A Sound (and Complete) Model of Contracts

Matthias Blume
Toyota Technological Institute at Chicago
blume@tti-c.org

David McAllester
Toyota Technological Institute at Chicago
mcallester@tti-c.org

Abstract

Even in statically typed languages it is useful to have certain invariants checked dynamically. Findler and Felleisen gave an algorithm for dynamically checking expressive higher-order types called contracts. If we postulate soundness (in the sense that whenever a term is accused of violating its contract it really does fail to satisfy it), then their algorithm implies a semantics for contracts. Unfortunately, the implicit nature of the resulting model makes it rather unwieldy.

In this paper we demonstrate that a direct approach yields essentially the same semantics without having to refer to contract-checking in its definition. The so-defined model largely coincides with intuition, but it does expose some peculiarities in its interpretation of *predicate contracts* where a notion of *safety* (which we define in the paper) "leaks" into the semantics of Findler and Felleisen's original *unrestricted* predicate contracts.

This counter-intuitive aspect of the semantics can be avoided by changing the language, replacing unrestricted predicate contracts with a restricted version. The corresponding loss in expressive power can be recovered by also providing a way of explicitly expressing safety as a contract—either in ad-hoc fashion or, e.g., by including general recursive contracts.

Categories and Subject Descriptors: D.3.1 Programming Languages: Formal Definitions and Theory — *semantics*

General Terms: Languages, Theory, Verification

Keywords: contracts, predicates, safety

1 Introduction

Static types can serve as a powerful tool for expressing program invariants that a compiler can verify. Yet, many invariants a compiler cannot enforce. It is therefore useful to allow for dynamic checks of runtime properties of programs, regardless of whether the language is statically typed or not. Many languages have mechanisms for reporting abnormal situations that arise at runtime. For example, in the ML programming language [13, 12] one typically raises an exception when an intended program invariant is violated. While these mechanisms enable people to program defensively in an ad-hoc manner, they are an inappropriate basis for designing, implementing, and composing program components.

Findler and Felleisen introduced the notion of *contracts* [9] as a more systematic way of expressing and monitoring runtime invariants. Contracts are a form of types too expressive for static verification, but an implementation such as the DrScheme system [8, 5] can provide a meaningful way of checking contracts dynamically. The contract checker automatically raises exceptions called *contract exceptions*.

Once an exception indicates the violation of an intended invariant one would like to identify the part of the program (the *module*) that is actually in error. Thus, a raised contract exception should blame a specific contract declaration. To be somewhat more concrete, consider a program of the form

$$\textbf{let } x_1 : t_1 = e_1 \textbf{ in } \ldots \textbf{let } x_n : t_n = e_n \textbf{ in } x_n$$

where each t_i is a closed contract expression acting as the *interface* of *module* e_i. Findler and Felleisen give an algorithm for assigning blame to one of the e_i in the case when a contract exception is raised. Intuitively, this means that e_i does not satisfy contract t_i, but the concept of contract satisfaction had not actually been defined formally. Still, we can view the algorithm as *implying* a semantics of contracts. In particular, we can say e satisfies t unless there is some program for which the algorithm claims otherwise. More formally, let $[\![t]\!]_{\mathrm{FF}}$ be the set of values that the Findler-Felleisen contract checking algorithm cannot blame for violating t.

It is important to show that the Findler-Felleisen algorithm is correct. This means that when the algorithm blames a contract declaration, that contract declaration is actually wrong. $[\![\cdot]\!]_{\mathrm{FF}}$ as informally defined above makes correctness vacuously true, because a declaration is "wrong" by definition if the algorithm blames it. A more meaningful notion of correctness must be based on an independent definition of the meaning of contracts, preferably defined in a mostly compositional manner, for example the one we give in this paper.

A compositional semantics is desirable because the structure of $[\![\cdot]\!]_{\mathrm{FF}}$ is not at all obvious. With just the informal definition given above, an answer to the question "Does e satisfy t?" is difficult because it requires consideration of *all* possible contexts for e. Indeed, some aspects of $[\![\cdot]\!]_{\mathrm{FF}}$ do turn out to be counterintuitive. An important part of the Findler-Felleisen contract language are *predicate contracts*, here written $\langle\phi\rangle$, which are checked by applying their predicate ϕ to the value in question. An exception is raised if the result is not true. In particular, checking $\langle\lambda x.\underline{1}\rangle$, where the predicate is always true, is a no-op. One might expect *every* value to satisfy $\langle\lambda x.\underline{1}\rangle$ and, consequently, the identity function $\lambda x.x$ to

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

$$x,y,\ldots \in \mathrm{Var} \quad ::= \quad \mathtt{x} \mid \mathtt{y} \mid \ldots \qquad \underline{i},\underline{j},\ldots \in I \quad ::= \quad \underline{0} \mid \underline{1} \mid \ldots \qquad F \quad ::= \quad + \mid - \mid < \mid \leq \mid \ldots$$

elements common to both external and internal languages

$$\begin{array}{rcll}
t^{\mathrm{e}} \in T^{\mathrm{e}} & ::= & \underline{\mathrm{int}} \mid \underline{\mathrm{safe}} \mid T^{\mathrm{e}} \xrightarrow{\mathrm{Var}} T^{\mathrm{e}} \mid \langle T^{\mathrm{e}}; \lambda \mathrm{Var}.P^{\mathrm{e}} \rangle & \\
\phi^{\mathrm{e}} \in P^{\mathrm{e}} & ::= & P_0{}^{\mathrm{e}} \mid (P^{\mathrm{e}}\ \mathrm{Var}) & \\
P_0{}^{\mathrm{e}} & \subset & E^{\mathrm{e}} & \textit{closed predicate terms} \\
 & & & \\
e^{\mathrm{e}} \in E^{\mathrm{e}} & ::= & \mathrm{Var} \mid I \mid \lambda \mathrm{Var}.E^{\mathrm{e}} \mid (E^{\mathrm{e}}\ E^{\mathrm{e}}) \mid F(E^{\mathrm{e}},\ldots,E^{\mathrm{e}}) & \\
 & & & \\
p \in P & ::= & \mathrm{Var} \mid \mathbf{let}\ \mathrm{Var} : T^{\mathrm{e}} = E^{\mathrm{e}}\ \mathbf{in}\ P &
\end{array}$$

Figure 1. External language

$$\begin{array}{rcll}
t \in T & ::= & \underline{\mathrm{int}} \mid \underline{\mathrm{safe}} \mid T \xrightarrow{\mathrm{Var}} T \mid \langle T; \lambda \mathrm{Var}.P \rangle & \\
\phi \in P & ::= & P_0 \mid (P\ \mathrm{Var})_{\perp} & \\
P_0 & \subset & E & \textit{closed predicate terms} \\
\xi \in X & ::= & \perp \mid \top^0 \mid \top^1 \mid \top^2 \mid \ldots & \\
e \in E & ::= & \mathrm{Var} \mid I \mid \lambda \mathrm{Var}.E \mid (E\ E)_X \mid F_X(E,\ldots,E) \mid & \\
 & & (W_T^{X,X}\ E) \mid E ?_X E &
\end{array}$$

Figure 2. Internal language

$$\begin{array}{rcl}
C^e_\xi(\underline{i};\Gamma) & = & \underline{i} \\
C^e_\xi(x;\Gamma) & = & x \qquad ;x \notin \mathrm{domain}(\Gamma) \\
C^e_\xi(x;\Gamma) & = & (W_t^{\xi,\xi'}\ x) \qquad ;\Gamma(x) = (\xi',t) \\
C^e_\xi((e_1^{\mathrm{e}}\ e_2^{\mathrm{e}});\Gamma) & = & (C^e_\xi(e_1^{\mathrm{e}};\Gamma)\ C^e_\xi(e_2^{\mathrm{e}};\Gamma))_\xi \\
C^e_\xi(\lambda x.e^{\mathrm{e}}) & = & \lambda x.C^e_\xi(e^{\mathrm{e}};\Gamma\mid_{\neq x}) \\
C^e_\xi(f(e_1^{\mathrm{e}},\ldots,e_k^{\mathrm{e}});\Gamma) & = & f_\xi(C^e_\xi(e_1^{\mathrm{e}};\Gamma),\ldots,C^e_\xi(e_k^{\mathrm{e}};\Gamma))
\end{array}$$

$$\begin{array}{rcl}
C^t(\underline{\mathrm{int}}) & = & \underline{\mathrm{int}} \\
C^t(\underline{\mathrm{safe}}) & = & \underline{\mathrm{safe}} \\
C^t(t_1^{\mathrm{e}} \xrightarrow{x} t_2^{\mathrm{e}}) & = & C^t(t_1^{\mathrm{e}}) \xrightarrow{x} C^t(t_2^{\mathrm{e}}) \\
C^t(\langle t^{\mathrm{e}}; \lambda x.e^{\mathrm{e}} \rangle) & = & \langle C^t(t^{\mathrm{e}}); \lambda x.C^e_\perp(e^{\mathrm{e}};\emptyset) \rangle \\
 & & \\
C^p(x;\Gamma) & = & (W_t^{\perp,\xi}\ x) \quad \Gamma(x) = (\xi,t) \\
C^p(\mathbf{let}\ x : t^{\mathrm{e}} = e_1^{\mathrm{e}}\ \mathbf{in}\ e_2^{\mathrm{e}};\Gamma) & = & ((\lambda x.e_2)\ e_1)_\perp \\
\multicolumn{3}{c}{\textit{where } e_1 = C^e_{\top^i}(e_1^{\mathrm{e}};\Gamma);\quad e_2 = C^p(e_2^{\mathrm{e}};\Gamma, x \mapsto (\top^i, C^t(t^{\mathrm{e}})));} \\
\multicolumn{3}{c}{i \textit{ uniquely identifies the module named } x}
\end{array}$$

Figure 3. Translation from external to internal language.

satisfy $(\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \lambda x.\underline{1} \rangle$. (Every legal argument would be a legal result.) But DrScheme disagrees. When evaluating the following example (translated to Scheme) in DrScheme, the identity f is blamed for violating $(\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \lambda x.\underline{1} \rangle$:

$$\begin{array}{c}
\mathbf{let}\ f : (\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \lambda x.\underline{1} \rangle = \lambda y.y\ \mathbf{in} \\
((f\ \lambda z.z)\ \lambda w.w)
\end{array}$$

We account for this effect by interpreting $\langle \lambda x.\underline{1} \rangle$ as the set of *safe* values (see Section 2.9). Not all values satisfying $\underline{\mathrm{int}} \to \underline{\mathrm{int}}$ are safe, justifying the claim that the identity violates $(\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \lambda x.\underline{1} \rangle$. As a non-trivial theorem we prove that the Findler-Felleisen algorithm is sound and complete with respect to our semantics. Soundness and completeness together mean that this semantics is equivalent to $[\![\cdot]\!]_{\mathrm{FF}}$.

For a given interpretation $[\![\cdot]\!]$ of contracts, we call the contract checking algorithm sound if blame on a module e_i is explained by the fact that e_i violates one of it contract interfaces. If e_i is closed this says that its evaluation result (written $[\![e_i]\!]$) is not in $[\![t_i]\!]$. If e_i contains free references to variables x_j (with $j < i$) it means that there are values $v_1 \in [\![t_1]\!], \ldots, v_{i-1} \in [\![t_{i-1}]\!]$ such that $e_i[v_j/x_j]_{j=1\ldots i-1}$ produces a result that is not in $[\![t_i]\!]$.[1] Soundness relative to $[\![\cdot]\!]$ can be stated as $\forall t.[\![t]\!] \subseteq [\![t]\!]_{\mathrm{FF}}$.

Conversely, we say that the algorithm is complete with respect to the semantics if the contract checker can detect every interface violation in at least one context. Concretely, let e have free variables $x_1,\ldots,x_{i-1}$. If there are values $v_1,\ldots,v_{i-1}$ satisfying $t_1,\ldots,t_{i-1}$ such that the result of $e[v_j/x_j]_{j=1\ldots i-1}$ is not in $[\![t]\!]$, then there are terms $e_1,\ldots,e_{i-1}$ and some p such that running the algorithm on

$$\begin{array}{c}
\mathbf{let}\ x_1 : t_1 = e_1\ \mathbf{in}\ \ldots \\
\mathbf{let}\ x_{i-1} : t_{i-1} = e_{i-1}\ \mathbf{in} \\
\mathbf{let}\ x_i : t = e\ \mathbf{in}\ p
\end{array}$$

results in e being blamed. Completeness relative to $[\![\cdot]\!]$ can be stated as $\forall t.[\![t]\!] \supseteq [\![t]\!]_{\mathrm{FF}}$. Soundness and completeness together imply $[\![\cdot]\!] = [\![\cdot]\!]_{\mathrm{FF}}$.

The remainder of this paper is organized as follows:

In Section 2 we formally introduce our term- and contract-languages together with a corresponding operational semantics of terms and an interpretation $[\![\cdot]\!]$ for contracts as sets of values. We also give several definitions of *safety*—a concept central to this paper—and prove them pairwise equivalent. In Section 3 we state the central lemma and use it to sketch the proof of soundness for $[\![\cdot]\!]$. The next two sections are devoted to proofs of the central lemma: in Section 4 we take a step back and prove it in a setting that assumes all predicates in contracts to be total. This simplification allows us to show the main idea of the proof without getting bogged down in details of dealing with contracts that have effects. It also allows us to prove that $[\![\cdot]\!]$ is maximal and therefore coincides with $[\![\cdot]\!]_{\mathrm{FF}}$. Section 5 then proves soundness (but not completeness) in the general case where predicates in contracts may diverge. Before we conclude in Section 7 we take a brief excursion and discuss the addition of recursive contracts in Section 6. Recursive types have various practical applications, for example, the encoding of object types [2], but we also find that they provide yet another angle from which to understand the notion of safety that is so central to our proofs and our results.

[1] As usual, we write $A[B/x]$ for the term A' obtained from A by substituting B in a capture-free manner for all free occurrences of x in A.

2 The formal setting

We consider programs at two different language levels: an external and an internal one.[2] The former can be thought of as a syntactically-sugared refinement of the latter.

2.1 Syntax

At their core, both external and internal languages (see Figures 1 and 2) consist of untyped λ-calculi with constants. As usual, there are variables $x, y, \ldots \in \text{Var}$, λ-abstractions $\lambda x.e$, and applications $(e_1\ e_2)$. For simplicity we restrict ourselves to integer constants $\underline{0}, \underline{1}, \ldots \in I$ and some primitive operations like $+$ (addition) or $>$ (comparison) over such integers. (In examples we often use infix notation for those.) For boolean values we use the convention: $\underline{1} = \textbf{true}, \textit{everything else} = \textbf{false}$.

Either language makes use of a sub-language of contract expressions consisting of $\underline{\text{int}}$ (the contract satisfied by all integer values); dependent function contracts $t_1 \xrightarrow{x} t_2$ (satisfied by functions that take values v satisfying t_1 to values satisfying $t_2[v/x]$) and their non-dependent special case $t_1 \to t_2$; the contract $\underline{\text{safe}}$ of safe values; and restricted predicate contracts $\langle t; \phi \rangle$ (satisfied by values v also satisfying t such that ϕ applied to v yields true). The unrestricted version of predicate contracts $\langle \phi \rangle$ shown in the introduction is not explicitly part of our languages and should be thought of as an abbreviation for the operationally equivalent $\langle \underline{\text{safe}}; \phi \rangle$.

External: Programs in external form are closed terms

$$\textbf{let}\ x_1 : t_1^{\text{e}} = e_1^{\text{e}}\ \textbf{in}\ \ldots \textbf{let}\ x_n : t_n^{\text{e}} = e_n^{\text{e}}\ \textbf{in}\ x_n$$

where the e_i^{e} are individual modules bound to "module identifiers" x_i. The module interface of e_i^{e} is governed by contract t_i^{e}. The scope of each **let**-bound x_i consists of everything to the right of e_i^{e} (i.e., $e_{i+1}^{\text{e}}, \ldots, e_n^{\text{e}}, x_n$). Predicates in predicate contracts within the t_i^{e} are taken from the expression language.

Module interfaces are the only place where contract expressions t_i^{e} can appear. Moreover, without loss of generality we require each such contract t_i^{e} to be closed. (The effect of a free occurrence of x_j in t_i^{e} can be simulated by abstracting from x_j in both e_i^{e} (using λ) and in t_i^{e} (using a dependent function contract).)

Internal: The internal language makes pervasive use of contract exceptions $\top^1, \top^2, \ldots$ as well as the "pseudo-exception" $\bot$. When an exception $\top^i$ is *raised*, the entire program immediately terminates, producing $\top^i$ as the final result. Raising $\bot$, however, causes the program to diverge. (We use $\bot$ as a technical device to make characterization and construction of "safe" expressions easier.)

One use of exceptions is to signal violations of *language contracts*: applications of non-functions or ill-typed (i.e., non-integer in our case) arguments to primitive operations. For this, they appear as annotations on all applications $(e_1\ e_2)_\xi$ and on all primitive operations $f_\xi(e_1, \ldots, e_k)$.[3]

There are also two additional expression forms:

- *Wrapped* expressions $(W_t^{\xi', \xi}\ e)$ represent *module contracts* and are at the heart of contract checking. They act as guards looking for evidence of violations of contract t by either e or the context. If evidence for e violating t is found, then exception ξ is raised. Similarly, when it is detected that the context tries to use e in a way that is not consistent with t, then ξ' is raised.
- The one-armed conditional $e_1 ?_\xi e_2$ evaluates to the value of e_2 if e_1 evaluates to true. If e_1 does not evaluate to true, then ξ is raised. (This form was added to make it easier to state the operational semantics of predicate contract wrappers.)

There is no **let**-form in the internal language. Instead, module boundaries and the contracts governing their interfaces are expressed using wrapped terms and function application.

2.2 From external to internal syntax

Figure 3 shows the "de-sugaring" translation from external to internal syntax. The idea is to arrange for $\top^i$ to be raised when the contract checker finds evidence for e_i^{e} not respecting its contracts.

There are three ways in which a module e_i^{e} of the external language can fail to respect its contracts:

1. Its value might not satisfy its export interface t_i^{e}.
2. It might try to use x_j (where x_j is one of its free variables) in a way that is not consistent with the import interface t_j^{e}.
3. It might use one of the language's primitive operations incorrectly[4] (trying to apply an integer, passing a non-integer or the wrong number of arguments to one of the built-in operations).

The translation from external to internal language reflects this classification: contract exception $\top^i$ appears

1. in $W_{t_i}^{\xi, \top^i}$ which is wrapped around uses of x_i,
2. in $W_{t_j}^{\top^i, \xi}$ which is wrapped around uses of x_j within e_i^{e},
3. and as an annotation on every application and built-in operation within the translation of e_i^{e}.

The translator is given in three parts: $C_\xi^e(e^{\text{e}}; \Gamma)$ annotates applications and primitive operations within e^{e} with exception ξ and replaces free occurrences of variables bound in Γ with wrapped versions of these variables; $C^t(t^{\text{e}})$ translates external contracts to internal ones; $C^p(p; \Gamma)$ translates **let**-expressions. Environments Γ are used to map **let**-bound module identifiers to their respective module exceptions and translated contracts. Thus, a closed external program p is translated using $C^p(p; \emptyset)$.

The statement of our central lemma (Lemma 5) requires that predicates within contracts do not raise contract exceptions of their own. This property, formally captured by the notion of *safe contracts*, is guaranteed in part by the fact that $C^t(\cdot)$ never inserts wrapper expressions while artificially using $\bot$ for all language contracts (applications and primitive operations), thus sending the program into an infinite loop—as opposed to having it raise an unaccounted-for contract exception—should a predicate violate one of these.

In a practical implementation it makes sense to use a separate $\top^{\text{contract}}$ instead of $\bot$ for this purpose, effectively putting contracts on contract predicates. To account for $\top^{\text{contract}}$, most of the definitions and proofs in this paper would have to be adjusted, making them superficially (but not intrinsically) more complicated. Since the increased complexity does not pay off, we do not explore this direction here.

2.3 Core semantics

We follow Felleisen and Hieb [6] and make use of *evaluation contexts* to specify the operational semantics of the internal language (see Figure 4). Every closed expression $e \in E$ that is not a value

[2] In practice there often will be a third level: a statically typed *surface language*. Here we assume that static types—if originally present—have been checked and erased. Appendix B briefly touches upon the likely interaction between static types and contracts. In general it suffices to assume a dynamically typed setting.

[3] A static type system can often eliminate the need for language contracts, but we do not make this assumption here.

[4] One can think of this as having contract wrappers on those primitive operations (and even implement it that way).

$$
\begin{array}{rcll}
V & ::= & I \mid \lambda\mathrm{Var}.E & \textit{values} \\
V^* & ::= & V \mid X & \textit{results} \\
C_e & ::= & \{\cdot\} \mid & \textit{evaluation contexts} \\
 & & (C_e\ E)_X \mid (V\ C_e)_X \mid & \\
 & & F_X(V,\ldots,V,C_e,E,\ldots,E) \mid (W_T^{X,X}\ C_e) \mid & \\
 & & C_e?_X E \mid V?_X C_e & \\
C & ::= & [\cdot] \mid & \textit{contexts} \\
 & & \lambda\mathrm{Var}.C \mid (C\ E)_X \mid (E\ C)_X \mid & \\
 & & F_X(E,\ldots,E,C,E,\ldots,E) \mid (W_T^{X,X}\ C) \mid & \\
 & & C?_X E \mid E?_X C &
\end{array}
$$

$$
\begin{array}{rcll}
v & \Downarrow_0 & v & \\
c_e\{(\underline{i}\ v)_{\top^j}\} & \Downarrow_0 & \top^j & \\
c_e\{f^j_{\top}(v_1,\ldots,\lambda x.e,\ldots,v_k)\} & \Downarrow_0 & \top^j & \\
c_e\{(W^{\xi,\top^j}_{\underline{\mathrm{int}}}\ \lambda x.e)\} & \Downarrow_0 & \top^j & \\
c_e\{(W^{\xi,\top^j}_{t_1\xrightarrow{x}t_2}\ \underline{i})\} & \Downarrow_0 & \top^j & \\
c_e\{v?_{\top^j}v'\} & \Downarrow_0 & \top^j & ;v \neq \underline{1}
\end{array}
$$

$$
\frac{e = c_e\{e'\} \quad e' \hookrightarrow e'' \quad c_e[e''] \Downarrow_n v}{e \Downarrow_{n+1} v}
$$

$$
\begin{array}{rcll}
f_\xi(\underline{i}_1,\ldots,\underline{i}_k) & \hookrightarrow & A(f,\underline{i}_1,\ldots,\underline{i}_k) & \\
((\lambda x.e)\ v)_\xi & \hookrightarrow & e[v/x] & \\
\underline{1}?_\xi v & \hookrightarrow & v & \\
(W^{\xi',\xi}_{\underline{\mathrm{int}}}\ \underline{i}) & \hookrightarrow & \underline{i} & \\
(W^{\xi',\xi}_{\underline{\mathrm{safe}}}\ v) & \hookrightarrow & v & \\
(W^{\xi',\xi}_{t_1\xrightarrow{z}t_2}\ \lambda x.e) & \hookrightarrow & & (\dagger) \\
 & & \lambda y.(W^{\xi',\xi}_{t_2[(W^{\perp,\xi'}_{t_1}\ y)/z]}\ ((\lambda x.e)\ (W^{\xi,\xi'}_{t_1}\ y))_\perp) & \\
(W^{\xi',\xi}_{\langle t;\lambda x.e\rangle}\ v) & \hookrightarrow & ((\lambda x.e)\ (W^{\perp,\xi}_{t}\ v))_\perp?_\xi(W^{\xi',\xi}_{t}\ v) &
\end{array}
$$

$$
\begin{array}{rcll}
(\underline{i}\ v)_\perp & \hookrightarrow & ((\lambda x.(x\ x)_\perp)\ \lambda x.(x\ x)_\perp)_\perp & \\
f_\perp(v_1,\ldots,\lambda x.e,\ldots,v_k) & \hookrightarrow & ((\lambda x.(x\ x)_\perp)\ \lambda x.(x\ x)_\perp)_\perp & \\
(W^{\xi,\perp}_{\underline{\mathrm{int}}}\ \lambda x.e) & \hookrightarrow & ((\lambda x.(x\ x)_\perp)\ \lambda x.(x\ x)_\perp)_\perp & \\
(W^{\xi,\perp}_{t_1\xrightarrow{x}t_2}\ \underline{i}) & \hookrightarrow & ((\lambda x.(x\ x)_\perp)\ \lambda x.(x\ x)_\perp)_\perp & \\
v?_\perp v' & \hookrightarrow & ((\lambda x.(x\ x)_\perp)\ \lambda x.(x\ x)_\perp)_\perp & ;v \neq \underline{1}
\end{array}
$$

Figure 4. Operational semantics of the internal language.

$v \in V$ has a unique decomposition into an evaluation context $c_e \in C_e$ and a current β_v-redex $e' \in E$; we write $e = c_e\{e'\}$ for this. Evaluation proceeds by repeatedly replacing the current redex with its corresponding 1-step reduction until a value is reached or a contract exception is raised.[5]

Evaluation immediately terminates with a non-value result of $\top^j$ if the contract exception $\top^j$ gets raised at any point during evaluation. Raising the pseudo-exception $\perp$ is modeled by replacing the current redex with an infinite loop.

Evaluation of e either diverges or produces a result r (either in V or some $\top^j$) after k steps. The latter fact is expressed by the relation $e \Downarrow_k r$. The valuation function $[\![\cdot]\!]$ from closed expressions to results is defined as follows:

$$e \Downarrow_k r \Rightarrow [\![e]\!] = r \qquad (\forall k, r.\neg(e \Downarrow_k r)) \Rightarrow [\![e]\!] = \perp$$

It is easy to check that the full set of rules given here is exhaustive. This means that there are no "stuck terms" in the internal language.

The meaning of built-in primitives is assumed to be given by the semantic function A.

2.4 Contract checking

The heart of the contract checker is the set of rules dealing with the case when the current redex is a wrapped expression $(W_t^{\xi',\xi}\ v)$. These rules are directed by the syntax of the contract t. If t is $\underline{\mathrm{safe}}$, then the wrapper acts as an identity function; if t is $\underline{\mathrm{int}}$, then the wrapper checks v for being an integer, raising ξ if it is not.

If t is a (potentially dependent) function contract $t_1 \xrightarrow{x} t_2$, then v is first checked for being a λ-term. If that is the case, then rule $(\dagger)$ applies: the wrapper constructs a function that first accepts an argument y and wraps it using contract t_1, then applies v to the wrapped y, and finally wraps the result using contract t_2 where (a wrapped version of) the original argument y has been substituted for x. The original exception superscripts appear in reversed order in the argument wrapper—a detail that is a crucial aspect of Findler-Felleisen-style contract checking since it reflects the role reversal between the producer of a value and its context. Such role reversals take place at the domain part of function contracts, the intuition behind it being that a value f acts as the context of any arguments that f is applied to, whereas the context of f is supplying these argument values. Formally, the rule follows the standard construction for projections. Indeed, contract wrappers are idempotent, so we can view them as retractions.[6]

If t is a restricted predicate contract $\langle t'; \phi\rangle$, then v is wrapped using t' and checked for satisfying the predicate ϕ.

When an arbitrary value v "meets" predicate code (which happens when a predicate is applied to v or when v is substituted into a dependent type), v gets wrapped with a special wrapper where the exception dedicated to blaming the context (i.e., the predicate code) is $\perp$. This technical device is part of our effort of maintaining *safety of contracts*. As hinted in Section 2.2, a practical implementation should use some $\top^{\mathrm{contract}}$ instead of $\perp$.

We give an operational semantics to external programs by way of their translation into E. Thus, $[\![p]\!]^{\mathrm{e}}$ is defined to be the same as $[\![C^p(p;\emptyset)]\!]$.

In the following examples, it is instructive to verify how the rule

[5] Notice that $e = c_e\{e'\}$ and $e' \hookrightarrow e''$ does not imply that substituting e'' for $\{\cdot\}$ in c_e has the form $c_e\{e''\}$ since in general e'' is not the next current redex. For this reason we use the notation $c_e[e'']$ when substituting into the hole of an evaluation context (just like we do when substituting into the hole of a general context $c \in C$).

[6] Unfortunately, the corresponding retracts do not coincide with the interpretation of contracts that we would like to use, which is why we have not pursued this direction here (but elsewhere [7]). For example, we want $\lambda x.x : \underline{\mathrm{int}} \to \underline{\mathrm{int}}$ to be true, but no term equivalent to $\lambda x.x$ is in the image of $W^{\xi,\xi'}_{\underline{\mathrm{int}}\to\underline{\mathrm{int}}}$ for any ξ, ξ'.

$$
\begin{array}{rcl}
e : t & \Leftrightarrow & [\![e]\!] \in [\![t]\!] \cup \{\bot\} \\
[\![\underline{\text{int}}]\!] & = & \{\underline{0}, \underline{1}, \ldots\} \\
[\![\underline{\text{safe}}]\!] & = & \textbf{Safe} \\
[\![t_1 \xrightarrow{x} t_2]\!] & = & \{\lambda y.e \mid \forall v \in [\![t_1]\!].e[v/y] : t_2[v/x]\} \\
[\![\langle t; \lambda x.e \rangle]\!] & = & \{v \in [\![t]\!] \mid [\![e[v/x]]\!] \in \{\underline{1}, \bot\}\} \\
[\![t^{\text{e}}]\!]^{\text{e}} & = & [\![C^t(t^{\text{e}})]\!]
\end{array}
$$

Figure 5. Semantics of contracts.

marked (†) in Figure 4 produces the results in cases (2) and (3):

$$((W_{\underline{\text{int}} \to \underline{\text{int}}}^{\top^1, \top^2}\ \underline{0})\ \underline{1}) \quad \Downarrow \quad \top^2 \tag{1}$$

$$((W_{\underline{\text{int}} \to \underline{\text{int}}}^{\top^1, \top^2}\ \lambda x.x+1)\ \lambda y.y) \quad \Downarrow \quad \top^1 \tag{2}$$

$$((W_{\underline{\text{int}} \to \underline{\text{int}}}^{\top^1, \top^2}\ \lambda x.\lambda y.x)\ \underline{2}) \quad \Downarrow \quad \top^2 \tag{3}$$

$$((W_{\underline{\text{int}} \to \underline{\text{int}}}^{\top^1, \top^2}\ \lambda x.x+1)\ \underline{1}) \quad \Downarrow \quad \underline{2} \tag{4}$$

2.5 Semantic equivalence

We write $e \cong e'$ to say that e and e' are *semantically equivalent*, i.e., that there is no context $c \in C$ that could distinguish between the two:

$$([\![c[e]]\!] \in [\![\underline{\text{int}}]\!] \cup X \vee [\![c[e']]\!] \in [\![\underline{\text{int}}]\!] \cup X) \ \Rightarrow\ [\![c[e']]\!] = [\![c[e]]\!]$$

We will also use the notation $e' \leq e$ for $e, e' \in E$ (or $t \leq t'$ for $t, t' \in T$) if e' (or t') can be obtained from e (or t) by replacing zero or more occurrences of $\top$ with $\bot$. We write $\lfloor e \rfloor$ and $\lfloor c \rfloor$ to denote the expression or context obtained from e or c by replacing *every* occurrence of $\top$ (regardless of label) with $\bot$ (implying $\lfloor e \rfloor \leq e$).

LEMMA 1. *If $e' \leq e$ then the following is true:*
$[\![e']\!] = \top^i \Rightarrow [\![e]\!] = \top^i \quad [\![e]\!] = \bot \Rightarrow [\![e']\!] = \bot \quad [\![e]\!] = \underline{i} \Leftrightarrow [\![e']\!] = \underline{i}$

PROOF SKETCH. By showing that the rules of the operational semantics preserve the $\leq$ relation on terms until an exception is raised. □

2.6 Semantic interpretation of contracts

The interpretation of a contract t is some set $[\![t]\!] \subseteq V$. A closed expression e is said to satisfy t (written $e : t$) if it either diverges or produces a result in $[\![t]\!]$. The rules in Figure 5 define $[\![t]\!]$ for contracts $t \in T$. The semantics $[\![\cdot]\!]^{\text{e}}$ for the external contract language T^{e} is handled by viewing it as a refinement of T. (This means that external types are interpreted as sets of internal values. See Appendix B for a justification.) The definition of **Safe** is given in Section 2.9. Notice that the semantics of contracts invokes the operational semantics for terms—reflecting the fact that contract satisfaction is determined based on runtime behavior.

Any diverging term satisfies all contracts while a term whose evaluation raises some contract exception $\top^i$ satisfies no contract. Fortunately, the same is true under $[\![\cdot]\!]_{\text{FF}}$. If e satisfies t under $[\![\cdot]\!]_{\text{FF}}$ then $(W_t^{\xi', \xi}\ v)$ should not raise ξ in any context $c \in C$ that does not contain ξ (assuming $\xi \neq \xi'$). However, this condition is violated if e itself raises ξ.

2.7 Findler-Felleisen-style contract checking

We found it remarkable that contract checking works at all, i.e., that one can prove it sound with respect to a simple compositional semantics. Checking higher-order contracts requires type tests at higher-order types. Membership in $[\![t_1 \to t_2]\!]$ is, however, by Rice's theorem [14] undecidable! The trick used by the Findler-Felleisen algorithm is to give up on this unattainable goal and settle for less. When a runtime error is generated, the contract checker reports that a certain claim of the form $v : t$ is false. However, even the ability to do that might come as a bit of a surprise since it seems to require being able to verify claims of the form $v : t_1 \to t_2$ after all. In particular, consider proving that $\neg(f : (t_1 \to t_2) \to t_3)$. This requires showing the existence of a witness v such that $v : t_1 \to t_2$ and $\neg((f\ v) : t_3)$. As pointed out before, we generally cannot know whether some v satisfies $t_1 \to t_2$. What we do know, however, is that even if v was not in $t_1 \to t_2$, at the time f got blamed for not being in $(t_1 \to t_2) \to t_3$ this fact had not yet been found out. In other words, the argument v of f has so far behaved like a value in $t_1 \to t_2$. The idea behind the soundness proof is to show that there is some v' that in this particular context acts just like v but which actually *does* satisfy $t_1 \to t_2$. The construction of v' is one of the technical difficulties of the soundness proof.

Let us look at two examples: First, let t_1 stand for the contract

$$\underline{\text{int}} \xrightarrow{i} (\langle \underline{\text{int}}; \lambda x.x < i \rangle \to \langle \underline{\text{int}}; \lambda x.x > \underline{0} \rangle)$$

in the program fragment:

$$\textbf{let}\ x_1 : t_1 = \lambda i.\lambda k.k - i\ \textbf{in let}\ x_2 : \underline{\text{int}} = ((x_1\ \underline{4})\ \underline{3})\ \textbf{in}\ x_2$$

This code will fail at runtime and report a contract violation. The arguments to x_1 pass their respective tests while the return value does not, so the contract checker produces $\top^1$, accusing x_1 for breaking t_1. This is correct because the arguments to x_1 constitute a *counterexample* to $x_1 : t_1$.

Take a look at this variant of the program:

$$\textbf{let}\ x_1 : t_1 = \lambda i.\lambda k.i - k\ \textbf{in let}\ x_2 : \underline{\text{int}} = ((x_1\ \underline{3})\ \underline{4})\ \textbf{in}\ x_2$$

The body of x_1 now produces a positive number as promised, if the arguments satisfy their contracts. The order of the arguments, however, have been inverted so that they no longer satisfy x_1's contract. The contract checker now raises $\top^2$ because x_2 abuses x_1. Notice how important it is for the argument contract to be checked before the result contract, as otherwise the wrong exception would have been raised.

In our second example, let t_1 stand for

$$(\underline{\text{int}} \to \langle \underline{\text{int}}; \lambda x.x \geq \underline{0} \rangle) \to \langle \underline{\text{int}}; \lambda x.x \geq \underline{0} \rangle$$

and consider:

$$\begin{array}{l}
\textbf{let}\ x_1 : t_1 = \lambda g.((g\ \underline{1}) - \underline{1})\ \textbf{in} \\
\quad \textbf{let}\ x_2 : \underline{\text{int}} = (x_1\ (\lambda x.(x - \underline{1})))\ \textbf{in} \\
\quad\quad x_2
\end{array}$$

Again, this is a call of a function with an argument that is *not* in its stated domain, but it escapes discovery because it is itself a function and never gets applied to a value that would witness this fact. Instead, the Findler-Felleisen algorithm detects a violation of the range contract of x_1, so the result is $\top^1$ which says that x_1 does not satisfy its contract—even though the checker has not really seen a counterexample! Blaming x_1 is nevertheless correct here since there exist other values, for example

$$\lambda x.\underline{0} \in [\![\underline{\text{int}} \to \langle \underline{\text{int}}; \lambda x.x \geq \underline{0} \rangle]\!]$$

that can witness the problem with x_1 in precisely the same way.

2.8 Behavioral correctness

An important property of contract checking is that it must not change the behavior of a program in an *essential* way. By this we

mean that as long as no exceptions are raised, there is no other way of operationally distinguishing between e and $(W_t^{\xi',\xi}\ e)$:

LEMMA 2. *Let $e' = (W_t^{\xi',\xi}\ e)$. If $[\![c[e']]\!] = \underline{i}$ (with $\underline{i} \in I$) then $[\![c[e]]\!] = \underline{i}$. Also, if $[\![c[e]]\!] = \underline{i}$ and $[\![c[e']]\!] \in V$, then $[\![c[e']]\!] = \underline{i}$.*

PROOF SKETCH. Using a bi-simulation between expressions that contain instances of W and corresponding terms with some of these wrappers stripped out. For brevity we omit the details of the proof here. □

2.9 Safety

The concept of safety that we use here formalizes the familiar practice of coding *as defensively as possible*. It means that before attempting any operation that could "go wrong" the program makes sure that it will, in fact, not go wrong. For example, a safe program in a dynamically typed language must verify that the arguments of $+$ are indeed numeric and take evasive action if they are not.[7] In the higher-order case the caveat is that one can never be sure that an unknown function does not itself raise an exception after being called. The definition of safety takes this into account.

Behavioral safety: An expression e is safe if and only if it is impossible to trigger any of its syntactically embedded contract exceptions. Thus, e must remain semantically unchanged if some or all of its $\top$s are replaced with $\bot$:
DEFINITION 1 (SAFETY, TAKE 1).

$$\mathbf{Safe}_1 = \{v \in V \mid \lfloor v \rfloor \cong v\}$$

Let $\mathbf{Safe}^{\text{syn}} = \{\lfloor v \rfloor \mid v \in V\}$ be the set of *syntactically safe* values, i.e., values that do not contain syntactic occurrences of $\top^i$. From definition 1 it is then immediately clear that $\mathbf{Safe}^{\text{syn}} \subseteq \mathbf{Safe}_1$.

Safe in syntactically safe contexts: The second definition characterizes safe values as those that do not trigger a contract exception in any syntactically safe context (i.e., contexts without syntactic occurrences of $\top$):
DEFINITION 2 (SAFETY, TAKE 2).

$$\mathbf{Safe}_2 = \{v \in V \mid \forall c \in C.[\![\lfloor c \rfloor [v]]\!] \in V \cup \{\bot\}\}$$

Safety as a greatest fixpoint: To explicitly deal with the problem of safety in a higher-order setting we would like to say that a function f is safe whenever the result of applying f to a safe value v is still safe. Unfortunately, this is not a definition for precisely the same reason that makes the interpretation of recursive types difficult. The operator whose fixpoint we are after is not monotonic. To get around this problem we weaken the condition and say that v is safe if it is a "flat" value ($\underline{0}, \underline{1}, \ldots$ in our case) or a function returning something safe whenever applied to a *syntactically* safe argument. Thus, we take $\mathbf{Safe}_3$ to be the greatest fixpoint νS of the monotonic operator S:
DEFINITION 3 (SAFETY, TAKE 3).

$$\begin{array}{rcl} S(Q) & = & \{v \in V \mid \forall w \in V.[\![(v\ \lfloor w \rfloor)_\bot]\!] \in Q \cup \{\bot\}\} \\ \mathbf{Safe}_3 & = & \nu S \end{array}$$

Properties of safety: All notions of safety are pairwise equivalent.

[7] Depending on what one considers "wrong," even statically typed programs must perform certain runtime tests to be safe. Example: index range checks in subscript expressions.

LEMMA 3.

$$\mathbf{Safe}_1 = \mathbf{Safe}_2 = \mathbf{Safe}_3$$

PROOF. This follows from lemmas 10 and 11, both shown and proved in appendix A.1. □

Since the three versions of safety are equivalent we drop the subscript and simply write **Safe**. We use the subscripted version when we want to indicate the properties of **Safe** that we use for a proof.

By definition, it is impossible to operationally distinguish between a $v \in \mathbf{Safe}_1$ and the corresponding $\lfloor v \rfloor$. By plugging this fact into the definition of $\mathbf{Safe}_3$ we conclude that **Safe** is also the greatest fixpoint of $\hat{S}$, defined as

$$\hat{S}(Q) = \{v \in V \mid \forall w \in Q.[\![(v\ w)_\bot]\!] \in Q \cup \{\bot\}\}$$

This coincides with our original intuition of safe values being those that remain safe when applied to other safe values, a fact that can be stated as follows:

LEMMA 4. $e, e' : \underline{\text{safe}} \Rightarrow (e\ e')_\bot : \underline{\text{safe}}$

PROOF. Follows immediately from **Safe** being the greatest fixpoint of $\hat{S}$ and $[\![\underline{\text{safe}}]\!] = \mathbf{Safe}$. □

3 Soundness and completeness

We would like to show that $[\![\cdot]\!]_{\text{FF}}$ and $[\![\cdot]\!]$ are the same, but this is true only if we make certain assumptions about predicates. A sufficient condition is all predicates being total. But even without assuming totality we can show contract checking to be sound, i.e., $[\![t]\!] \subseteq [\![t]\!]_{\text{FF}}$. This means that any difference between $[\![t]\!]_{\text{FF}}$ and $[\![t]\!]$ can always be explained by non-terminating predicate code.[8] In any case, blame assignment is sound as every blame is justified by a corresponding contract violation:
THEOREM 1 ($[\![t]\!] \subseteq [\![t]\!]_{\text{FF}}$). *If*

$$[\![\mathbf{let}\ x_1 : t_1^{\text{e}} = e_1^{\text{e}}\ \mathbf{in} \ldots \mathbf{let}\ x_n : t_n^{\text{e}} = e_n^{\text{e}}\ \mathbf{in}\ x_n]\!]^{\text{e}} = \top^i$$

then

$$\exists v_1 \in [\![t_1^{\text{e}}]\!]^{\text{e}}, \ldots, v_{i-1} \in [\![t_{i-1}^{\text{e}}]\!]^{\text{e}}$$

such that

$$[\![e_i[v_j/x_j]_{j=1\ldots i-1}]\!] \notin [\![t_i^{\text{e}}]\!]^{\text{e}} \cup \{\bot\}$$

where $e_i = C^e_{\top^i}(e_i^{\text{e}}; \emptyset)$.
Moreover, there are $v_1^{\text{e}}, \ldots, v_{i-1}^{\text{e}} \in E^{\text{e}}$ such that

$$C^e_{\top^i}(v_j^{\text{e}}; \emptyset) \cong v_j$$

i.e.

$$e_i[v_j/x_j]_{j=1\ldots i-1} \cong C^e_{\top^i}(e_i^{\text{e}}[v_j^{\text{e}}/x_j]_{j=1\ldots i-1}; \emptyset)$$

Furthermore, we can get a completeness result if P^0 is assumed to contain only total predicates. (For example, we could take P^0 to be the set of functions that return false if one of their arguments is not an integer, and which otherwise compute a boolean combination of the results of comparing its arguments with one another. Such a class could be defined by a suitable syntactic restriction on E.) In this case every contract violation has the potential for causing corresponding blame:

[8] For example, the contract checker cannot determine that $\lambda x.\lambda y.y$ does not satisfy $\langle \underline{\text{int}} \to \underline{\text{int}}; \lambda z.(\lambda x.(x\ x)\ \lambda x.(x\ x))\rangle$ because it always gets stuck in the infinite loop that is the body of the predicate.

$$\frac{\mathrm{st}(\varepsilon;t)}{t \in T_{\mathrm{safe}}} \qquad \mathrm{st}(x_1,\ldots,x_k;\underline{\mathrm{int}}) \qquad \mathrm{st}(x_1,\ldots,x_k;\underline{\mathrm{safe}})$$

$$\frac{\mathrm{st}(x_1,\ldots,x_k;t) \quad \lambda x_1.\cdots\lambda x_k.\lambda x.e \in \mathbf{Safe}}{\mathrm{st}(x_1,\ldots,x_k;\langle t;\lambda x.e\rangle)}$$

$$\frac{\mathrm{st}(x_1,\ldots,x_k;t_1) \quad \mathrm{st}(x_1,\ldots,x_k,x;t_2)}{\mathrm{st}(x_1,\ldots,x_k;t_1 \xrightarrow{x} t_2)}$$

Figure 6. Safe contracts.

THEOREM 2 ($[\![t]\!] \supseteq [\![t]\!]_{\mathrm{FF}}$). *If all predicates in P^0 are total and $\exists v_1 \in [\![t_1^{\mathrm{e}}]\!]^{\mathrm{e}},\ldots,v_{i-1} \in [\![t_{i-1}^{\mathrm{e}}]\!]^{\mathrm{e}}$ with*

$$[\![e[v_j/x_j]_{j=1\ldots i-1}]\!] \notin [\![t^{\mathrm{e}}]\!]^{\mathrm{e}} \cup \{\bot\} \text{ where } e = C^{e}_{\top^i}(e^{\mathrm{e}};\emptyset)$$

for some $e^{\mathrm{e}} \in E^{\mathrm{e}}$ with free variables $x_1,\ldots,x_{i-1}$, then there are $e_1^{\mathrm{e}},\ldots,e_{i-1}^{\mathrm{e}} \in E^{\mathrm{e}}$ and $p \in P$ such that:

$$\left[\!\!\left[\begin{array}{l} \textbf{let } x_1 : t_1^{\mathrm{e}} = e_1^{\mathrm{e}} \textbf{ in } \ldots \textbf{let } x_{i-1} : t_{i-1}^{\mathrm{e}} = e_{i-1}^{\mathrm{e}} \textbf{ in} \\ \quad \textbf{let } x_i : t^{\mathrm{e}} = e^{\mathrm{e}} \textbf{ in } p \end{array} \right]\!\!\right]^{\mathrm{e}} = \top^i$$

(The $e_1^{\mathrm{e}},\ldots,e_{i-1}^{\mathrm{e}}$ can be picked from the set of closed expressions.)

3.1 The central lemma

Before we can state the central lemma we need to introduce a safety restriction on contracts (Figure 6). Safety guarantees that predicates within contracts do not raise exceptions of their own. The formula $\mathrm{st}(x_1,\ldots,x_k;t)$ expresses that t, which may have free variables in $\{x_1,\ldots,x_k\}$, is safe. A closed contract t is in T_{safe} if $\mathrm{st}(\varepsilon;t)$ where ε denotes the empty sequence of variables.

By slight abuse of notation, let's write $W_t^{\xi',\xi}$ for $\lambda x.(W_t^{\xi',\xi}\ x)$ and $W_{t_1}^{\xi'_1,\xi_1} \circ W_{t_2}^{\xi'_2,\xi_2}$ for $\lambda x.(W_{t_1}^{\xi_1,\xi_1}\ (W_{t_2}^{\xi'_2,\xi_2}\ x))$.

An easy induction on the structure of contract t shows that contract wrappers have a *telescoping* property:

$$W_t^{\xi_1,\xi_2} \circ W_t^{\xi_3,\xi_4} = W_t^{\xi_1,\xi_4}$$

Thus, wrappers $W_t^{\top^i,\bot}$ and $W_t^{\bot,\top^j}$ can be seen as two "halves" of $W_t^{\top^i,\top^j}$. The central lemma states that one half coerces safe values into values satisfying the contract while the other half coerces contract-satisfying values into safe values:[9]

LEMMA 5 (CENTRAL LEMMA). *For any $\xi \in X$ and any $t \in T_{\mathrm{safe}}$:*

a. $v : t \Rightarrow (W_t^{\bot,\xi}\ v) : \underline{\mathrm{safe}}$

b. $v : \underline{\mathrm{safe}} \Rightarrow (W_t^{\xi,\bot}\ v) : t$

Once again abusing notation, we can render this as:

$$W_t^{\bot,\xi} : t \to \underline{\mathrm{safe}} \qquad W_t^{\xi,\bot} : \underline{\mathrm{safe}} \to t$$

3.2 Proof of soundness

The proof for Theorem 1 uses Lemma 5(b) to construct the required values $v_1,\ldots,v_{i-1}$ and then finishes by applying Lemma 5(a).

[9] One is tempted to look for an embedding-projection pair here, but notice that neither $W_t^{\top^i,\bot} \circ W_t^{\bot,\top^j} = W_t^{\top^i,\top^j}$ nor $W_t^{\bot,\top^j} \circ W_t^{\top^i,\bot} = W_t^{\bot,\bot}$ is an identity on a domain we are interested in.

PROOF THEOREM 1 (SKETCH). First we define a substitution σ defined by the equation

$$\sigma(x_i) = C^{e}_{\top^i}(e_i^{\mathrm{e}};\emptyset)[(W_{C^t(t_j)}^{\top^i,\top^j}\ \sigma(x_j))/x_j]_{j=1\ldots i-1}$$

Let e be the expression $(W_{C^t(t_n^{\mathrm{e}})}^{\bot,\top^n}\ \sigma(x_n))$. Note that e is the **let**-expansion of the original program's internal form.

Suppose $[\![e]\!] = \top^i$. Intuitively, there is a particular occurrence of an exception label in e, the offending $\top^i$, which gets returned as the exception value. We can write e as

$$c\left[\left(W_{t_i}^{\top^j,\top^i}\ C^{e}_{\top^i}(e_i^{\mathrm{e}};\emptyset)[(W_{t_k}^{\top^i,\top^k}\ \sigma(x_k))/x_k]_{k=1\ldots i-1}\right)\right]$$

such that the offending $\top^i$ is not in $c \in C$. Since the offending $\top^i$ is neither in c nor in any of the $\sigma(x_k)$, using $e_i = C^{e}_{\top^i}(e_i^{\mathrm{e}};\emptyset)$, we have

$$[\![\lfloor c \rfloor\left[\left(W_{t_i}^{\bot,\top^i}\ e_i[(W_{t_k}^{\top^i,\bot}\ \lfloor\sigma(x_k)\rfloor)/x_k]_{k=1\ldots i-1}\right)\right]]\!] = \top^i$$

Pick v_k for $k = 1\ldots i-1$ to be $(W_{t_k}^{\top^i,\bot}\ \lfloor\sigma(x_k)\rfloor)$. By Lemma 5(b) we find $v_k \in [\![t_k^{\mathrm{e}}]\!]^{\mathrm{e}}$ as required. Each v_k has a semantically equivalent external version v_k^{e} (see Appendix B). Substituting v_k into the above equation yields

$$[\![\lfloor c \rfloor\left[\left(W_{t_i}^{\bot,\top^i}\ e_i[v_k/x_k]_{k=1\ldots i-1}\right)\right]]\!] = \top^i$$

which means that $e_i[v_k/x_k]_{k=1\ldots i-1} : [\![t_i^{\mathrm{e}}]\!]^{\mathrm{e}}$ would contradict Lemma 5(a). □

The proof sketch for Theorem 2 is shown in Section 4.2.

4 Assuming total predicates

In this section we consider the case that each $\rho \in P^0$ in a predicate contract $\langle t;\lambda x_n.(\ldots(\rho\ x_1)_\bot\ \ldots x_n)_\bot\rangle$ is a total function from n arbitrary values to $\underline{\mathrm{int}}$. This assumption implies that contracts are always in T_{safe}. Moreover, relying on Lemma 2 we can equivalently write the operational semantics for contract wrappers in a simpler way:

$$\begin{array}{rcl} (W^{\xi',\xi}_{t_1 \xrightarrow{z} t_2}\ \lambda x.e) & \hookrightarrow & \lambda y.(W^{\xi',\xi}_{t_2[y/z]}\ ((\lambda x.e)\ (W_{t_1}^{\xi,\xi'}\ y))_\bot) \\ (W^{\xi',\xi}_{\langle t;\lambda x.e\rangle}\ v) & \hookrightarrow & ((\lambda x.e)\ v)_\bot ?_\xi (W_t^{\xi',\xi}\ v) \end{array}$$

4.1 A simple proof of the central lemma

We now give a proof of Lemma 5 under the assumption of totality for predicates:

PROOF CENTRAL LEMMA. By simultaneous induction on the structure of t. We only show the two most important cases. (All other cases are trivial.) The first is $t = t_1 \xrightarrow{z} t_2$ and $v = \lambda x.e$:

a. Consider any syntactically safe w: By induction hypothesis (part b.) we have $(W_{t_1}^{\xi,\bot}\ w) : t_1$, so using the contract on $\lambda x.e$ we get $((\lambda x.e)\ (W_{t_1}^{\xi,\bot}\ w)) : t_2[(W_{t_1}^{\xi,\bot}\ w)/z]$. If this expression diverges, then by definition it satisfies $t_2[w/z]$. Otherwise, we get the same result by noting that $[\![t_2[w/z]]\!]$ must be equal to $[\![t_2[(W_{t_1}^{\xi,\bot}\ w)/z]]\!]$. This again follows from Lemma 2 since otherwise one of the total integer-result predicates would have to be able to distinguish between w and $(W_{t_1}^{\xi,\bot}\ w)$. Using the induction hypothesis (part a.) we find that

$$(W^{\bot,\xi}_{t_2[w/z]}\ ((\lambda x.e)\ (W_{t_1}^{\xi,\bot}\ w)))$$

is safe. By definition of $\mathbf{Safe}_3$, using the (simplified) rule for $W^{\xi',\xi}_{t_1 \xrightarrow{z} t_2}$ this means that $(W^{\bot,\xi}_{t_1 \xrightarrow{z} t_2}\ \lambda x.e)$ is safe.

b. Consider any $w : t_1$: By induction hypothesis (part a.) we know that $(W^{\perp,\xi}_{t_1}\ w)$ is safe, so by Lemma 4 we find $((\lambda x.e)\ (W^{\perp,\xi}_{t_1}\ w))_\perp$ to be safe as well. By induction hypothesis (part b.) this means that:

$$(W^{\xi,\perp}_{t_2[w/z]}\ ((\lambda x.e)\ (W^{\perp,\xi}_{t_1}\ w))_\perp) : t_2[w/z]$$

Using our semantics for $t_1 \xrightarrow{z} t_2$ and the corresponding (simplified) operational rule we get the desired result, namely $(W^{\xi,\perp}_{t_1 \xrightarrow{z} t_2}\ \lambda x.e) : t_1 \xrightarrow{z} t_2$

The other interesting case is $t = \langle t'; \lambda x.e\rangle$:

a. Since $v \in [\![\langle t'; \lambda x.e\rangle]\!]$ we also have $v \in [\![t']\!]$ and $[\![((\lambda x.e)\ v)_\perp]\!] = \underline{1}$. But $(W^{\perp,\xi}_{\langle t';\lambda x.e\rangle}\ v)$ makes a transition to $((\lambda x.e)\ v)_\perp ?_\xi (W^{\perp,\xi}_{t'}\ v)$ and finally evaluates to $[\![W^{\perp,\xi}_{t'} v]\!]$, which is safe by part a. of the induction hypothesis.

b. $((\lambda x.e)\ v)_\perp$ must evaluate to an integer (by our totality assumption). If that result is not $\underline{1}$, then

$$((\lambda x.e)\ v)_\perp ?_\perp (W^{\xi,\perp}_{t'}\ v)$$

raises the $\perp$ exception which is in $[\![t']\!]$. The outcome $\underline{1}$ makes the final result $(W^{\xi,\perp}_{t'}\ v)$, which by induction hypothesis (part b.) is in $[\![t']\!]$. Furthermore, Lemma 2 tells us that $((\lambda x.e)\ (W^{\xi,\perp}_{t'}\ v))$ cannot evaluate to anything other than $\underline{1}$, so the result is indeed in $[\![\langle t'; \lambda x.e\rangle]\!]$. □

4.2 Completeness

We now show that $[\![\cdot]\!]$ is maximal under the totality assumption. First we need the following lemma:

LEMMA 6. *If $v : t'$ but $\neg(c[v] : t)$, then $\neg(c[(W^{\top^i,\top^j}_{t'}\ v)] : t)$*

The proof for this is shown in appendix A.2.

PROOF THEOREM 2. To complete the proof of Theorem 2, recall that we have $v_1 \in [\![t^{\mathrm{e}}_1]\!]^{\mathrm{e}}, \ldots, v_{i-1} \in [\![t^{\mathrm{e}}_{i-1}]\!]^{\mathrm{e}}$ such that $C^e_{\top^i}(e^{\mathrm{e}};\emptyset)[v_j/x_j]_{j=1,\ldots,i-1}$ does not satisfy $C^t(t^{\mathrm{e}})$. We pick $e^{\mathrm{e}}_1, \ldots, e^{\mathrm{e}}_{i-1}$ equivalent to $\lfloor v_1 \rfloor \ldots, \lfloor v_{i-1} \rfloor$. (See Appendix B for how this can be done.) Now consider the **let**-expansion of e^{e} which is equivalent to

$$C^e_{\top^i}(e^{\mathrm{e}};\emptyset)[(W^{\top^i,\top^j}_{C^t(t_j)}\ \lfloor v_j \rfloor)/x_j]_{j=1\ldots i-1}$$

It is easy to see that $\lfloor v_j \rfloor : t_j$, so according to Lemma 6 this expression, let's call it $\hat{e}$, does not satisfy $C^t(t^{\mathrm{e}}_i)$.

What remains to be shown is the existence of a context c such that $c[(W^{\top^{i+1},\top^i}_{t_i}\ \hat{e})]$ evaluates to $\top^i$. From such a c one can then easily construct a p that completes the proof, for example $p = \mathbf{let}\ x_{i+1} : \underline{\mathrm{int}} = ((\lambda y.\underline{0})\ c[x_i])_\perp\ \mathbf{in}\ x_{i+1}$. (For this we need c to be syntactically safe. Again, see Appendix B for details.)

The construction of c proceeds by induction on the structure of t_i. We make use of the fact that the constructed context is always strict in its hole. First we note that if evaluating $\hat{e}$ raises an exception, then this exception must be $\top^i$ since all other available $\top^j$ would, by Theorem 1, blame one of the v_j, and those do satisfy their respective contracts. We now consider the case $[\![\hat{e}]\!] \in V$ and construct c according to t_i.

$\underline{\mathrm{int}}$ It suffices to make c strict in its hole so that the wrapper for x_i will be evaluated, causing $\top^i$ to be raised. For example, we can simply use $[\cdot]$.

$$\frac{}{e \preceq e} \qquad \frac{e'_1 \preceq e_1 \quad \cdots \quad e'_k \preceq e_k}{f_\perp(e'_1,\ldots,e'_k) \preceq f_\xi(e_1,\ldots,e_k)}$$

$$\frac{e'_1 \preceq e_1 \quad e'_2 \preceq e_2}{(e'_1\ e'_2)_\perp \preceq (e_1\ e_2)_\xi} \qquad \frac{e' \preceq e \quad t' \preceq t}{(W^{\perp,\xi}_{t'}\ e') \preceq (W^{\xi',\xi}_{t}\ e)}$$

$$\frac{e' \preceq e \quad t \in T_{\mathrm{safe}}}{(W^{\perp,\perp}_{t}\ e') \preceq e} \qquad \frac{e' \preceq e \quad t' \preceq t}{(W^{\xi',\perp}_{t'}\ e') \preceq (W^{\xi',\xi}_{t}\ e)}$$

$$\frac{}{t \preceq t} \qquad \frac{t' \preceq t \quad \phi' \preceq \phi}{\langle t';\phi'\rangle \preceq \langle t;\phi\rangle}$$

$$\frac{t'_1 \preceq t_1 \quad t'_2 \preceq t_2}{t'_1 \xrightarrow{x} t'_2 \preceq t_1 \xrightarrow{x} t_2}$$

Figure 7. A partial order on expressions and contracts.

$\underline{\mathrm{safe}}$ We pick c to be a context witnessing $[\![\hat{e}]\!] \notin \mathbf{Safe}_2$. Since the witnessing context itself is (syntactically) safe, it must be strict in its hole to be able to trigger the exception.

$\langle t;\phi\rangle$ We use the induction hypothesis, construct the c' corresponding to t, and make $c = c'$. Because of totality, ϕ applied to $[\![\hat{e}]\!]$ must be either true or false. If it is false, the wrapper will trigger $\top^i$ (because c is strict in its hole). If the predicate returns true, then $[\![\hat{e}]\!]$ must violate t, so by induction hypothesis c will cause $\top^i$ to be raised.

$t_1 \xrightarrow{x} t_2$ If $[\![\hat{e}]\!]$ is of the form $\lambda y.e'$, then there must be some $v \in [\![t_1]\!]$ such that $((\lambda y.e')\ v)_\perp$ does not satisfy $t_2[v/x]$. By Lemma 6 this means that $((\lambda y.e')\ (W^{\top^i,\top^{i+1}}_{t_1}\ v))_\perp$ also violates $t_2[v/x]$. Using the induction hypothesis for this contract-expression combination, we pick a c' in such a way that

$$c'[(W^{\top^{i+1},\top^i}_{t_2[v/x]}\ ((\lambda y.e')\ (W^{\top^i,\top^{i+1}}_{t_1}\ v))_\perp)]$$

raises $\top^i$. But then $c'[((W^{\top^{i+1},\top^i}_{t_1 \xrightarrow{x} t_2}\ \lambda y.e')\ \lfloor v \rfloor)_\perp]$ will also trigger $\top^i$. This means that we can pick c to be $c'[([\cdot]\ \lfloor v \rfloor)_\perp]$. If $\hat{e}$ does not evaluate to $\lambda y.e'$, then any strict c such as $[\cdot]$ will do. □

This concludes our demonstration that—given totality of predicates—our semantics for contracts $[\![\cdot]\!]$ is the same as $[\![\cdot]\!]_{\mathrm{FF}}$.

5 Not assuming total predicates

In the absence of totality, there are two potential problems with predicates in contracts: they might diverge, or they might raise contract exceptions of their own. We cannot completely avoid either problem. However, by maintaining the *safety of contracts* we manage to contain the damage well enough to keep soundness intact. As hinted earlier, contract safety relies on details in the translation of external types ($C^t(\cdot)$, where $\perp$ is used as the exception annotation on predicate code; see Section 2.2) and the way the operational semantics inserts wrappers that raise $\perp$ when predicate code misbehaves (see Section 2.4).

Without totality, neither the simplifications of the operational rules used in Section 4 nor conclusions such as

$$[\![t_2[w/z]]\!] = [\![t_2[(W^{\xi,\perp}_{t_1}\ w)/z]]\!]$$

are true. To prove Lemma 5 under these conditions, we have to strengthen the induction hypothesis, using a partial order $\preceq$ on terms and contract expressions. The definition of this relation, which is a generalization of the $\leq$ introduced in Section 2, is shown in Figure 7. Roughly, we say $e' \preceq e$ (or $t' \preceq t$) if e' (or t') can be obtained from e (or t) by turning some or all occurrences of $\top$ into $\perp$ and, at the same time, inserting zero or more wrappers of the form $W^{\perp,\perp}_{\hat{t}}$ where $\hat{t} \in T_{\mathrm{safe}}$.

Using this relation we can state a generalization of Lemmas 1 and 2 as follows:

LEMMA 7. *If $e' \preceq e$ and $[\![e']\!] = \underline{i}$ for some $\underline{i} \in I$, then $[\![e]\!] = \underline{i}$. Also, if $[\![e]\!] = \underline{i}$ and $[\![e']\!] \in V$, then $[\![e']\!] = \underline{i}$.*

The proof for this proceeds like that for Lemma 2 (using a bisimulation between terms related via $\preceq$). We omit the details here and just point out that the basic idea is to have e' either diverge or, as long as it does not diverge, behave exactly like e.

5.1 The stronger version of the central lemma

Now we are ready to state the stronger version of Lemma 5:

LEMMA 8 (STRONGER VERSION OF CENTRAL LEMMA). *For any $\xi \in X$, any $t' \in T_{\text{safe}}$, and t such that $t' \preceq t$*

a. $v : t \Rightarrow [\![(W^{\perp,\xi}_{t'}\ v)]\!] : \underline{\text{safe}}$

b. $v : \underline{\text{safe}} \Rightarrow [\![(W^{\xi,\perp}_{t'}\ v)]\!] : t$

PROOF. As in the proof given in Section 4 we only consider the two most important cases.

$\boxed{t_1 \xrightarrow{z} t_2, \lambda x.e}$ By definition, we have $t'_1 \preceq t_1, t'_2 \preceq t_2$.

a. Using the fact that $t'_2[(W^{\perp,\perp}_{t'_1}\ v)/z] \preceq t_2[(W^{\xi,\perp}_{t'_1}\ v)/z]$ we need to show that the result of applying

$$(W^{\perp,\xi}_{t'_1 \xrightarrow{z} t'_2}\ \lambda x.e)$$

to a safe value v is safe. This can be seen as follows:

$$\underbrace{(W^{\perp,\xi}_{t'_2[(W^{\perp,\perp}_{t'_1}\ v)/z]}\ \underbrace{((\lambda x.e)\ (W^{\xi,\perp}_{t'_1}\ \overbrace{v}^{t_1}))}_{t_2[(W^{\xi,\perp}_{t'_1}\ v)/z]}}_{\underline{\text{safe}}}$$

For this, from inside-out, we are using the assumption about v, the induction hypothesis (b.), the assumption about the contract on $\lambda x.e$, and the induction hypothesis (a.).

b. Let v be a value in t_1. Then, by induction hypothesis (a.), $(W^{\perp,\xi}_{t'_1}\ v)$ is safe, so by definition of $\mathbf{Safe}_1$ it is, in fact, equal to $(W^{\perp,\perp}_{t'_1}\ v)$, which means that we have:

$$t'_2[(W^{\perp,\xi}_{t'_1}\ v)/z] \cong t'_2[(W^{\perp,\perp}_{t'_1}\ v)/z] \preceq t_2[v/z]$$

Using this we need to show that the result of applying $(W^{\xi,\perp}_{t'_1 \xrightarrow{z} t'_2}\ \lambda x.e)$ to v satisfies $t_2[v/z]$, which can be seen from the following:

$$\underbrace{(W^{\xi,\perp}_{t'_2[(W^{\perp,\xi}_{t'_1}\ v)/z]}\ \underbrace{((\lambda x.e)\ (W^{\perp,\xi}_{t'_1}\ \overbrace{v}^{t_1}))}_{\underline{\text{safe}}}}_{t_2[v/z]}\ \overbrace{\ }^{\underline{\text{safe}}}$$

Again, from inside-out, we used the contract satisfaction assumption about v, induction hypothesis (a.), the safety assumption about $\lambda x.e$, and induction hypothesis (b.).

Remark: Notice that under the assumption of $t'_1 \xrightarrow{z} t'_2$ being in T_{safe} we find that all contracts in wrapper expressions are also in T_{safe}.

$\boxed{\langle t;\phi\rangle}$ By definition we have $t' \preceq t$ and $\phi' \preceq \phi$.

a. Let $v \in [\![\langle t;\phi\rangle]\!]$, which means that $v \in [\![t]\!]$ and $[\![(\phi\ v)]\!] \in \{\perp, \underline{1}\}$. Consider $(W^{\perp,\xi}_{\langle t';\phi'\rangle}\ v)$ which expands into

$$\underbrace{(\phi'\ \underbrace{(W^{\perp,\xi}_{t'}\ \overbrace{v}^{t})}_{\underline{\text{safe}}})}_{\underline{\text{safe}}}?_\xi\underbrace{(W^{\perp,\xi}_{t'}\ \overbrace{v}^{t})}_{\underline{\text{safe}}}$$

As before, the annotations show the conclusions we can draw from induction hypotheses and contract satisfaction assumptions. The only way the shown expression might not be safe is by having $(\phi'\ (W^{\perp,\xi}_{t'}\ v))$, which by the properties of safety is the same as $(\phi'\ (W^{\perp,\perp}_{t'}\ v))$, yielding a proper value other than $\underline{1}$. By Lemma 7 this would imply that $(\phi\ v)$ also returns a value other than $\underline{1}$, and that contradicts the assumptions.

b. Let $v \in \mathbf{Safe}$ and consider $(W^{\xi,\perp}_{\langle t';\phi'\rangle}\ v)$ which expands into

$$\underbrace{\overbrace{(\phi'\ \overbrace{(W^{\perp,\perp}_{t'}\ \overbrace{v}^{\underline{\text{safe}}})}^{\underline{\text{safe}}})}^{\underline{\text{safe}}}}_{\underline{\text{safe}}}?_\perp(W^{\xi,\perp}_{t'}\ v)$$

Clearly, if the final value here is not $\perp$, then it must be true that

$$[\![(\phi'\ (W^{\perp,\perp}_{t'}\ v))]\!] = \underline{1}$$

and the result is $(W^{\xi,\perp}_{t'}\ v)$. But in that case, since $\phi' \preceq \phi$ by Lemma 7 we also have $[\![(\phi\ (W^{\xi,\perp}_{t'}\ v))]\!] = \underline{1}$, which means that the value satisfies $\langle t;\phi\rangle$. □

Lemma 5 is implied by Lemma 8. As a result, we have a proof for Theorem 1 (stating the soundness of contract checking) even in the more general setting where contract predicates might not terminate, and where the substitution of unsafe terms into predicate terms can cause contract exceptions from predicate code. The key here is to carefully control the latter effect: contract exceptions raised by predicate code always correctly point to genuine contract violations in other parts of the program.

6 Recursive contracts

Adding recursive contracts $\mu\alpha.t$ to the contract language and accounting for this change in the operational semantics is relatively straightforward.[10] Here are the changes:

$$\begin{array}{rcl} \text{TyVar} & ::= & \alpha \mid \beta \mid \ldots \\ T^{\text{e}} & ::= & \ldots \mid \mu\text{TyVar}.T^{\text{e}} \mid T^{\text{e}} \vee T^{\text{e}} \\ T & ::= & \ldots \mid \mu\text{TyVar}.T \mid T \vee T \\ \\ C^t(\mu\alpha.t^{\text{e}}) & = & \mu\alpha.C^t(t^{\text{e}}) \\ C^t(t^{\text{e}}_1 \vee t^{\text{e}}_2) & = & C^t(t^{\text{e}}_1) \vee C^t(t^{\text{e}}_2) \end{array}$$

[10] We also add a form of sum contracts $t_1 \vee t_2$ where $[\![t_1]\!]$ and $[\![t_2]\!]$ are recursively separable (meaning there is a computable total predicate on V which is true for all of $[\![t_1]\!]$ and false for all of $[\![t_2]\!]$). Such sums are sometimes called *tidy sums*.

$$(W^{\xi',\xi}_{\mu\alpha.t}\ v) \quad \hookrightarrow \quad (W^{\xi',\xi}_{t[\mu\alpha.t/\alpha]}\ v)$$

$$(W^{\xi',\xi}_{t_1\vee t_2}\ v) \quad \hookrightarrow \quad \begin{cases} (W^{\xi',\xi}_{t_1}\ v) & : \quad \text{if } v \in [\![t_1]\!] \\ (W^{\xi',\xi}_{t_2}\ v) & : \quad \text{if } v \notin [\![t_1]\!] \end{cases}$$

Because of the rule in the operational semantics that identifies $\mu\alpha.F(\alpha)$ with $F(\mu\alpha.F(\alpha))$ (where F is a contract function), structural induction on contract expression breaks down in the presence of recursive contracts. If we could restrict F to covariant contract functions, then we would be able to salvage the situation using co-induction. For many uses of recursive contracts this is sufficient. However, there are useful applications of recursive types (and contracts) where F is not covariant. For example, several popular encodings of object types have this property.

There is another reason why considering recursive contracts in the context of contract checking is useful: it provides us with a different (but consistent!) view on the problem of how to interpret Findler and Felleisen's original unrestricted predicate contracts.

6.1 Indexing

We have made extensive use of structural induction, so our proofs do not work in the presence of recursive contracts. Fortunately, it is straightforward (although tedious) to adopt Appel and McAllester's indexed model of recursive types [1] and to modify proofs accordingly.

In the indexed model, a contract t is interpreted as a set $[\![t]\!]_{\text{idx}}$ of indexed terms $\langle k, v\rangle$. The idea is that v is a k-approximation of a value satisfying t, i.e., that no context c can tell in k or fewer steps that v does not satisfy t. The original interpretation of contracts can then be recovered as $[\![t]\!] = \bigcap_k \{v \mid \langle k, v\rangle \in [\![t]\!]_{\text{idx}}\}$. Along with this interpretation of contracts as index-value pairs goes an indexed contract-satisfaction relation $e{:}_k t$, which is defined as $e{\Downarrow_j}v \wedge \langle k-j, v\rangle \in [\![t]\!]_{\text{idx}}$.

Of course, an adaptation of the proof also requires an indexed model of safety. But with recursive contracts in the language we can *define* safety *as a contract*:

$$\underline{\text{safe}}' = \mu\alpha.\underline{\text{int}} \vee (\alpha \to \alpha)$$

Although this looks suspiciously like the equation often used to characterize the domain of untyped λ-terms [15], it should be noted that here it does not refer to the set of all possible values but rather a proper subset thereof. (Of course, the original untyped λ-calculus does not have contract exceptions, so all terms are safe there.)

Significant effort in our non-indexed proofs was spent on showing that (abusing notation)

$$\mathbf{Safe} = \mu\alpha.\underline{\text{int}} \vee (\mathbf{Safe}^{\text{syn}} \to \alpha)$$

where $\mathbf{Safe}^{\text{syn}} = \{\lfloor w \rfloor \mid w \in V\}$ (see Lemmas 3 and 4).

Here $F = \Lambda\alpha.\underline{\text{int}} \vee (\mathbf{Safe}^{\text{syn}} \to \alpha)$ is covariant, so co-induction works. In the indexed proof, safety can be treated directly as a contract, so no such detours are necessary. Lemma 5 can be restated as

LEMMA 9 (CENTRAL LEMMA WITH RECURSIVE CONTRACTS). *If $t \in T_{\text{safe}}$ then*

$$e{:}_k t \Rightarrow (W^{\perp,\xi}_t\ e){:}_{k+1}\underline{\text{safe}} \qquad e{:}_k\underline{\text{safe}} \Rightarrow (W^{\perp,\xi}_t\ e){:}_{k+1} t$$

The proof for this proceeds by induction on k, using a close adaptation of the techniques presented by Appel and McAllester [1]. We omit the details here since they do not add any new insights.

6.2 Recursion, safety, and predicates

We presented our interpretation of $\underline{\text{safe}}$ as **Safe** in an ad-hoc fashion. Clearly, we made the right decision since we ended up with a sound (and under reasonable assumptions complete) model. However, we could have motivated our choice by looking at the statement of Lemma 5 and noting that the contract wrapper for the always-true predicate is simply the identity. If the identity must map safe values to values satisfying the contract and vice versa, then satisfying the contract and being safe should better mean the same.

Still, as before, our choice is guided by our desire for a particular theorem or lemma to hold. Although we feel that this *does* constitute strong motivation, it still might seem like a matter of personal taste.

The treatment of recursive contracts, together with the observation that $\underline{\text{safe}}$ is such a contract, gives yet another explanation for $[\![\underline{\text{safe}}]\!] = \mathbf{Safe}$, this time deriving this fact directly from the rest of the existing model and not depending on any proof details.

Observe that $W^{\xi',\xi}_{\mu\alpha.\underline{\text{int}}\vee(\alpha\to\alpha)}$ is an identity function. The formal proof for this proceeds by induction on the number of unfoldings of the recursive contract. Informally, the wrapper is clearly the identity on $\underline{\text{int}}$. All other values are of the form $\lambda x.e$ where the wrapper makes two copies of itself—one wrapped around the argument, the other wrapped around the result. But if the copies are identities, then so is the original.

We should consider contracts as being identical if their corresponding wrappers are semantically equivalent. Thus, $\underline{\text{safe}}$ is equivalent to $\mu\alpha.\underline{\text{int}} \vee (\alpha \to \alpha)$, showing that our choice of interpretation for $\underline{\text{safe}}$ is more than just a trick to make the proof for Lemma 5 go through. Another consequence of being able to define $\underline{\text{safe}}$ as a recursive contract is that we can keep the full expressive power of the original Findler-Felleisen system while dropping our ad-hoc addition of $\underline{\text{safe}}$ from the language of contracts.

As noted in the introduction, their system has *unrestricted* predicate contracts $\langle\phi\rangle$ which are operationally equivalent to our $\langle\underline{\text{safe}};\phi\rangle$. We originally started with a hunt for the proper semantics of $\langle\phi\rangle$. By the above equivalence, the answer turns out to be $\{v \in \mathbf{Safe} \mid (\phi\ v) \in \{\underline{1}, \bot\}\}$ and not, as naively expected, $\{v \in V \mid (\phi\ v) \in \{\underline{1}, \bot\}\}$. Notice that "counterexamples" like that in Section 1 work in dynamically typed settings such as DrScheme but not in the calculus given in Findler and Felleisen's paper [9] because they fail to statically type-check there.

A contract places *complementary burdens of responsibility* on an expression and its context. This is the key to understanding why $[\![\langle\lambda x.\underline{1}\rangle]\!] = \mathbf{Safe}$. The contract checker does nothing at all here, so expression and context must jointly do their best at avoiding contract exceptions. This idea is precisely captured by our notion of safety.

7 Conclusions and outlook

We developed an independent model of Findler and Felleisen's contracts for higher-order functions and proved the soundness of their contract checker. Under reasonable assumptions, it is also complete. In short, the contract checker discovers all violations and always assign blame properly.

The main technical insight from these proofs is in the simple and apparently fundamental theoretical properties of contract wrappers expressed in the central lemma (Lemma 5). The central lemma shows that there is strong interaction between the semantics of contracts and the notion of safety. Furthermore, the fact that Findler-Felleisen-style unrestricted predicate contracts $\langle\phi\rangle$ are operationally equivalent to our $\langle\underline{\text{safe}};\phi\rangle$ implies that the semantics of $\langle\phi\rangle$ has to mention safety. In our system we can avoid this "leakage" of the soundness proof into contract semantics by eliminating unrestricted predicate contracts, letting the restricted version take their place.

The full expressiveness of the original system can be restored by making it possible to express safety explicitly as a contract—either using a new ad-hoc phrase like safe or via recursive contracts.

Under reasonable assumptions about predicates, our model $[\![\cdot]\!]$ for contracts is exactly equivalent to the one implied by the contract checking algorithm. Moreover, while completeness does not stay intact, soundness is not compromised even if we drop those extra assumptions. It should be noted, however, that this result crucially relies on the fact that our language is essentially pure, the only effects being non-termination and contract exceptions. If the language has constructs with general effects (mutation, I/O), then a compositional semantics that preserves soundness seems out of reach at this point. In the calculus shown here, contracts cannot interfere with a program's execution other than by changing the termination behavior. To make them into a reliable debugging tool even in the general case with arbitrary effects, one definitely needs to preserve this property. One should be able to remove contracts without altering the semantics of the program in an essential way. A separate investigation of the restrictions on predicates that one needs for this is currently under way [7].

There are several possible future directions for this work. We have not extended the algorithm to handle polymorphism, although it may not be difficult to use higher-order wrappers, i.e., functions from wrappers to wrappers, to treat contracts of the form $\forall\alpha.t$ interpreted as $\bigcap_{t'\in T}[\![t[t'/\alpha]]\!]$. Our soundness proof is for a language with call-by-value semantics. Since most real-world languages that are pure (e.g., Clean [3] or Haskell [11]) are also lazy, it seems desirable to translate our results to a lazy setting. We believe that doing so will not be difficult.

Of course, a natural direction for further work is to implement contracts in a strongly typed language such as ML or Haskell.

7.1 Program verification

It also seems possible to apply ideas from contract checking to static program verification. In particular, symbolic evaluation of programs with contract wrappers might be able to statically verify that a particular contract exception $\top^i$ can never be raised, i.e., that module e_i^e satisfies t_i in the $[\![\cdot]\!]_{\mathrm{FF}}$ model. Assuming completeness this implies contract satisfaction in the $[\![\cdot]\!]$ model as well.

One way of showing that $\top^i$ cannot be raised is to eliminate it from the program. (There are no operational rules that generate new exceptions.) One might hope to rely on the telescoping property of contract wrappers, but this law is applicable only if the wrappers in question are indexed by the same contract:

$$W_t^{\xi_2',\xi_2} \circ W_t^{\xi_1',\xi_1} = W_t^{\xi_2',\xi_1}$$

Now consider t_1 and t_2 with $[\![t_1]\!] \subseteq [\![t_2]\!]$. In this case we would like to argue that the left side of

$$W_{t_2}^{\xi_2',\xi_2} \circ W_{t_1}^{\xi_1',\xi_1}$$

is redundant because of the "stronger" wrapper on the right. However, the right side is stronger only from the point of view of the wrapped value while it is actually the left side that is stronger from the context's point of view. Thus, we cannot simply eliminate either t_1 or t_2, but we *can* argue that neither ξ_1' nor ξ_2 could ever be raised here. It is possible to express this, e.g., as

$$W_{t_2}^{\xi_2',\bot} \circ W_{t_1}^{\bot,\xi_1}$$

but doing so seems clumsy. A leaner notation separates the two roles of $W_t^{\xi',\xi}$ (watching the value and watching the context) by defining each contract wrapper as the composition of two parts:

$$W_t^{\xi',\xi} = W_t^{-\xi'} \circ W_t^{+\xi}$$

Operational rules for $W_t^{-\xi'}$ and $W_t^{+\xi}$ are easy to set up. The main idea is to alternate between W^- and W^+ instead of swapping exception superscripts in contravariant positions. Here are the rules (abusing notation when "raising" ξ) for a simple contract language with only int and $\to$:

$$\begin{array}{rcll}
(W_{\underline{\mathrm{int}}}^{+\xi}\ \underline{i}) & \to & \underline{i} & \\
(W_{t_1\to t_2}^{+\xi}\ \lambda x.e) & \to & \lambda y.(W_{t_2}^{+\xi}\ ((\lambda x.e)\ (W_{t_1}^{-\xi}\ y))_\bot) & \\
(W_t^{+\xi}\ v) & \to & \xi & ;\textit{otherwise} \\
 & & & \\
(W_{t_1\to t_2}^{-\xi}\ \lambda x.e) & \to & \lambda y.(W_{t_2}^{-\xi}\ ((\lambda x.e)\ (W_{t_1}^{+\xi}\ y))_\bot) & \\
(W_t^{-\xi}\ v) & \to & v & ;\textit{otherwise}
\end{array}$$

W^- and W^+ commute regardless of their contract subscripts, and assuming $[\![t_1]\!] \subseteq [\![t_2]\!]$ we have:

$$W_{t_2}^{+\xi'} \circ W_{t_1}^{+\xi} = W_{t_1}^{+\xi} \quad \text{and} \quad W_{t_2}^{-\xi'} \circ W_{t_1}^{-\xi} = W_{t_2}^{-\xi'}$$

Notice that the requirements on the context expressed, by W^-, are *preconditions* while the requirements on the value, expressed by W^+, are *postconditions*. Thus, the above laws precisely capture the fact that one has to keep the weakest precondition and the strongest postcondition [10, 4]. Using $[\![t_1]\!] \subseteq [\![t_2]\!]$ we get

$$\begin{array}{rcl}
W_{t_2}^{\xi_2',\xi_2} \circ W_{t_1}^{\xi_1',\xi_1} & = & W_{t_2}^{-\xi_2'} \circ W_{t_2}^{+\xi_2} \circ W_{t_1}^{-\xi_1'} \circ W_{t_1}^{+\xi_1} \\
 & = & W_{t_2}^{-\xi_2'} \circ W_{t_1}^{-\xi_1'} \circ W_{t_2}^{+\xi_2} \circ W_{t_1}^{+\xi_1} \\
 & = & W_{t_2}^{-\xi_2'} \circ W_{t_1}^{+\xi_1}
\end{array}$$

which concisely captures the idea of eliminating contract exceptions (here ξ_2 and ξ_1') that can never be raised.

8 Acknowledgments

We greatly benefited from extensive discussions with Robby Findler as well as from helpful advice given by Matthias Felleisen and several anonymous reviewers.

9 References

[1] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5):657–683, 2001.

[2] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, pages 415–438, 1997.

[3] T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer. CLEAN: A Language for Functional Graph Rewriting. In G. Kahn, editor, *Proc. of the Conf. on Functional Programming Languages and Computer Architecture (FPCA'87), Portland, Oregon, USA*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1987.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[5] M. Felleisen, R. B. Findler, M. Flatt, and S. Kristnamurthi. The DrScheme project: An overview. *SIGPLAN Notices*, 33(6):17–23, 1998.

[6] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

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

[8] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. of the 7th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 48–59. ACM Press, 2002.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):578–580, October 1969.

[11] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[12] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report No. 117, INRIA, Feb. 1990.

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[14] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[15] D. S. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer, 1972.

A Additional proofs

A.1 $\mathbf{Safe}_1 = \mathbf{Safe}_2 = \mathbf{Safe}_3$

We split this statement into two parts:

LEMMA 10. $\mathbf{Safe}_1 = \mathbf{Safe}_2$

PROOF $\subseteq$. By definition we have $v = \lfloor v \rfloor$, so if $[\![\lfloor c \rfloor [v]]\!] = \top$ then also $[\![\lfloor c \rfloor [\lfloor v \rfloor]]\!] = \top$, but $\top$ does not even occur in $\lfloor c \rfloor [\lfloor v \rfloor]$. □

PROOF $\supseteq$. Suppose $v \neq \lfloor v \rfloor$ and c is a witnessing context that distinguishes between the two. By Lemma 1 it must be the case that $[\![c[v]]\!] = \top^i$ for some i. $\top^i$ is generated from some particular occurrence of $\top^i$ in either v or c, so it must also be the case that either $[\![\lfloor c \rfloor [v]]\!] = \top^i$ or $[\![c[\lfloor v \rfloor]]\!] = \top^i$.[11] But since c is the witnessing context for v and $\lfloor v \rfloor$ being different, the latter is impossible. This concludes the proof. □

LEMMA 11. $\mathbf{Safe}_2 = \mathbf{Safe}_3$.

PROOF $\subseteq$. Indirect: If $v \notin \mathbf{Safe}_3$ then there must be a finite sequence of values $v_1, \ldots, v_k$ such that

$$[\![(\ldots(v\ \lfloor v_1 \rfloor)_\perp \ldots \lfloor v_k \rfloor)_\perp]\!] = \top$$

but $[\![(\ldots(\lfloor \cdot \rfloor\ \lfloor v_1 \rfloor)_\perp \ldots \lfloor v_k \rfloor)_\perp]\!]$ is a syntactically safe context. □

PROOF $\supseteq$. Indirect: Pick a $v \in \mathbf{Safe}_3 \setminus \mathbf{Safe}_2$ and a corresponding $c \in C$ with $\lfloor c \rfloor [v] \Downarrow_n \top^i$ for some i so that n is minimized (i.e., we pick an unsafe but operator-safe value together with the context that demonstrates non-membership in $\mathbf{Safe}_2$ in the smallest number of evaluation steps).

The number n cannot be 0: there are no occurrences of $\top$ in $\lfloor c \rfloor$, so if $\lfloor c \rfloor [v] \Downarrow_0 \top^i$ then also $v \Downarrow_0 \top^i$, which contradicts the assumption that $v \in \mathbf{Safe}_3$.

For the case of $n > 0$ there is a unique evaluation context $\hat{c}_e \in C_e$ and corresponding $\hat{e} \in E$ such that $\hat{c}_e\{\hat{e}\} = \lfloor c \rfloor [v]$ where $\hat{e}$ is the next β_v-reduction to do in $\lfloor c \rfloor [v]$ [6]. The proof proceeds by case analysis on the possible shapes of $\hat{e}$ and shows that the transition system defining the operational semantics can perform at least one step which gives rise to another pair $(v', \lfloor c' \rfloor)$ with $v' \in \mathbf{Safe}_3 \setminus \mathbf{Safe}_2$ such that $\lfloor c' \rfloor [v'] \Downarrow_{n-1} \top^i$.

If $\hat{e}$, which cannot be a subexpression of the value v, is a subexpression of $\lfloor c \rfloor$, this is immediately clear. The remaining cases are those where v is a subexpression of $\hat{e}$. For brevity we only show the analysis for the two most interesting situations:

1. If $v = \lambda x.b$ and $\hat{e} = (v\ \lfloor v' \rfloor)_\perp$ for some subexpression $\lfloor v' \rfloor$ of $\lfloor c \rfloor$, then $\hat{c}_e$ is also syntactically safe. Moreover, since $v \in \mathbf{Safe}_3$ we can consider $d = b[v'/x]$ and find that $[\![d]\!] \in \mathbf{Safe}_3$ as well. This means that for some k with $0 < k < n$ we have $d \Downarrow_k d'$ and $d' \in \mathbf{Safe}_3$. But $\hat{c}_e[d'] \Downarrow_{n-k-1} \top^i$, which is the contradiction that we are looking for.
2. If $\hat{e} = ((\lambda x.b)\ v')_\perp$ where v is a subexpression of v' ($v' = c_0[v]$), then b,$\hat{c}_e$, and c_0 are syntactically safe. We know that $\hat{c}_e[b[v'/x]] \Downarrow_{n-1} \top^i$. Since the value $\top^i$ is generated from some single occurrence of $\top^i$ which must be within one of the copies of v within $b[v'/x]$, we can replace all occurrences of $\top$ in every other copy of v by $\perp$, thus rewriting $b[c_0[v]/x]$ as $\lfloor c_1 \rfloor [c_0[v]]$. This means that $\hat{c}_e[\lfloor c_1 \rfloor [c_0[v]]] \Downarrow_{n-1} \top^i$, which is the contradiction we are looking for since $\hat{c}_e[c_1[c_0[\cdot]]]$ is a syntactically safe context. □

[11] Making this informal argument precise is not difficult but tedious. An extended version of this paper with these details included can be obtained from the authors.

A.2 Preservation of contract violation

Lemma 6 states that, under the totality assumption of Section 4, if $v : t'$ and $\neg(c[v] : t)$ then $\neg(c[(W_{t'}^{\top^i, \top^j}\ v)] : t)$.

PROOF. By induction on the structure of t:

int Because of totality the extra wrapper cannot cause non-termination. But by Lemma 2, if $c[(W_{t'}^{\top^i, \top^j}\ v)]$ were to return an integer, then so would $c[v]$.

safe We use the definition for $\mathbf{Safe}_2$ and consider the witnessing context c' where $[\![\lfloor c' \rfloor [c[v]]]\!] = \top^k$ while $[\![\lfloor c' \rfloor [c[(W_{t'}^{\top^i, \top^j}\ v)]]]\!] \in V \cup \{\perp\}$. The remainder of this case proceeds like the proof for Lemma 2 (e.g., using bisimulation), showing that given totality of predicates the second term must raise either $\top^k$ or $\top^i$. (It cannot raise $\top^j$ since v satisfies t'.)

$\langle t; \phi \rangle$ By Lemma 2 and totality, the results of ϕ have to agree in both cases. Now use the induction hypothesis with t.

$t_1 \xrightarrow{x} t_2$ The only non-trivial case is where c has the form $\lambda x.c'$, and by definition there has to be a $w : t_1$ such that $\neg(c'[v][w/x] : t_2[w/x])$ while $c'[(W_{t'}^{\top^i, \top^j}\ v)][w/x] : t_2[w/x]$. Consider $c'' = c'[w/x]$ and use the induction hypothesis with t_2. □

B Witness expressions

On a number of occasions, in particular in the proof of Theorem 1 shown in Section 3.2 and also in the proof of Theorem 2 in Section 4.2, we construct values $v \in V$, e.g., as witnesses for the fact that some contract is being violated. Since the main theorem is stated in terms of the external language, we need semantically equivalent $e^e \in E^e$ for these v. The construction of such equivalents is possible because all our witnesses $v \in V$ are syntactically safe, and given sufficient language support safety can be "coded up."

A more realistic language E^e than the one we have—for simplicity of the presentation—restricted ourselves to would have some form of conditional that can branch on equality to $\underline{1}$. If the language also comes with a mechanism for separating functions from integers, e.g., via some *typecase* construct, then all witnesses $\lfloor v \rfloor$ have an operationally equivalent counterpart in E^e. (Typecase might be troublesome for the static type system of the surface language, but if the surface language is indeed statically typed and "safe" in the sense "well-typed programs do not go wrong", then typecase is not even needed since all of its outcomes would be statically known.)

Let us be more concrete. Suppose **(tycase** $e_1\ e_2\ e_3$**)** evaluates to $[\![e_2]\!]$ if $[\![e_1]\!] \in [\![\underline{\text{int}}]\!]$ and to $[\![e_3]\!]$ if $[\![e_1]\!] = \lambda x.e'$. Now consider an application $(e_1^e\ e_2^e)$ and re-code it as $((\lambda f.(\textbf{tycase}\ f\ \Omega\ (f\ e_2^e)))\ e_1^e)$ where Ω is a diverging term, e.g., $\Omega = ((\lambda x.(x\ x))\ \lambda x.(x\ x))$ and run that through the translator C. The implicit exception inserted by the translator ends up being "protected" by our explicit test. This means that the result is equivalent to $(e_1\ e_2)_\perp$ (where e_1 is the translation of e_1^e and e_2 that of e_2^e). Implicit exceptions in primitive operations can be protected in a similar fashion.

Finally, in the proof for Theorem 1 we need to be able to represent wrappers of the form $W_t^{\perp,\perp}$ and $W_t^{\top^i,\perp}$. Given conditionals and typecase, coding up a wrapper in a type-directed fashion is straightforward. Raising $\perp$ just means going into an infinite loop. Raising $\top^i$ can be simulated by, e.g., evaluating $(\underline{0}\ \underline{0})$. Since the overall expression gets translated using $C^e_{\top^i}(\cdot;\emptyset)$ (see Theorem 1), the exception annotation on $(\underline{0}\ \underline{0})$ will indeed be $\top^i$.