Space-Efficient Latent Contracts
Research Paper/Extended Abstract

Michael Greenberg

Pomona College
michael@cs.pomona.edu

Abstract. Standard higher-order contract monitoring breaks tail recursion and leads to space leaks that can change a program's asymptotic complexity. Space efficient semantics restore tail recursion and bound the amount of space used by contacts. Space efficient contract monitoring for contracts enforcing simple types are well studied. Prior work establishes a space-efficient semantics in a manifest setting [8]; we adapt that work to a latent calculus in the form of non-dependent contract PCF (CPCF) [1,2].

1 Introduction

Findler and Felleisen [4] brought design-by-contract [12] into the higher-order world, allowing programs to write pre- and post-conditions on functions to be checked at runtime. Pre- and post-conditions are easy in first-order languages, where it's very clear who is to blame when a contract is violated: if the pre-condition fails, blame the caller; if the post-condition fails, blame the callee. In higher-order languages, however, it's harder to tell who calls whom! Who should be to blame when a pre-condition on a higher-order function fails? For example, consider the following contract:

$$(\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x > 0) \mapsto \mathsf{pred}(\lambda y{:}\mathsf{Int}.\ y \geq 0)) \mapsto \mathsf{pred}(\lambda z{:}\mathsf{Int}.\ z \bmod 2 = 0)$$

This contract applies to a function (call it f, with type $(\mathsf{Int}{\to}\mathsf{Int}){\to}\mathsf{Int}$) that takes another function (call it g, with type $\mathsf{Int}{\to}\mathsf{Int}$) as input. The contract says that g will only be called with positives and only return naturals; f must return an even number. If f returns an odd number, f is to blame; if g returns a negative number, then it, too is to blame. But what if g is *called* with a non-positive number, say, -1? Who is to blame then? Findler and Felleisen's insight was that even in a higher-order setting, there are only two parties to blame. Here, g was given to f, any bad values given to g here are due to some nefarious action on f's part—blame f! That is, the higher-order case generalizes pre- and post-conditions so that the negative positions of a contract all blame the caller while the positive all blame the callee.

While implementations of contracts have proven quite successful (particularly so in Racket [5,14]), there is a problem: contracts leak space. Why?

The default implementation of contracts works by wrapping a function in a function proxy; to check that $f = \lambda x{:}\mathsf{Int}.\ x\ \mathsf{plus}\ 1$ satisfies the contract $C = \mathsf{pred}(\lambda z{:}\mathsf{Int}.\ z\ \mathsf{mod}\ 2 = 0) \mapsto \mathsf{pred}(\lambda z{:}\mathsf{Int}.\ z\ \mathsf{mod}\ 2 = 0)$, we monitor the function by wrapping it in a *function proxy*. When this proxy is called with an input v, we first check that v satisfies C's domain contract (i.e., that v is even), then we run f on v to get some result v', and then check that v' satisfies C's codomain contract (that the result is even). Here the contract will always fail, in one of two ways: either v or v' will always be odd.

Contracts leak space in two ways. First, there is no bound on the number of function proxies that can appear on a given function. Second, function calls that may *look like* tail calls—with no need to allocate any stack space—may actually need to check a codomain contract! If there's no bound on function proxies, then there's no bound on how many codomain contracts might show up, so there's no bound on stack allocation... and there's no way to apply tail-call optimization!

The rest of this paper discusses a formulation of contracts that enjoys sound space efficiency; that is, where we slightly change the implementation of contracts so that (a) programs are observationally equivalent, but (b) contracts consume a bounded amount of space. In this abstract, we've omitted some of the more detailed examples and motivation—we refer curious readers to Greenberg [8].

When the contracts are constrained to simple types (as in gradual typing [16]) are well studied [10,11,17,6,15]; Greenberg [8] developed a space-efficient semantics for general, non-dependent contracts.

Gradual types mediating between untyped and simply typed languages are essentially a restricted form of contracts; Greenberg [8] adapted ideas from space-efficient gradual typing to non-dependent contracts. He chose to use a manifest calculus, where contracts and types are conflated; however, contracts are typically implemented in latent calculi, where contracts are distinct from whatever types may exist. Greenberg "believe[s] it would be easy to design a latent version of eidetic λ_{H}, following the translations in Greenberg, Pierce, and Weirich (GPW) [9]; in this paper, we show that belief to be well founded by giving a space-efficient semantics for a non-dependent variant of contract PCF (CPCF) [1,2].

As a whole, this paper can be seen as a composition of Greenberg [8] and GPW [9]: we take a desirable semantics for non-dependent λ_{H}, pass it through the translations (which are exact in this case!), and find a corresponding desirable semantics for λ_{C}. Rather than using exactly λ_{C}, we have modernized our presentation by using CPCF, the main model used in the community studying latent contracts.

We follow Greenberg's general structure, defining two forms of non-dependent CPCF: *classic* CPCF is the typical semantics; *space-efficient* CPCF is space efficient, following the eidetic semantics.

We offer this model and its metatheory as our primary contribution—these results are not a major technical leap beyond Greenberg's original work, but rather a translation into another vernacular. That said, there are some new contributions here. Adding in nontermination is a real step past what Green-

berg's calculi are capable of, and makes it clear that the result from the POPL 2015 paper isn't an artifact of strong normalization (where we can, in theory, bound the size of the any term's evaluation in advance, not just contracts). The simpler type system here makes it clear which invariants are necessary for space-efficiency and which are bookkeeping for type soundness. By separating contracts and types, we can potentially give tighter bounds—the types function from Greenberg could conceivably collect types that are never used in a contract, while we collect exactly contracts.

2 Classic and space-efficient Contract PCF

We present classic and space-efficient CPCF as separate calculi sharing syntax and some typing rules (Figure 1, with standard rules omitted), and a single, parameterized operational semantics with some rules held completely in common (omitted to save space) and others specialized to each system (Figure 2). The formal presentation is modal, with two modes: C for classic and E for space-efficient. While much is shared between the two modes—types, the core syntax of expressions, e, most of the typing rules—we use colors to highlight parts that belong to only one system. Classic CPCF is typeset in salmon while space-efficient CPCF is in periwinkle.

2.1 Contract PCF (CPCF)

Plain CPCF is an extension of Plotkin's 1977 PCF [13], developed first by Dimoulas and Felleisen [1,2]. It is a simply typed language with recursion. The typing rules are straightforward (Figure 1). The operational semantics for the generic fragment also uses conventional rules (omitted to save space). Dimoulas and Felleisen use evaluation contexts to offer a concise description of their system; we write out our relation in full, giving congruence rules (E*L, E*R, EIf) and error propagating rules (E*Raise) explicitly—we will need to restrict congruence for casts, and our methods are more transparent in this way than using the nested evaluation contexts of Herman et al. [10,11], which are error prone [7].

Contracts are CPCF's distinguishing feature. Contracts, C, are installed via monitors, written $\mathsf{mon}^l(C, e)$; such a monitor can be read as saying "please ensure that e satisfies the contract C; if not, the blame lies with label l". Monitors can only be applied at appropriate types (TMon).

There are two kinds of contracts in CPCF: *predicate contracts*, written $\mathsf{pred}(e)$, and *function contracts*, written $C_1 \mapsto C_2$. Predicate contracts $\mathsf{pred}(e)$ hold a predicate on base types which identifies which values are okay or not. For example, $\mathsf{pred}(\lambda x{:}\mathsf{Nat}.\ x > 0)$ identifies the positives. Note that these predicates are exclusively over *base types*, not functions. Function contracts $C_1 \mapsto C_2$ are satisfied by functions satisfying their parts: functions whose inputs all satisfy C_1 and whose outputs all satisfy C_2. Naturally, these checks for satisfaction occur at runtime, not statically.

Syntax

Types

$$
\begin{array}{rcl}
B & ::= & \mathsf{Bool} \mid \mathsf{Nat} \mid \ldots \\
T & ::= & B \mid T_1 \to T_2
\end{array}
$$

Terms

$$
\begin{array}{rcl}
e & ::= & x \mid k \mid e_1\ op\ e_2 \mid e_1\ e_2 \mid \lambda x{:}T.\ e \mid \mu(x{:}T).\ e \mid \mathsf{if}\ e_1\ e_2\ e_3 \mid \\
 & & \mathsf{err}^l \mid \mathsf{mon}^l(C, e) \mid \mathsf{mon}(c, e) \\
op & ::= & \mathsf{add1} \mid \mathsf{sub1} \mid \ldots \\
k & ::= & \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots \\
w & ::= & v \mid \mathsf{err}^l \\
v & ::= & k \mid \lambda x{:}T.\ e \mid \mathsf{mon}^l(C_1 \mapsto C_2, v) \mid \mathsf{mon}(c_1 \mapsto c_2, \lambda x{:}T.\ e) \\
C & ::= & \mathsf{pred}(e) \mid C_1 \mapsto C_2 \\
c & ::= & r \mid c_1 \mapsto c_2 \\
r & ::= & \mathsf{nil} \mid \mathsf{pred}^l(e); r
\end{array}
$$

Typing rules $\boxed{\Gamma \vdash e : T}$ $\boxed{\vdash C : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{TVar}$$

$$\frac{}{\Gamma \vdash k : \mathsf{ty}(k)}\ \textsc{TConst}$$

$$\frac{}{\Gamma \vdash \mathsf{err}^l : T}\ \textsc{TBlame}$$

$$\frac{\Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T_1 \to T_2}\ \textsc{TAbs} \qquad \frac{\Gamma, x{:}T \vdash e : T}{\Gamma \vdash \mu(x{:}T).\ e : T}\ \textsc{TRec}$$

$$\frac{\mathsf{ty}(op) = T_1 \to T_2 \to T \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ op\ e_2 : T}\ \textsc{TOp} \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \textsc{TApp}$$

$$\frac{\Gamma \vdash e : T \quad \vdash C : T}{\Gamma \vdash \mathsf{mon}^l(C, e) : T}\ \textsc{TMon} \qquad \frac{\Gamma \vdash e : T \quad \vdash c : T}{\Gamma \vdash \mathsf{mon}(c, e) : T}\ \textsc{TMonC}$$

$$\frac{\emptyset \vdash e : B \to \mathsf{Bool}}{\vdash \mathsf{pred}(e) : B}\ \textsc{TPred} \qquad \frac{\vdash C_1 : T_1 \quad \vdash C_2 : T_2}{\vdash C_1 \mapsto C_2 : T_1 \to T_2}\ \textsc{TFun}$$

$$\frac{}{\vdash \mathsf{nil} : B}\ \textsc{TCNil} \qquad \frac{\forall \mathsf{pred}^{l'}(e') \in r.\ \mathsf{pred}(e) \not\supset \mathsf{pred}(e') \quad \vdash \mathsf{pred}(e) : B \quad \vdash r : B}{\vdash \mathsf{pred}^l(e); r : B}\ \textsc{TCPred}$$

$$\frac{\vdash c_1 : T_1 \quad \vdash c_2 : T_2}{\vdash c_1 \mapsto c_2 : T_1 \to T_2}\ \textsc{TCFun}$$

Fig. 1. Syntax and typing rules of classic and space-efficient CPCF

$$\frac{}{\mathsf{mon}^l(\mathsf{pred}(e_1), v_2) \longrightarrow_{\mathsf{C}} \mathsf{if}\ (e_1\ v_2)\ v_2\ \ \mathsf{err}^l}\ \textsc{EMonPred}$$

$$\frac{}{\mathsf{mon}^l(C_1 \mapsto C_2, v_1)\ v_2 \longrightarrow_{\mathsf{C}} \mathsf{mon}^l(C_2, v_1\ \mathsf{mon}^l(C_1, v_2))}\ \textsc{EMonApp}$$

$$\frac{e \longrightarrow_{\mathsf{C}} e'}{\mathsf{mon}^l(C, e) \longrightarrow_{\mathsf{C}} \mathsf{mon}^l(C, e')}\ \textsc{EMon} \qquad \frac{}{\mathsf{mon}^l(C, \mathsf{err}^{l'}) \longrightarrow_{\mathsf{C}} \mathsf{err}^{l'}}\ \textsc{EMonRaise}$$

$$\frac{}{\mathsf{mon}^l(C, e) \longrightarrow_{\mathsf{E}} \mathsf{mon}(\mathsf{label}^l(C), e)}\ \textsc{EMonLabel}$$

$$\frac{}{\mathsf{mon}(\mathsf{nil}, v_1) \longrightarrow_{\mathsf{E}} v_1}\ \textsc{EMonCNil}$$

$$\frac{}{\mathsf{mon}((\mathsf{pred}^l(e); r), v_1) \longrightarrow_{\mathsf{E}} \mathsf{if}\ (e\ v_1)\ \mathsf{mon}(r, v_1)\ \ \mathsf{err}^l}\ \textsc{EMonCPred}$$

$$\frac{}{\mathsf{mon}(c_1 \mapsto c_2, v_1)\ v_2 \longrightarrow_{\mathsf{E}} \mathsf{mon}(c_2, v_1\ \mathsf{mon}(c_1, v_2))}\ \textsc{EMonCApp}$$

$$\frac{e \neq \mathsf{mon}(c', e'') \quad e \longrightarrow_{\mathsf{E}} e'}{\mathsf{mon}(c, e) \longrightarrow_{\mathsf{E}} \mathsf{mon}(c, e')}\ \textsc{EMonC} \qquad \frac{}{\mathsf{mon}(c, \mathsf{err}^l) \longrightarrow_{\mathsf{E}} \mathsf{err}^l}\ \textsc{EMonCRaise}$$

$$\frac{}{\mathsf{mon}(c_2, \mathsf{mon}(c_1, e)) \longrightarrow_{\mathsf{E}} \mathsf{mon}(\mathsf{join}(c_1, c_2), e)}\ \textsc{EMonCJoin}$$

Fig. 2. Operational semantics of classic and space-efficient CPCF

2.2 Classic Contract PCF ($\mathrm{CPCF}_{\mathsf{C}}$)

Classic CPCF gives a straightforward semantics to contracts, largely following the seminal work by Findler and Felleisen [4]. To check a predicate contract, we simply run the test (EMonPred), returning either the value or an appropriately labeled error. Function contracts are deferred, treating $\mathsf{mon}^l(C_1 \mapsto C_2, v)$ as a *value*; we call this value a *function proxy*. When a function proxy is given an argument, it unwraps the proxy, checking the argument against the domain contract, running the function, and then checking the return value on the codomain contract. Standard congruence rules allow for evaluation inside of monitors (EMon) and the propagation of errors (EMonRaise).

2.3 Space-efficient Contract PCF ($\mathrm{CPCF}_{\mathsf{E}}$)

How can we recover tail calls in CPCF? $\mathrm{CPCF}_{\mathsf{C}}$ will happily wrap arbitrarily many function proxies around a value, and there's no bound on the number of

$$\begin{aligned}
\mathsf{label}^l(\mathsf{pred}(e_1)) &= \mathsf{pred}^l(e_1) \\
\mathsf{label}^l(C_1 \mapsto C_2) &= \mathsf{label}^l(C_1) \mapsto \mathsf{label}^l(C_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{join}(\mathsf{nil}, r_2) &= r_2 \\
\mathsf{join}(\mathsf{pred}^l(e); r_1, r_2) &= \mathsf{pred}^l(e); \mathsf{drop}(\mathsf{join}(r_1, r_2), \mathsf{pred}(e)) \\
\mathsf{join}(c_{11} \mapsto c_{12}, c_{21} \mapsto c_{22}) &= \mathsf{join}(c_{21}, c_{11}) \mapsto \mathsf{join}(c_{12}, c_{22})
\end{aligned}$$

$$\begin{aligned}
\mathsf{drop}(\mathsf{nil}, \mathsf{pred}(e)) &= \mathsf{nil} \\
\mathsf{drop}(\mathsf{pred}^l(e_1); r, \mathsf{pred}(e)) &= \begin{cases} \mathsf{drop}(r, \mathsf{pred}(e)) & \mathsf{pred}(e) \supset \mathsf{pred}(e_1) \\ \mathsf{pred}^l(e_1); \mathsf{drop}(r, \mathsf{pred}(e)) & \mathsf{pred}(e) \not\supset \mathsf{pred}(e_1) \end{cases}
\end{aligned}$$

Fig. 3. Contract labeling and predicate stack management

codomain contract checks that can accumulate. The key idea is *joining* contracts. We'll make two changes to the language: we'll bound function proxies and we'll bound stacks. To ensure a function value can have only one proxy, we'll change the semantics of monitoring: when we try to apply a function contract monitor to a value, we'll *join* the new monitor and the old one. To put bounds on the size of a stack of contract checks, we'll explicitly model stacks of predicate contracts, being careful to avoid redundancy. Since there are a finite number of predicate contracts in a given starting program, there can only be finitely many predicates in a given stack. Fortuitously, our notion of join solves both of our problems—killing two birds with one stone.

We have strived to use Greenberg's notation exactly, but we have still made two small, clarifying changes:

– the cons operator for predicate stacks is a semi-colon, to avoid ambiguity;
– there were formerly two things named join, but one has been folded into the other.

Before we can give the semantics for our join operation in detail, we need to establish a notion of predicate implication: when does one contract imply another? We'll use this relation to avoid redundancy (and so bound space consumption).

Definition 1 (Predicate implication). *Let* $\boxed{\mathsf{pred}(e_1) \supset \mathsf{pred}(e_2)}$ *be a relation on predicate such that:*

(Reflexivity) *If* $\vdash \mathsf{pred}(e) : B$ *then* $\mathsf{pred}(e) \supset \mathsf{pred}(e)$.
(Transitivity) *If* $\mathsf{pred}(e_1) \supset \mathsf{pred}(e_2)$ *and* $\mathsf{pred}(e_2) \supset \mathsf{pred}(e_3)$, *then* $\mathsf{pred}(e_1) \supset \mathsf{pred}(e_3)$.
(Adequacy) *If* $\mathsf{pred}(e_1) \supset \mathsf{pred}(e_2)$ *then* $\forall k \in \mathcal{K}_B.\ e_1\ k \longrightarrow_{\mathsf{m}} \mathsf{true}$ *implies* $e_2\ k \longrightarrow_{\mathsf{m}} \mathsf{true}$.
(Decidability) *For all* $\vdash \mathsf{pred}(e_1) : B$ *and* $\vdash \mathsf{pred}(e_2) : B$, *it is decidable whether* $\mathsf{pred}(e_1) \supset \mathsf{pred}(e_2)$ *or* $\mathsf{pred}(e_1) \not\supset \mathsf{pred}(e_2)$.

The entire development of space-efficiency is parameterized over this implication relation, $\supset$, that characterizes when one first-order contract subsumes

another. We write $\not\supset$ for the negation of $\supset$. There is at least one workable implication relation: syntactic equality.

The $\supset$ relation is a *total pre-order* (a/k/a a *preference relation*)—it would be a total order, but it may not necessarily enjoy anti-symmetry. For example, we could have $\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \geq 0) \supset \mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x\ \mathsf{plus}\ 1 > 0)$ and vice versa, even though the two predicates aren't equal. Alternatively, you could view $\supset$ as a total order *up-to contextual equivalence*.

To achieve our space-efficient semantics over the same set of terms, we introduce *labeled contracts*, written c (as opposed to C), comprising function contracts as usual ($c_1 \mapsto c_2$) and predicate stacks. *Predicate stacks* r are lists of *labeled predicates* $\mathsf{pred}^l(e)$: they are either empty (written nil) or a labeled predicate $\mathsf{pred}^l(e)$ cons-ed on to another predicate stack.

We add an evaluation rule taking ordinary contract monitors $\mathsf{mon}^l(C, e)$ to labeled-contract monitors $\mathsf{mon}(c, e)$ by means of the labeling function label (EMonLabel).

Space-efficiency comes by restricting congruence to only apply when there are abutting monitors (cf. EMonC here in CPCF_E to EMon in CPCF_C). When two monitors collide, we *join* them (EMonCJoin). Checking function contracts is as usual (EMonCApp is the same as EMonApp, only the latter works over labeled contracts); checking predicate stacks proceeds straightforwardly predicate-by-predicate (EMonCNil and EMonCPred).

The only interesting typing rules for CPCF_E are those for predicate stacks, which enforce the invariant that there are no redundant predicates in a stack: $\mathsf{pred}^l(e); r$ is only well typed when there are no checks in r that do the same work as $\mathsf{pred}^l(e)$, i.e., $\mathsf{pred}(e) \not\supset \mathsf{pred}(e')$ for all $\mathsf{pred}^{l'}(e') \in r$.

3 Soundness for space efficiency

CPCF_C and CPCF_E are operationally equivalent, even though their cast semantics differ. We can make this connection formal by proving that if every CPCF term either: (a) diverges in both CPCF_C and CPCF_E or (b) goes to a equivalent term in both CPCF_C and CPCF_E.

One minor technicality: some of the forms in our language are necessary only for runtime or only appear in one of the two calculi. We characterize *source programs* as those which omit runtime terms (and, it turns out, can be typed in either mode).

Definition 2 (Source program). *A well typed source program does not use* TBlame *or* TMonC *(and so* TCNil*,* TCPred*, and* TCFun *cannot be used).*

Greenberg identified the key property for proving soundness of a space efficient semantics: to be sound, the space-efficient semantics must recover a notion of congruence for checking. In his manifest setting, he calls it *cast congruence*; since CPCF uses contract monitors rather than casts, we call it here *monitor congruence*.

Lemma 1 (Monitor congruence (single step)). *If* $\emptyset \vdash e_1 : T$ *and* $\vdash c : T$ *and* $e_1 \longrightarrow_{\mathsf{E}} e_2$*, then* $\mathsf{mon}(c, e_1) \longrightarrow_{\mathsf{E}}^* w$ *iff* $\mathsf{mon}(c, e_2) \longrightarrow_{\mathsf{E}}^* w$*.*

Proof. By cases on the step taken to find $e_1 \longrightarrow_{\mathsf{E}} e_2$*. In the easy case, there's no merging of coercions and the same rule can apply in both derivations. In the more interesting case, two contract monitors join. In either case, it suffices to show that the terms are ultimately confluent, since determinism will do the rest.*

It is particularly satisfying that the key property for showing soundness of space efficiency can be proved independently of the inefficient semantics. Implementors can therefore work entirely in the context of the space-efficient semantics, knowing that as long as they have congruence, they're sound.

We, however, go to the trouble to show the observational equivalence of CPCF_{C} and CPCF_{E}. The proof is by logic relations (omitted for space), which gives us contextual equivalence—the strongest equivalence we could ask for.

Lemma 2 (Similar contracts are logically related). *If* $C_1 \sim C_2 : T$ *and* $v_1 \sim v_2 : T$ *then* $\mathsf{mon}^l(C_1, v_1) \sim \mathsf{mon}^l(C_2, v_2) : T$*.*

Proof. By induction on the (type index of the) invariant relation $C_1 \sim C_2 : T$*.*

Lemma 3 (Unwinding). *If* $\emptyset \vdash \mu(x{:}T).\ e : T$*, then* $\mu(x{:}T).\ e --> m * w$ *iff there exists an* n *such that unrolling the fixpoint only* n *times converges to the same value, i.e.,* $e[\mu(x{:}T).\ \ldots\ e[\mu(x{:}T).\ e/x]\ \ldots/x] \longrightarrow_{\mathsf{m}}^* w$*.*

Theorem 1 (Classic and space-efficient CPCF terms are logically related).

1. *If* $\Gamma \vdash e : T$ *as a source program then* $\Gamma \vdash e \simeq e : T$*.*
2. *If* $\vdash C : T$ *then* $C \sim C : T$*.*

Proof. By mutual induction on the typing relations.

4 Bounds for space efficiency

We have seen that CPCF_{E} behaves the same as CPCF_{C} (Theorem 1), but is CPCF_{E} actually space efficient? To see why, observe that a given source program e starts with a finite number of predicate contracts in it. As e runs, no new predicates appear, but these predicates may accumulate in stacks. In the worst case, a predicate stack could contain every predicate contract from the original program e exactly once... but no more than that! Function contracts are also bounded: e starts out with function contracts of a certain height, and evaluation can only shrink that height. The leaves of function contracts are labeled with predicate stacks, so the largest contract we could ever see is of maximum height with maximal predicate stacks at every leaf. As the program runs, abutting monitors are joined, giving us a bound on the total number of monitors in a program (one per non-monitor AST node).

We can make these ideas formal by first defining what we mean by "all the predicates in a program" (we'll write it $\mathsf{preds}(e)$, with details omitted for space), and then showing that evaluation doesn't introduce predicates (Lemma 4).

Lemma 4 (Reduction is monotonically decreasing in predicates). *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow_{\mathsf{m}} e'$ *then* $\mathsf{preds}(e') \subseteq \mathsf{preds}(e)$.

Proof. By induction on the step taken.

To be concrete, let $P_B = |\{e \in \mathsf{preds}(e) \mid\vdash \mathsf{pred}(e) : B\}|$ be the number of distinct predicates at type B; let $P^* = \max_B P_B$ be the maximal number of distinct predicates at any base type. No predicate stack can have more than P^* entries. Let $W^* = \log_2 \max_B P_B$ be the maximal number of predicates that appear at a maximal base type B. Predicate stacks can therefore be represented in $P^* \cdot W^*$ space in the worst case. We can bound the number of predicate stacks by the size of the largest function type used for a contract int he program, $s = \max\{\mathsf{size}(C) \mid C \in e\wedge \vdash C : T_1 \to T_2\}$. Supposing that a label can be represented in L bits, then each predicate stack can be represented in $T = W^* \cdot L$ bits, and all contract monitors can be represented in at $s \cdot P^* \cdot T$ bits. Readers familiar with Greenberg's paper (and earlier work, like Herman et al. [10]) will notice that these bounds are slightly different. Our new bounds are more precise, tracking the number of holes in an actual contract ($s = \mathsf{size}(C)$) rather than simply computing the height of a type ($2^{\mathsf{height}(T)}$).

5 Related work

For the technique of space efficiency itself, we refer the reader to Greenberg [8] for a full description of related work.

CPCF was first introduced in several papers by Dimoulas et al. in 2011 [1,2], and has later been the subject of studies of blame for dependent function contracts [3] and static analysis [18]. Our exact behavioral equivalence means we could use results from Tobin-Hochstadt et al.'s static analysis in terms of CPCF_{C} to optiimze space efficient programs in CPCF_{E}. More interestingly, the predicate implication relation $\supset$ seems to be doing some of the work that a static analysis might do, so there may be a deeper relationship.

One piece of very closely related work is Racket's `tail-marks-match?` predicate[1], which offers a less thorough form of space efficiency. We hope that the presentation of CPCF_{E} is close enough to Racket's internal model to provide insight into how to achieve space efficiency for at least some contracts in Racket.

Finally, the manifest type system in Greenberg's work is somewhat disappointing compared to the type system given here: Greenberg works much harder than we do to prove a stronger type soundness theorem... but that theorem isn't enough to help materially in proving the soundness of space efficiency.

6 Conclusion

We have translated Greenberg's original result [8] from a manifest calculus [9] to a latent one [1,2]. In so doing, we have: offered a simpler explanation of the original result; isolated the parts of the type system required for space bounds, which

[1] From `racket/collects/racket/contract/private/arrow.rkt`.

were intermingled with complexities from conflating contracts and types; and, extended the original result, both in terms of features covered (nontermination) and in terms of the precision of bounds.

Acknowledgments

The existence of this paper is due to comments from Sam Tobin-Hochstadt and David Van Horn that I chose to interpret as encouragement.

References

1. Dimoulas, C., Felleisen, M.: On contract satisfaction in a higher-order world. TOPLAS 33(5), 16:1–16:29 (Nov 2011)
2. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: no more scapegoating. In: Principles of Programming Languages (POPL) (2011)
3. Dimoulas, C., Tobin-Hochstadt, S., Felleisen, M.: Complete monitors for behavioral contracts. In: Programming Languages and Systems, vol. 7211 (2012)
4. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: International Conference on Functional Programming (ICFP) (2002)
5. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010), `http://racket-lang.org/tr1/`
6. Garcia, R.: Calculating threesomes, with blame. In: International Conference on Functional Programming (ICFP) (2013)
7. Greenberg, M.: Manifest Contracts. Ph.D. thesis, University of Pennsylvania (November 2013)
8. Greenberg, M.: Space-efficient manifest contracts. In: POPL (2015)
9. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: Principles of Programming Languages (POPL) (2010)
10. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Programming (TFP). pp. 404–419 (2007)
11. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. Higher Order Symbol. Comput. 23(2), 167–189 (Jun 2010)
12. Meyer, B.: Eiffel: the language. Prentice-Hall, Inc. (1992)
13. Plotkin, G.: Lcf considered as a programming language. Theoretical Computer Science 5(3), 223 – 255 (1977)
14. Racket contract system (2013), `http://pre.plt-scheme.org/docs/html/guide/contracts.html`
15. Siek, J., Thiemann, P., Wadler, P.: Blame, coercion, and threesomes: Together again for the first time (2015), `http://homepages.inf.ed.ac.uk/wadler/topics/blame.html#coercions`
16. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
17. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Principles of Programming Languages (POPL). pp. 365–376 (2010)
18. Tobin-Hochstadt, S., Van Horn, D.: Higher-order symbolic execution via contracts. In: OOPSLA. pp. 537–554. OOPSLA '12, ACM (2012)