-order dependent contracts, the monitored expressions may be applications, $\mathtt{mon}^{k,l}_{j}(\kappa, \|e_0\|^k\, e_r)$, or variables, $\mathtt{mon}^{k,l}_{j}(\kappa, x)$. Fortunately, such deviations are only temporary. In the case of the application the argument e_r is always well-formed under l and, when it is absorbed by $\|e_0\|^k$, the monitor expression is once again well-formed. In the case of the free variable we can show that x is always replaced with a value of the form $\|v_0\|^k$, which conforms to the standard form.

To formulate a subject reduction theorem, we must generalize both the judgment for well-formed programs, $l \vdash e$, and the one for well-formed contracts, $k; l; j \rhd \kappa$. First, we equip the two relations with an environment that records the label of bound variables:

$$\frac{l; \Gamma \uplus \{x : l\} \vdash e}{l; \Gamma \vdash \lambda x.e} \qquad \frac{l; \Gamma \setminus \{x\} \vdash e}{l; \Gamma \vdash \mu x.e}$$

With environments, it becomes possible to check variable occurrences in monitor terms.

Second, we add a rule for checking expressions that already have an owner:

$$\frac{k; \Gamma \vdash e}{l; \Gamma \vdash \|e\|^k}$$

While these terms show up only within monitors in source programs, they flow into many positions during evaluations. Using this new rule, we can check these cases, too.

All other rules—except those concerning monitors—propagate the environment and otherwise check expressions in the same way as the corresponding rules of the preceding section. The rule for variables ignores the environment. Variables not in monitor terms can be replaced with well-formed terms of any owner. For details, see figure 9.

The purpose of the environment is to check expressions without ownership annotations in monitor terms. Here is the key rule:

$$\frac{k; \Gamma \Vdash e \qquad \{k\}; \{l\}; j; \Gamma \rhd \kappa}{l; \Gamma \vdash \mathtt{mon}^{k,l}_{j}(\kappa, e)}$$

To check whether the wrapped expression is well-formed, it delegates to the auxiliary relation $k; \Gamma \Vdash e$. The contract is checked as before, though, with an environment.

With $k; \Gamma \Vdash e$, we can check the ordinary ownership terms but also applications and variables as introduced during reductions. For

$\boxed{j;\Gamma \vdash e}$

$$\frac{j;\Gamma \vdash e_1 \quad j;\Gamma \vdash e_2}{j;\Gamma \vdash e_1\; e_2} \qquad \frac{j;\Gamma \vdash e_1}{j;\Gamma \vdash \texttt{zero?}(e_1)}$$

$$\frac{}{j;\Gamma \vdash \texttt{error}^k} \qquad \frac{j;\Gamma \vdash e_1 \quad j;\Gamma \vdash e_2 \quad j;\Gamma \vdash e_3}{j;\Gamma \vdash \texttt{if}\; e_1\; e_2\; e_3}$$

$$\frac{j;\Gamma \vdash e_1 \quad j;\Gamma \vdash e_2}{j;\Gamma \vdash e_1 + e_2} \qquad \frac{j;\Gamma \vdash e_1 \quad j;\Gamma \vdash e_2}{j;\Gamma \vdash e_1 - e_2}$$

$$\frac{j;\Gamma \vdash e_1 \quad j;\Gamma \vdash e_2}{j;\Gamma \vdash e_1 \wedge e_2} \qquad \frac{j;\Gamma \vdash e_1 \quad j;\Gamma \vdash e_2}{j;\Gamma \vdash e_1 \vee e_2}$$

$$\frac{}{j;\Gamma \vdash \texttt{n}} \qquad \frac{}{j;\Gamma \vdash \texttt{tt}} \qquad \frac{}{j;\Gamma \vdash \texttt{ff}} \qquad \frac{}{j;\Gamma \vdash x}$$

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

$$\frac{\overline{l};\overline{k};j;\Gamma \rhd \kappa_1 \quad \overline{k};\overline{l};j;\Gamma \rhd \kappa_2}{\overline{k};\overline{l};j;\Gamma \rhd \kappa_1 \mapsto \kappa_2}$$

$$\frac{\overline{l};\overline{k} \cup \{j\};j;\Gamma \rhd \kappa_1 \quad \overline{k};\overline{l};j;\Gamma \rhd \kappa_2}{\overline{k};\overline{l};j;\Gamma \rhd \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)}$$

$$\frac{j;\Gamma \vdash e \quad \overline{k} \subseteq \overline{k'}}{\overline{k};\overline{l};j;\Gamma \rhd \lfloor \texttt{flat}(\|e\|^j) \rfloor^{\overline{k'}}}$$

Figure 9. Obligations coincide with labels on monitors (2)

function applications, the label serves as ownership label for the operator and the operand, similar to the standard application rule:

$$\frac{k;\Gamma \vdash e_1 \quad k;\Gamma \vdash e_2}{k;\Gamma \Vdash \|e_1\|^k\; e_2}$$

For free variables, the environment serves as the source of the ownership label:

$$\frac{\Gamma(x) = k}{k;\Gamma \Vdash x}$$

After all, the variable in this position is going to be replaced by a value via a function application, and the substitution is going to use a value with the specified label. Finally, for a guarded term with an ownership annotation, it suffices to check if it is well-formed with respect to the specified owner:

$$\frac{k;\Gamma \vdash e}{k;\Gamma \Vdash \|e\|^k}$$

A well-formed program in the sense of the preceding section is a well-formed program in the sense of revised judgment, too.

PROPOSITION 4. *For all e and l, $l \vdash e$ implies $l;\varnothing \vdash e$.*

PROOF IDEA. By straightforward induction on the height of the derivation $l \vdash e$. ▮

A program that is well-formed according to $l;\Gamma \vdash e$ reduces to well-formed programs. This statement holds for a contract system using the *lax* reduction rule as well as for or those using *indy*.

THEOREM 5. *Let e be a program such that $l_o;\varnothing \vdash e$. Then:*

1. *if $e \longmapsto_l e_0$, then $l_o;\varnothing \vdash e_0$;*
2. *if $e \longmapsto_i e_0$, then $l_o;\varnothing \vdash e_0$.*

PROOF. (1) We proceed by case analysis on the reduction of e:

- $E^l[\|\texttt{n}_1\|^{\vec{k}} + \|\texttt{n}_2\|^{\vec{l}}] \longmapsto lE^l[\texttt{n}]$. By assumption $l_o;\varnothing \vdash e$, for which lemma 6 implies that $l;\varnothing \vdash \|\texttt{n}_1\|^{\vec{k}} + \|\texttt{n}_2\|^{\vec{l}}$. We can use the same label to check $\texttt{n}$ via the inference rules, i.e., $l;\varnothing \vdash \texttt{n}$. Hence, $l_o;\varnothing \vdash E^l[\texttt{n}]$.
- The cases for other primitive operations are similar to the first.
- $E^l[\mu x.e] \longmapsto_l E^l[\{\|\mu x.e\|^l/x\}e]$: By assumption and lemma 6, $l;\varnothing \vdash \mu x.e$. From lemma 7 we get that $l;\varnothing \vdash \{\|\mu x.e\|^l/x\}e$ and, in turn, $l_o;\varnothing \vdash E^l[\{\|\mu x.e\|^l/x\}e]$.
- $E^l[\|\lambda x.e_0\|^{\vec{k}}\; v] \longmapsto_l E^l[\|\{\|v\|^{l\overleftarrow{k}}/x\}e_0\|^{\vec{k}}]$: Again by assumption and lemma 6, we conclude that $l;\varnothing \vdash \|\lambda x.e_0\|^{\vec{k}}\; v$ and, therefore, $l;\varnothing \vdash \|\lambda x.e_0\|^{\vec{k}}$ and $l;\varnothing \vdash v$.

  Next we distinguish two cases, depending on the length of $\vec{k}$. First assume the vector is empty. In that case, the inference rules imply $l;\{x : l\} \vdash e_0$. Combining this judgment with $l;\varnothing \vdash v$, we may conclude that $l;\varnothing \vdash \{\|v\|^l/x\}e_0$ via lemma 8. Finally from here it is easy to get $l_o;\varnothing \vdash E^l[\{\|v\|^l/x\}e_0]$, the desired conclusion.

  Second, let k_1 be the first element of $\vec{k}$. In that case, the inference rules imply $k_1;\{x : k_1\} \vdash e_0$. Since $l;\varnothing \vdash v$ still holds, we conclude again via lemma 8 that $k_1;\varnothing \vdash \{\|v\|^{l\overleftarrow{k}}/x\}e_0$. Since k_1 is the outermost element of $\vec{k}$, we finally get the desired conclusion, $l_o;\varnothing \vdash E^l[\|\{\|v\|^{l\overleftarrow{k}}/x\}e_0\|^{\vec{k}}]$.
- $E^l[\texttt{mon}_j^{k,l}(\lfloor\texttt{flat}(e_c)\rfloor^{\overline{l'}}, v)] \longmapsto_l E^l[\texttt{if}\;(e_c\; v)\; v\; \texttt{error}^k]$: The assumptions imply $l;\varnothing \vdash \texttt{mon}_j^{k,l}(\lfloor\texttt{flat}(e_c)\rfloor^{\overline{l'}}, v)$ via lemma 6 and hence, $e_c = \|e'_c\|^j$ with $j;\varnothing \vdash e'_c$. Furthermore, the same reasoning yields $v = \|v_0\|^k$ and $k;\varnothing \vdash v_0$. Since the rules for well-formed expressions imply $l;\varnothing \vdash \texttt{if}\;(e_c\; v)\; v\; \texttt{error}^k$ is well-formed, the desired conclusion follows immediately.
- $E^l[\texttt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)] \longmapsto_l E^l[\lambda x.\texttt{mon}_j^{k,l}(\kappa_2, v\,\texttt{mon}_j^{l,k}(\kappa_1, x))]$: With the usual reasoning, we get $l;\varnothing \vdash \texttt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)$, $v = \|v_0\|^k$, and $k;\varnothing \vdash v_0$. The contract check yields two pieces of knowledge:
  $$\{l\};\{k\};j;\varnothing \rhd \kappa_1$$
  and
  $$\{k\};\{l\};j;\varnothing \rhd \kappa_2$$
  From an additional application of the inference rules for well-formedness we get $k;\{x : l\} \Vdash v\,\texttt{mon}_j^{l,k}(\kappa_1, x)$ and, with the help of lemma 9, $l;\{x : l\} \vdash \texttt{mon}_j^{k,l}(\kappa_2, v\,\texttt{mon}_j^{l,k}(\kappa_1, x))$. Finally from a last application of the inference rules for well-formedness we get $l;\varnothing \vdash \lambda x.\texttt{mon}_j^{k,l}(\kappa_2, v\,\texttt{mon}_j^{l,k}(\kappa_1, x))$.
- Finally, let $e = E^l[\texttt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)]$ and observe that
  $$e \longmapsto_l E^l[\lambda x.\texttt{mon}_j^{k,l}(\{x/^c x\}\kappa_2, v\,\texttt{mon}_j^{l,k}(\kappa_1, x))].$$
  Via lemma 6 we derive $l;\varnothing \vdash \texttt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)$. Thus, $v = \|v\|^k$ with $k;\varnothing \vdash v$, but also
  $$\{l\};\{k,j\};j;\varnothing \rhd \kappa_1$$
  and
  $$\{k\};\{l\};j;\varnothing \rhd \kappa_2\ .$$
  The rest of this argument uses the same strategy as the preceding case, except that we use lemmas 10 and 14 to derive the key result, $l;\{x : l\} \vdash \texttt{mon}_j^{k,l}(\{x/^c x\}\kappa_2, v\,\texttt{mon}_j^{l,k}(\kappa_1, x))$.

(2) The proof of part (2) differs from the proof of part (1) only in the case for monitors with dependent contracts. Therefore, let $e = E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)]$ and recall that the contraction proceeds as follows:

$$e \longmapsto_i E^l[\lambda x.\mathtt{mon}_j^{k,l}(\{\mathtt{mon}_j^{l,j}(\kappa_1, x)/^c x\}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))].$$

Once again, we derive $l;\varnothing \vdash \mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)$ via lemmas 6. Hence, $v = \|v\|^k$ with $k;\varnothing \vdash v$, but also

$$\{l\};\{k,j\};j;\varnothing \rhd \kappa_1$$

and

$$\{k\};\{l\};j;\varnothing \rhd \kappa_2\ .$$

By the well-formedness, $k;\{x:l\} \Vdash v\,\mathtt{mon}_j^{l,k}(\kappa_1, x)$. Now, with the help of lemmas 12 and 14, we can derive

$$l;\{x:l\} \vdash \mathtt{mon}_j^{k,l}(\{\mathtt{mon}_j^{l,j}(\kappa_1, x)/^c x\}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))\ .$$

Thus we conclude the proof with another application of the inference rules for well-formedness. ▮

The proofs of the central lemmas depend on a series of auxiliary lemmas about the properties of well-formed terms and contracts, substitution, and contract substitution.

LEMMA 6. *If $l;\varnothing \vdash E^k[e]$ then $k;\varnothing \vdash e$.*

PROOF IDEA. By induction on the size of E^k. ▮

LEMMA 7. *If $l;\Gamma \vdash e$, $k;\Gamma \vdash e_0$, and $x \notin dom(\Gamma)$, $l;\Gamma \vdash \{\|e_0\|^k/x\}e$.*

PROOF IDEA. By induction on the height of $l;\Gamma \vdash e$. ▮

LEMMA 8. *If $l;\Gamma \uplus \{x:k\} \vdash e$, $v = \|v_0\|^k$ and $k;\Gamma \vdash v_0$, then*

$$l;\Gamma \vdash \{v/x\}e\ .$$

PROOF IDEA. By induction on the height of $l;\Gamma \uplus \{x:k\} \vdash e$. Note that $l;\Gamma \uplus \{x:k\} \vdash \mathtt{mon}_j^{k,l}(\kappa, x)$ is a base case. ▮

LEMMA 9. *If $l;\Gamma \vdash e$ and $x \notin dom(\Gamma)$, then $l;\Gamma \uplus \{x:k\} \vdash e$.*

PROOF IDEA. By induction on the height of $l;\Gamma \vdash e$. ▮

LEMMA 10. *If $\overline{k};\overline{l};j;\Gamma \rhd \kappa$ and $x \notin dom(\Gamma)$, then*

$$\overline{k};\overline{l};j;\Gamma \uplus \{x:l'\} \rhd \{x/^c x\}\kappa\ .$$

PROOF IDEA. By induction on the height of $\overline{k};\overline{l};l;\Gamma \rhd \kappa$. For the flat contracts case we employ lemma 11. ▮

LEMMA 11. *If $l;\Gamma \vdash e$ and $x \notin dom(\Gamma)$, $l;\Gamma \uplus \{x:k\} \vdash \{\|x\|^l/x\}e$.*

PROOF IDEA. First, we generalize the lemma's statement: If $l;\Gamma \vdash e$ and $x \notin dom(\Gamma)$, $l;\Gamma \uplus \{x:k\} \vdash \{\|x\|^j/x\}e$. Then we proceed by induction on the height of $l;\Gamma \vdash e$. ▮

LEMMA 12. *If $\overline{k};\overline{l};j;\Gamma \rhd \kappa$ and $k;\Gamma \uplus \{x:l\} \vdash \mathtt{mon}_j^{k,l}(\kappa', x)$, with $x \notin dom(\Gamma)$ then $\overline{k};\overline{l};j;\Gamma \uplus \{x:k\} \rhd \{\mathtt{mon}_j^{k,l}(\kappa', x)/^c x\}\kappa$*

PROOF IDEA. By induction on the height of $\overline{k};\overline{l};j;\Gamma \rhd \kappa$. For the flat contracts case we employ lemma 13. ▮

LEMMA 13. *If $l;\Gamma \vdash e$, $l;\Gamma \uplus \{x:k\} \vdash \mathtt{mon}_j^{k,l}(\kappa', x)$, & $x \notin dom(\Gamma)$, then $l;\Gamma \uplus \{x:k\} \vdash \{\|\mathtt{mon}_j^{k,l}(\kappa', x)\|^l/x\}e$*

PROOF IDEA. First, we generalize the lemma's statement as follows: If $l';\Gamma \vdash e$, $l;\Gamma \uplus \{x:k\} \vdash \mathtt{mon}_j^{k,l}(\kappa', x)$, & $x \notin dom(\Gamma)$, then $l';\Gamma \uplus \{x:k\} \vdash \{\|\mathtt{mon}_j^{k,l}(\kappa', x)\|^l/x\}e$ We proceed by induction on the height of $l';\Gamma \vdash e$. ▮

LEMMA 14. *If $\overline{k};\overline{l};j;\Gamma \rhd \kappa$, $\overline{k'} \subseteq \overline{k}$ and $\overline{l'} \subseteq \overline{l}$ then $\overline{k'};\overline{l'};j;\Gamma \rhd \kappa$.*

PROOF IDEA. By induction on the height of $\overline{k};\overline{l};j;\Gamma \rhd \kappa$. ▮

4.3 Main Theorems

When a program is well-formed, its monitors obviously satisfy the blame correctness criterion.

THEOREM 15. *$\longmapsto_l$ and $\longmapsto_i$ are blame correct.*

PROOF. This theorem is a straightforward consequence of theorem 5. To wit, the subject reduction theorem says that a program satisfies the subject including when its redex is a monitor term containing a flat contract. From the proof of the subject reduction theorem, we know that the subject implies

$$l;\varnothing \vdash \mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c)\rfloor^{\overline{l'}}, \|v\|^k)$$

The label on the context is the same as the client label by lemma 6 but we also need to know that the server label l is a member of the contract's obligations $\overline{l'}$. This has to be the case given that the monitor term is well-formed. Note how the proof is independent of the reduction semantics as long as it satisfies the subject reduction property. ▮

The *picky* contract system fails to satisfy an analogous theorem.

THEOREM 16. *There exists a program e such that $l_o \vdash e$ and*

$$e \overset{*}{\longmapsto}_p E^l[\mathtt{mon}_j^{k,l_2}(\lfloor \mathtt{flat}(e)\rfloor^{\overline{l}}, \|v\|^{l_1})]$$

but $k \neq l_1$.

PROOF. Here is one such program where $k \neq l_o$:

$$\Pi_l^3 = \mathtt{mon}_l^{k,l_o}(\kappa_l, \|\lambda h_1.h_1\ \lambda x.5\ (\lambda g.g\ 1)\|^k)\ (\lambda f.\lambda h_2.h_2\ \lambda x.6)$$

The restriction on the labels intuitively corresponds to the composition of the two different modules k and l_o through the contract κ_l. Note that Π_l^3 is a family of programs, one per label l. This label is the label of the contract monitor and, consequently, must be the owner of all embedded flat contracts. In principle, l could be the label of the client, the server, or any other non-top-level (l_o) label but in the end, our choice must obey subject reduction.

While Π_l^3 performs no interesting computation, its contract κ_l plays a critical role. To explain the contract though, it is best to start with the type of h_1:

$$[(num \to num) \to \{([num \to num] \to num) \to num\}] \to num$$

The type tells us that h_1 consumes a complex higher-order function and produces a number. Instead of plain numbers, however, we wish to deal with positive numbers only. We thus know that κ_l must have at least something like the following shape:

$$[(\mathtt{P?}_l \mapsto \mathtt{P?}_l) \mapsto \{([\mathtt{P?}_l \mapsto \mathtt{P?}_l] \mapsto \mathtt{P?}_l) \mapsto \mathtt{P?}_l\}] \mapsto \mathtt{P?}_l$$

Next we add two dependencies:

$$[(\mathtt{P?}_l \mapsto \mathtt{P?}_l) \overset{d}{\mapsto} (\lambda f.\{([\mathtt{P?}_l \mapsto \mathtt{P?}_l] \overset{d}{\mapsto} (\lambda g.\mathtt{P?}_l)) \mapsto \mathtt{P?}_l\})] \mapsto \mathtt{P?}_l$$

The two key points to notice are: (1) in this contract, g is in the scope of f and (2) while f originates in the server (wrapped expression), g originates in the client (context) and both flow into the contract.

Equipped with this informal and approximate understanding, we can now turn to the actual contract:

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

Note how κ_l^2 invokes f on a positive number and g on 0.

Now we show that l must be set to l_o in order to satisfy blame correctness. First, note that for for all $l \in \mathbb{L}$, $l_o \vdash \Pi^3_l$ if $k \neq l_o$. Second, the reduction of Π^3_l eventually checks that 1 is greater than 0, i.e., that the post-condition contract κ^2_l is satisfied:

$$\Pi^3_l \overset{*}{\longmapsto}_p E^l_0[\mathtt{mon}^{l_o,k}_l(\lfloor \mathtt{P?}_l \rfloor^{l_o}, \| \| 1 \|^l \|^l)]$$

In order for the *picky* system to satisfy the blame correctness condition, l must be equal to l_o, which means the term looks like this:

$$E^{l_o}_0[\mathtt{mon}^{l_o,k}_{l_o}(\lfloor \mathtt{P?}_{l_o} \rfloor^{l_o}, \| \| 1 \|^{l_o} \|^{l_o})]$$

Unfortunately, the next few steps of the reduction process produces a state that is inconsistent with blame correctness. Specifically, κ^2_l also checks that g's pre-condition holds for 0:

$$\begin{aligned}E^{l_o}_0[\mathtt{mon}^{l_o,k}_{l_o}(\lfloor \mathtt{P?}_{l_o} \rfloor^{l_o}, \| \| 1 \|^{l_o} \|^{l_o})] \overset{*}{\longmapsto}_p \\ E^{l_o}_1[\mathtt{mon}^{k,l_o}_{l_o}(\lfloor \mathtt{P?}_{l_o} \rfloor^{l_o}, \| \| 0 \|^{l_o} \|^{l_o})]\end{aligned}$$

This last state, however, is inconsistent with the subject because k cannot equal l_o. Indeed, the next few reduction steps result in a failed check. The contract monitor blames k, which isn't the owner l_o of the value. The *picky* system fails to assign blame properly. ∎

In essence the proof of the theorem shows that there is no correct strategy for associating (pieces of) contracts with components in a picky semantics. No matter which labeling strategy we use, a contract violation may blame a component that has no control over the faulty value.

5. Consigliere

Our model introduces two concepts that are potentially useful for practical programmers: obligations and ownership. According to our judgments for well-formed contracts and expressions, a *contractual obligation* is a static property of contract text. When confronted with complex contracts, a programmer may benefit from displaying such information in the IDE. In contrast, *ownership information* is a dynamically evolving property, and it is particularly useful in a debugger for determining the creator and current owner of values as they flow from one component to another.

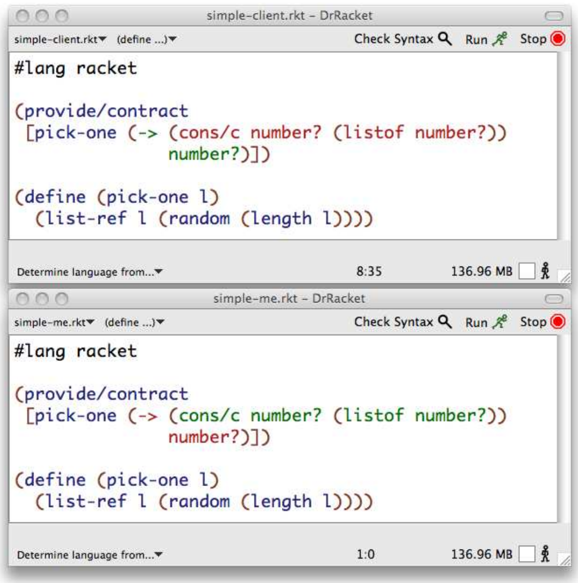
To test the practicality of displaying obligation information, we have implemented a tool that analyzes modules and contracts and annotates them with obligation information. The tool is a plug-in for DrRacket (Findler et al. 2002), our IDE for the Racket programming language. In this section, we refer to the tool as *consigliere*, though in DrRacket, it is simply a part of the *CheckSyntax* tool.

The *consigliere* tool operates on Racket modules in two different modes. In *server* mode, *consigliere* analyzes the module from the perspective of the supplier of services. It retrieves all the contracts that are attached to exported identifiers and uses

- *red* to tag the server's obligations;
- *green* to highlight the server's assumptions, i.e., the parts of the contract that the other party is responsible for; and
- *yellow* to signal that a part of the contract is both an assumption and an obligation.

In client mode, *consigliere* analyzes the module from the perspective of a consumer of services. Once it has the results, it proceeds as in server mode except that it uses red for the client's obligations and green for the client's assumptions.

For a simple example, consider the module in figure 10. The module `provide/contract` specification lists one exported function, `pick-one`. According to its contract, the function consumes a non-empty list of numbers and returns an number. A code inspection shows that the function actually returns a random element of the given list.


simple-client.rkt - DrRacket
simple-client.rkt▾ (define ...)▾
Check Syntax
Run
Stop
Determine language from...▾
8:35
136.96 MB

```
#lang racket

(provide/contract
 [pick-one (-> (cons/c number? (listof number?))
               number?)])

(define (pick-one l)
  (list-ref l (random (length l))))
```

simple-me.rkt - DrRacket
simple-me.rkt▾ (define ...)▾
Check Syntax
Run
Stop
Determine language from...▾
1:0
136.96 MB

```
#lang racket

(provide/contract
 [pick-one (-> (cons/c number? (listof number?))
               number?)])

(define (pick-one l)
  (list-ref l (random (length l))))
```

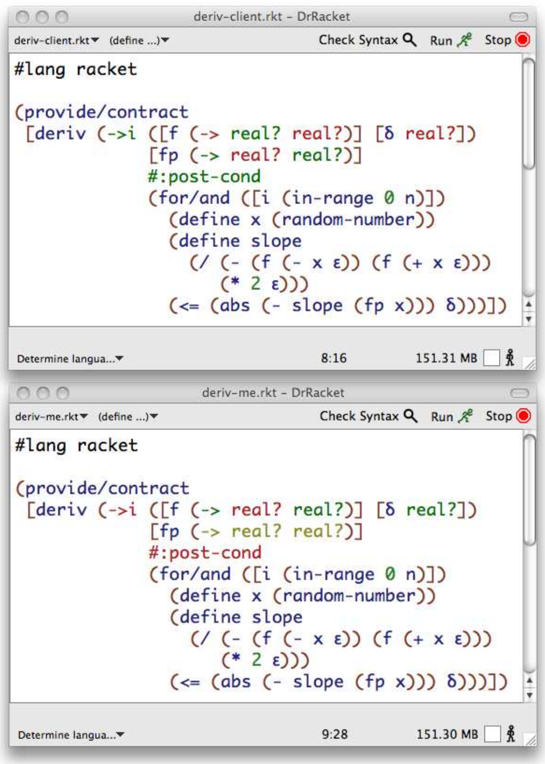
Figure 10. Obligations and assumptions for `one-of`

The top of figure 10 shows the result of using the tool in client mode. As expected, the non-empty list part of the contract is colored in red, because any user of `pick-one` is required to use a non-empty list of numbers as an argument. Also, because the client may assume that `pick-one` is a function and not some arbitrary value, *consigliere* colors the `->` contract constructor with green. The reverse reasoning explains the coloring of the lower part of figure 10, which shows the result of using *consigliere* in server mode.

For a second example, consider a `deriv` function with the contract from section 1. Its contract is significantly more complex than *pick-one*, because it comes with two higher-order components and a post-condition. Note that this contract uses the `->i` contract combinator, turning the contract provider into a party with obligations and assumptions. Here the contract provider is the server module, meaning `->i` is interpreted using the *+indy* semantics. Practically speaking, the obligations and assumptions of the server also include the obligations and assumptions of the contract provider.

The top screenshot of figure 11 shows the result of using the tool in client mode. The color assignment roughly follows the same contravariant traversal of pattern as the contract in figure 10. When it comes to the `#:post—cond` code, however, the color of the keyword indicates that the post-condition is an assumption for the client just like the other post-condition of the contract.

The lower screenshot depicts the server's obligations and assumptions, which are more interesting than the client's. The difference is due to the server's dual role as both a service provider *and* as the owner of the contract. Recall that the contract provider is responsible for meeting the pre-conditions of `f` and `fp` in the post-condition, while the server may assume that `f` and `fp` meet their post-conditions. As a result, the colors used for `f` from the server's perspective coincide with those used in the client mode. For `fp`, however, both parts are colored yellow because the obligation of the server is the contract provider's assumption and vice versa. Since the server is both the server and the contract provider party in this case, the user must be ready to treat the pre-condition and post-condition of `fp` either as an assumption or as an obligation depending on the use.



```
#lang racket

(provide/contract
 [deriv (->i ([f (-> real? real?)] [δ real?])
             [fp (-> real? 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)))])
```

```
#lang racket

(provide/contract
 [deriv (->i ([f (-> real? real?)] [δ real?])
             [fp (-> real? 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)))])
```

Figure 11. Obligations and assumptions for `deriv`

6. Related Work

Provenance is the book-keeping of origin, context and history information of data. It plays an important role for the correct and secure behavior of large software and hardware systems (Simmhan et al. 2005; Bose and Frew 2005). The study of provenance from a formal linguistic perspective is still at an initial stage (Cheney et al. 2009). Our notion of ownership is a means of keeping around some origin and context information about program values. So it can be viewed as a form of provenance. Also ownership can be used as a basis for studying formal properties of this kind of provenance. However our technique and its use to prove properties of contract systems is not related to any provenance tracking technology.

Tracking information flow in a computer system is a specialized kind of provenance. Secure information flow as pioneered by Denning (1976) is the restriction of flow of data in a computer system only between agents that have the appropriate level of clearance. There are both software and hardware techniques for imposing secure information flow. Our instrumentation of the dynamic semantics resembles techniques used for proving sound type systems that enforce secure information flow.

Zdancewic et al. (1999) introduce (program) principals as a means to prove type abstraction properties related to information flow with a syntactic proof technique. In a principal semantics, different principals "own" different components and exported values carry the principal of their component of origin. Since the principals semantics prevents reductions that involve values with different principals, a client component is obliged to use a server's functions on the server's values. In short, the semantics dynamically enforces a form of information hiding. It is now easy to see how a principals semantics supports a syntactic soundness proof for abstract types. If the interface between the server component and the client employs abstract types and if the type system soundly enforces type abstraction, stuck states become unreachable during computation.

Although both principals and ownership annotations point to the source of values and functions, the principal semantics differs substantially from our ownership semantics. Most importantly, principals may change the evaluation of a program; ownership does not. When a function from one server is applied to a value from a client or a different server, the principal semantics is stuck. In contrast, ownership annotations are simply propagated in our reduction rules; they do not affect computation. Ownership information is instead used to formulate a criteria for determining the correctness of blame assignment. We can imagine using an ownership semantics to formulate syntactic soundness proofs for type abstraction, while we do not see any advantages over the principals semantics for this application.

As explained in the introduction, Greenberg et al. (2010) study the full relationship between lax and picky contract systems on one hand and manifest contract systems on the other. For the latter, the type system propagates some of the contract constraints. While Gronski and Flanagan (2007) established a tight relationship for a world without dependent contracts, Greenberg et al. (2010) demonstrate that the full picture is rather complex. In particular, they show that as far as contract violations are concerned, manifest contracts sit strictly between lax contracts and picky contracts.

In general, this work extends our decade-old linguistic investigation of behavioral software contracts. Meyer (1988, 1991, 1992) introduced software contracts via the design of the Eiffel programming language and the creation of a contract-oriented software engineering curriculum. Since then contracts have been used both for extended static checking (Detlefs et al. 1998; Barnett et al. 2004) and runtime monitoring of higher-order programs (Findler and Felleisen 2002). Blume and McAllester (2006) introduce *picky* contract monitoring and explore a quotient model for higher-order contracts and use it to prove properties of Findler and Felleisen's higher-order contracts. Findler et al. (2004; 2006) propose an alternative view, namely, "contracts as projections," which relates contracts to Scott (1976)'s denotation model of types. Gronski and Flanagan (2007) relate Findler and Felleisen's higher-order contract to type casts. Their result motivates a type-oriented form of extended static checking (Knowles et al. 2006), which Greenberg et al. consider a manifest form of contract. Xu et al. (2009) use Blume and McAllester's ideas to develop static contract checking for Haskell using symbolic evaluation. Hinze et al. (2006) and Chitil et al. (2003) both introduce contracts to Haskell but end up with two different contract systems. The first performs eager contract checking while the second is lazy. Degen et al. (2010) compare eager and lazy contract checking for lazy languages through a series of formal properties but do not reach a definite conclusion.

Finally, in the context of JML (Leavens et al. 1999), Rudich et al. (2008) develop a method for proving the well-formedness of pure-method specifications and they discuss how their technique can benefit from automated theorem proving. The goal of this line of research is significantly different than ours. It concerns extraction of proof obligations for the verification of JML contracts internal consistency. Unfortunately, JML contracts capture only first-order properties, and it is unclear if their technique is applicable in a higher-order world. Furthermore, their static semantics lacks a formalization of the contract parties which plays an important role in our work.

7. Conclusion

This paper introduces a new semantics for dependent contracts in response to Greenberg et al. (2010)' s comparison of two alternatives. Our work acknowledges the motivation behind the *picky* contract system and turns the *lax* system into a choice for the confident programmer. Like the *picky* system, the new *indy* system protects arguments and results inside dependency assertions. In contrast to the *picky* system, each contract is treated as an independent party with its own obligations to meet.

Most importantly, we introduce a semantics that tracks value ownership and contract obligations, and we formulate the first ever correctness criterion for blame assignment. Our major theorems show that the *indy* system guarantees that contract monitors blame only components that are in control while the *picky* system fails to satisfy this intuitive correctness property for blame assignment.

Our results suggest several changes to the implementation of contracts in Racket. First, even though the *lax* semantics is blame correct, we now support the *indy* semantics for dependent contracts to ensure that only guilty parties are blamed. Second, by instantiating the contract party, we obtain flavors of an *indy* semantics that support the entire variety of Racket contracts in use: server-side contracts, client-side contracts, and contracts for ML-like modules where signatures have an independent existence.Third, the notion of well-formed contractual obligations is the basis of a DrRacket tool that can remind module programmers of their obligations and assumptions in complex, higher-order contracts.

Acknowledgments

Thanks to Matthew Flatt for suggesting how to exploit Racket's macro system to implement *consigliere*. We appreciate fruitful discussions with Riccardo Pucella, Sam Tobin-Hochstadt, Stevie Strickland and Carl Eastlund. We are grateful to the anonymous reviewers of POPL 2011 for their useful feedback.

References

H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, pages 49–69, 2004.

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

M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Survey*, 37(1):1–28, 2005.

J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: a future history. In *Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications: Onward! Session (OOPSLA Onward!)*, pages 957–964, 2009.

O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *Revised Papers of the 15th International Workshop on Implementation of Functional Languages (IFL)*, pages 1–19, 2003.

M. Degen, P. Thiemann, and S. Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *Journal of Logic and Algebraic Programming*, page to appear, 2010.

D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

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

C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*. accepted (with revisions) for publication.

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

R. B. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS)*, pages 226–241, 2006.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 48–59, 2002.

R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, Mar. 2002.

R. B. Findler, M. Felleisen, and M. Blume. An investigation of contracts as projections. Technical Report TR-2004-02, University of Chicago, Computer Science Department, 2004.

C. Flanagan. Hybrid type checking. In *Proceedings of the 33th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 245–256, 2006.

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

M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*, pages 353–364, 2010.

J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Proceedings of the 8th Symposium on Trends in Functional Programming (TFP)*, pages 54–69, 2007.

R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *In Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS)*, pages 208–235, 2006.

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/`.

G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonids, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

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

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

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

G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

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

A. Rudich, A. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In *Proceedings of the 15th International Symposium on Formal Methods (FM)*, pages 68–83, 2008.

D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.

Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, 2005.

T. S. Strickland and M. Felleisen. Contracts for first-class modules. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS)*, pages 27–38. ACM, 2009.

S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN Internation Conference on Functional Programming (ICFP)*, pages 117–128, 2010.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL)*, pages 395–407, 2008.

D. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*, pages 41–52, 2009.

S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 197–207, 1999.