Space-Efficient Latent Contracts

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 without dependency [8]; we adapt that work to a latent calculus and extend it work with dependent contracts in contract PCF (CPCF) [1,2].

1 Introduction

Findler and Felleisen [4] brought design-by-contract [13] 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.

Dependent contracts—where the codomain contract can refer to the function's argument—are particularly useful. For example, we can state the contract on the sqrt function as: $x{:}\mathsf{pred}(\lambda y{:}\mathsf{Real}.\ y \geq 0) \mapsto \mathsf{pred}(\lambda z{:}\mathsf{Real}.\ \mathsf{abs}\ (x - z * z) < \epsilon)$ That is, the square root function sqrt takes a non-negative real, x, and returns a non-negative real z that's within ϵ of the square root of x. (The dependent variable x is bound in the codomain; the variable y is local to the predicate.)

$$\begin{array}{l} \mathsf{let\ odd} \quad = \mathsf{mon}^{l_{\mathsf{odd}}}(x{:}\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \geq 0) \mapsto \mathsf{pred}(\lambda b{:}\mathsf{Bool}.\ b\,\mathsf{or}\,(x \bmod 2 = 0)), \\ \qquad\qquad\quad \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ \mathsf{false}\ \ (\mathsf{even}\ (x - 1))) \\ \mathsf{and\ even} = \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ \mathsf{true}\ \ (\mathsf{odd}\ (x - 1)) \end{array}$$

$$\begin{array}{ll} & \mathsf{odd}\ 5 \\ \longrightarrow^*_{\mathsf{C}} & \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{even}\ 4) \\ \longrightarrow^*_{\mathsf{C}} & \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \\ & \quad \mathsf{odd}\ \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \geq 0), 3))) \\ \longrightarrow^*_{\mathsf{C}} & \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \mathsf{even}\ 2)) \\ \longrightarrow^*_{\mathsf{C}} & \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 1]}(\dots), \\ & \quad \mathsf{odd}\ \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \geq 0), 1)))) \\ \longrightarrow^*_{\mathsf{C}} & \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 1]}(\dots), \mathsf{even}\ 0))) \end{array}$$

Fig. 1. Contracts break tail recursion

1.1 Contracts leak space

While implementations of contracts have proven quite successful (particularly so in Racket [6,15]), 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 $f = \lambda x{:}\mathsf{Int}.\ x + 1$ satisfies the contract $C = \mathsf{pred}(\lambda z{:}\mathsf{Int}.\ z \bmod 2 = 0) \mapsto \mathsf{pred}(\lambda z{:}\mathsf{Int}.\ z \bmod 2 = 0)$, 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. More grievously, contracts break tail recursion. To demonstrate the issue with with tail calls, we'll use the simplest example of mutual recursion: detecting parity.

$$\begin{array}{l} \mathsf{let\ odd} \quad = \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ \mathsf{false}\ \ (\mathsf{even}\ (x - 1)) \\ \mathsf{and\ even} = \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ \mathsf{true}\ \ (\mathsf{odd}\ (x - 1)) \end{array}$$

Functional programmers will expect this program to run in constant space, because it is *tail recursive*. Adding a contract breaks the tail recursion.[1] If we add a contract to odd and call odd 5, what contract checks accumulate (Figure 1)? Notice how the checks accumulate in the codomain? Even though the mutually recursive calls to even and odd are syntactically tail calls, we can't bound the number of codomain checks that occur. That is, we can't bound the size of the

[1] Readers may observe that the contract betrays a deeper knowledge of numbers than the functions themselves. We offer this example as the smallest conceivable, not as a naturally occurring issue.

stack, and tail recursion is broken! Even though there's only one function proxy on `odd`, our contracts create a space leak.

1.2 Overview and contributions

Space efficiency for gradual types [19] (a/k/a contracts constrained to type tests) is well studied [10,11,20,7,18]; Greenberg [8] developed a space-efficient semantics for general, 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 (dependent!) variant of contract PCF (CPCF) [1,2].

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].

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

We offer two primary contributions: adapting Greenberg's work to a latent calculus and extending space efficiency to dependent contracts.

There are some other, smaller, contributions as well. Adding in nontermination moves beyond Greenberg's strongly normalizing calculi. 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 type system invariants are necessary for space-efficiency and which are bookkeeping for proving that the more complicated manifest type system is sound. Finally, by separating contracts and types, we potentially give tighter space 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 2 and Figure 3), and a single, parameterized operational semantics with some rules held completely in common (omitted to save space) and others specialized to each system (Figure 4). 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.

$$
\begin{array}{llcl}
\textbf{Types} & B & ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \ldots \\
 & T & ::= & B \mid T_1 \to T_2 \\
\textbf{Terms} & e & ::= & x \mid k \mid e_1 \mathit{op}\, e_2 \mid e_1\ e_2 \mid \lambda x{:}T.\ e \mid \mu(x{:}T).\ e \mid \mathsf{if}\ e_1\ e_2\ e_3 \mid \\
 & & & \mathsf{err}^l \mid \mathsf{mon}^l(C, e) \mid \mathsf{mon}(c, e) \\
 & \mathit{op} & ::= & \mathsf{add1} \mid \mathsf{sub1} \mid \ldots \\
 & k & ::= & \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots \\
 & w & ::= & v \mid \mathsf{err}^l \\
 & v & ::= & k \mid \lambda x{:}T.\ e \mid \mathsf{mon}^l(x{:}C_1 \mapsto C_2, v) \mid \mathsf{mon}(x{:}c_1 \mapsto c_2, \lambda x{:}T.\ e) \\
 & C & ::= & \mathsf{pred}_\sigma(e) \mid x{:}C_1 \mapsto C_2 \\
 & c & ::= & r \mid x{:}c_1 \mapsto c_2 \\
 & r & ::= & \mathsf{nil} \mid \mathsf{pred}^l_\sigma(e); r
\end{array}
$$

Fig. 2. Syntax of classic and space-efficient CPCF

2.1 Contract PCF (CPCF)

Plain CPCF is an extension of Plotkin's 1977 PCF [14], developed first by Dimoulas and Felleisen [1,2] (our syntax is in Figure 2). It is a simply typed language with recursion. The typing rules are straightforward (Figure 3). 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].

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* over base type, written $\mathsf{pred}_\sigma(e)$, and *function contracts*, written $x{:}C_1 \mapsto C_2$.

Predicate contracts $\mathsf{pred}_\sigma(e)$ have two parts: a predicate on base types, e, which identifies which values are okay or not; and a closing substitution σ which keeps track of values substituted into contracts. For example:

– $\mathsf{pred}_\iota(\lambda x{:}\mathsf{Int}.\ x > 0)$ identifies the positives;
– $\mathsf{pred}_\iota(\lambda x{:}\mathsf{Int}.\ x > y)$ identifies numbers greater than an unspecified number y; and,
– $\mathsf{pred}_{[y \mapsto 47]}(\lambda x{:}\mathsf{Int}.\ x > y)$ identifies numbers greater than 47.

When the closing substitution σ is the identity mapping, we write $\mathsf{pred}(e)$ instead of $\mathsf{pred}_\iota(e)$. In CPCF$_\mathsf{C}$, closing substitutions will map each variable to either (a) itself or (b) a value. Predicates are solely over *base types*, not functions.

Function contracts $x{:}C_1 \mapsto C_2$ are satisfied by functions satisfying their parts: functions whose inputs all satisfy C_1 and whose outputs all satisfy C_2. Function contracts are dependent: the codomain contract C_2 can refer back to the input to the function. For example, the contract $x{:}\mathsf{pred}(\lambda z{:}\mathsf{Int}.\ z > 0) \mapsto \mathsf{pred}(\lambda y{:}\mathsf{Int}.\ y > x)$ is satisfied by increasing functions on the positives. Note that x is bound in the codomain, but z is not.[2] When functions aren't dependent, we omit the binder at the front, e.g., $\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x > 0) \mapsto \mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x > 0)$ to mean operators on positives. These checks for satisfaction occur at runtime, not statically.

We use explicit, delayed substitutions to keep track of which values are substituted into predicate contracts. We make these substitutions explicit for two reasons. The primary motivation is to emphasize the way our predicate implication relation (Definition 1) could be realized as an operation on closures; secondarily, explicit substitutions make it easier to prove the space-efficient calculus sound. In particular, we have:

$$\mathsf{pred}_\sigma(e)[v/x] = \begin{cases} \mathsf{pred}_{\sigma[x\mapsto v]}(e) & x \in \mathrm{fv}(\sigma(e)) \\ \mathsf{pred}_\sigma(e) & \text{otherwise} \end{cases}$$

Alpha equivalence allows us to rename variables in the domain of σ as long as we consistently rename those variables inside of the predicate e. We explicitly keep only those values which are closing up free variables in e as a way of modeling closures. A dependent predicate closes over some finite number of variables; a compiled representation would generate a closure with some number of slots in the closing environment. Restricting substitutions to exactly those variables that appear free in the predicate serves another purpose: it will allow us to recover space-efficiency bounds for programs without dependent contracts (Section 4.1).

2.2 Classic Contract PCF (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 test it (EMonPred), returning either the value or an appropriately labeled error. Function contracts are deferred: $\mathsf{mon}^l(x{:}C_1 \mapsto C_2, v)$ is a *value*, called a *function proxy.* When a function proxy is given an argument, it unwraps the proxy, monitoring the argument with the domain contract, running the function, and then monitoring the return value with the codomain (EMonApp).

Our semantics may seem to be a *lax*, where no monitor is applied to dependent uses of the argument in the codomain monitor [9]. In fact, it is agnostic: we could be *picky* by requiring that function contract monitors $\mathsf{mon}^l(x{:}C_1 \mapsto C_2, e)$ have the substitution $[x \mapsto \mathsf{mon}^l(C_1, x)]$ throughout C_2; we could be *indy* by having $[x \mapsto \mathsf{mon}^{l'}(C_1, x)]$ throughout C_2 [2]. We default to a lax rule to make our proof of soundness easier, but we'll have as a corollary that classic and space-efficient semantics yield the same result regardless of what the closing substitutions do in the codomain (Section 3).

Standard congruence rules allow for evaluation inside of monitors (EMon) and the propagation of errors (EMonRaise).

[2] Concrete syntax for such predicates can be written much more nicely, but we ignore such concerns here.

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

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{TVar} \qquad \frac{}{\Gamma \vdash k : \mathsf{ty}(k)}\ \textsc{TConst} \qquad \frac{}{\Gamma \vdash \mathsf{err}^l : T}\ \textsc{TBlame}$$

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

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

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : T}\ \textsc{TIf}$$

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

Contract typing $\boxed{\Gamma \vdash C : T}$ $\boxed{\Gamma \vdash c : T}$

$$\frac{\Gamma, \Gamma' \vdash e : B \rightarrow \mathsf{Bool} \quad \Gamma' \vdash \sigma}{\Gamma \vdash \mathsf{pred}_\sigma(e) : B}\ \textsc{TPred} \qquad \frac{\Gamma \vdash C_1 : T_1 \quad \Gamma, x{:}T_1 \vdash C_2 : T_2}{\Gamma \vdash x{:}C_1 \mapsto C_2 : T_1 \rightarrow T_2}\ \textsc{TFun}$$

$$\frac{}{\Gamma \vdash \mathsf{nil} : B}\ \textsc{TCNil} \qquad \frac{\Gamma \vdash \mathsf{pred}_\sigma(e) : B \quad \Gamma \vdash r : B}{\Gamma \vdash \mathsf{pred}^l_\sigma(e); r : B}\ \textsc{TCPred}$$

$$\frac{\Gamma \vdash c_1 : T_1 \quad \Gamma, x{:}T_1 \vdash c_2 : T_2}{\Gamma \vdash x{:}c_1 \mapsto c_2 : T_1 \rightarrow T_2}\ \textsc{TCFun}$$

Closing substitutions $\boxed{\Gamma \vdash \sigma}$

$$\frac{}{\emptyset \vdash \iota}\ \textsc{TId} \qquad \frac{\Gamma \vdash \sigma \quad x{:}T \vdash v : T}{\Gamma, x{:}T \vdash \sigma[x \mapsto v]}\ \textsc{TMap}$$

Fig. 3. Typing rules of classic and space-efficient CPCF

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

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

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

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

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

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

$$\frac{}{\mathsf{mon}(x{:}c_1 \mapsto c_2, v_1)\ v_2 \longrightarrow_{\mathsf{E}} \mathsf{mon}(c_2[v_2/x], v_1\ \mathsf{mon}(c_1, v_2))}\ \textsc{EMonCApp}$$

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

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

Fig. 4. Operational semantics of classic and space-efficient CPCF

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

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

We have strived to use Greenberg's notation exactly, but we have still made some changes in adapting to dependent contracts: the cons operator for predicate stacks is a semi-colon, to avoid ambiguity; there were formerly two things

$$\begin{aligned}
\mathsf{label}^l(\mathsf{pred}_\sigma(e_1)) &= \mathsf{pred}^l_\sigma(e_1)\\
\mathsf{label}^l(x{:}C_1 \mapsto C_2) &= x{:}\mathsf{label}^l(C_1) \mapsto \mathsf{label}^l(C_2)\\[1em]
\mathsf{join}(\mathsf{nil}, r_2) &= r_2\\
\mathsf{join}(\mathsf{pred}^l_\sigma(e); r_1, r_2) &= \mathsf{pred}^l_\sigma(e); \mathsf{drop}(\mathsf{join}(r_1, r_2), \mathsf{pred}_\sigma(e))\\
\mathsf{join}(x{:}c_{11} \mapsto c_{12}, x{:}c_{21} \mapsto c_{22}) &= x{:}\mathsf{join}(c_{21}, c_{11}) \mapsto \mathsf{join}(\mathsf{wrap}(c_{12}, x, c_{22}), c_{22})\\[1em]
\mathsf{drop}(\mathsf{nil}, \mathsf{pred}_\sigma(e)) &= \mathsf{nil}\\
\mathsf{drop}(\mathsf{pred}^l_{\sigma_2}(e_2); r, \mathsf{pred}_{\sigma_1}(e_1)) &= \\
&\begin{cases} \mathsf{drop}(r, \mathsf{pred}_{\sigma_2}(e_2)) & \mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)\\ \mathsf{pred}^l_{\sigma_2}(e_2); \mathsf{drop}(r, \mathsf{pred}_{\sigma_1}(e_1)) & \mathsf{pred}_{\sigma_1}(e_1) \not\supset \mathsf{pred}_{\sigma_2}(e_2)\end{cases}\\[1em]
\mathsf{wrap}(\mathsf{pred}^l_\sigma(e), x, c) &= \begin{cases} \mathsf{pred}^l_\sigma(e) & x \notin \mathrm{fv}(\sigma(e))\\ \mathsf{pred}^l_{\sigma[x \mapsto \mathsf{mon}(\mathsf{join}(c', c), e)]}(e) & \sigma(x) = \mathsf{mon}(c', e)\\ \mathsf{pred}^l_{\sigma[x \mapsto \mathsf{mon}(c, \sigma(x))]}(e) & \text{otherwise}\end{cases}\\
\mathsf{wrap}(\mathsf{nil}, x, c) &= \mathsf{nil}\\
\mathsf{wrap}(\mathsf{pred}^l_\sigma(e); r, x, c) &= \mathsf{wrap}(\mathsf{pred}^l_\sigma(e), x, c); \mathsf{wrap}(r, x, c)\\
\mathsf{wrap}(y{:}c_1 \mapsto c_2, x, c) &= y{:}\mathsf{wrap}(c_1, x, c) \mapsto \mathsf{wrap}(c_2, x, c)
\end{aligned}$$

Fig. 5. Contract labeling and predicate stack management

named join, but one has been folded into the other; and predicates have closing substitutions. 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?

Definition 1 (Predicate implication). *Let* $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$ *be a relation on predicates such that:*

(Reflexivity) *If* $\emptyset \vdash \mathsf{pred}_\sigma(e) : B$ *then* $\mathsf{pred}_\sigma(e) \supset \mathsf{pred}_\sigma(e)$.

(Transitivity) *If* $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$ *and* $\mathsf{pred}_{\sigma_2}(e_2) \supset \mathsf{pred}_{\sigma_3}(e_3)$, *then* $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_3}(e_3)$.

(Substitutivity) *When* $\Gamma_{i\,1}, x{:}T', \Gamma_{i\,2} \vdash \mathsf{pred}_{\sigma_i}(e_i) : T$ *and* $\emptyset \vdash v : T'$, *if* $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$ *then* $\mathsf{pred}_{\sigma_1}(e_1)[v/x] \supset \mathsf{pred}_{\sigma_2}(e_2)[v/x]$.

(Adequacy) *If* $\emptyset \vdash \mathsf{pred}_{\sigma_i}(e_i) : T$ *and* $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$ *then* $\forall k \in \mathcal{K}_B.\ \sigma_1(e_1)\ k \longrightarrow_{\mathsf{m}} \mathsf{true}$ *implies* $\sigma_2(e_2)\ k \longrightarrow_{\mathsf{m}} \mathsf{true}$.

(Decidability) *For all* $\emptyset \vdash \mathsf{pred}_{\sigma_1}(e_1) : B$ *and* $\emptyset \vdash \mathsf{pred}_{\sigma_2}(e_2) : B$, *it is decidable whether* $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$ *or* $\mathsf{pred}_{\sigma_1}(e_1) \not\supset \mathsf{pred}_{\sigma_2}(e_2)$.

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 $\supset$. 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 $\mathsf{pred}_\iota(\lambda x{:}\mathsf{Int}.\ x \geq 0) \supset \mathsf{pred}_\iota(\lambda x{:}\mathsf{Int}.\ x + 1 > 0)$ and vice versa, even though the two predicates aren't equal. You can also view $\supset$ as a total order *up-to contextual equivalence*.

We have one more requirement than Greenberg did: monotonicity under substitution, which we call *substitutivity*. Substitutions weren't an issue in his non-dependent system, but we must require that if a join can happen without having a value for x, then the same join happens when we know x's value.

There is at least one workable implication relation: syntactic equality. We say $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$ iff $e_1 = e_2$ and $\sigma_1 = \sigma_2$. Since we've been careful to store only those values that are actually referenced in the closure of the predicate, the steps to determine these equalities are finite and computable at run time. For example, suppose we wish to show that $\mathsf{pred}_{[y \mapsto 47]}(\lambda x{:}\mathsf{Int}.\ x > y) \supset \mathsf{pred}_{[y \mapsto 47]}(\lambda x{:}\mathsf{Int}.\ x > y)$. The code part—the predicate $\lambda x{:}\mathsf{Int}.\ x > y$—is the same; an implementation might observe that the function pointers are equal. The environment has only one slot, for y, with the value 47 in it; an implementation might compare the two environments slot-by-slot. We *could* have given an operational semantics which behaves more like an implementation, explicitly generating conditionals and merge operations in the terms themselves, but we believe this slightly more abstract presentation is more digestible.

We extend Greenberg's notion of join to account for dependency with a new function, wrap. Greenberg, Pierce, and Weirich identified two variants of latent contracts in the literature: *picky*, where we monitor value substituted in the codomain with the domain contract; and *lax*, where the actual parameter value substituted into the codomain is unmonitored [9]. There is a third variant, *indy*, which applies a monitor to the argument value but uses a different blame label [2]. These different models of substitution all exhibit different behavior for *abusive* contracts, where the codomain contract violates the domain contract.

When we think about space-efficiency, we have to think about another source of substitutions in the codomain: multiple function proxies. How do the monitors unfold when we have two function proxies? In the classic lax semantics, we find:

$$\begin{array}{ll} & \mathsf{mon}(x{:}c_{11} \mapsto c_{12}, \mathsf{mon}(x{:}c_{21} \mapsto c_{22}, f))\ v \\ \longrightarrow_{\mathsf{C}} & \mathsf{mon}(c_{12}[v/x], \mathsf{mon}(x{:}c_{21} \mapsto c_{22}, f)\ \mathsf{mon}(c_{11}, v)) \\ \longrightarrow_{\mathsf{C}} & \mathsf{mon}(c_{12}[v/x], \mathsf{mon}(c_{22}[\mathsf{mon}(c_{11}, v)/x], f\ \mathsf{mon}(c_{21}, \mathsf{mon}(c_{11}, v)))) \end{array}$$

The second step cheats a little. If the domain is of base type, we'd check the arguments first. That technicality aside: even though we're using the lax semantics, we substitute contracts into the codomain. For the space-efficient semantics to be sound, it must behave *exactly* like the classic semantics. That means that no matter what joins happen, $\mathrm{CPCF}_{\mathsf{E}}$ must replicate the contract substitutions done in $\mathrm{CPCF}_{\mathsf{C}}$. We can construct an abusive contract in $\mathrm{CPCF}_{\mathsf{C}}$—even though it has lax semantics—by having the inner function proxy abuse the outer one. Why was blame raised? Because c_2's codomain contract *abused* c_1's domain contract. Even though $\mathrm{CPCF}_{\mathsf{C}}$ has a lax semantics, wrapping multiple function proxies leads to monitoring domains from one contract in the codomain of another—a situation ripe for abuse.

Space-efficiency means joining contracts, so how can we emulate this classic-semantics substitution behavior? We use the wrap function, forcing a substitution when two function contracts are joined. By keeping track of these substitutions

$$
\begin{aligned}
C_1 &= f{:}(\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x > 0) \mapsto \mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x > 0)) \mapsto \mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x > 0)\\
C_2 &= f{:}(\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ \mathsf{true}) \mapsto \mathsf{pred}(\lambda x{:}\mathsf{Int}.\ \mathsf{true})) \mapsto \mathsf{pred}(\lambda x{:}\mathsf{Int}.\ f\ 0 = 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{mon}^{l_1}(C_1, \mathsf{mon}^{l_2}(C_2, \lambda f{:}(\mathsf{Int}\to\mathsf{Int}).\ f\ 5))\ (\lambda x{:}\mathsf{Int}.\ x)\\
\longrightarrow_{\mathsf{C}}\ &\mathsf{mon}^{l_1}(C_{12}[(\lambda x{:}\mathsf{Int}.\ x)/f],\\
&\quad \mathsf{mon}^{l_2}(C_2, \lambda f{:}(\mathsf{Int}\to\mathsf{Int}).\ f\ 5)\ \mathsf{mon}^{l_1}(C_{11}, (\lambda x{:}\mathsf{Int}.\ x)))\\
\longrightarrow_{\mathsf{C}}^*\ &\mathsf{mon}^{l_1}(C_{12}[(\lambda x{:}\mathsf{Int}.\ x)/f], \mathsf{mon}^{l_2}(C_{22}[\mathsf{mon}^{l_1}(C_{11}, \lambda x{:}\mathsf{Int}.\ x)/f], 5))\\
\longrightarrow_{\mathsf{C}}\ &\mathsf{mon}^{l_1}(C_{12}[(\lambda x{:}\mathsf{Int}.\ x)/f],\\
&\quad \mathsf{if}\ ((\lambda x{:}\mathsf{Int}.\ \mathsf{mon}^{l_1}(C_{11}, \lambda x{:}\mathsf{Int}.\ x)\ 0 = 0)\ 5)\ 5\ \ \mathsf{err}^{l_2})\\
\longrightarrow_{\mathsf{C}}\ &\mathsf{mon}^{l_1}(C_{12}[(\lambda x{:}\mathsf{Int}.\ x)/f], \mathsf{if}\ (\mathsf{mon}^{l_1}(C_{11}, \lambda x{:}\mathsf{Int}.\ x)\ 0 = 0)\ 5\ \ \mathsf{err}^{l_2})\\
\longrightarrow_{\mathsf{C}}^*\ &\mathsf{err}^{l_2}
\end{aligned}
$$

Fig. 6. Abusive function proxies in CPCF$_{\mathsf{C}}$

at every join, any joins that happen in the future will be working on contracts which already have appropriate substitutions.

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 ($x{:}c_1 \mapsto c_2$) 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. Note that substitution for labeled predicate contracts is explicit and delayed, as for ordinary contracts:

$$
\begin{aligned}
\mathsf{pred}^l_\sigma(e)[v/x] &= \begin{cases} \mathsf{pred}^l_{\sigma[x\mapsto v]}(e) & x \in \mathrm{fv}(\sigma(e))\\ \mathsf{pred}^l_\sigma(e) & \text{otherwise}\end{cases}\\
\mathsf{nil}[v/x] &= \mathsf{nil}\\
(\mathsf{pred}^l_\sigma(e); r)[v/x] &= \mathsf{pred}^l_\sigma(e)[v/x]; r[v/x]
\end{aligned}
$$

We *do not* do any joining when a substitution occurs—substitution works exactly as usual for predicate stacks. In CPCF$_{\mathsf{E}}$, closing substitutions map each variable to (a) itself ($[x \mapsto x]$), (b) a monitor on itself ($[x \mapsto \mathsf{mon}(c, x)]$), or (c) a value.

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). We also change the evaluation rule for unwrapping function proxies: we no longer explicitly substitute a check into the codomain.

Space-efficiency comes by restricting congruence to only apply when there are abutting monitors (cf. EMonC here in CPCF$_{\mathsf{E}}$ to EMon in CPCF$_{\mathsf{C}}$). 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$_{\mathsf{E}}$ 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$_{\mathsf{C}}$ and CPCF$_{\mathsf{E}}$ 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$_{\mathsf{C}}$ and CPCF$_{\mathsf{E}}$ or (b) reduces to an equivalent term in both CPCF$_{\mathsf{C}}$ and CPCF$_{\mathsf{E}}$.

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.

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, we call it *monitor congruence.*

Lemma 1 (Monitor congruence (single step)). *If* $\emptyset \vdash e_1 : T$ *and* $\emptyset \vdash c : T$ *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 joining 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$_{\mathsf{C}}$ and CPCF$_{\mathsf{E}}$. The proof is by logical relations (omitted for space), which gives us contextual equivalence—the strongest equivalence we could ask for.

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

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

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).

Predicate extraction $\boxed{\mathsf{preds}(e), \mathsf{preds}(C), \mathsf{preds}(c) : \mathcal{P}(e \times (\mathit{Var} \rightharpoonup e))}$

$$\begin{aligned}
\mathsf{preds}(x) &= \emptyset \\
\mathsf{preds}(k) &= \emptyset \\
\mathsf{preds}(\lambda x{:}T.\ e) &= \mathsf{preds}(e) \\
\mathsf{preds}(\mathsf{mon}^l(C, e)) &= \mathsf{preds}(C) \cup \mathsf{preds}(e) \\
\mathsf{preds}(\mathsf{mon}(c, e)) &= \mathsf{preds}(c) \cup \mathsf{preds}(e) \\
\mathsf{preds}(e_1\ e_2) &= \mathsf{preds}(e_1) \cup \mathsf{preds}(e_2) \\
\mathsf{preds}(e_1\ \mathit{op}\ e_2) &= \mathsf{preds}(e_1) \cup \mathsf{preds}(e_2) \\
\mathsf{preds}(\mathsf{if}\ e_1\ e_2\ \ e_3) &= \\
&\mathsf{preds}(e_1) \cup \mathsf{preds}(e_2) \cup \mathsf{preds}(e_3) \\
\mathsf{preds}(\mathsf{err}^l) &= \emptyset
\end{aligned}
\qquad
\begin{aligned}
\mathsf{preds}(\mathsf{pred}_\sigma(e)) &= \{(e, \sigma)\} \cup \\
&\textstyle\bigcup_{[x \mapsto v] \in \sigma} \mathsf{preds}(v) \\
\mathsf{preds}(x{:}C_1 \mapsto C_2) &= \mathsf{preds}(C_1) \cup \mathsf{preds}(C_2) \\
\\
\mathsf{preds}(\mathsf{nil}) &= \emptyset \\
\mathsf{preds}(\mathsf{pred}^l_\sigma(e); r) &= \{(e, \sigma)\} \cup \\
&\textstyle\bigcup_{[x \mapsto e'] \in \sigma} \mathsf{preds}(e') \cup \\
&\mathsf{preds}(r) \\
\mathsf{preds}(x{:}c_1 \mapsto c_2) &= \mathsf{preds}(c_1) \cup \mathsf{preds}(c_2)
\end{aligned}$$

Contract size $\boxed{\mathsf{size}(C) : \mathbb{N}}$

$$\begin{aligned}
\mathsf{size}(\mathsf{pred}_\sigma(e)) &= 1 \\
\mathsf{size}(x{:}C_1 \mapsto C_2) &= \mathsf{size}(C_1) + \mathsf{size}(C_2)
\end{aligned}$$

Fig. 7. Predicate extraction and contract size

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

Proof. By mutual induction on the typing relations.

4 Bounds for space efficiency

We have seen that CPCF$_\mathsf{E}$ behaves the same as CPCF$_\mathsf{C}$ (Theorem 1), but is CPCF$_\mathsf{E}$ 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", and then showing that evaluation doesn't introduce predicates (Lemma 6). We let $\mathsf{preds}(e)$ be the set of predicates in a term, where a predicate is represented as a term and a closing substitution.

4.1 The simple case

Greenberg showed that for *simple* contracts—without dependency—we can put a bounds on space [8]. We can recover his result in our more general framework.

We say program e *uses simple contracts* when all predicates in e are closed every predicate stack has no redundancies. The following theorems only hold for programs that use simple contracts.

Lemma 4. $\mathsf{preds}(e[e'/x]) \subseteq \mathsf{preds}(e) \cup \mathsf{preds}(e')$

Proof. By induction on e, using the absorptive property of set union.

The critical case is when e is a predicate contract. Ordinarily, substitution here would store $[x \mapsto e']$ in σ. Since e uses simple contracts, the predicate has no free variables, and the substitution doesn't hold on to anything.

Lemma 5. *If $\emptyset \vdash c_1 : T$ and $\emptyset \vdash c_2 : T$ then* $\mathsf{preds}(\mathsf{join}(c_1, c_2)) \subseteq \mathsf{preds}(c_1) \cup \mathsf{preds}(c_2)$.

Proof. By induction on c_1, ignoring wrap*'s substitution by Lemma 4.*

With these proofs in hand, we can prove that reduction is non-increasing: as a program reduces, no new contracts appear (and contracts may disappear).

Lemma 6 (Reduction is non-increasing in simple 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 $P_B = |\{e \in \mathsf{preds}(e) \mid \Gamma \vdash \mathsf{pred}_\sigma(e) : B\}|$ 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, $s = \max\{\mathsf{size}(C) \mid C \in e \wedge \Gamma \vdash C : T_1 \rightarrow T_2\}$. 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 ($2^{\mathsf{height}(T)}$).

4.2 The dependent case

In the dependent case, we can't bound the number of contracts by the size of contracts used in the program. Consider the following term, where $n \in \mathbb{N}$:

```
let downTo = μ(f:Int→Int).
  mon^l(x:pred(λx:Int. x ≥ 0) ↦ pred(λy:Int. x ≥ y),
    λx:Int. if (x = 0) 0  (f (x − 1))) in
downTo n
```

How many different contracts will appear in a run of this program? As downTo runs, we'll n different forms of the predicate $\mathsf{pred}^l_{\sigma_i}(\lambda y{:}\mathsf{Int}.\ x \ge y)$. We'll have one, $\sigma_n = [x \mapsto n]$ on the first call, another $\sigma_{n-1} = [x \mapsto n - 1]$ on the second call, all the way down to $\sigma_0 = [x \mapsto 0]$ on the nth call. But the size of n has no effect on the size of the contracts in the program. The number of distinct contracts that appear will be unbounded.

In the simple case, we get bounds automatically, using the smallest possible implication relation—the identity relation, syntactic equality. In the dependent case, it's up to the programmer to identify which implications are

We can recover space efficiency for downTo by saying $\mathsf{pred}_{\sigma_1}(\lambda y{:}\mathsf{Int}.\ x \ge y) \supset \mathsf{pred}_{\sigma_2}(\lambda y{:}\mathsf{Int}.\ x \ge y)$ iff $\sigma_1(x) \subset \sigma_2(x)$. Then we the codomain checks from recursive calls will be able to join:

$$\begin{aligned}
\mathsf{downTo}\ n &\longrightarrow^*_{\mathsf{E}} \mathsf{mon}^l(\mathsf{pred}_{[x\mapsto n]}(\dots), \dots)\\
&\longrightarrow^*_{\mathsf{E}} \mathsf{mon}^l(\mathsf{pred}_{[x\mapsto n]}(\dots), \mathsf{mon}^l(\mathsf{pred}_{[x\mapsto n-1]}(\dots), \dots))\\
&\longrightarrow^*_{\mathsf{E}} \mathsf{mon}^l(\mathsf{pred}_{[x\mapsto n-1]}(\dots), \dots)
\end{aligned}$$

Why are we able to recover space efficiency in this case? Because we can come up with an easily decidable implication rule for our specific predicates that matches how our function checks narrower and narrower properties as it recurses.

Recall the mutually recursive $\mathsf{even}/\mathsf{odd}$ example (Figure 1). We can make this example space-efficient by adding the implication that:

$$\mathsf{pred}_{\sigma_1}(\lambda b{:}\mathsf{Bool}.\ b \mathrel{\mathsf{or}} (x \mathbin{\mathsf{mod}} 2 = 0)) \supset \mathsf{pred}_{\sigma_2}(\lambda b{:}\mathsf{Bool}.\ b \mathrel{\mathsf{or}} (x \mathbin{\mathsf{mod}} 2 = 0))$$

iff $\sigma_1(x) + 2 = \sigma_2(x)$. Suppose we put contracts on both even and odd:

$$\begin{aligned}
&\mathsf{let\ odd} = \mathsf{mon}^{l_{\mathsf{odd}}}(x{:}\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \ge 0) \mapsto \mathsf{pred}(\lambda b{:}\mathsf{Bool}.\ b \mathrel{\mathsf{or}} (x \mathbin{\mathsf{mod}} 2 = 0)),\\
&\qquad\qquad\qquad\qquad \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ \mathsf{false}\ \ (\mathsf{even}\ (x - 1)))\\
&\mathsf{and\ even} =\\
&\quad \mathsf{mon}^{l_{\mathsf{even}}}(x{:}\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \ge 0) \mapsto \mathsf{pred}(\lambda b{:}\mathsf{Bool}.\ b \mathrel{\mathsf{or}} ((x + 1) \mathbin{\mathsf{mod}} 2 = 0)),\\
&\qquad\qquad\qquad\qquad \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ \mathsf{true}\ \ (\mathsf{odd}\ (x - 1)))
\end{aligned}$$

Now our trace of contracts won't be homogeneous; we'll find something like the following (which elides domain contracts):

$$\begin{aligned}
\mathsf{odd}\ 5 &\longrightarrow^*_{\mathsf{C}} \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{even}\ 4)\\
&\longrightarrow^*_{\mathsf{C}} \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{even}}}(\mathsf{pred}_{[x\mapsto 4]}(\dots), \mathsf{odd}\ 3))\\
&\longrightarrow^*_{\mathsf{C}} \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{even}}}(\mathsf{pred}_{[x\mapsto 4]}(\dots),\\
&\qquad \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \mathsf{even}\ 2)))\\
&\longrightarrow^*_{\mathsf{C}} \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{even}}}(\mathsf{pred}_{[x\mapsto 4]}(\dots),\\
&\qquad \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \mathsf{mon}^{l_{\mathsf{even}}}(\mathsf{pred}_{[x\mapsto 2]}(\dots), \mathsf{odd}\ 1))))\\
&\longrightarrow^*_{\mathsf{C}} \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 5]}(\dots), \mathsf{mon}^{l_{\mathsf{even}}}(\mathsf{pred}_{[x\mapsto 4]}(\dots),\\
&\qquad \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 3]}(\dots), \mathsf{mon}^{l_{\mathsf{even}}}(\mathsf{pred}_{[x\mapsto 2]}(\dots),\\
&\qquad\ \mathsf{mon}^{l_{\mathsf{odd}}}(\mathsf{pred}_{[x\mapsto 1]}(\dots), \mathsf{even}\ 0)))))
\end{aligned}$$

To make these checks space efficient, we'd need several implications; we write $\mathsf{odd_p}$ for $\lambda b{:}\mathsf{Bool}.\ b \mathrel{\mathsf{or}} (x \mathbin{\mathsf{mod}} 2 = 0)$ and $\mathsf{even_p}$ for $\lambda b{:}\mathsf{Bool}.\ b \mathrel{\mathsf{or}} ((x + 1) \mathbin{\mathsf{mod}} 2 =$

0). The following table gives conditions on the implication relation for the row predicate to imply the column predicate:

$\supset$	$\mathsf{pred}_{\sigma_2}(\mathsf{odd_p})$	$\mathsf{pred}_{\sigma_2}(\mathsf{even_p})$
$\mathsf{pred}_{\sigma_1}(\mathsf{odd_p})$	$\sigma_1(x) + 2 = \sigma_2(x)$	$\sigma_1(x) + 1 = \sigma_2(x)$
$\mathsf{pred}_{\sigma_1}(\mathsf{even_p})$	$\sigma_1(x) + 1 = \sigma_2(x)$	$\sigma_1(x) + 2 = \sigma_2(x)$

Having all four of these implications allows us to eliminate any pair of checks generated by the recursive calls in odd and even, reducing the codomain checking to constant space (just one check).

We could define a different implication relation, where, say, $\mathsf{pred}_{\sigma_1}(\mathsf{odd_p}) \supset \mathsf{pred}_{\sigma_2}(\mathsf{odd_p})$ iff $\sigma_1(x) \bmod 2 = \sigma_2(x) \bmod 2$. Such an implication would apply more generally than those in the table above.

As usual, there is a trade-off between time and space. It's possible to write contracts where the conditions on the implication relation amount to re-checking the implication relation. Consider the following tail-recursive factorial function:

$$
\begin{array}{l}
\mathsf{let\ any} = \lambda z{:}\mathsf{Int}.\ \mathsf{true} \\
\mathsf{let\ fact} = \mu(f{:}\mathsf{Int}{\to}\mathsf{Int}{\to}\mathsf{Int}). \\
\qquad\qquad \mathsf{mon}^l(x{:}\mathsf{pred}(\mathsf{any}) \mapsto acc{:}\mathsf{pred}(\mathsf{any}) \mapsto \mathsf{pred}(\lambda y{:}\mathsf{Int}.\ x \geq 0), \\
\qquad\qquad\ \ \lambda x{:}\mathsf{Int}.\ \lambda acc{:}\mathsf{Int}. \\
\qquad\qquad\quad \mathsf{if}\ (x = 0)\ acc\ \ (f\ (x - 1)\ (x * acc)))
\end{array}
$$

If you fact with a negative number, you indeed won't get a value back out. The contract isn't *wrong*, per se, but a call of fact 3 yields monitors that check, from outside to inside, that $3 \geq 0$ and $2 \geq 0$ and $1 \geq 0$ and $0 \geq 0$. When should we say that $\mathsf{pred}_{\sigma_1}(\lambda y{:}\mathsf{Int}.\ x \geq 0) \supset \mathsf{pred}_{\sigma_1}(\lambda y{:}\mathsf{Int}.\ x \geq 0)$? We could check that $\sigma_1(x) \geq \sigma_2(x)$... but the time cost is just like checking the original contract.

5 Where should the implication relation come from?

The simplest source of an implication relation is to have the contract system read pragmas or other directives from the programmer; the implication relation can be derived as the reflexive transitive closure of a programmer's rules. We can imagine something like the following, where programmers can specify how several different predicates interrelate:

```
contract y:Int{x1 >= y} implies y:Int{x2 >= y} when
  x1 <= x2

contract y:Int{x1 >  y} implies y:Int{x2 >= y} when
  x1 <= x2 + 1

contract y:Int{x1 >  y} implies y:Int{x2 >  y} when
  x1 <= x2
```

We can imagine a default collection of such implications that come with the language; library programmers should be able to write their own, as well.

It is probably unwise to allow programmers to write arbitrary implications: what if they're wrong? A good implementation would only accept verified implications, using a theorem prover or an SMT solver to avoid bogus implications.

Rather than having programmers write their own implications, we could try to *automatically* derive the implications. Given a program, a fixed number of predicates occur, even if an unbounded number of predicate/closing substitution pairings might occur at runtime. Collect all possible predicates from the source program, and consider each pair of predicates over the same base type, $\mathsf{pred}(e_1)$ and $\mathsf{pred}(e_2)$ such that $\Gamma \vdash e_i : B \to \mathsf{Bool}$. We can derive from the typing derivation the shapes of the respective closing substitutions, σ_1 and σ_2. What are the conditions on σ_1 and σ_2 such that $\mathsf{pred}_{\sigma_1}(e_1) \supset \mathsf{pred}_{\sigma_2}(e_2)$? We are looking for a property $P(\sigma_1, \sigma_2)$ such that:

$$\forall k \in \mathcal{K}_B,\ P(\sigma_1, \sigma_2) \wedge \sigma_1(e_1)\ k \longrightarrow^*_{\mathsf{E}} \mathsf{true} \Rightarrow \sigma_2(e_2)\ k \longrightarrow^*_{\mathsf{E}} \mathsf{true}$$

Ideally, P is also efficiently decidable—at least more efficiently than deciding both predicates. The problem of finding P can be reduced to that of finding the weakest precondition for the safety of the following program:

```
fun x:B  =>
  let y0 = v_10 (* representation of σ1 *)
      ...
      yn = v_1n
      z0 = v_20 (* representation of σ2 *)
      ...
      zn = v_2m in
  if e1 x then if e2 x then () else error else ()
```

Since P would be the *weakest* precondition, we would know that we had found the most general condition for the implication relation. Whether or not the most general condition is the *best* condition depends on context. We should also consider a cost model for P; programmers may want to occasionally trade space for time, not bothering to join predicates that would be expensive to test.

Finding implication conditions resembles liquid type inference [16,22,12]: programmers get a small control knob (which expressions can go in P) and then an SMT solver does the rest. The settings are different, though: liquid types are about verifying programs, while we're executing checks at runtime.

5.1 Implementation

Implementation issues abound. How should the free variables in terms be represented? What kind of refactorings and optimizations can the compiler do, and how might they interfere with the set of contracts that appear in a program? When is the right moment in compilation to fix the implication relation? More generally, what's the design space of closure representations and calling conventions for languages with contracts?

6 Extensions

Generalizing our space-efficient semantics to sums and products does not seem particularly hard: we'd need contracts with corresponding shapes, and the join operation would push through such shapes. Recursive types are more interesting. Findler et al.'s lazy contract checking keeps contracts from changing the asymptotic time complexity of the program [5]; we may be able to adapt their work to avoid changes in asymptotic space complexity, as well.

The predicates here range over base types, but we could also allow predicates over other types. If we allow predicates over higher types, how should the adequacy constraint on predicate implication (Definition 1) change?

Impredicative polymorphism in the style of System F would require even more technical changes. The introduction of type variables would make our reasoning about names and binders trickier. In order to support predicates over type variables, we'd need to allow predicates over higher types—and so our notion of adequacy of $\supset$ would change. In order to support predicates over quantified types, we'd need to change adequacy again. Adequacy would end up looking like the logical relation used to show relational parametricity: when would we have $\forall\alpha.\,T_1 \supset \forall\alpha.\,T_2$? If we substitute T_1' for α on the left and T_2' for α on the right (and T_1' and T_2' are somehow related), then $T_1[T_1'/\alpha] \supset T_2[T_2'/\alpha]$. Not only would the technicalities be tricky, implementations would need to be careful to manage closure representations correctly (e.g., what happens if polymorphic code differs for boxed and unboxed types?).

We don't treat blame as an interesting algebraic structure—it's enough for our proofs to show that we always produce the same answer. Changing our calculus to have a more interesting notion of blame, like *indy* semantics [2] or involutive bame labels [24,23], would be a matter of pushing a shallow change in the semantics through the proofs.

Finally, it would make sense to define substitution as follows:

$$\begin{aligned}
\mathsf{pred}^l_\sigma(e)[v/x] &= \begin{cases} \mathsf{pred}^l_{\sigma[x\mapsto v]}(e) & x \in \mathrm{fv}(\sigma(e)) \\ \mathsf{pred}^l_\sigma(e) & \text{otherwise} \end{cases} \\
\mathsf{nil}[v/x] &= \mathsf{nil} \\
(\mathsf{pred}^l_\sigma(e); r)[v/x] &= \mathsf{join}(\mathsf{pred}^l_\sigma(e)[v/x]; \mathsf{nil}, r[v/x])
\end{aligned}$$

Every time we substitute a value into a predicate stack, we check for any new redundancies that have been revealed. We haven't gone through the proofs to show that this change would be sound. Not only would the correctness proofs change (Lemma 2 in particular), so would the type soundness proof: we would need to simultaneously prove that substitution and joins preserve types.

7 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 [21]. Our exact behavioral equivalence means we could use results from Tobin-Hochstadt et al.'s static analysis in terms of CPCF$_{\mathsf{C}}$ to optiimze space efficient programs in CPCF$_{\mathsf{E}}$. 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.

Sekiyama et al. [17] also use delayed substitutions in their polymorphic manifest contract calculus, but for different technical reasons.

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. Developing the approach to dependency used here was much easier in a latent calculus, though several bugs along the way would have been caught early by a stronger type system. Complexity trade-offs of this sort are an old story.

7.1 Racket's implementation

If contracts leak space, how is it that they are used so effectively throughout PLT Racket? Racket is designed to avoid space leaks. In Racket, contracts tend to go on module boundaries. Calls inside of a module then don't trigger contract checks—including recursive calls, like in the $\mathsf{even}/\mathsf{odd}$ example.

Racket *will* monitor recursive calls across module boundaries, and these checks can indeed lead to space leaks. Phrased in terms of our system, Racket implements a contract check on a recursive function as follows:

$$\begin{aligned}\mathsf{downTo} = \mathsf{mon}^{l}(&x{:}\mathsf{pred}(\lambda x{:}\mathsf{Int}.\ x \ge 0) \mapsto \mathsf{pred}(\lambda y{:}\mathsf{Int}.\ x \ge y),\\ &\mu(f{:}\mathsf{Int}{\to}\mathsf{Int}).\ \lambda x{:}\mathsf{Int}.\ \mathsf{if}\ (x = 0)\ 0\ \ (f\ (x - 1)))\end{aligned}$$

Note that calling $\mathsf{downTo}\ n$ will merely check that the final result is less than n—none of the intermediate values. Our version of downTo above puts the contract *inside* the recursive knot, forcing checks every time.

Racket's `tail-marks-match?` predicate[3], which offers a less thorough form of space efficiency. We hope that the presentation of CPCF$_{\mathsf{E}}$ is close enough to Racket's internal model to provide insight into how to achieve space efficiency for at least some contracts in Racket. In particular, Racket will avoid redundant checks on the following program:

```
(define (count-em-integer? x)
  (printf "checking␣~s\n" x)
  (integer? x))

(letrec
  ([f (contract (-> any/c count-em-integer?)
```

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

```
        (lambda (x)
          (printf "x:␣~s\n" x)
            (if (zero? x) x (f (- x 1))))
        'pos
        'neg)])
  (f 3))
```

But wrapping the underlying function with the same contract twice leads to a space leak.The contract mechanisms used here are the low-level ones, to highlight the compromise Racket makes for space efficiency: their contract compilation macros “strike a balance between common ways that tail recursion is broken and checking that would not be too expensive in the case that there wouldn’t have been any pileup of redundant wrappers”.[4]

Contracts are first-class in Racket: monitors take two expressions, one for the contract and one to be monitored. Computing new contracts at run time breaks our framing of space-efficiency: if we can’t predetermine which contracts arise at runtime, we can’t fix an implication relation in advance.

8 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; and, extended the original result, both in terms of features covered (dependency and 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. Robby Findler provided the Racket example and helped correct and clarify a draft; Sam Tobin-Hochstadt also offered corrections and suggestions.

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. Springer Berlin Heidelberg (2012)
4. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: International Conference on Functional Programming (ICFP) (2002)

[4] Robby Findler, personal correspondence, 2016-05-19.

5. Findler, R.B., Guo, S.Y., Rogers, A.: Lazy contract checking for immutable data structures. In: Implementation and Application of Functional Languages, pp. 111–128. Springer-Verlag (2008)
6. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010), `http://racket-lang.org/tr1/`
7. Garcia, R.: Calculating threesomes, with blame. In: International Conference on Functional Programming (ICFP) (2013)
8. Greenberg, M.: Space-efficient manifest contracts. In: Principles of Programming Languages (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. Jhala, R.: Refinement types for haskell. In: Programming Languages Meets Program Verification (PLPV). pp. 27–27. ACM (2014)
13. Meyer, B.: Eiffel: the language. Prentice-Hall, Inc. (1992)
14. Plotkin, G.: Lcf considered as a programming language. Theoretical Computer Science 5(3), 223 – 255 (1977)
15. Racket contract system (2013), `http://pre.plt-scheme.org/docs/html/guide/contracts.html`
16. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Programming Language Design and Implementation (PLDI) (2008)
17. Sekiyama, T., Igarashi, A., Greenberg, M.: Polymorphic manifest contracts, revised and resolved (2016), in submission
18. Siek, J., Thiemann, P., Wadler, P.: Blame, coercion, and threesomes: Together again for the first time. In: Programming Language Design and Implementation (PLDI) (2015)
19. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
20. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Principles of Programming Languages (POPL). pp. 365–376 (2010)
21. Tobin-Hochstadt, S., Van Horn, D.: Higher-order symbolic execution via contracts. In: OOPSLA. pp. 537–554. OOPSLA '12, ACM (2012)
22. Vazou, N., Rondon, P.M., Jhala, R.: European Symposium on Programming (ESOP), chap. Abstract Refinement Types, pp. 209–228. Springer Berlin Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-37036-6_13`
23. Wadler, P.: A Complement to Blame. In: SNAPL. LIPIcs, vol. 32 (2015)
24. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: European Symposium on Programming (ESOP) (2009)