On Contract Satisfaction in a Higher-Order World

CHRISTOS DIMOULAS and MATTHIAS FELLEISEN, Northeastern University, Boston, MA

Behavioral software contracts have become a popular mechanism for specifying and ensuring logical claims about a program's flow of values. While contracts for first-order functions come with a natural interpretation and are well understood, the various incarnations of higher-order contracts adopt, implicitly or explicitly, different views concerning the meaning of contract satisfaction. In this paper, we define various notions of contract satisfaction in terms of observational equivalence and compare them with each other and notions in the literature. Specifically, we introduce a small model language with higher-order contracts and use it to formalize different notions of contract satisfaction. Each of them demands that the contract parties satisfy certain observational equivalences.

1. ASSERTIONS, CONTRACTS, AND MEANING

An `assert` statement captures a logical claim concerning the values of program variables as an executable boolean expression. When the evaluation of the expression produces false, something is wrong and the evaluation is aborted. This point is easy to understand for every working programmer, and it explains the widespread use of assertions [Rosenblum 1995]. Programmers realize that assertions ensure basic program invariants; assertions assist with debugging when failures do happen; and assertions help reason about code.

Likewise, the behavioral[1] software contracts of Eiffel are assertions that govern the flow of values across component[2] interfaces [Meyer 1988; 1991; 1992b]. A programmer writes down boolean expressions, and the compiler and the execution system enforce contracts at run-time. A failure of these run-time checks triggers a report about a contract violation, pinpointing the violator and explaining the nature of the violation [Findler et al. 2001].

Over the past decade, contracts have received significant attention in the research community. Projects on extended static checking [Detlefs et al. 1998; Barnett et al. 2004] exploit contracts to statically predict program errors. Similarly, Findler and Felleisen [2002] introduce contracts

---

[1]We use the terminology of Beugnard et al. [1999] for software contracts.

[2]We use the neutral word component to refer to the pieces of a software system. Think module, class, procedure, function, or similar units of code.

---

for higher-order values (closures, objects, streams), a theme which numerous researchers have expanded on [Blume and McAllester 2006; Findler et al. 2004; Findler and Blume 2006; Gronski and Flanagan 2007; Hinze et al. 2006; Xu et al. 2009; Greenberg et al. 2010].

While Findler and Felleisen [2002] introduce higher-order contracts as a monitoring and blame-assignment problem, they do not provide an independent definition of what it means for a component to satisfy its contract. Blume and McAllester [2006] introduce a quotient model for a contract language and provide the first definition of contract satisfaction, which resembles the standard semantics for type refinements. This work also points out that the original proposal by Findler and Felleisen seems to miss contract violations. Findler et al. [2004; 2006] explain this discrepancy with a "contracts as projections" approach [Scott 1976]. Gronski and Flanagan [2007] relate Findler and Felleisen's higher-order contract to type casts. Their result serves as the basis for a type-oriented form of extended static checking [Knowles et al. 2006]. While the latter consider only one direction of the relationship, Greenberg et al. [2010] investigate both directions and explain how the addition of dependent higher-order contracts vastly complicates the picture. Xu et al. [2009] transplant Blume and McAllester's ideas to a lazy setting, though with the goal of using contracts for theorem-proving purposes. Hinze et al. [2006] and Chitil et al. [2003] add contracts to lazy languages and end up with two different systems in terms of contract satisfaction.

Our paper introduces the novel idea of using the observational equivalence relation of Plotkin's PCF [1977] to study contracts. Observational equivalence is the most fundamental model of reasoning about programs, subsuming equational (λ) calculi, operational quotient models, and equivalences based on denotational semantics [Morris 1968]. Using observational equivalence, we can define different notions of contract satisfaction as solutions of observational equations. The simplicity of the definition allows us to examine the relationships among the definitions and to illuminate various contract-related phenomena.

2. THE IDEAS

Higher-order contract systems borrow notation from type systems to express assertions for functions. Consider this example adapted from the Racket [Flatt and PLT 2010] code base:

```
(provide/contract ...
  [encode
    (-> valid-string? code-string?)] ...
```

This interface fragment specifies an export called `encode`, whose contract says it is a function from a valid string to a code string. The `valid-string?` and `code-string?` predicates can perform any arbitrary computation and are in general impossible to check statically.
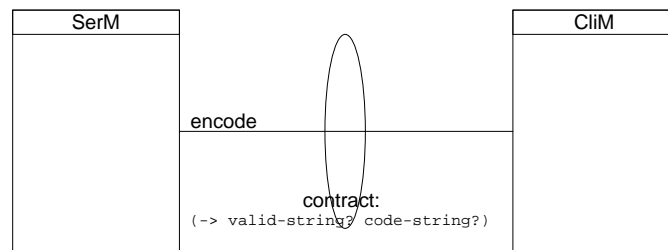
The above diagram illustrates one way to monitor such contracts. A service-providing component, dubbed `SerM`, creates a link with a service-consuming component, called `CliM`. The link represents the export of the `encode` function, and the contract is depicted as a monitoring system that watches the communication on this link.

The nature of this link is bidirectional in the sense that the client provides a string to be encoded and the server returns the encoded string of the given argument to the client. According to the

contract, the argument has to be a valid string and the result of encode has to be a code string. Pictorially speaking, we can rearrange the first diagram into a second one that shows how software contracts impose obligations on both parties.
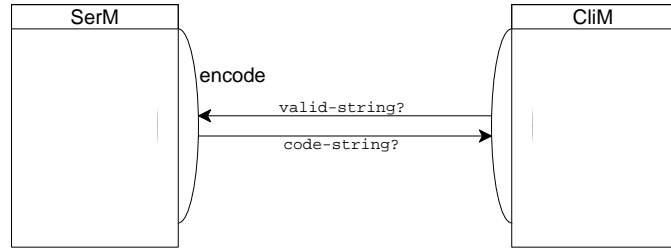
The second diagram reveals that the creators of components will not be able to program to the expectations of a common contract *unless* they can tease out the respective obligations by reading the contract text [Meyer 1992a].

While a contract system must allow the separation of a contract into obligations for the service provider and obligations for the client of the contract, the client and server pieces must contain enough information to reconstruct the entire contract. The contract system is responsible for checking that the strings that cross the channels satisfy the corresponding predicate and if not, blame the party that failed its obligations.

In a first-order world, distributing obligations is easy: the client is responsible for the function argument that is checked against the domain contract and the server is responsible for the result that is checked against the range contract. In a higher-order world, though, things get complicated. Consider the following example:

```
(provide/contract ...
  [generator
     (-> (-> nat? prime?)
         (-> valid-string? code-string?))] ...
```

This interface fragment exports a service called `generator` that, when provided with a prime numbers stream, returns an encode function similar to the one from the previous example. The contract says that `generator` expects a function from natural numbers to prime numbers and returns another function from valid strings to code strings.
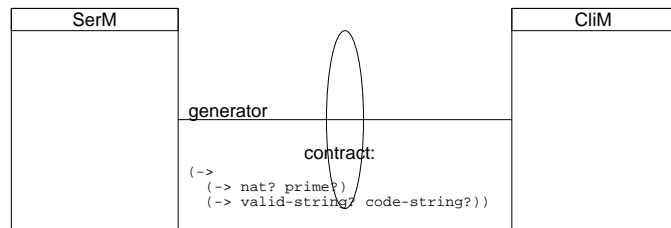
The simplistic approach to separate client and server obligations that we followed in the first-order world does not apply here because both the argument and the result are functions, i.e., new services that come with their own postconditions and preconditions. We can observe, however, that the argument values of the argument function are provided by the server and the argument values of the result function are provided by the client. With a similar reasoning about the result values of the

argument and result functions, we conclude that the following diagram depicts a realistic separation of the obligations of the server and the client for the higher-order example.
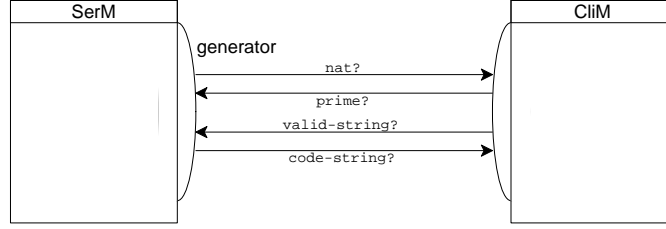
Our separation analysis—an informal reformulation of the contracts as projections perspective [Findler et al. 2004; Findler and Blume 2006]—implies one obvious criteria for what it means for a component such as *SerM* or *CliM* to meet the obligations of a contract. Let $\mathrm{mon}(c,m)$ be the expression that uses contract c to monitor the flow of values from component m to its clients and back. Then the separation suggests that m satisfies c if monitoring the component with c is observationally equivalent to monitoring just the obligations for the other party:

$$\mathrm{mon}(c, SerM) \simeq \mathrm{mon}(obl^{Client}[\![c]\!], SerM) \quad (\dagger)$$
$$\mathrm{mon}(c, CliM) \simeq \mathrm{mon}(obl^{Server}[\![c]\!], CliM) \quad (\ddagger)$$

After all, the code for the server component is responsible for meeting the obligations on the server, and if the server-side contract is removable without visible effect in any context, the code must be correct as far as those obligations are concerned (eq. ($\dagger$)). Similarly, satisfying a client contract must be equivalent to monitoring just the server contract (eq. ($\ddagger$)).

Of course, a software engineer may argue that it is the composition of the server component and the client component, i.e., the composite program, that must meet the contract and that considering components in isolation is futile. Pragmatically speaking, this engineering perspective says errors are only errors when there is evidence of their existence. In a different but related context de Alfaro and Henzinger [2001] label this approach "optimistic."

In order to formulate this second notion of contract satisfaction, we need a notion of component composition. Lacking a concrete syntax for now, we write $SerM \circ_m CliM$ for the composition of server component *SerM* with client *CliM*. Equipped with this notation, we say that a server component *SerM* optimistically satisfies a contract c in composition with client *CliM* if

$$\mathrm{mon}(c, SerM) \circ_m CliM \simeq \mathrm{mon}(obl^{Client}[\![c]\!], SerM) \circ_m CliM$$

and conversely, the *CliM* component satisfies c in conjunction with server *SerM* if

$$\mathrm{mon}(c, CliM) \circ_m SerM \simeq \mathrm{mon}(obl^{Server}[\![c]\!], CliM) \circ_m SerM$$

Note how this second notion of contract satisfaction uses ternary relations among three items: server, client, and contract. In contrast, the first notion is expressed with just binary relations between components and contracts. The second notion describes contract satisfaction for a composition of a client and a server while the first notion defines contract satisfaction for a server or a client in isolation. In addition, the first notion implies the second while the reverse does not hold.

The rest of the paper translates these informal ideas about contract satisfaction into the concrete syntax of a minimal programming language, specifically, a call-by-value variant of PCF [Plotkin 1977] equipped with contracts, which we call CPCF. Section 3 recalls the relevant bits about PCF, extends it with contracts, and introduces observational equivalence; section 4 shows how to tease apart higher-order contracts. In this setting, we explore formally the two notions of contract satisfaction of this section: sections 5 and 6 and their relation. Lastly, in section 7 we develop a meaning for contract checking à la Chitil et al. [2003].

3. CONTRACT PCF

PCF is a programming language derived from the simply-typed λ calculus [Plotkin 1977]:

$$
\begin{aligned}
\textbf{Types} \quad \tau &::= o \mid \tau \to \tau \\
o &::= num \mid bool \\
\textbf{Terms} \quad e &::= 0 \mid -1 \mid +1 \mid \ldots \mid \texttt{tt} \mid \texttt{ff} \mid (\lambda x : \tau.e) \mid x \mid e+e \mid e-e \\
&\mid e \wedge e \mid e \vee e \mid \texttt{zero?}(e) \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \mu x : \tau.e \mid (e\ e)
\end{aligned}
$$

We assume a simple[3] type system for the language; the details are omitted and types are omitted in examples unless needed for disambiguation. As usual, programs are closed terms and contexts C are terms with holes in the position of an e.

Next we equip the language with a by-value semantics. While *any* formalization of the semantics should work, we rely on reduction semantics [Plotkin 1975; Felleisen et al. 2009]. For conciseness, we focus on the absolutely necessary elements and skip the rest of the standard formalization. A value is a numeric constant, a boolean, or a function:

$$
\begin{aligned}
\textbf{Values} \quad v &::= 0 \mid -1 \mid +1 \mid \ldots \mid \texttt{tt} \mid \texttt{ff} \mid (\lambda x : \tau.e) \\
\textbf{Ev. Ctxt.} \quad F &::= [\,] \mid F+e \mid v+F \mid F-e \mid v-F \mid F \wedge e \mid F \vee e \\
&\mid \texttt{zero?}(F) \mid \texttt{if } F \texttt{ then } e \texttt{ else } e \mid (F\ e) \mid (v\ F) \mid
\end{aligned}
$$

An evaluation context F is a context with the hole in a leftmost-outermost position. The notions of reduction generate the compatible closure with evaluation contexts, which is the standard reduction relation of the CPCF calculus. For example,

$$
\begin{aligned}
F[((\lambda x : \tau.e)\ v)] &\longmapsto F[\{v/x\}e] \\
F[\mu x : \tau.e] &\longmapsto F[\{\mu x : \tau.e/x\}e]
\end{aligned}
$$

are the standard reduction relations for function application and recursion.

3.1. CPCF: PCF with Contracts

Adding contracts to PCF requires extensions to the language of types and terms:

$$
\textbf{Types} \quad \tau ::= \ldots \mid con(\tau)
$$

$$
\begin{aligned}
\textbf{Terms} \quad e &::= \ldots \mid \texttt{mon}^l(\kappa, e) \mid {}^l\texttt{error}_p \\
\kappa &::= \texttt{flat}(e) \mid \kappa \mapsto \kappa \mid \kappa \xmapsto{d} (\lambda x : \tau.\kappa) \\
p &::= \texttt{s} \mid \texttt{c}
\end{aligned}
$$

where $l \in \mathbb{N}$. The result is CPCF, PCF with contracts.

CPCF integrates contracts and terms with a contract monitor form: $\texttt{mon}^l(\kappa, e)$. Its purpose is to observe the values that flow from e to its context and back and to make sure they satisfy contract κ. The label on the statement uniquely identifies the monitor[4] so that in the case of a contract violation the monitor can issue an error report that pinpoints the problem source.

---

[3]We conjecture that type polymorphism and type inference are entirely orthogonal to our goal.
[4]In an implementation, this label *is* the source location of the monitor.

Here are the typing rules for this language extension:

$$\frac{}{\Gamma \vdash {}^l \mathtt{error}_s : \tau} \qquad \frac{\Gamma \vdash \kappa : con(\tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{mon}^l(\kappa, e) : \tau} \qquad \frac{\Gamma \vdash e : o \to bool}{\Gamma \vdash \mathtt{flat}(e) : con(o)}$$

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

$$\frac{\Gamma \vdash \kappa_1 : con(\tau_1) \quad \Gamma; x : \tau_1 \vdash \kappa_2 : con(\tau_2)}{\Gamma \vdash \kappa_1 \overset{d}{\mapsto} (\lambda x : \tau_1.\kappa_2) : con(\tau_1 \to \tau_2)}$$

Errors are available at all types. A monitoring term demands that the first position is a contract for a certain type and that the second term is a term at that type. Next, a flat contract is just a predicate on base types. In contrast, a functional contract combines two existing contracts and creates a contract for a function type, restricting both its domain type and range type. Because contracts can relate arguments and results, CPCF also comes with dependent functional contracts, which supply the function argument to the range contract so that it can monitor the desired relationship.

A program is a term of ground type. Furthermore, we assume that a *program contains each label at most once*. An appropriate set of rules for checking uniqueness of labels is straightforward and omitted. Of course, as the program is reduced to a value, labels are copied and occur multiple times in intermediate execution states.

Our syntax ensures that contracts show up in $\mathtt{mon}^l(\kappa, e)$ terms only. Intuitively, this term enforces the contract κ between the "server component" e and the context of the term, which plays the role of a "client component." For flat, first-order values, this just means checking that the value of e has the desired property. For functions, it requires checking the arguments to the function and its result. Finally, for higher-order functional values, the monitoring requires additional checks on the use of functions as arguments. If this monitoring process discovers that a predicate from an embedded flat contract does not hold, it raises the exception ${}^l \mathtt{error}_p$. The label l identifies the original location of the contract and the subscript p indicates which of the two parties has violated the contract: $\mathtt{s}$ means e broke the contract and $\mathtt{c}$ blames the context.

Our formal semantics adapts the system of Findler and Felleisen [2002] to the syntax of CPCF. We start with some additional terms and evaluation contexts:

$$\begin{array}{lll} \textbf{Terms} & e ::= \ldots \mid \mathtt{mon}^l_p(\kappa, e) & \\ \textbf{Ev. Ctxt.} & F ::= \ldots \mid \mathtt{mon}^l_p(\kappa, F) & \end{array} \qquad p \in \{\mathtt{s}, \mathtt{c}\}$$

The additional contract terms carry a "blame subscript", which determines whose fault it is when a run-time check fails. The new terms satisfy appropriate typing constraints.

Here are the relevant notions of reduction:

$$\frac{}{F[\cdots] \longmapsto F[\cdots]}$$

$$\begin{array}{llll} \mathtt{mon}^l(\kappa, v) & \cdot & \mathtt{mon}^l_s(\kappa, v) & [mon] \\ \mathtt{mon}^l_p(\mathtt{flat}(e), v) & \cdot & \mathtt{if}\ (e\ v)\ \mathtt{then}\ v\ \mathtt{else}\ {}^l \mathtt{error}_p & [flat] \\ \mathtt{mon}^l_p(\kappa_1 \mapsto \kappa_2, v) & \cdot & (\lambda x.\mathtt{mon}^l_p(\kappa_2, (v\ \mathtt{mon}^l_{\bar{p}}(\kappa_1, x)))) & [ho] \\ \mathtt{mon}^l_p(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) & \cdot & (\lambda x.\mathtt{mon}^l_p(\kappa_2, (v\ \mathtt{mon}^l_{\bar{p}}(\kappa_1, x)))) & [lax] \end{array}$$

While the first relation rewrites a monitoring term to one with a blame subscript, the remaining three use it to enforce contracts:

(1) A flat contract applies the given predicate to the given value. If this reduces to $\mathtt{ff}$, the exception blames party p at location l.

(2) A contract for a function v rewrites into a function that checks the domain contract for the parameter of this function and the range contract for the result of applying v to it.

(3) A dependent contract for a function works like a functional contract, but it substitutes the argument for v also into the range contract. [5]

The two last rules use $\bar{p}$, which denotes the complement of p, meaning if $p = \mathtt{s}$ then $\bar{p} = \mathtt{c}$ and vice versa. For first-order functions, this implies that the monitoring system blames the context if something is wrong with the argument and the function body if the result does not pass the range check. In general, for higher-order functions, negative positions in the contract come with a client blame label and positive positions with a server label.[6]

As above, these notions of reduction generate the compatible closure with evaluation contexts, which in turn represents the standard reduction relation of the CPCF calculus. In addition, we need this relation:

$$F[\,{}^l\mathtt{error}_p] \longmapsto {}^l\mathtt{error}_p$$

That is, if an error shows up in the hole of an evaluation context, the program stops.

From here we get a semantics, i.e., a function mapping programs to answers:

$$\textbf{Answers} \quad a ::= v \mid {}^l\mathtt{error}_p\,.$$

We say that a program converges $(e \Downarrow a)$ iff $e \longmapsto^* a$; if the answer does not matter, we write $(e \Downarrow)$. Conversely, a program diverges $(e \Uparrow)$ iff for all e', there exists e'' such that when $e \longmapsto^* e'$, $e' \longmapsto e''$.

3.2. Pragmatics of Contracts

Before delving into our explorations of contract satisfaction, we illustrate CPCF with a few examples. While this purely expression-oriented language lacks modules and contract boundaries, it can mimic them with terms in the role of server components and contexts for client components. To link a server component e with a client component C, we just plug the former into the hole of the latter: $C[e]$. Or, if the component composition is to respect a contract κ, we fill the hole with a monitoring term:

$$C[\mathtt{mon}^l(\kappa, e)].$$

Monitoring means that the predicate portions of the contract are applied to the appropriate base-type arguments and results of functions. When these run-time checks fail, errors are raised that indicate which party violates which (uniquely labeled) contract. By implication, the monitoring semantics of Findler and Felleisen [2002] says that a component does not satisfy a contract if the monitoring semantics signals an appropriately labeled error.

---

[5]Some authors [Blume and McAllester 2006; Hinze et al. 2006; Xu et al. 2009] use a different reduction rule for dependent contracts:

$$\mathtt{mon}^l_p(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) \longmapsto_{picky} (\lambda x.\mathtt{mon}^l_p(\{\mathtt{mon}^l_{\bar{p}}(\kappa_1, x)/x\}\kappa_2, (v\,\mathtt{mon}^l_{\bar{p}}(\kappa_1, x)))) \quad [picky]$$

In this variant of the contract monitoring system, the argument of a function guarded by a dependent function contract is passed to the postcondition contract wrapped with the precondition contract. Doing so makes the contract monitoring system enforce contracts while evaluating contract predicates and may catch additional violations of the assertions by the predicates.

This paper sticks to the *lax* reduction rules for dependent contracts, and the contract satisfaction criteria in the following sections are capturing the philosophy of a *lax* system. None of the theorems of the paper fails for the *picky* semantics, however. This is not a surprise as the notion of contract satisfaction is much more relaxed in a *lax* world than in a *picky* world.

[6]The terms "negative" and "positive" are used for contracts as if they were types.

The following example is loosely adapted from Blume and McAllester [2006]: [7]

SerM	κ	*CliM*
$\lambda f.\text{if zero?}(f\,1)\text{ then }f\text{ else }f$	κ_0	$(([\,]\,(\lambda y.y))\,2)$

where $\kappa_0 = (\text{flat}(O?) \mapsto \text{flat}(O?)) \mapsto (Any \mapsto Any)$
and $O?$ checks if a number is odd and Any is $\text{flat}(\lambda x.\text{tt})$

The example shows that higher-order contract monitoring may yield unexpected results. The surprise is due to the $Any \mapsto Any$ result contract, which any function should satisfy, yet the monitoring system seems to disagree. Many programmers would expect that since the server does not explicitly abuse its argument and promises little about its result, the server cannot be held responsible for anything. But, as the client receives the result of the server and applies it to 2, the monitoring system recognizes a contract failure and raises an exception blaming the server. As far as the latter is concerned, it must ensure that the identity function that flows from the client to the server is always applied to odd numbers. Even though the server hands this function back to the client as part of the result, it remains responsible for the function's domain. In other words, the server lets its argument flow unprotected into the context and therefore takes the blame for failing to keep its promise to its client that it will always apply its argument to odd numbers.

Similarly, Xu et al. [2009, p. 2] describe a situation with two contract violations where the monitor discovers only one:

SerM	κ	*CliM*
$\lambda f.(f\,1) - 1$	$(Any \mapsto \text{flat}(PZ?)) \mapsto \text{flat}(PZ?)$	$([\,]\,(\lambda x.x - 1))$

where $PZ?$ checks that a number is ≥ 0

First, they agree with the monitoring semantics in that this example should signal an error blaming the server for producing -1, even though its result contract promises numbers that are greater or equal to 0. Second, they claim a violation on the client side, namely, that the function $(\lambda x.x - 1)$ does not map arbitrary integers to positive ones.

Here is a modification of the same example without the distracting server-side violation:

SerM	κ	*CliM*
$\lambda f.(f\,1) + 1$	$(Any \mapsto \text{flat}(PZ?)) \mapsto \text{flat}(PZ?)$	$([\,]\,(\lambda x.x - 1))$

Xu et al. [2009] insist that the client violates the contract when it hands the server a function that returns the value -1 when given the value 0. They justify their claim with an explicit reference to the SAGE contract checking approach [Knowles et al. 2006; Gronski and Flanagan 2007]. The CPCF monitoring semantics, however, reduces this program to 1 and does not signal a violation.

Finally, Xu et al.'s addition of contracts to Haskell raises the question of how to deal with laziness. One obvious question is whether the monitoring system should signal an error if an element in a lazy stream fails its contract yet the program never evaluates the element. Laziness, though, also highlights the problem of contracts that force more stream elements than the function they are to protect, something that plain CPCF exhibits too.

Although CPCF does not come with lazy streams, we can mimic lazy streams with functions on the natural numbers. In this context, consider one last example:

SerM	κ	*CliM*
$\lambda s.(\text{fst }s)$	κ_1	$([\,]\,(\lambda i. - i))$

where $\kappa_1 = (Any \mapsto Any) \overset{d}{\mapsto} \lambda s.\text{flat}(\lambda r.(PZ?\,(\text{fst }s)) \wedge (PZ?\,(\text{fst }(\text{rst }s))))$
and fst is $\lambda s.(s\,0)$ and rst is $\lambda s.\lambda i.(s\,(i+1))$

---

[7] The differences between the original example and the one given here are due to the statically typed nature of our language. More precisely Blume and McAllester [2006] use an untyped Any predicate that accepts any value as the postcondition contract of the server instead of the $Any \mapsto Any$ function contract. We follow the analysis of Findler and Blume [2006] who show that, in this setting, the most interesting interpretation of Any is $Any \mapsto Any$.

While this contract forces the first *two* elements of the stream and triggers a contract violation error because the second one is less than 0, the function itself uses only *one* element and as such would not have raised an error. Because the contract is overly eager, monitoring the contract adds behavior above and beyond the function itself. This implies that in a lazy setting a dependent contract should not explore the function argument more than the function does. Again, the literature differs on this view, with Xu et al. [2009] suggesting that the above example should trigger an error and with Chitil et al. [2003] arguing that the contract should be ignored for being too strict. Degen et al. [2010] study the different approaches to contract checking for lazy languages: they formulate and implement an eager and a delayed contract checking system for a lazy language and compare them using monadic semantics. On this basis, they show that eager contract checking does not satisfy contract idempotence unless severe restrictions are imposed on contracts. They also demonstrate that even though lazy contract checking does not demand such restrictions, it gives rise to weird phenomena because less contract predicates than expected are checked. We show in section 7 that even if we accept the idea of lazy[8] contract checking, there is a still wide spectrum of possibilities. For the rest of the paper we refer to this kind of contract checking as *s*hy to avoid associating this idea exclusively with lazy languages.[9]

3.3. CPCF and Observational Equivalence

At this point we are ready to introduce the key relation of our framework: observational equivalence. We consider a term e_1 observationally equivalent to a term e_2 if the two are indistinguishable with respect to answers for all program contexts:

Definition 3.1. (Observational Equivalence in CPCF, $\simeq$) Two terms, e_1 and e_2, are observational equivalent in CPCF, $e_1 \simeq e_2$, if for all contexts C,

— $C[e_1] \Downarrow^l \text{error}_p$ if and only if $C[e_2] \Downarrow^l \text{error}_p$, and
— $C[e_1] \Downarrow v$ if and only if $C[e_2] \Downarrow v'$.

That is, if either program converges to a contract error, the other one must reduce to the same contract error. Otherwise, they are equivalent if both terminate or both diverge. For our proofs, we also need a value-termination preorder on terms:

Definition 3.2. (Value Approximation in CPCF, $\preceq$). A term e_1 is below a term e_2 in the term value-termination preorder in CPCF, written $e_1 \preceq e_2$, if for all contexts C, $C[e_1] \Downarrow v$ implies $C[e_2] \Downarrow v'$.

While in the context of pure PCF, this preorder coincides with observational approximation (whose symmetric closure is observational equivalence), in CPCF the two relations differ due the presence of errors.

Proving contextual equivalence and value approximation for CPCF is at the core of the development of our theory. Koutavas [2008, ch. 2] introduces a technique to prove contextual equivalence for a by-value language based on the untyped λ calculus, where observational equivalence is defined via co-termination. The additions of contracts, errors, and types to the language are conservative extensions. Thus their result can be also transferred to CPCF and we use it for establishing our theorems.[10]

To keep the paper self-contained, we briefly present the technique of Koutavas [2008, ch. 2] as adapted to CPCF. The goal of the method is to define a binary relation R that includes the two terms in question and to demonstrate that R satisfies ADEQUACY. If so, its COMPATIBLE closure,

---

[8]This notion of lazy contract checking is distinct from the one of Findler et al. [2007]. For them, lazy checking is a device for checking contracts on data structures only when needed.

[9]All linguistic phenomena observed in lazy languages are also present in eager languages but with a type one order higher. Consider for instance how lazy streams can be represented as first-order functions.

[10]Our results are not tied to the method of Koutavas. Any method for proving contextual equivalence, for instance the one of Ahmed [2006], could be used instead.

called R^{ctx}, is a bi-simulation and as such is closed under evaluation and relates only terms that co-terminate. From this we conclude that the bi-simulation is identical to observational equivalence [Morris 1968] and that in particular the two terms in question are observationally equivalent.

Definition 3.3. (Compatible Closure) If R is a binary relation over closed terms of the same type, R^{ctx} is defined as $C[e] R^{ctx} C[e']$ if $\vdash e : \tau, \vdash e' : \tau$, and $e R e'$.

Definition 3.4. (Adequate Relations) A binary relation R on closed terms of the same type is adequate if for all closed expression e, e' such that $e R^{ctx} e'$ if $e \Downarrow a$ for some answer a there exists some answer a' such that $e' \Downarrow a'$ and $a R^{ctx} a'$, and if $e' \Downarrow a'$ there exists a such that $e \Downarrow a$ and $a R^{ctx} a'$.

In general, we prove adequacy for R by showing that $IH_R(k)$ if $IH_R(k-1)$ and that $IH_{R^{-1}}(k)$ if $IH_{R^{-1}}(k-1)$, where R^{-1} is the inverse of R. The claim $IH_R(k)$ is the following induction hypothesis:

> $IH_R(k)$: for all $i \leq k$, if $e R^{ctx} e'$ and $e \overset{i}{\longmapsto} a$, then there exists a' such that $e' \overset{*}{\longmapsto} a'$ and $\underline{a R^{ctx} a'}$.

The key insight is that this induction can be simplified. A proof based on the induction hypothesis demands considering all appropriate $e R^{ctx} e'$. The following theorem allows us to use smaller proof obligations and restrict the proof only to $e R e'$.

THEOREM 3.5. *(Simplified Adequacy for CPCF) A binary relation R on closed terms of the same type is adequate iff for all k,*

— *for all $i \leq k$, if $\vdash e : o, \vdash e' : o$, $e R e'$, $e \overset{i}{\longmapsto} a_r$ then there exists a'_r such that $e' \overset{*}{\longmapsto} a'_r$ and $a_r R^{ctx} a'_r$;*
— *for all $i \leq k$, if $\vdash e : \tau_1 \rightarrow \tau_2, \vdash e' : \tau_1 \rightarrow \tau_2, \vdash v_a : \tau_1, \vdash v'_a : \tau_2, e R e', v_a R^{ctx} v'_a$ and $(e\ v_a) \overset{i}{\longmapsto} a_r$*
  then there exists a'_r such that $(e'\ v'_a) \overset{*}{\longmapsto} a'_r$ and $a_r R^{ctx} a'_r$;
— *the above two claims also hold for R^{-1}*

when $IH_R(k-1)$ is assumed.

PROOF SKETCH. The proof is a mild modification of the corresponding proof of Koutavas [2008, ch. 2, sec. 2.5]. ∎

The method is usually applied as follows:

(1) create an R that relates the terms in question;
(2) execute the computations described in the theorem;
(3) look for a synchronization point after one or more reduction steps;
(4) and apply the induction hypothesis $IH_R(k-1)$.

If this fails, the "stuck state" tends to suggest additional elements to be added to R. After modifying R appropriately, repeat steps 2 and 3 for the newly added elements of R.

The same method can also be adapted to prove that a CPCF term value approximates another one.

Definition 3.6. (Adequate Approximation Relations) A binary relation R on terms of the same type is adequate iff for all terms e, e' such that $e R^{ctx} e'$ if $e \Downarrow v$ there exists v' such that $e' \Downarrow v'$ and $v R^{ctx} v'$.

Based on the definition of adequacy we give the appropriate induction hypothesis:

> $IH_R(k)$: for all $i \leq k$, if $e R^{ctx} e'$ and $e \overset{i}{\longmapsto} v$, then there exists v' such that $e' \overset{*}{\longmapsto} v'$ and $\underline{v R^{ctx} v'}$.

Due to the directional nature of the approximation relation, it suffices to show that if $IH_R(k-1)$ then $IH_R(k)$.

As with contextual equivalence, we can simplify the induction hypothesis.

THEOREM 3.7. *(Simplified Adequacy for Approximation) A binary relation R on closed terms of the same type is adequate iff for all k,*

— *for all $i \leq k$, if $\vdash e : o$, $\vdash e' : o$, $e\,R\,e'$, $e \overset{i}{\longmapsto} v_r$ then there exists v'_r such that $e' \overset{*}{\longmapsto} v'_r$ and $v_r\,R^{ctx}\,v'_r$;*

— *for all $i \leq k$, if $\vdash e : \tau_1 \to \tau_2$, $\vdash e' : \tau_1 \to \tau_2$, $\vdash v_a : \tau_1$, $\vdash v'_a : \tau_2$, $e\,R\,e'$, $v_a\,R^{ctx}\,v'_a$ and $(e\,v_a) \overset{i}{\longmapsto} v_r$ then there exists v'_r such that $(e'\,v'_a) \overset{*}{\longmapsto} v'_r$ and $v_r\,R^{ctx}\,v'_r$*

when $IH_R(k-1)$ is assumed.

PROOF SKETCH. The proof is a mild modification of the forward direction of the corresponding proof of Koutavas [2008, ch. 2, sec. 2.5]. ∎

To demonstrate the proof method, we prove a simple approximation for CPCF. With the value-termination preorder, we can confirm that monitoring a contract for a value changes the observable behavior in a predictable manner. That is, if the program with a contract terminates normally, the program without contracts produces the same value; otherwise the effects of the contract (divergence, extra errors) or a contract error may show up.

PROPOSITION 3.8. $\mathtt{mon}^l(\kappa, e) \preceq e$

PROOF SKETCH. According to the method, it suffices to define a binary relation R on terms that relates at least the two terms from the proposition's statement. Then we show that the following proof obligation $P(k)$ holds for all k:

$\underline{P(k)}$: if for all $i \leq k$, if $e\,R^{ctx}\,e'$ and $e \overset{i}{\longmapsto} v$, then there exists v' such that $e' \overset{*}{\longmapsto} v'$ and $v\,R^{ctx}\,v'$.

where R^{ctx} is the compatible closure of R.

Here is a relation R, generated from the proposition's claim:

$$\frac{e\,R^{ctx}\,e'}{\mathtt{mon}^l(\kappa, e)\,R\,e'} \qquad \frac{e\,R^{ctx}\,e'}{\mathtt{mon}^l_p(\kappa, e)\,R\,e'}$$

The standard way to show that the proof obligation holds is to proceed by induction on k. By theorem 3.7, it is sufficient to consider only pairs e, e' such that $e\,R\,e'$. We show one case of the inductive step of the proof for the second clause of theorem 3.7. In this case, we focus on terms $\mathtt{mon}^l_p(\kappa_1 \mapsto \kappa_2, e_o)$ and e'_o such that $\mathtt{mon}^l_p(\kappa_1 \mapsto \kappa_2, e_o)\,R\,e'_o$.

Based on the antecedent of the second clause of 3.7, we pick v_a and v'_a, calculate the terms $(\mathtt{mon}^l_p(\kappa_1 \mapsto \kappa_2, e)\,v_a)$ and $(e'_o\,v'_a)$ and show that if the first term converges to a value then the second term also converges to a value and the two values are related under R^{ctx}.

We focus on the sub-case where $e_o \overset{*}{\longmapsto} v_o$ and we compute for the first term:

$$(\mathtt{mon}^l_p(\kappa_1 \mapsto \kappa_2, e)\,v_a) \overset{\geq n+1}{\longmapsto} \mathtt{mon}^l_p(\kappa_2, (v_o\,\mathtt{mon}^l_{\bar{p}}(\kappa_1, v_a)))$$

By the definition of R, we know that since $\mathtt{mon}^l_p(\kappa_1 \mapsto \kappa_2, e_o)\,R\,e'_o$, $e_o\,R^{ctx}\,e'_o$ holds too. By applying the induction hypothesis on terms e_o, e'_o we conclude that if $e_o \overset{*}{\longmapsto} v_o$ then $e'_o \overset{*}{\longmapsto} v'_o$ and $v_o\,R^{ctx}\,v'_o$.

Thus, we can now compute for $(e'_o\,v'_a)$:

$$(e'_o\,v'_a) \overset{*}{\longmapsto} (v'_o\,v'_a)$$

We know that $v_o\,R^{ctx}\,v'_o$. By the definition of R, we conclude that $\mathtt{mon}^l_{\bar{p}}(\kappa_1, v_a)\,R^{ctx}\,v'_a$. and also $\mathtt{mon}^l_p(\kappa_2, (v_o\,\mathtt{mon}^l_{-p}(\kappa_1, v_a)))\,R^{ctx}\,(v'_o\,v'_a)$. Now the desired conclusion that $(v'_o\,v'_a)$ converges to a value if $\mathtt{mon}^l_p(\kappa_2, (v_o\,\mathtt{mon}^l_{\bar{p}}(\kappa_1, v_a)))$ converges to a value and the two values are related under R^{ctx} follows from another use of the induction hypothesis. ∎

4. SEPARATING CONTRACTS

Every contract has two parties: a client and a server. Like in real life, conformance with a contract is only possible if a contract party can read a contract's text before executing it and extract its obligations.

In CPCF, contracts are trees with flat contracts at the leaves. Each of these leaves applies to either the server or the client component of a contract monitoring boundary, but not both. To tease out the server or client obligations from a contract, it suffices to leave the tree structure intact and to replace the inapplicable flat contracts with predicates that accept all values. The following definition expresses this idea as a meta-function $\mathsf{obl}^p[\![\cdots]\!]$ from contracts to contracts.

Definition 4.1. (Contract Splitting, $\mathsf{obl}^p[\![\cdots]\!]$)

$$\mathsf{obl}^p[\![\kappa]\!] \;=\; \begin{cases} \mathsf{obl}^{\mathsf{s}}_{\mathsf{s}}[\![\kappa]\!] & \text{if } p = \mathsf{s} \\ \mathsf{obl}^{\mathsf{c}}_{\mathsf{s}}[\![\kappa]\!] & \text{if } p = \mathsf{c} \end{cases}$$

$$\text{where} \qquad \begin{aligned}
\mathsf{obl}^{p_1}_{p_2}[\![\kappa_1 \mapsto \kappa_2]\!] &= \mathsf{obl}^{p_1}_{\bar{p}_2}[\![\kappa_1]\!] \mapsto \mathsf{obl}^{p_1}_{p_2}[\![\kappa_2]\!] \\
\mathsf{obl}^{p_1}_{p_2}[\![\kappa_1 \xmapsto{d} (\lambda x.\kappa_2)]\!] &= \mathsf{obl}^{p_1}_{\bar{p}_2}[\![\kappa_1]\!] \xmapsto{d} (\lambda x.\mathsf{obl}^{p_1}_{p_2}[\![\kappa_2]\!]) \\
\mathsf{obl}^{p}_{p}[\![\mathtt{flat}(e)]\!] &= \mathtt{flat}(e) \\
\mathsf{obl}^{p}_{\bar{p}}[\![\mathtt{flat}(e)]\!] &= \mathtt{flat}(\lambda x.\mathtt{tt}) \\
p &\in \{\mathsf{s},\mathsf{c}\}
\end{aligned}$$

The converse is to take two homologous contract trees and compose them.

Definition 4.2. (Contract composition, $\circ$)

$$\begin{aligned}
\mathtt{flat}(e) \circ \mathtt{flat}(e') &= \mathtt{flat}((\lambda x.(e'\,x) \wedge (e\,x))) \\
\kappa_1 \mapsto \kappa'_1 \circ \kappa_2 \mapsto \kappa'_2 &= \kappa_1 \circ \kappa_2 \mapsto \kappa'_1 \circ \kappa'_2 \\
\kappa_1 \xmapsto{d} (\lambda x.\kappa'_1) \circ \kappa_2 \xmapsto{d} (\lambda x.\kappa'_2) &= \kappa_1 \circ \kappa_2 \xmapsto{d} (\lambda x.\kappa'_1 \circ \kappa'_2)
\end{aligned}$$

Special care is taken to make sure that the dependent variables of the two parts are identical.

The composition of the server and client obligations of a contract is observationally equivalent to the contract.

PROPOSITION 4.3. $\mathtt{mon}^l(\kappa, e) \simeq \mathtt{mon}^l(\mathsf{obl}^p[\![\kappa]\!] \circ \mathsf{obl}^{\bar{p}}[\![\kappa]\!], e)$

PROOF SKETCH. The proof proceeds along the lines of the proof of proposition 3.8.

Here is a binary relation R that expresses our proposition:

$$\frac{\kappa \, R^{ctx} \, \kappa' \qquad e \, R^{ctx} \, e'}{\mathtt{mon}^l(\kappa, e) \, R \, \mathtt{mon}^l(\mathsf{obl}^p[\![\kappa']\!] \circ \mathsf{obl}^{\bar{p}}[\![\kappa']\!], e')}$$

$$\frac{\kappa \, R^{ctx} \, \kappa' \qquad e \, R^{ctx} \, e'}{\mathtt{mon}^l_{p'}(\kappa, e) \, R \, \mathtt{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa']\!], e')}$$

We use again the proof obligation:

$P(k)$: if for all $i \leq k$, if $e \, R^{ctx} \, e'$ and $e \xmapsto{i} a$, then there exists a' such that $e' \xmapsto{*} a'$ and $a \, R^{ctx} \, a'$.

However, due to the bidirectional nature of observational equivalence, we need to prove that $P(k)$ holds for all k both for R and its inverse relation R^{-1}.

We show the most interesting case of the inductive step of the proof for the second clause of 3.5. We focus on terms with dependent contracts $\mathtt{mon}^l_{p'}(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), e)$ and $\mathtt{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa'_1 \xmapsto{d} (\lambda x.\kappa'_2)]\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa'_1 \xmapsto{d} (\lambda x.\kappa'_2)]\!], e'_o)$ that are related under the above relation R: $\mathtt{mon}^l_{p'}(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), e_o) \, R \, \mathtt{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa'_1 \xmapsto{d} (\lambda x.\kappa'_2)]\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa'_1 \xmapsto{d} (\lambda x.\kappa'_2)]\!], e'_o)$.

Based on the antecedent of the second clause of theorem 3.5, we pick v_a and v_a' such that $v_a R^{ctx} v_a'$ and then we proceed by calculating terms $(\mathsf{mon}^l_{p'}(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), e_o)\, v_a)$ and $(\mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!], e_o')\, v_a')$ and showing that either the two terms diverge or they both converge to answers related under R^{ctx}.

We split the proof into two cases, one per direction:

— From left to right, we pick the sub-case where $e_o \xmapsto{n} v_o$ and proceed by calculating $(\mathsf{mon}^l_{p'}(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), e_o)\, v_a)$:

$$(\mathsf{mon}^l_{p'}(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), e_o)\, v_a) \xmapsto{n \geq 1} \mathsf{mon}^l_{p'}(\{v_a/x\}\kappa_2, (v_o\, \mathsf{mon}^l_{\bar{p}'}(\kappa_1, v_a)))$$

By the definition of R, we know that since the two terms $\mathsf{mon}^l_{p'}(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), e_o)$ and $\mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!], e_o')$ are related under R, we can infer that $e_o R^{ctx} e_o'$. By applying the induction hypothesis on terms e_o and e_o', we conclude that if $e_o \xmapsto{*} v_o$ then $e_o' \xmapsto{*} v_o'$ and $v_o R^{ctx} v_o'$.

Now, we can calculate $(\mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!], e_o')\, v_a')$:

$$
\begin{aligned}
&(\mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1' \xmapsto{d} (\lambda x.\kappa_2')]\!], e_o')\, v_a') \\
&\xmapsto{*} (\lambda x.\mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa_2']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_2']\!], (v_o'\, \mathsf{mon}^l_{\bar{p}'}(\mathsf{obl}^p_{p'}[\![\kappa_1']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1']\!], x)))\, v_a') \\
&\longmapsto \mathsf{mon}^l_{p'}(\{v_a'/x\}(\mathsf{obl}^p_{p'}[\![v_a']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![v_a']\!]), \\
&\qquad (v_o'\, \mathsf{mon}^l_{\bar{p}'}(\mathsf{obl}^p_{p'}[\![\kappa_1']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1']\!], v_a'))) \\
&= \mathsf{mon}^l_{p'}(\{v_a'/x\}\mathsf{obl}^p_{p'}[\![\kappa_2']\!] \circ \{v_a'/x\}\mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_2']\!], \\
&\qquad (v_o'\, \mathsf{mon}^l_{\bar{p}'}(\mathsf{obl}^p_{p'}[\![\kappa_1']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1']\!], v_a'))) \\
&= \mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\{v_a'/x\}\kappa_2']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\{v_a'/x\}\kappa_2']\!], \\
&\qquad (v_o'\, \mathsf{mon}^l_{\bar{p}'}(\mathsf{obl}^p_{p'}[\![\kappa_1']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1']\!], v_a')))
\end{aligned}
$$

The two syntactic equalities are derived by two simple lemmas regarding substitution and the obligation separation and composition functions. Both lemmas can be easily proved by structural induction on contracts.

Now, the desired conclusion that if $\mathsf{mon}^l_{p'}(\{v_a/x\}\kappa_2, (v_o\, \mathsf{mon}^l_{\bar{p}'}(\kappa_1, v_a))) \Downarrow a_r$ then also $\mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\{v_a'/x\}\kappa_2']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\{v_a'/x\}\kappa_2']\!], (v_o'\, \mathsf{mon}^l_{\bar{p}'}(\mathsf{obl}^p_{p'}[\![\kappa_1']\!] \circ \mathsf{obl}^{\bar{p}}_{p'}[\![\kappa_1']\!], v_a')))$ returns answer a_r' and $a_r R^{ctx} a_r'$, and similarly for divergence, follows from another use of the induction hypothesis.

— The right to left direction follows an analogous argument. ∎

The idea of separating contracts into obligations for servers and clients is implicit in Meyer's writing and is adapted to a higher-order setting by Blume and McAllester [2006] and Findler and Blume [2006]. The latter treat obligations as error projections and attempts to compose them are defined in terms of function composition. This approach does not work for dependent contracts, however. More specifically Findler et al. [2004] show that dependent contracts and their obligations are not error projections.

Function composition of our obligations also does not give the expected outcome. In more detail, we can demonstrate that $\mathsf{mon}^l(\kappa, e) \not\simeq \mathsf{mon}^l(\mathsf{obl}^s[\![\kappa]\!], \mathsf{mon}^l(\mathsf{obl}^c[\![\kappa]\!], e))$ and $\mathsf{mon}^l(\kappa, e) \not\simeq \mathsf{mon}^l(\mathsf{obl}^c[\![\kappa]\!], \mathsf{mon}^l(\mathsf{obl}^s[\![\kappa]\!], e))$. Consider the following composition where $\kappa_0 = (\mathtt{flat}(PZ?) \mapsto \mathtt{flat}(PZ?)) \xmapsto{d} (\lambda f.\mathtt{flat}(\lambda x.(f\, 0) < 0))$:

SerM	κ	CliM
$\lambda f.(f\, 2)$	κ_0	$([\,]\, (\lambda y.y - 1))$

In this case, $\mathsf{obl^s}[\![\kappa_0]\!] = (\mathtt{flat}(PZ?) \mapsto \mathit{Any}) \overset{d}{\mapsto} (\lambda f.\mathtt{flat}(\lambda x.(f\,0) < 0))$ and $\mathsf{obl^c}[\![\kappa_0]\!] = (\mathit{Any} \mapsto \mathtt{flat}(PZ?)) \overset{d}{\mapsto} (\lambda f.\mathit{Any})$. Calculating $(\mathsf{mon}^l(\kappa_0, (\lambda f.(f\,2)))\,(\lambda y.y - 1))$ returns the value 1 while $(\mathsf{mon}^l(\mathsf{obl^s}[\![\kappa_0]\!], \mathsf{mon}^l(\mathsf{obl^c}[\![\kappa_0]\!], (\lambda f.(f\,2))))\,(\lambda y.y - 1))$ raises $\,{}^l\mathtt{error_s}$ and $(\mathsf{mon}^l(\mathsf{obl^c}[\![\kappa_0]\!], \mathsf{mon}^l(\mathsf{obl^s}[\![\kappa_0]\!], (\lambda f.(f\,2))))\,(\lambda y.y - 1))$ raises $\,{}^l\mathtt{error_c}$.

In contrast, our *approach* to treat separation and composition of obligations as *syntactic* operations on contracts handles all kinds of contracts, as proposition 4.3 shows. We therefore choose this syntactic approach over the semantic one of Findler et al. [2004] and Blume and McAllester [2006].

Before we end the section, let us prove two useful lemmas about obligations. First, the client-side and server-side obligations of a contract have fewer effects (contract errors or divergence) than the complete contract when used to monitor a component.

LEMMA 4.4. $\mathsf{mon}^l(\kappa, e) \preceq \mathsf{mon}^l(\mathsf{obl}^p[\![\kappa]\!], e)$

PROOF SKETCH. Here is a relation that brings together the critical elements:

$$\frac{\kappa R^{ctx} \kappa' \quad e R^{ctx} e'}{\mathsf{mon}^l(\kappa, e)\, R\, \mathsf{mon}^l(\mathsf{obl}^p[\![\kappa']\!], e')} \qquad \frac{\kappa R^{ctx} \kappa' \quad e R^{ctx} e'}{\mathsf{mon}^l_{p'}(\kappa, e)\, R\, \mathsf{mon}^l_{p'}(\mathsf{obl}^p_{p'}[\![\kappa']\!], e')}$$

The rest of the proof is a standard application from the strategy in subsection 3.3. ∎

Second, using only a client-side or server-side contract in place of the full contract imposes fewer restrictions. Technically, imposing just the server-side obligations means client-side errors cannot show up and vice versa.

LEMMA 4.5. *For all program contexts C, $C[\mathsf{mon}^l(\mathsf{obl}^p[\![\kappa]\!], v)] \not\Downarrow {}^l\mathtt{error}_{\bar{p}}$.*

PROOF SKETCH. Recall that programs have unique labels on all contracts and errors. The proof uses a Curry-Feys subject reduction strategy to prove that $\,{}^l\mathtt{error}_{\bar{p}}$ doesn't show up as a redex. The core of the subject is S:

$$\begin{aligned}
{}^l\mathtt{error}_p &\in S \\
\mathsf{mon}^l(\mathsf{obl}^p[\![\kappa]\!], e) &\in S \;\; \text{if } \kappa, e \in S^{ctx}_{/l} \\
\mathsf{mon}^l_p(\mathsf{obl}^p[\![\kappa]\!], e) &\in S \;\; \text{if } \kappa, e \in S^{ctx}_{/l} \\
\mathsf{mon}^l_{\bar{p}}(\mathsf{obl}^p[\![\kappa]\!], e) &\in S \;\; \text{if } \kappa, e \in S^{ctx}_{/l} \\
\mathtt{if}\,((\lambda x.\mathtt{tt})\,v)\,\mathtt{then}\,v\,\mathtt{else}\,{}^l\mathtt{error}_{\bar{p}} &\in S \;\; \text{if } v \in S^{ctx}_{/l} \\
\mathtt{if}\,\mathtt{tt}\,\mathtt{then}\,v\,\mathtt{else}\,{}^l\mathtt{error}_{\bar{p}} &\in S \;\; \text{if } v \in S^{ctx}_{/l} \\
&\quad\quad \text{for } p, l \text{ from theorem's statement}
\end{aligned}$$

The subject itself is the "mostly compatible" closure of S, dubbed $S^{ctx}_{/l}$, which we define as $C[\bar{e}] \in S^{ctx}_{/l}$ if $\bar{e} \in S$ and l does not occur in C. The subject reduction step shows that the standard reduction relation preserves membership in $S^{ctx}_{/l}$. ∎

5. TIGHT CONTRACT SATISFACTION

Recent research efforts have focused on the verification of components independent of their deployment context [Flanagan et al. 2002; Barnett et al. 2004]. Specifically their efforts aim to prove that a component lives up to its behavioral software contracts. One way to formulate this notion of contract satisfaction is to say that connecting the verified server component with *any* client component never triggers a contract error that blames the server. With CPCF notation, v satisfies κ if and only if for all contexts C, $C[\mathsf{mon}^l(\kappa, v)]$ does not evaluate to $\,{}^l\mathtt{error_s}$.

Exactly along these lines, we propose a definition of contract satisfaction that uses only CPCF's notion of observational equivalence. Specifically, we say that a server component v satisfies a contract κ if the server's part of the contract, $\mathsf{obl^s}[\![\kappa]\!]$, is never observable for any client. Conversely,

a client component C satisfies a contract κ if the client's part of the contract, $\mathsf{obl}^c[\![\kappa]\!]$, is never observable for any server.

Definition 5.1. (Tight Contract Satisfaction)

— $v \vDash_T \kappa$ if $\mathsf{mon}^l(\kappa, v) \simeq \mathsf{mon}^l(\mathsf{obl}^c[\![\kappa]\!], v)$
— $C \vDash_T \kappa$ if $C[\mathsf{mon}^l(\kappa, x)] \simeq C[\mathsf{mon}^l(\mathsf{obl}^s[\![\kappa]\!], x)]$

In a world of first-order functions, a world where function types look like $o \to \ldots \to o$, a programmer can ensure tight contract satisfaction with a simple `if` expression. Assume we are given

$$\kappa^* = \mathtt{flat}(e_c) \mapsto \mathtt{flat}(e_s) : con(o \to o)$$

Then for a function $v : o \to o$, the following alternative v^*

$$(\lambda x.\mathtt{let}\ y = (v\ x)\ \mathtt{in}\ \mathtt{if}\ (e_s\ y)\ \mathtt{then}\ y\ \mathtt{else}\ {}^l\mathtt{error}_s)$$

tightly guarantees the contract κ^*, as a straightforward calculation shows. First observe that v^* is equivalent to

$$(\lambda x.((\lambda y.\mathsf{mon}^l_s(\mathtt{flat}(e_s), y))\ (v\ x)))$$

which is indistinguishable from

$$(\lambda x.\mathsf{mon}^l_s(\mathtt{flat}(e_s), (v\ x)))$$

For the rest of the development of our argument we use the latter term when we refer to the definition of v^*.

Second, we can show for CPCF that $\longmapsto \subset \simeq$. This allows us to consider terms produced by reduction steps as observationally equivalent.

Third, it is easy to prove, now, that

$$\mathsf{mon}^l_p(\mathtt{flat}(\lambda x.\mathtt{tt}), e) \simeq e \qquad [\mathtt{tt}_{intro}]$$

and

$$\mathsf{mon}^l_p(\mathtt{flat}(\lambda x.e_c), e) \simeq \mathsf{mon}^l_p(\mathtt{flat}(\lambda x.e_c), \mathsf{mon}^l_p(\mathtt{flat}(\lambda x.e_c), e)) \qquad [\mathtt{flat}_{elim}]$$

Fourth, we know that

$$((\lambda x.F[x])\ e) \simeq F[e] \qquad [\beta_\Omega] \quad [\text{Sabry and Felleisen 1993}]$$

Finally, the above imply $v^* \vDash_T \kappa^*$ as follows:

$$
\begin{aligned}
&\mathsf{mon}^l(\kappa^*, v^*) \\
\simeq\ &\mathsf{mon}^l_s(\kappa^*, v^*) &&[\text{red. rule } mon] \\
\simeq\ &(\lambda x.\mathsf{mon}^l_s(\mathtt{flat}(e_s), (v^*\ \mathsf{mon}^l_c(\mathtt{flat}(e_c), x)))) &&[\text{red. rule } ho] \\
\simeq\ &(\lambda x.\mathsf{mon}^l_s(\mathtt{flat}(e_s), \mathsf{mon}^l_s(\mathtt{flat}(e_s), (v\ \mathsf{mon}^l_c(\mathtt{flat}(e_c), x))))) &&[v^* \text{ def., } \beta_\Omega] \\
\simeq\ &(\lambda x.\mathsf{mon}^l_s(\mathtt{flat}(e_s), (v\ \mathsf{mon}^l_c(\mathtt{flat}(e_c), x)))) &&[\mathtt{flat}_{elim}] \\
\simeq\ &(\lambda x.\mathsf{mon}^l_s(\mathtt{flat}(\lambda x.\mathtt{tt}), \mathsf{mon}^l_s(\mathtt{flat}(e_s), (v\ \mathsf{mon}^l_c(\mathtt{flat}(e_c), x))))) &&[\mathtt{tt}_{intro}] \\
\simeq\ &(\lambda x.\mathsf{mon}^l_s(\mathtt{flat}(\lambda x.\mathtt{tt}), \mathsf{mon}^l_s(\mathtt{flat}(e_s), (v\ \mathsf{mon}^l_c(\mathtt{flat}(e_c), x))))) &&[v^* \text{ def., } \beta_\Omega] \\
\simeq\ &\mathsf{mon}^l(\mathtt{flat}(\lambda x.\mathtt{tt}) \mapsto \mathtt{flat}(e_s), v^*) &&[\text{red. rule } ho] \\
\simeq\ &\mathsf{mon}^l(\mathsf{obl}^c[\![\kappa^*]\!], v^*) &&[\kappa^*, \mathsf{obl}^c \text{ def.}]
\end{aligned}
$$

The simplicity of v^* may explain why contract satisfaction is easily dismissed as trivial even though the higher-order version is surprisingly complex and does not allow such trivial protection.

For a higher-order world, tight contract satisfaction promotes a stringent and basically pessimistic view. In a sense, it forces component creators to defend against all possible—malicious and unintentional—problems in the clients with which their components might collaborate. While

section 3 already spells out an example that illustrates the severity of this notion of contract satisfaction, let us explore another example as a reminder. Consider these pieces:

$$C^\dagger = \mathtt{let}\ g = [\ ]\ \mathtt{in}\ \mathtt{let}\ f = (\lambda x.x - 1)\ \mathtt{in}\ (g\ f)$$
$$v^\dagger = (\lambda f.(f\ 2) - 1)$$
$$\kappa^\dagger = (\mathtt{flat}(PZ?) \mapsto \mathtt{flat}(PZ?)) \mapsto \mathtt{flat}(PZ?)$$

where $PZ?$ checks whether a number is greater or equal to 0.

Connecting the two components via the contract produces a plain CPCF program:

$$\mathtt{let}\ g = \mathtt{mon}^l((\mathtt{flat}(PZ?) \mapsto \mathtt{flat}(PZ?)) \mapsto \mathtt{flat}(PZ?), (\lambda f.(f\ 2) - 1))$$
$$\mathtt{in}\ \mathtt{let}\ f = (\lambda x.x - 1)\ \mathtt{in}\ (g\ f)$$

According to the semantics of CPCF, this program evaluates to 0, i.e., the contract monitoring process does not observe any contract violation. It is straightforward to see, however, that $v^\dagger \nVdash_T \kappa^\dagger$. To prove this point, we exhibit the context

$$C_o^\dagger = \mathtt{let}\ g = [\ ]\ \mathtt{in}\ \mathtt{let}\ f = (\lambda x.0)\ \mathtt{in}\ (g\ f)$$

as a witness for distinguishing the behavior of $\mathtt{mon}^l(\kappa^\dagger, v^\dagger)$ and $\mathtt{mon}^l(\mathtt{obl}^c[\![\kappa^\dagger]\!], v^\dagger)$. Specifically, the program

$$C_o^\dagger[\mathtt{mon}^l(\kappa^\dagger, v^\dagger)]$$

reduces to $^l\mathtt{error}_+$, while

$$C_o^\dagger[\mathtt{mon}^l(\mathtt{obl}^c[\![\kappa^\dagger]\!], v^\dagger)]$$

produces -1 because $\mathtt{obl}^c[\![\kappa^\dagger]\!]$ is

$$(\mathtt{flat}((\lambda x.\mathtt{tt})) \mapsto \mathtt{flat}(PZ?)) \mapsto \mathtt{flat}((\lambda x.\mathtt{tt}))$$

and checks only that the client sends numbers greater or equal to 0 to the server while the latter may send any value to the former.

In short, tight contract satisfaction expects components to provide universal guarantees about their behavior. This idea shows up in both the work of Blume and McAllester [2006] as well as Xu et al. [2009]. We can also characterize tight contract satisfaction via two theorems that are similar to the result of their piece of research.

THEOREM 5.2. *If $v \nVdash_T \kappa$ then there exists C such that $C[\mathtt{mon}^l(\kappa, v)] \not\simeq C[\mathtt{mon}^l(\mathtt{obl}^c[\![\kappa]\!], v)]$ and for all v', $C[\mathtt{mon}^l(\kappa, v)] \Downarrow\!\!\!/ \ v'$.*

PROOF. First, by definition of tight contract satisfaction we know that there is at least one whole program context C such that $C[\mathtt{mon}^l(\kappa, v)] \not\simeq C[\mathtt{mon}^l(\mathtt{obl}^c[\![\kappa]\!], v)]$. Second, we proceed with proof by contradiction. By lemma 4.4, we show that if $C[\mathtt{mon}^l(\kappa, v)] \Downarrow v'$, then $C[\mathtt{mon}^l(\mathtt{obl}^c[\![\kappa]\!], v)] \Downarrow v''$. Since C is a whole program v' and v'' are values of base types and thus $v' = v''$. Based on that we conclude that for all v', $C[\mathtt{mon}^l(\kappa, v)] \Downarrow\!\!\!/ \ v'$. ∎

THEOREM 5.3. *If $v \vDash_T \kappa$, then $C[\mathtt{mon}^l(\kappa, v)] \Downarrow\!\!\!/ \ ^l\mathtt{error}_s$ for all contexts C.*

PROOF. By definition of tight contract satisfaction and lemma 4.5. ∎

Blume and McAllester [2006, p. 3] start their investigation of contract satisfaction from the implied notion of the Findler-Felleisen contracts. More specifically, they define that a server satisfies its contract if the composition with any client never triggers a contract error that blames the server, just like we did at the opening paragraph of this section. They give an alternative definition of this notion of contract satisfaction based on set theory and prove the equivalence of the two definitions via soundness and completeness theorems. Taken together the theorems of this section establish that tight contract satisfaction defines a relationship that is analogous to the one of Blume and

McAllester [2006]. Any remaining differences are due to the latter's requirements that contracts do not signal errors on their own while our contracts are unrestricted, allowing errors and divergence.

6. LOOSE CONTRACT SATISFACTION

Although tight contract satisfaction establishes universal guarantees, many component programmers will happily accept the less ambitious standard of correct collaboration between a *specific* server component and some *specific* client components. Put differently, it often suffices to link a client with a server via some contract and to merely monitor the values that flow across the boundary whenever the program is run.

Following de Alfaro and Henzinger [2001], we refer to this approach as "optimistic" and characterize it in the following definition with a ternary relation between a server, a client, and a contract.

Definition 6.1. (Loose Contract Satisfaction)

— $C, v \vDash^{\mathsf{s}}_L \kappa$ iff $C[\mathtt{mon}^l(\kappa, v)] \simeq C[\mathtt{mon}^l(\mathsf{obl}^{\mathsf{c}}[\![\kappa]\!], v)]$
— $C, v \vDash^{\mathsf{c}}_L \kappa$ iff $C[\mathtt{mon}^l(\kappa, v)] \simeq C[\mathtt{mon}^l(\mathsf{obl}^{\mathsf{s}}[\![\kappa]\!], v)]$

Note that C represents only the client, *not* the entire program context.

One way to ascertain loose contract satisfaction is to make the client context so large that the rest of the program cannot possibly interfere with or observe the flow of values between the server and the client. A different—and practical—way is to bake enough checks into the client and the server so that the rest of the program cannot observe problems with the contracts between the two. That is, the creators of the client module and the server "conspire" so that the combination of the two appears to be a single coherent component. In practice this kind of reasoning motivates nested or hierarchical component systems.

The next two theorems confirm that this definition coincides with the implied Findler-Felleisen notion of contract satisfaction. First, if a component composition that forms a whole program does not loosely satisfy its contract, it either diverges or raises an error.

THEOREM 6.2. *Let* $\vdash C[\mathtt{mon}^l(\kappa, v)] : o$. *If* $C, v \nvDash^{\mathsf{s}}_L \kappa$, $C[\mathtt{mon}^l(\kappa, v)] \Uparrow$ *or* $C[\mathtt{mon}^l(\kappa, v)] \Downarrow \mathtt{error}$. *If* $C, v \nvDash^{\mathsf{c}}_L \kappa$, *then* $C[\mathtt{mon}^l(\kappa, v)] \Uparrow$ *or* $C[\mathtt{mon}^l(\kappa, v)] \Downarrow \mathtt{error}$. *(The label and subscript on the resulting error are not necessarily related to the components.)*

PROOF. By definition of loose contract satisfaction and lemma 4.4. ∎

Second, if a component composition loosely satisfies its contract, an execution never blames either of the two parties.

THEOREM 6.3. *If* $C, v \vDash^{\mathsf{s}}_L \kappa$, *then for all contexts* C', $C'[C[\mathtt{mon}^l(\kappa, v)]] \not\Downarrow^l \mathtt{error}_{\mathsf{s}}$. *Similarly, if* $C, v \vDash^{\mathsf{c}}_L \kappa$, *then for all contexts* C', $C'[C[\mathtt{mon}^l(\kappa, v)]] \not\Downarrow^l \mathtt{error}_{\mathsf{c}}$.

PROOF. By definition of loose contract satisfaction and lemma 4.5. ∎

The introduction of a second notion of contract satisfaction raises the natural question whether one implies the other. Given the names "tight" and "loose", we should expect that the former implies the later but not necessarily the inverse. The following theorem formalizes this expectation.

THEOREM 6.4. *Tight contract satisfaction implies loose contract satisfaction for all possible contexts, but the converse doesn't hold:*

— $v \vDash_T \kappa$ *implies* $C, v \vDash^{\mathsf{s}}_L \kappa$ *for all* C; $C \vDash_T \kappa$ *implies* $C, v \vDash^{\mathsf{c}}_L \kappa$ *for all* v.
— *There exists* κ, v, *and* C *such that* $C, v \vDash^{\mathsf{s}}_L \kappa$ *does not imply* $v \vDash_T \kappa$ *and such that* $C, v \vDash^{\mathsf{c}}_L \kappa$ *does not imply* $C \vDash_T \kappa$.

PROOF. The first statement follows from the definitions of contextual equivalence, tight contract satisfaction, and loose contract satisfaction.

For the second statement, consider the example $C^\dagger[\mathtt{mon}^l(\kappa^\dagger, v^\dagger)]$ of section 5. In this example the server and the client satisfy their contract under loose contract satisfaction but they don't satisfy it under tight contract satisfaction. ∎

7. SHY CONTRACT SATISFACTION

A higher-order function consumes objects with infinite behavior. In practice, a higher-order function may consume streams or a method may consume objects in the sense of an object-oriented language. In CPCF, only functions have infinite behavior, so higher-order functions are function-consuming functions.

As Chitil et al. [2003] observe, assertions for a higher-order function G may explore a larger portion of the argument f than G does. The last concrete example in section 3.2 demonstrates this observation, repeated here for convenience:

$$\frac{SerM \qquad \kappa \qquad CliM}{(\lambda s.(\mathtt{fst}\ s)) \qquad \kappa_1 \qquad ([\ ]\ (\lambda i.-i))}$$

where $\kappa_1 = (Any \mapsto Any) \overset{d}{\mapsto} \lambda s.\mathtt{flat}((PZ?\ (\mathtt{fst}\ s)) \wedge (PZ?\ (\mathtt{fst}\ (\mathtt{rst}\ s))))$
and $\mathtt{fst}$ is $\lambda s.(s\ 0)$ and $\mathtt{rst}$ is $\lambda s.\lambda i.(s\ (i+1))$

In turn, Chitil et al. [2003] argue that the run-time system for a *lazy* language should ignore overly eager assertions. More concretely, the very moment an assertion evaluates more of an argument than the function itself, assertion checking is abandoned and the assertion is considered satisfied. We call this notion "shy" contract satisfaction and apply it to the by-value language CPCF.

In this section, we explore the relationship of shy checking to tight and loose contract satisfaction. At least at first glance, a shy notion of contract satisfaction is even looser than loose contract satisfaction but a moment's reflection shows that this conclusion is overly simplistic. To start with, the preceding sections should clarify that we can add shyness to *both* tight and loose contract checking. Furthermore, we can apply the idea to individual contract checks for each evaluation of a server or, following Chitil et al. [2003], we can check all the assertions *at the end of the program execution* when it is absolutely clear which part of every higher-order argument is explored. Yet another alternative would be to consider overly eager contracts as errors.

Here we choose to focus on the combination of shy checking with loose and tight contract satisfaction on a per contract basis to demonstrate the flexibility of our idea, though, it could also be used to investigate the alternatives. We start from the observation that a terminating function application $(f\ v)$ explores only a finite part of its argument, i.e., a finite value v^* that is "below" the given value v such that $(f\ v^*)$ is still observationally equivalent to $(f\ v)$ with respect to the given program. The goal is to change the contract satisfaction criteria so that they notice when a contract explores v beyond the extent of v^*. Inspired by Chitil et al. [2003], we create values v^* that raise exceptions when the contract over-explores the argument, and we turn exceptions into approval.

The formal development of our idea requires two steps. The first subsection introduces an extension of CPCF with exceptions and exception handling, dubbed CPCF^x. This language is only useful in describing the mechanisms of shy contract satisfaction; exceptions and exception handlers are not added to the surface syntax. The second subsection uses the semantic framework of the first subsection to define two additional notions of contact satisfaction on CPCF: shy tight and shy loose contract satisfaction. The last two subsections deal with the implications of the new notions.

7.1. CPCF with Exceptions and Exception Handlers

The set of terms and answers of CPCF^x are extensions of the corresponding sets of CPCF.

$$\begin{aligned}\mathbf{Terms} \quad & e ::= \ldots \mid \mathtt{exn} \mid \mathtt{catch}(e) \\ \mathbf{Answers} \quad & a ::= \ldots \mid \mathtt{exn}\end{aligned}$$

Like errors, exn has all possible types. Exception handlers, though, work only for boolean-typed sub-terms:

$$\frac{\Gamma \vdash e : bool}{\Gamma \vdash \texttt{catch}(e) : bool}$$

As for the semantics of CPCF^x, we formulate it via $\longmapsto_x$, an extension of the reduction semantics for CPCF. In the context of a reduction semantics, exceptions erase portions of the evaluation context:

$$\textbf{Ev. Ctxt.} \quad F ::= \ldots \mid \texttt{catch}(F)$$
$$\textbf{Exn. Ctxt.} \quad E ::= \text{ all of } F \text{ except for } \texttt{catch}(F)$$

That is, the set of evaluation contexts from CPCF is extended with a $\texttt{catch}(\cdots)$ construction so that evaluation may happen in the sub-term of an exception handler. Exception contexts, however, are all evaluation contexts *without* the $\texttt{catch}(\cdots)$ construction and it is those contexts that exceptions erase:

$$\begin{aligned} F[E[\texttt{exn}]] &\longmapsto_x F[\texttt{exn}] \\ F[\texttt{catch}(v)] &\longmapsto_x F[v] \\ F[\texttt{catch}(\texttt{exn})] &\longmapsto_x F[\texttt{tt}] \end{aligned}$$

The exception handler doesn't react when its body reduces to a (boolean) value; otherwise its body reduces to an exception, and it is turned into a $\texttt{tt}$ value.

Also, we modify the reduction relation for monitoring flat contracts so that it can exploit exceptions handlers:

$$F[\texttt{mon}_p^l(\texttt{flat}(e), v)] \longmapsto_x F[\texttt{let } x = \texttt{catch}((e \; v)) \texttt{ in if } x \texttt{ then } v \texttt{ else }^l\texttt{error}_p] \quad [lax_x]$$

That is, flat contract checking now runs predicates within the extent of an exception handler. If checking the value for a property raises an exception, the contract check is abandoned and considered successful.

CPCF^x, like CPCF, is an eager language and we only use it as a target language for CPCF. None of its extra constructs are available or visible to the programmer.

Note that since CPCF terms do not contain or raise exn, CPCF^x does not change the behavior of CPCF programs. More concretely, let S_{CPCF}^{ctx} the compatible closure of a relation S on CPCF^x terms restricted to CPCF contexts. Consider the following relation: $e \; S \; \texttt{catch}(e')$ where $e \; S_{CPCF}^{ctx} \; e'$, which brings together CPCF terms that are identical modulo the injection of $\texttt{catch}$ frames. It is easy to prove the following theorem.

THEOREM 7.1. *Let e in CPCF. If $e \Downarrow a$ then $e \Downarrow_x a'$ and $a \, S_{CPCF}^{ctx} \, a'$.*

PROOF SKETCH. The proof proceeds with a standard simulation argument using S_{CPCF}^{ctx} as the simulation relation. ∎

From the above theorem and theorems 6.2 and 6.3, we can also conclude that, despite the extra infrastructure, CPCF^x implements loose contract checking.

Finally, the additions to CPCF affect the notion of observational equivalence. We introduce a new definition of observational equivalence for CPCF^x.

Definition 7.2. (Observational Equivalence in CPCF^x, $\simeq_x$) Two terms, e_1 and e_2, are observationally equivalent in CPCF^x, written $e_1 \simeq_x e_2$, if for all contexts C,

— $C[e_1] \Downarrow_x {}^l\texttt{error}_p$ if and only if $C[e_2] \Downarrow_x {}^l\texttt{error}_p$, and
— $C[e_1] \Downarrow_x \texttt{exn}$ if and only if $C[e_2] \Downarrow_x \texttt{exn}$, and
— otherwise: if both programs diverge or both converge.

Similar modifications are also required for the definition of the value approximation.

Definition 7.3. (Value Approximation in CPCF^x, $\preceq_x$). A term e_1 is bellow a term e_2 in the term value-termination preorder in CPCF^x, written $e_1 \preceq_x e_2$, if for all program contexts C,

— if $C[e_1] \Downarrow_x$ exn then $C[e_2] \Downarrow_x$ exn, and
— if $C[e_1] \Downarrow_x v$ then $C[e_2] \Downarrow_x v'$.

The proof method of subsection 3.3 cannot be adapted to CPCF^x. In place of the shortcut, we must manually build a bi-simulation. We demonstrate how to adapt the proof of proposition 3.8 of CPCF in this manner. Proofs of the all theorems about CPCF can be lifted to CPCF^x in an analogous manner.

PROPOSITION 7.4. *For* κ *and e in* CPCF^x, $\text{mon}^l(\kappa, e) \preceq_x e$

PROOF SKETCH. The proposition can be proved by applying the following lemma to the pair of terms of the $C[\text{mon}^l(\kappa, e)]$ and $C[e]$. ∎

LEMMA 7.5. *Let R a binary relation of pairs of terms of the form:* $\text{mon}^l(\kappa, e_v) R e'_v$ *and* $\text{mon}^l_p(\kappa, e_v) R e'_v$ *if* $e_v R^{ctx} e'_v$ *where* R^{ctx} *is the compatible closure of R. If* $e R^{ctx} e'$, *if* $e \Downarrow_x$ exn, *then* $e' \Downarrow$ exn, *and if* $e \Downarrow_x v_r$ *then* $e' \Downarrow v'_r$ *and* $v_r R^{ctx} v'_r$.

PROOF SKETCH. We proceed by induction using the following hypothesis:

  <u>$IH(k)$</u>: for all $i \leq k$, if $e R^{ctx} e'$ and if $e \overset{i}{\longmapsto}_x$ exn then $e' \overset{*}{\longmapsto}_x$ exn and if $e \overset{i}{\longmapsto}_x v_r$ then $e' \overset{*}{\longmapsto}_x v'_r$ and $v_r R^{ctx} v'_r$.

The most interesting case is when $e = F[\text{mon}^l_p(\kappa, v_r)]$ and $e' = F'[v'_r]$. By the reduction relation we know that

$$F[\text{mon}^l_p(\text{flat}(e_c), v_r)]$$
$$\longmapsto_x F[\text{if catch}((e_c\ v_r)) \text{ then } v_r \text{ else } {}^l\text{error}_p]$$

The result of this term depends on the evaluation of $(e_c\ v_r)$. There are five cases: $(e_c\ v_r) \Uparrow$, $(e_c\ v_r) \Downarrow_x$ error, $(e_c\ v_r) \Downarrow_x$ ff, $(e_c\ v_r) \Downarrow_x$ tt and $(e_c\ v_r) \Downarrow_x$ exn. The first three are not possible cases since by assumption $e \Downarrow_x$ exn or $e \Downarrow_x v$. For the last two cases, by the reduction rules for if and catch, we conclude that

$$F[\text{if catch}((e_c\ v_r)) \text{ then } v_r \text{ else } {}^l\text{error}_p] \longmapsto_x F[v_r]$$

Since $e R^{ctx} e'$, we conclude that $F[v_r] R^{ctx} F'[v'_r]$. The desired result is obtained by applying the induction hypothesis. ∎

The compilation of CPCF terms to CPCF^x is trivial; programs are simply taken as-is.

7.2. Shy Contract Satisfaction: The Definition

The introduction of CPCF^x enables us to define a notion of shy contract satisfaction. Specifically, the machinery allows us to describe the "finite" value that is like v_a in a terminating application $(v_f\ v_a)$ but raises exceptions if it is explored beyond the portion that v_f explores. First, we order values according to the exceptions that they can raise. Intuitively a value v_1 exception approximates value v_2 if v_1 behaves like v_2 in any context except that in some case it causes exn to be raised while v_2 does not.

Definition 7.6. (Exception Approximation) A CPCF^x value v_1 of type τ *exception approximates* v_2 of the same type—$v_1 \preceq_{\text{exn}} v_2$—iff:

— for τ a base type, $v_1 \simeq_x v_2$;
— for a function type $\tau \to \tau'$, for all v_a, v'_a such that $v_a \preceq^{ctx}_{\text{exn}} v'_a$
    — if $(v_1\ v_a) \Downarrow_x$ exn then $(v_2\ v'_a) \Downarrow_x a$ or $(v_2\ v'_a) \Uparrow_x$, and
    — if $(v_1\ v_a) \Downarrow_x a$ where $a \neq$ exn then $(v_2\ v'_a) \Downarrow_x a'$ and $a \preceq^{ctx}_{\text{exn}} a'$, and
    — if $(v_1\ v_a) \Uparrow_x$ then $(v_2\ v'_a) \Uparrow_x$

where $\preceq^{ctx}_{\text{exn}}$ is the compatible closure of $\preceq_{\text{exn}}$.

Second, we use the exception approximation to describe the portion of an argument that is explored during a terminating function application.

Definition 7.7. (Weakened Argument) For function v_f and argument v_a, the *weakened argument*, $\mathbb{W}[v_f, v_a]$, is a minimal element in $\{v^* | v^* \preceq_{\mathrm{exn}} v_a$ and $(v_f\ v_a) \simeq_x (v_f\ v^*)\}$, i.e., there is no w in the set such that $w \preceq_{\mathrm{exn}} \mathbb{W}[v_f, v_a]$ and $w \not\simeq_x \mathbb{W}[v_f, v_a]$.

The weakened argument is required to be minimal in order to hold the minimum necessary information that is needed so that the corresponding application is not affected by the weakening. This implies that the weakened argument captures only the part of the original argument that the function explored.

In general, finding the weakened argument is not computable. It becomes computable in some cases, like for functions that return flat values. Furthermore, even in these cases, the weakened argument cannot be expressed in CPCF^x. Additional features like reflection or support baked into the runtime system is required to calculate it. For example, an execution environment could keep track of arguments and results with a hash table and feed the hash table approximation to the contract instead of the actual argument. Since the definition of the weakened argument does not interfere with the semantics of CPCF^x and the question of computability is irrelevant for a specification of contract satisfaction, we ignore it here.

Note: As mentioned before, Chitil et al. [2003] do not just weaken the argument with respect to a function application but with respect to the evaluation context of the application. Thus, if

$$F[(v_f\ v_a)] \longmapsto^*_x F[v]$$

the weakened argument would be

$$\mathbb{W}[(\lambda x.F[(v_f\ x)]), v_a]$$

according to our notation.

Their design choice makes the weakened argument always computable since they restrict themselves to functions that return base values. However, in the general case, they consider a bigger context than we do. Thus the weakened argument they produce contains much more information than our weakened argument. ∎

Now we are ready to develop a shy notion of contract satisfaction. The key is to replace all arguments to a monitored function v_f with a weakened argument. If $\kappa \equiv \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$ then $\mathrm{mon}^l(\kappa, v_f)$ is equivalent to $(\lambda x.\mathrm{mon}^l_s(\kappa_2, (v_f\ \mathrm{mon}^l_c(\kappa_1, x))))$, assuming x is not free in v_f. We then use $(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v_f, \mathrm{mon}^l_c(\kappa_1, x)]/x\}\kappa_2, (v_f\ \mathrm{mon}^l_c(\kappa_1, x))))$ to weaken the argument visible to the post-condition contract κ_2. Doing so guarantees that v_f gets to see everything it needs from the argument and that κ_2 cannot inspect anything else. Since this form of weakening is orthogonal to contract satisfaction, we apply it to both notions of contract satisfaction.

Definition 7.8. (Shy-Tight Contract Satisfaction) Let v, C and κ be in CPCF. Let v be a function of type $\tau_1 \to \tau_2$ and let $\kappa \equiv \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$ be its contract.

— $v \vDash_{ST} \kappa$ if and only if

$$\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\kappa_1, x)]/x\}\kappa_2, (v\ \mathrm{mon}^l_c(\kappa_1, x)))$$
$$\simeq_x$$
$$\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x)]/x\}\mathrm{obl}^c_s[\![\kappa_2]\!], (v\ \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x)))$$

— $C \vDash_{ST} \kappa$ if and only if

$$C[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[z, \mathrm{mon}^l_c(\kappa_1, x)]/x\}\kappa_2, (z\ \mathrm{mon}^l_c(\kappa_1, x))))]$$
$$\simeq_x$$
$$C[(\lambda x.\ \mathrm{mon}^l_s(\{\mathbb{W}[z, \mathrm{mon}^l_c(\mathrm{obl}^s_c[\![\kappa_1]\!], x)]/y\}\mathrm{obl}^s_s[\![\kappa_2]\!],)$$
$$(z\ \mathrm{mon}^l_c(\mathrm{obl}^s_c[\![\kappa_1]\!], x)))]$$

Definition 7.9. (Shy-Loose Contract Satisfaction) Let v, C and κ be in CPCF. Let v be a function of type $\tau_1 \to \tau_2$ and let $\kappa \equiv \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$ be its contract.

— $C, v \models_{SL}^{s} \kappa$ if and only if

$$C[(\lambda x.\mathrm{mon}_s^l(\{\mathbb{W}[v, \mathrm{mon}_c^l(\kappa_1, x)]/x\}\kappa_2, (v\,\mathrm{mon}_c^l(\kappa_1, x))))] \simeq_x$$
$$C[(\lambda x.\mathrm{mon}_s^l(\{\mathbb{W}[v, \mathrm{mon}_c^l(\mathrm{obl}_c^c[\![\kappa_1]\!], x)]/x\}\mathrm{obl}_s^c[\![\kappa_2]\!], (v\,\mathrm{mon}_c^l(\mathrm{obl}_c^c[\![\kappa_1]\!], x))))]$$

— $C, v \models_{SL}^{c} \kappa$ if and only if

$$C[(\lambda x.\mathrm{mon}_s^l(\{\mathbb{W}[v, \mathrm{mon}_c^l(\kappa_1, x)]/x\}\kappa_2, (v\,\mathrm{mon}_c^l(\kappa_1, x))))] \simeq_x$$
$$C[(\lambda x.\mathrm{mon}_s^l(\{\mathbb{W}[v, \mathrm{mon}_c^l(\mathrm{obl}_c^s[\![\kappa_1]\!], x)]/x\}\mathrm{obl}_s^s[\![\kappa_2]\!], (v\,\mathrm{mon}_c^l(\mathrm{obl}_c^s[\![\kappa_1]\!], x))))]$$

7.3. Shy Contract Satisfaction: The Implications

While loose contract satisfaction is already permissive, shy-loose contract satisfaction captures a yet more permissive contract interpretation than Findler and Felleisen's method of "execute and check." The two following components and their contract demonstrate a case where this is true:

$$\begin{aligned} C^{\S} &= \texttt{let } g = [\,]\,\texttt{in let } f = (\lambda x.\texttt{if } x = 1 \texttt{ then } 1 \texttt{ else } -1)\texttt{in } (g\,f) \\ v^{\S} &= (\lambda f.(f\,1)) \\ \kappa^{\S} &= \kappa_1 \overset{d}{\mapsto} (\lambda f.\kappa_2) \\ &\text{where } \kappa_1 = \texttt{flat}(PZ?) \mapsto \texttt{flat}(PZ?) \text{ and } \kappa_2 = \texttt{flat}(\lambda x.(f-3) \geq 0) \end{aligned}$$

On the one hand, $v^{\S}$ and $C^{\S}$ satisfy $\kappa^{\S}$ under shy-loose contract satisfaction. For the three given applications $(g\,\mathrm{mon}_c^l(\kappa_1, v^{\S}))$, $(g\,\mathrm{mon}_c^l(\mathrm{obl}_c^c[\![\kappa_1]\!], v^{\S}))$, and $(g\,\mathrm{mon}_c^l(\mathrm{obl}_c^s[\![\kappa_1]\!], v^{\S}))$, the weakened argument is

$$w^{\S} = (\lambda x.\texttt{if } x = 1 \texttt{ then } 1 \texttt{ else exn}).$$

because $v^{\S}$ is only explored at 1. No matter which part of the contract is neutralized—server or client—the flat contract checking succeeds due to the exceptions raised by $w^{\S}$ and the composition itself runs to successful completion. As a result, the composition satisfies the contract according to shy-loose contract satisfaction. Also, note that the server in the above composition satisfies its contract according even to shy-tight contract satisfaction.

On the other hand, if the component composition $C^{\S}[\mathrm{mon}^l(\kappa^{\S}, v^{\S})]$ is run, the execution ends with $^l\mathrm{error}_s$. Hence, the above composition is a counter-example for the analogues of theorems 5.3 and 6.3 for shy-tight contract satisfaction and shy-loose contract satisfaction respectively.

In addition, as stated before, CPCF^x implements loose contract checking. Because of that the server of the above composition fails to satisfy the range contract κ_2 of g according to loose contract satisfaction.

A second implication of our examples is that shy-loose contract satisfaction does not imply loose contract satisfaction and shy-tight contract satisfaction does not imply tight contract satisfaction.

It is possible, though, to re-establish the analogues of theorems 5.2 and 6.2 for shy contract satisfaction. First, if the composition of a server with any client through a contract always reduces to a value then the server satisfies the contract according to shy-tight contract satisfaction.

THEOREM 7.10. *Let v, $\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$ and C be in CPCF. If $v \not\models_{ST} \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$, then there exists C such that*

$$C[(\lambda x.\mathrm{mon}_s^l(\{\mathbb{W}[v, \mathrm{mon}_c^l(\kappa_1, x)]/x\}\kappa_2, (v\,\mathrm{mon}_c^l(\kappa_1, x))))] \not\simeq_x$$
$$C[(\lambda x.\mathrm{mon}_c^l(\{\mathbb{W}[v, \mathrm{mon}_c^l(\mathrm{obl}_c^c[\![\kappa_1]\!], x)]/x\}\mathrm{obl}_s^s[\![\kappa_2]\!], (v\,\mathrm{mon}_s^l(\mathrm{obl}_c^c[\![\kappa_1]\!], x))))]$$

and $C[\mathrm{mon}^l(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \Downarrow_x v'$.

PROOF SKETCH. We proceed with a proof by contradiction. The proof is based on a tedious but uninteresting simulation between a CPCF^x term e and a CPCF^x term e' that differs from e in that

some values v in contracts have been replaced with values v' such that $v' \preceq_{\mathrm{exn}} v$. We express that using a binary relation S:

$$
\begin{aligned}
\mathrm{mon}^l(\kappa, e) \;\; &S \;\; \mathrm{mon}^l(\kappa', e') \\
&\text{if } \kappa\, T^{ctx}_{CPCF}\, \kappa' \text{ and } e\, S^{ctx}_{CPCF}\, e' \\
\mathrm{mon}^l_p(\kappa, e) \;\; &S \;\; \mathrm{mon}^l_p(\kappa', e') \\
&\text{if } \kappa\, T^{ctx}_{CPCF}\, \kappa' \text{ and } e\, S^{ctx}_{CPCF}\, e' \\
\texttt{let } x = \texttt{catch}((e\ v)) \texttt{ in if } x \texttt{ then } v \texttt{ else }{}^l\texttt{error}_p \;\; &S \\
\texttt{let } x = \texttt{catch}((e'\ v')) \texttt{ in if } x \texttt{ then } v \texttt{ else }{}^l\texttt{error}_p \\
&\text{if } e\, Y^{ctx}_{CPCF}\, e' \text{ and } v\, S^{ctx}_{CPCF}\, v' \\
v \;\; &T \;\; v' \\
&\text{if } v' \preceq_{\mathrm{exn}} v \\
\texttt{let } x = \texttt{catch}((e\ v)) \texttt{ in if } x \texttt{ then } v \texttt{ else }{}^l\texttt{error}_s \;\; &Y \\
\texttt{let } x = \texttt{catch}((e'\ v')) \texttt{ in if } x \texttt{ then } v \texttt{ else }{}^l\texttt{error}_s \\
&\text{if } e\, T^{ctx}_{CPCF}\, e' \text{ and } v\, T^{ctx}_{CPCF}\, v' \\
v \;\; &Y \;\; \mathrm{exn} \\
v \;\; &Y \;\; v' \\
&\text{if } v' \preceq_{\mathrm{exn}} v
\end{aligned}
$$

The simulation itself is the closure of S over CPCF contexts, dubbed S^{ctx}_{CPCF}. We prove, first, that if $e\, S^{ctx}_{CPCF}\, e'$ and $e \Downarrow_x v_o$ then $e' \Downarrow_x v'_o$ and $v_o S^{ctx}_{CPCF} v'_o$. Second, we show with a simple subject reduction that for all CPCF terms e, $e \Downarrow_x \mathrm{exn}$. Third, by the properties of $\preceq_x$, we conclude that if $C[\mathrm{mon}^l(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), v)] \Downarrow_x v_o$ then $C[\mathrm{mon}^l(\mathrm{obl}^c[\![\kappa_1 \xmapsto{d} (\lambda x.\kappa_2)]\!], v)] \Downarrow_x v''_o$. A consequence of this is that if

$$C_o[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\kappa_1, x)]/x\}\kappa_2, (v\, \mathrm{mon}^l_c(\kappa_1, x))))] \Downarrow_x v'_o$$

then

$$C_o[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x)]/x\}\mathrm{obl}^c_s[\![\kappa_2]\!], (v\, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x))))] \Downarrow_x v'''_o$$

Fourth, based on the first step we prove the following:
if $C[\mathrm{mon}^l(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), v)] \Downarrow_x v_o$ then

$$C[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\kappa_1, x)]/x\}\kappa_2, (v\, \mathrm{mon}^l_c(\kappa_1, x))))] \Downarrow_x v'_o$$

and if $C[\mathrm{mon}^l(\mathrm{obl}^c[\![\kappa_1 \xmapsto{d} (\lambda x.\kappa_2)]\!], v)] \Downarrow_x v''_o$ then

$$C[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x)]/x\}\mathrm{obl}^c_s[\![\kappa_2]\!], (v\, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x))))] \Downarrow_x v'''_o.$$

Fifth by the definition of shy-tight contract satisfaction, we know that there is at least a whole program context C_o s.t. $C_o[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\kappa_1, x)]/y\}\kappa_2, (v\, \mathrm{mon}^l_c(\kappa_1, x))))] \not\approx_x C_o[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\mathrm{obl}^c_s[\![\kappa_1]\!], x)]/x\}\mathrm{obl}^c_s[\![\kappa_2]\!], (v\, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x))))]$.

Sixth, for the program context C_o we assume that $C_o[\mathrm{mon}^l(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), v)] \Downarrow_x v_o$.

Seventh, based on our assumption and by the second, third and fourth step we prove that if $C_o[\mathrm{mon}^l(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), v)] \Downarrow_x v_o$ then

$$C_o[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\kappa_1, x)]/x\}\kappa_2, (v\, \mathrm{mon}^l_c(\kappa_1, x))))] \Downarrow_x v'_o$$

and

$$C_o[(\lambda x.\mathrm{mon}^l_s(\{\mathbb{W}[v, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x)]/x\}\mathrm{obl}^c_s[\![\kappa_2]\!], (v\, \mathrm{mon}^l_c(\mathrm{obl}^c_c[\![\kappa_1]\!], x))))] \Downarrow_x v'''_o$$

Since C_o is a whole program v'_o and v''_o are value of base types and thus $v'_o = v''_o$. But this conclusion contradicts the fact drawn by the theorem statement and stated in the fifth step of our proof. Thus, we conclude that our assumption was invalid and because of that $C_o[\mathrm{mon}^l(\kappa_1 \xmapsto{d} (\lambda x.\kappa_2), v)] \Downarrow_x v'$. ∎

THEOREM 7.11. *Let v, C and κ be in CPCF. Let $\vdash C[\text{mon}^l(\kappa,v)] : o$. If $C,v \not\vDash^{\mathsf{s}}_{SL} \kappa$, then $C[\text{mon}^l(\kappa,v)] \Uparrow_x$ or $C[\text{mon}^l(\kappa,v)] \Downarrow_x$ error. Similarly, if $C,v \not\vDash^{\mathsf{c}}_{SL} \kappa$, then $\text{mon}^l(\kappa,v) \Uparrow_x$ or $C[\text{mon}^l(\kappa,v)] \Downarrow_x$ error. (The label and subscript on the resulting error are not necessarily related to the components.)*

PROOF SKETCH. We proceed with a proof by contradiction and we re-use the simulation S from the proof of theorem 7.10. The first four steps are identical to the first four steps of theorem 7.10. For the fifth step, we assume that $C[\text{mon}^l(\kappa,v)] \Downarrow_x v_o$. Sixth, based on our assumption and by the second, third and fourth step we prove that if $C[\text{mon}^l(\kappa,v)] \Downarrow_x v_o$ then $C,v \vDash^{\mathsf{s}}_{SL} \kappa$. But this contradicts the theorem's statement that assumes $C,v \not\vDash^{\mathsf{s}}_{SL} \kappa$. Similarly for the second part of the theorem. ∎

Finally, the tight notion of shy contract satisfaction implies the loose one.

THEOREM 7.12. *Let v, C and κ be in CPCF. Shy tight contract checking implies shy-loose contract checking but not vice versa.*

— *$v \vDash_{ST} \kappa$ implies $C,v \vDash^{\mathsf{s}}_{SL} \kappa$ for all contexts C; $C \vDash_{ST} \kappa$ implies $C,v \vDash^{\mathsf{c}}_{SL} \kappa$ for all values v.*
— *There exists κ, v, and C such that $C,v \vDash^{\mathsf{s}}_{SL} \kappa$ but $v \vDash_{ST} \kappa$ does not hold and neither does $C \vDash_{ST} \kappa$.*

PROOF SKETCH. The first part follows from the definitions of contextual equivalence and tight and loose shy contract satisfaction.

For the second part, consider the example of section 5. We use the weakened argument $w^\dagger = (\lambda x.\text{if } x = 2 \text{ then } 1 \text{ else exn})$. In this example the server and the client satisfy their contract in terms of shy-loose contract satisfaction but there are other clients and servers that can trigger violations of the contract. For instance consider the composition of the server $v^\dagger$ with the client $C_o^\dagger = \text{let } g = [\,] \text{ in let } f = (\lambda x.0) \text{ in } (g\,f)$. Thus the composition does not satisfy the contract according to shy-tight contract satisfaction. ∎

7.4. The Landscape

The sheer existence of four notions of contract satisfaction for the same language raises the question of how they relate to each other. We know from sections 6 and 7 that tight contract satisfaction implies loose contract satisfaction in pure CPCF and that shy-tight satisfaction implies shy-loose satisfaction. In this subsection, we shed some light on the remaining relationships.

We can show that the relationship between shy-loose satisfaction and loose satisfaction is not straightforward. For some examples—e.g., the one in section 5—the two notions coincide. Also shy-loose contract satisfaction doesn't imply loose satisfaction as we demonstrate in section 7.3. Unfortunately, the expected converse doesn't hold either. The problem is that contracts may have computational effects of their own: they may signal errors, they may raise exceptions, and they may diverge. Here is an example that uses effects to show how loose contract satisfaction does not imply shy-loose contract satisfaction:

$$C^\ddagger = \text{let } g = [\,] \text{ in let } f = (\lambda x.x) \text{ in } (g\,f)$$
$$v^\ddagger = (\lambda f.(f\,1))$$
$$\kappa^\ddagger = (\text{flat}(PZ?) \mapsto \text{flat}((\lambda x.\Omega_{bool}))) \overset{d}{\mapsto} (\lambda f.\text{flat}((\lambda x.((f\,3) + \Omega_{num}) \geq 0)))$$

While both the client and the server loosely satisfy the contract, only the client satisfies it in the sense of shy-loose satisfaction; the server, however, fails.

As for loose contract satisfaction, we argue that the relevant compositions are observationally equivalent because they always diverge. The complete program diverges when the contract monitor tries to check the postcondition on the argument of g. A divergent computation is also triggered

when the client's or server's portions of the contract are disabled:

$$\text{obl}^c[\![\kappa^\ddagger]\!] = (\text{flat}((\lambda x.\text{tt})) \mapsto \text{flat}((\lambda x.\Omega_{bool}))) \overset{d}{\mapsto} (\lambda f.\text{flat}((\lambda x.\text{tt})))$$

$$\text{obl}^s[\![\kappa^\ddagger]\!] = (\text{flat}(PZ?) \mapsto \text{flat}((\lambda x.\text{tt}))) \overset{d}{\mapsto} (\lambda f.\text{flat}((\lambda x.((f\,3) + \Omega_{num}) \geq 0)))$$

As for shy-loose contract satisfaction, the weakened element for the application $(g\,f)$ is $w^\ddagger = (\lambda x.\text{if } x = 1 \text{ then } 1 \text{ else exn})$. Under this setting, both $C^\ddagger[\text{mon}^l(\kappa^\ddagger, w^\ddagger)]$ and the client variant $C^\ddagger[\text{mon}^l(\text{obl}^c[\![\kappa^\ddagger]\!], w^\ddagger)]$ diverge, meaning the client satisfies the contract. In contrast, the server-protected variant, $C^\ddagger[\text{mon}^l(\text{obl}^s[\![\kappa^\ddagger]\!], w^\ddagger)]$, returns 1 because the postcondition of g over-eagerly tries to inspect $w^\ddagger$ at 3. In short, even the empty context can differentiate the two compositions and therefore the server fails to satisfy the contract in the shy-loose sense.

8. OTHER RELATED WORK

Concurrency researchers have been investigating the use of behavioral pre-order for a long time [Hennessy 1988]. Most recently, this idea has been applied to define satisfaction for contracts for web services [Carpineti et al. 2006; Bravetti and Zavattaro 2007; Castagna et al. 2008; Castagna and Padovani 2009].

Contracts for web services are abstract traces of processes. They are extracted from the source code of a process with the help of type annotations or a type inference algorithm. As such they are constructed in isolation for each component and they are guaranteed to meet the behavior of the process. When two processes that broadcast their contracts are brought together, their contracts are checked statically for compatibility using various forms of test pre-ordering, which implies the behavioral compatibility of the two processes.

In contrast, our software contracts are a possibly independent specification of the interaction of two components. It is not guaranteed that either component meets the contract. In this setting, contract satisfaction has to do with deciding if each of the two components meets its obligations with respect to the shared contract. In case one of the components fails to satisfy its obligation, a blame error is signaled.

The two views are dual. The first checks for compatibility of two components with separate and verified interfaces using the two interfaces. The second checks for conformance of each component with a common agreed-upon interface that regulates the interaction of the two components.

9. CONCLUSION

This paper introduces a unified framework for understanding behavioral software contract satisfaction. Instead of relying on a particular model or semantics (denotational, operational), the framework exploits observational equivalence, meaning that the results hold for all possible functionally-equivalent (adequate) semantics. Most importantly, our framework teases out the primary notions of contract satisfaction—tight and loose—and establishes a relationship between them. Additionally, the framework provides an opportunity to explore a "shy" approach to contract checking and to compare it with the basic notions.

Concerning the different incarnations of the Findler-Felleisen system, our framework classifies the approaches to contract satisfaction as shown in figure 1. Columns correspond to the instantiation of our framework that each system uses. Rows correspond to the primary purpose of the system. Section 7 explains the placement of Chitil et al. [2003], which is too low here. Hinze et al. [2006] are placed close to the border-line between loose and shy-loose because their contract system for Haskell adds a little bit of shyness to contract checking: it does not check if the argument of a function guarded by a contract satisfies the precondition of the contract unless the argument is "used" by the function. The SAGE system [Knowles et al. 2006] combines dynamic contract checking with extended static checking and thus it resides between tight contract satisfaction and loose contract satisfaction on the horizontal axis and between run-time checking and static analysis on the vertical axis. Degen et al. [2010] develop an eager contract system for an eager language and an eager and a

monitoring	satisfaction			
	tight	loose	shy-loose	shy-tight
static analysis	[1]	[2]		
run-time checking		[3],[5]	[4],[5]	[5],[6]

1 : [Xu et al. 2009], 2: [Knowles et al. 2006], 3: [Findler and Felleisen 2002]
4: [Hinze et al. 2006], 5: [Degen et al. 2010], 6: [Chitil et al. 2003]

Fig. 1.   Classification of Higher-Order Contract Systems

delayed contract checking system for a lazy language and thus they appear three times in the table. Their eager contract system for the lazy language is similar to that of Hinze et al. [2006]. Their delayed contract checking system for the lazy language is based on the same notion of contract satisfaction as Chitil et al. [2003], but its implementation differs significantly. The classification points out that certain places remain unexplored, which may be explicable with different approaches to software engineering.

Concerning the software engineering of contracts, the results suggest three different conclusions. Loose contract satisfaction is a recursively enumerable problem for whole programs. As such the Findler-Felleisen approach is a perfectly adequate basis for checking contacts. Logically the approach is in Σ_0^1, making it somewhat simpler than tight contract satisfaction, which demands a for-all quantifier and which is thus in Π_0^1. Of course, if a software tool or a software engineer can confirm that a higher-order function or a first-class object tightly satisfies some behavioral contract, the guarantees are universally useful for the working programmer. Finally, our framework suggests that the addition of shy contract checking to a language is a complex undertaking with a large design space; the community needs additional evidence that investing in this form of contracts is worthwhile.

REFERENCES

AHMED, A. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th European Symposium on Programming (ESOP)*. 69–83.

BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. 2004. The Spec# programming system: an overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*. 49–69.

BEUGNARD, A., JÉZÉQUEL, J.-M., PLOUZEAU, N., AND WATKINS, D. 1999. Making components contract aware. *IEEE Computer 32,* 7, 38–45.

BLUME, M. AND MCALLESTER, D. 2006. Sound and complete models of contracts. *Journal of Functional Programming 16,* 4-5, 375–414.

BRAVETTI, M. AND ZAVATTARO, G. 2007. Towards a unifying theory for choreography conformance and contract compliance. In *Pre-proceedings of the 6th Symposium on Software Composition*. 34–50.

CARPINETI, S., CASTAGNA, G., LANEVE, C., AND PADOVANI, L. 2006. A formal account of contracts for web services. In *In Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM)*. 148–162.

CASTAGNA, G., GESBERT, N., AND PADOVANI, L. 2008. A theory of contracts for web services. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 261–272.

CASTAGNA, G. AND PADOVANI, L. 2009. Contracts for mobile processes. In *Proceedings of the 20th International Conference in Concurrency Theory (CONCUR)*. 211–228.

CHITIL, O., MCNEILL, D., AND RUNCIMAN, C. 2003. Lazy assertions. In *Revised Papers of the 15th International Workshop on Implementation of Functional Languages (IFL)*. 1–19.

DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Egineering (ESEC/FSE)*. 109–120.

DEGEN, M., THIEMANN, P., AND WEHR, S. 2010. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *Journal of Logic and Algebraic Programming*, to appear.

DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. 1998. Extended static checking. Tech. Rep. 158, Compaq SRC Research Report.

FELLEISEN, M., FINDLER, R. B., AND FLATT, M. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

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

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

FINDLER, R. B., FELLEISEN, M., AND BLUME, M. 2004. An investigation of contracts as projections. Tech. Rep. TR-2004-02, University of Chicago, Computer Science Department.

FINDLER, R. B., GUO, S., AND ROGERS, A. 2007. Lazy contract checking for immutable data structures. In *Revised Papers of the 16th International Workshop on Implementation of Functional Languages (IFL)*. 111–128.

FINDLER, R. B., LATENDRESSE, M., AND FELLEISEN, M. 2001. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Egineering (ESEC/FSE)*. 229–236.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 COnference on Proramming Language Design and Implementation (PLDI)*. 234–245.

FLATT, M. AND PLT. 2010. Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. http://racket-lang.org/tr1/.

GREENBERG, M., PIERCE, B. C., AND WEIRICH, S. 2010. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 353–364.

GRONSKI, J. AND FLANAGAN, C. 2007. Unifying hybrid types and contracts. In *In 8th Symposium on Trends in Functional Programming(TFP)*. 54–69.

HENNESSY, M. 1988. *Algebraic Theory of Processes*. MIT Press.

HINZE, R., JEURING, J., AND LÖH, A. 2006. Typed contracts for functional programming. In *In Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS)*. 208–235.

KNOWLES, K., TOMB, A., GRONSKI, J., FREUND, S. N., AND FLANAGAN, C. 2006. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic.

KOUTAVAS, V. 2008. Reasoning about imperative and higher-order programs. Ph.D. thesis, Northeastern University.

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

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

MEYER, B. 1992a. Applying design by contract. *IEEE Computer 25,* 10, 40–51.

MEYER, B. 1992b. *Eiffel: The Language*. Prentice Hall.

MORRIS, J. H. 1968. Lambda-calculus models of programming languages. Ph.D. thesis, Massachusetts Institute of Technology.

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

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

ROSENBLUM, D. S. 1995. A practical approach to programming with assertion. *IEEE Transactions on Software Engineering 21,* 1, 15–31.

SABRY, A. AND FELLEISEN, M. 1993. Reasoning about programs in continuation passing-style. *Lisp and Symbolic Computation 6,* 3/4, 289–360.

SCOTT, D. S. 1976. Data types as lattices. *SIAM Journal of Computing 5,* 3, 522–587.

XU, D., PEYTON JONES, S., AND CLAESSEN, K. 2009. Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 41–52.