Sound and Complete Models of Contracts

MATTHIAS BLUME and DAVID McALLESTER

Toyota Technological Institute at Chicago
(*e-mail:* {blume,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. They did not, however, give a semantics of contracts. The lack of a semantics makes it impossible to define and prove soundness and completeness of the checking algorithm. (Given a semantics, a sound checker never reports violations that do not exist under that semantics; a complete checker is—in principle—able to find violations when violations exist.)

Ideally, a semantics should capture what programmers intuitively feel is the meaning of a contract or otherwise clearly point out where intuition does not match reality.

In this paper we give an interpretation of contracts for which we prove the Findler-Felleisen algorithm sound and (under reasonable assumptions) complete. While our semantics mostly matches intuition, it also exposes a problem with *predicate contracts* where an arguably more intuitive interpretation than ours would render the checking algorithm unsound. In our semantics we have to make use of a notion of *safety* (which we define in the paper) to avoid unsoundness.

We are able to eliminate the "leakage" of safety into the semantics by changing the language, replacing the original version of *unrestricted* predicate contracts with a restricted form. The corresponding loss in expressive power can be recovered by making safety explicit as a contract. This can be done either in ad-hoc fashion or by including general *recursive contracts*. The addition of recursive contracts has far-reaching implications, deeply affecting the formulation of our model and requiring different techniques for proving soundness.

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 (Milner *et al.*, 1997; Leroy, 1990) 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 *higher-order contracts* (Findler & Felleisen, 2002) as a more systematic way of expressing and monitoring runtime

invariants. Higher-order contracts are a generalization of first-order contracts which, for example, had long been a feature of the programming language Eiffel (Meyer, 1992). Contracts in general, and higher-order contracts in particular, can be thought of as a form of types too expressive for static verification. Implementations such as the DrScheme system (Findler *et al.*, 2002; Felleisen *et al.*, 1998) can provide a meaningful way of monitoring contracts dynamically, at runtime. 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

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

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. To be able to gain some intuition and make this notion precise, let us first describe the contract checking algorithm in some detail.

A Findler-Felleisen-style contract checker consists of an ordinary higher-order language such as Scheme or Standard ML, augmented with a special language construct called a *guard*, and rules for handling guards in the operational semantics. Programs of the previously mentioned form ($\star$) are handled via translation into the calculus with guards.[1] Section 2 provides precise definition of the two language levels, the translation from the first to the second, as well as the semantics of the second that we use later in the paper. For the remainder of this introduction we limit ourselves to a somewhat higher-level description.

In the simplest setting, a contract t on some expression e is either a first-order contract $\langle\phi\rangle$ or a higher-order contract $t_1 \to t_2$. In the first-order case, ϕ is a predicate, i.e., a function that maps the value of e to either "true" or "false."

A guard for any contract t wrapped around an expression e is used to monitor compliance of e with t:

$$(\mathcal{W}_t\ e)$$

In the specific case where $t = \langle\phi\rangle$, the operational meaning of this expression is to first evaluate e to a value v, then to apply ϕ to v, and depending on the outcome of this application either to raise a contract exception or to produce v as the result of the guard.

Higher-order contracts are contracts on function values and express properties of

[1] Findler and Felleisen use a similar translation of contract annotations into an internal form of guarded expressions (which they call *obligation expressions*). The definition of their translation function $\mathcal{I}_e$ can be found in Findler's Ph.D. Thesis (Findler, 2002).

the behavior of these functions. More precisely, a function f satisfies the higher-order contract $t_1 \to t_2$ if any application of f to any argument satisfying t_1 satisfies t_2. Fundamentally, such properties are undecidable (Rice, 1953), so there is no hope of having a one-shot test that f could either "pass" or "fail." Therefore, the operational meaning of $(\mathcal{W}_{t_1 \to t_2}\ f)$ (after confirming that f is indeed a function value and not, say, an integer) is to return a new function f' defined by the following equation:

$$(f'\ y) = (\mathcal{W}_{t_2}\ (f\ (\mathcal{W}_{t_1}\ y)))$$

We make two important observations here:

1. Suppose the argument of f does not satisfy t_1, causing the corresponding guard to raise an exception. This exception does *not* imply that f has failed *its* contract $t_1 \to t_2$. Instead, it indicates that the context is trying to call f with an argument that did not meet the *precondition* for f returning a value satisfying t_2.
2. When a guarded expression evaluates to a value without raising a contract exception, it is still not guaranteed that the original value satisfied the corresponding contract. The evaluation of a guard often "bakes" other guards into its result, effectively suspending them until the value is later applied as a function. Thus, a guard can fail long after its original evaluation has completed.

The Findler-Felleisen algorithm accounts for point 1 by annotating each guard with information that uniquely identifies the guarded source expression as well as its context. Moreover, for every guard there is a complement guard where the roles of expression and context are swapped. In our notation we use superscripts and annotate each guard $\mathcal{W}_t^{\xi',\xi}$ with exceptions ξ, ξ'. $\mathcal{W}_t^{\xi',\xi}$ raises ξ upon detecting that the guarded expression violated t. Likewise, ξ' is raised when the context is found to "abuse" the value, i.e., if it fails to meet the preconditions that are encoded in t. The complement of $\mathcal{W}_t^{\xi',\xi}$ is $\mathcal{W}_t^{\xi,\xi'}$. Thus, $(\mathcal{W}_{t_1 \to t_2}\ f)$ becomes $(\mathcal{W}_{t_1 \to t_2}^{\xi',\xi}\ f)$, and the resulting function f' is defined to satisfy the following equation:

$$(f'\ y) = (\mathcal{W}_{t_2}^{\xi',\xi}\ (f\ (\mathcal{W}_{t_1}^{\xi,\xi'}\ y)))$$

In short, while the result guard uses the original expression-context relationship, that relationship is reversed in the argument guard.

The other observation (point 2) is directly related to the semantics of contracts implied by the contract checking algorithm. We will write $[\![t]\!]_{\mathrm{FF}}$ for the set of values that—according to the Findler-Felleisen algorithm—satisfy t. Let us now define what $[\![\cdot]\!]_{\mathrm{FF}}$ is. Since the contract checker can only falsify claims of the form $v \in [\![t]\!]_{\mathrm{FF}}$, we say that value v is not in $[\![t]\!]_{\mathrm{FF}}$ if and only if there exists a guard $\mathcal{W}_t^{\xi',\xi}$ together with a context c such that evaluating the guarded v in c causes ξ to be raised. This leads to the following definition:

Definition 1 (*Implied semantics of contracts*)

$$v \in [\![t]\!]_{\mathrm{FF}} \ \Leftrightarrow\ \forall \xi, \xi', c\ .\ (\xi \neq \xi' \wedge \xi \notin c)\ \Rightarrow\ c[(\mathcal{W}_t^{\xi',\xi}\ v)] \text{ does not raise } \xi$$

The side conditions $\xi' \neq \xi$ and $\xi \notin c$ ("ξ does not occur in c") are necessary to avoid contexts that "fabricate evidence" against v. Without them there would be no value that satisfies any contract at all as it is very easy to produce, for any ξ, a context $c_\xi[\cdot]$ which always raises ξ no matter how its hole is filled.

It is important to show that the Findler-Felleisen algorithm is correct, where correctness means that when the algorithm blames a contract declaration, that contract declaration is actually wrong. $[\![\cdot]\!]_{\mathrm{FF}}$ as 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. In this paper we give such a semantics.

The structure of a non-compositional semantics like $[\![\cdot]\!]_{\mathrm{FF}}$ is difficult to understand. With just definition 1 at hand, an answer to the question "Does e satisfy t?" is not easy because it requires consideration of every possible context. Unfortunately, we cannot just ignore this problem and hope that intuition will help us out since in our experience most people's intuition actually disagrees with $[\![\cdot]\!]_{\mathrm{FF}}$:

Consider a predicate contract $\langle \mathbf{true} \rangle$ whose predicate is true for all values. The corresponding guard $\mathcal{W}^{\xi',\xi}_{\langle \mathbf{true} \rangle}$ is a no-op. One might expect *every* value to satisfy $\langle \mathbf{true} \rangle$ and, consequently, the identity function $\lambda x.x$ to satisfy $(\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \mathbf{true} \rangle$. (We write $\underline{\mathrm{int}}$ for the contract satisfied by all integer values.) But DrScheme disagrees! When evaluating the following example (translated to Scheme) in DrScheme, the identity f is blamed for violating $(\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \mathbf{true} \rangle$:

$$\begin{array}{l} \mathbf{let}\ f : (\underline{\mathrm{int}} \to \underline{\mathrm{int}}) \to \langle \mathbf{true} \rangle = \lambda y.y\ \mathbf{in} \\ \quad ((f\ \lambda z.z)\ \lambda w.w) \end{array} \qquad (\star\star)$$

For any t, if the identity function does not satisfy $t \to \langle \mathbf{true} \rangle$, then there must be at least one value satisfying t which does not satisfy $\langle \mathbf{true} \rangle$. How can this be? How can there be values not satisfying $\langle \mathbf{true} \rangle$ given that $\mathcal{W}^{\xi',\xi}_{\langle \mathbf{true} \rangle}$ does nothing? To answer this question it is useful to recall observation 2:

> "When a guarded expression evaluates to a value without raising a contract exception, it is still not guaranteed that the original value satisfied the corresponding contract."

Even $\langle \mathbf{true} \rangle$, the contract whose guard does nothing, can be violated by "self-incriminating" values—values that already happen to contain an exception that they are capable of triggering. Since the semantics always raises an exception for ill-formed applications (as opposed to getting "stuck"), an example for such a value is $\lambda x.(1\ 2)$ in:

$$\mathbf{let}\ g : \langle \mathbf{true} \rangle = \lambda x.(1\ 2)\ \mathbf{in}\ (g\ 0)$$

Moreover, such values are routinely constructed by the contract checking algorithm itself. For example, the guard $\mathcal{W}^{\xi',\xi}_{\underline{\mathrm{int}} \to \langle \mathbf{true} \rangle}$ around function f produces a function value f' defined by

$$\begin{array}{rcl} (f'\ y) & = & (\mathcal{W}^{\xi',\xi}_{\langle \mathbf{true} \rangle}\ (f\ (\mathcal{W}^{\xi,\xi'}_{\underline{\mathrm{int}}}\ y))) \\ & = & (f\ (\mathcal{W}^{\xi,\xi'}_{\underline{\mathrm{int}}}\ y)) \end{array}$$

Here ξ occurs in f', but it cannot be triggered since only function contracts have non-trivial preconditions, and $\underline{\text{int}}$ is not a function contract. Using a function contract, e.g., $\underline{\text{int}} \to \underline{\text{int}}$ in place of $\underline{\text{int}}$ changes this situation. Function value g' obtained from guarding an identity function $g = \lambda x.x$ with $\mathcal{W}^{\xi',\xi}_{(\underline{\text{int}}\to\underline{\text{int}})\to\langle\mathbf{true}\rangle}$ is defined by:

$$\begin{aligned}(g'\ y) &= (\mathcal{W}^{\xi',\xi}_{\langle\mathbf{true}\rangle}\ (g\ (\mathcal{W}^{\xi,\xi'}_{\underline{\text{int}}\to\underline{\text{int}}}\ y))) \\ &= (g\ (\mathcal{W}^{\xi,\xi'}_{\underline{\text{int}}\to\underline{\text{int}}}\ y)) \\ &= (\mathcal{W}^{\xi,\xi'}_{\underline{\text{int}}\to\underline{\text{int}}}\ y)\end{aligned}$$

Applying g' to another identity function h gives an h' defined by:

$$\begin{aligned}(h'\ z) &= ((g'\ h)\ z) \\ &= ((\mathcal{W}^{\xi,\xi'}_{\underline{\text{int}}\to\underline{\text{int}}}\ h)\ z) \\ &= (\mathcal{W}^{\xi,\xi'}_{\underline{\text{int}}}\ (\mathcal{W}^{\xi',\xi}_{\underline{\text{int}}}\ z))\end{aligned}$$

Indeed, here it is quite easy to trigger ξ simply by applying h' to a non-integer. By no coincidence, this is exactly the construction of our "counterexample" ($\star\star$).

In our semantics of contracts, $\langle\mathbf{true}\rangle$ is interpreted as the set of *safe* values (see Section 2.9). As we will see, safe values are precisely those that are incapable of triggering any of their embedded exceptions. Since not every value satisfying $\underline{\text{int}} \to \underline{\text{int}}$ is safe[2], this interpretation supports and justifies DrScheme's claim that the identity violates $(\underline{\text{int}} \to \underline{\text{int}}) \to \langle\mathbf{true}\rangle$. As a non-trivial theorem we prove that the Findler-Felleisen algorithm is sound and complete with respect to this semantics. Soundness and completeness together mean that the semantics is equivalent to $[\![\cdot]\!]_{\text{FF}}$.

Let us now come back to programs of the form ($\star$):

$$\begin{array}{l}\mathbf{let}\ x_1 : t_1 = e_1\ \mathbf{in}\ \ldots \\ \quad\mathbf{let}\ x_n : t_n = e_n\ \mathbf{in} \\ \quad\quad x_n\end{array}$$

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 its 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 $[\![t_i]\!]$. (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.) Soundness relative to $[\![\cdot]\!]$ can be stated as $\forall t.[\![t]\!] \subseteq [\![t]\!]_{\text{FF}}$.

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 $v_1, \ldots, v_{i-1}$

[2] For example, we have already seen that the identity function guarded with $\mathcal{W}^{\xi',\xi}_{\underline{\text{int}}\to\underline{\text{int}}}$ is not safe although it does satisfy $\underline{\text{int}} \to \underline{\text{int}}$.

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

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

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 $[\![\cdot]\!] = [\![\cdot]\!]_{\mathrm{FF}}$.

Another way of stating completeness is to say that contract violation is a recursively enumerable property.[3] By Rice's theorem there is no hope for it to be decidable. On the flip side, completeness implies that contract *satisfaction* is not even recursively enumerable: we already know that it is undecidable, and it is the complement of contract violation.

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. (This semantics, in particular the way it handles contract guards, is—*mutatis mutandis*—the same as Findler and Felleisen's contract checking algorithm.) 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 complete 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.

Section 6 adds a recursion operator to the contract language and provides an operational semantics for it. It then accounts for the resulting changes by formulating an indexed model of contracts which is based on previous work on models for recursive types (Appel & McAllester, 2001). The section also gives new proofs for many of the lemmas since the original ones no longer work in this setting. Recursive types have various practical applications, for example, the encoding of object types (Bruce *et al.*, 1997), so it is to hope that recursive contracts can play a similar role. Yet more motivation for considering them stems from the observation that they provide another angle from which to approach and understand the notion of safety that is so central to our proofs and our results.

In Section 7, after summarizing our results, we conclude by speculating on the potential use of contracts for static verification.

[3] The proof for this proceeds by considering (for a given contract and its guard) all possible expressions and all possible contexts and letting them all run "in parallel" using the standard *dovetailing* construction explained in many textbooks (Hartley Rogers, 1987).

$$
\begin{array}{rcl}
x, y, \ldots & : & \text{variables} \\
\underline{i}, \underline{j}, \ldots & : & \text{numeric literals} \\
+ \mid - \mid < \mid \leq \mid \ldots & : & \text{function symbols}
\end{array}
$$

elements common to both external and internal languages

$$
\begin{array}{rcll}
t^{\mathrm{e}} & ::= & \underline{\mathrm{int}} \mid \underline{\mathrm{safe}} \mid t_1^{\mathrm{e}} \xrightarrow{x} t_2^{\mathrm{e}} \mid \langle t^{\mathrm{e}} \mid \lambda x.\rho^{\mathrm{e}} \rangle & \\
\rho^{\mathrm{e}} & ::= & \rho_0^{\mathrm{e}} \mid (\rho^{\mathrm{e}}\ x) & \\
\rho_0^{\mathrm{e}} & ::= & e^{\mathrm{e}} & (e^{\mathrm{e}}\ \textit{closed}) \\
 & & & \\
e^{\mathrm{e}} & ::= & x \mid \underline{i} \mid \lambda x.e^{\mathrm{e}} \mid (e_1^{\mathrm{e}}\ e_2^{\mathrm{e}}) \mid & \\
 & & f(e_1^{\mathrm{e}}, \ldots, e_k^{\mathrm{e}}) & \\
 & & & \\
p & ::= & x \mid \mathbf{let}\ x : t^{\mathrm{e}} = e^{\mathrm{e}}\ \mathbf{in}\ p &
\end{array}
$$

Fig. 1. External language

$$
\begin{array}{rcll}
t & ::= & \underline{\mathrm{int}} \mid \underline{\mathrm{safe}} \mid t_1 \xrightarrow{x} t_2 \mid \langle t \mid \lambda x.\rho \rangle & \\
\rho & ::= & \rho_0 \mid (\rho\ x)_{\bot} & \\
\rho_0 & ::= & e & (e\ \textit{closed}) \\
\xi & ::= & \bot \mid \top^0 \mid \top^1 \mid \top^2 \mid \ldots & \\
e & ::= & x \mid \underline{i} \mid \lambda x.e \mid (e_1\ e_2)_{\xi} \mid & \\
 & & f_{\xi}(e_1, \ldots, e_k) \mid & \\
 & & (\mathcal{W}_t^{\xi', \xi}\ e) \mid e_1 ?_{\xi} e_2 &
\end{array}
$$

Fig. 2. Internal language

2 The formal setting

We consider programs at two different language levels: an external and an internal one. The former can be thought of as a syntactically-sugared refinement of the latter. (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 A briefly touches upon the likely interaction between static types and contracts. In general it suffices to assume a dynamically typed setting.)

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 $x, y, \ldots$, λ-abstractions $\lambda x.e$, and applications $(e_1\ e_2)$. For simplicity we restrict ourselves to integer constants $\underline{0}, \underline{1}, \ldots$ 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: $\underline{1} = \mathbf{true}$, *everything else* $= \mathbf{false}$.

Either language makes use of a sub-language of contract expressions consisting of $\underline{\mathrm{int}}$ (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 $\underline{\mathrm{safe}}$ of safe values; and restricted predicate contracts $\langle t \mid \phi \rangle$ (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 $\langle \underline{\mathrm{safe}} \mid \phi \rangle$. (The phrase "operationally equivalent" refers to the respective operational semantics of $\mathcal{W}_{\langle \phi \rangle}$ and $\mathcal{W}_{\langle \underline{\mathrm{safe}} \mid \phi \rangle}$. These are equivalent since $\mathcal{W}_{\underline{\mathrm{safe}}}$ is defined to be a no-op. From

this it follows that $[\![\langle \mathbf{true} \rangle]\!] = [\![\underline{\text{safe}}]\!]$. But despite its suggestive name, we have yet to show that interpreting $\underline{\text{safe}}$ as the set of safe values is sound.)

External: Programs in external form are closed terms

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

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., $e_{i+1}^{\mathrm{e}}, \ldots, e_n^{\mathrm{e}}, x_n$). 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. Example: Let $e_i^{\mathrm{e}} = f(x_j)$ and $t_i^{\mathrm{e}} = \langle \underline{\text{int}} \mid \lambda x.x < x_j \rangle$. We can eliminate the free occurrence of x_j in t_i^{e} by turning e_i^{e} into $\lambda y.f(y)$ with a corresponding contract $t_j \xrightarrow{y} \langle \underline{\text{int}} \mid \lambda x.x < y \rangle$, and then replacing every mention of x_i with $(x_i\ x_j)$.

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 $f_\xi(e_1, \ldots, e_k)$. A static type system can often eliminate the need for most language contracts (array out-of-bounds errors being a notable exception), but we do not make this assumption here.

There are also two additional expression forms:

- *Wrapped* expressions $(\mathcal{W}_t^{\xi',\xi}\ e)$ 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.

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

$$
\begin{array}{rcl}
\mathcal{C}(\underline{\mathrm{int}}) & = & \underline{\mathrm{int}} \\
\mathcal{C}(\underline{\mathrm{safe}}) & = & \underline{\mathrm{safe}} \\
\mathcal{C}(t_1^{\mathrm{e}} \xrightarrow{x} t_2^{\mathrm{e}}) & = & \mathcal{C}(t_1^{\mathrm{e}}) \xrightarrow{x} \mathcal{C}(t_2^{\mathrm{e}}) \\
\mathcal{C}(\langle t^{\mathrm{e}} \mid \lambda x.e^{\mathrm{e}}\rangle) & = & \langle \mathcal{C}(t^{\mathrm{e}}) \mid \lambda x.\mathcal{C}_\perp(e^{\mathrm{e}};\emptyset)\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}(x;\Gamma) & = & (\mathcal{W}_t^{\perp,\xi}\ x) \quad \textit{where } \Gamma(x) = (\xi, t) \\
\mathcal{C}(\textbf{let } x : t^{\mathrm{e}} = e_1^{\mathrm{e}} \textbf{ in } e_2^{\mathrm{e}};\Gamma) & = & ((\lambda x.e_2)\ e_1)_\perp \\
\multicolumn{3}{c}{\textit{where } e_1 = \mathcal{C}_{\top^i}(e_1^{\mathrm{e}};\Gamma);\ \ e_2 = \mathcal{C}(e_2^{\mathrm{e}};\Gamma, x \mapsto (\top^i, \mathcal{C}(t^{\mathrm{e}})));} \\
\multicolumn{3}{c}{\textit{i uniquely identifies the module named x}}
\end{array}
$$

$$
\begin{array}{rcl}
\Gamma & \in & \mathrm{Var} \xrightarrow{\mathrm{fin}} X \times T \\
\mathrm{Var} - \text{variables } x, y, \ldots & & X - \text{exceptions } \xi \quad T - \text{internal types } t
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{dom}(\Gamma\mid_{\neq x}) & = & \mathrm{dom}(\Gamma) \setminus \{x\} \\
\Gamma\mid_{\neq x}(y) & = & \Gamma(y) \qquad \forall y \neq x
\end{array}
\qquad
\begin{array}{rcll}
\mathrm{dom}(\Gamma, x \mapsto (\xi, t)) & = & \mathrm{dom}(\Gamma) \cup \{x\} & \\
(\Gamma, x \mapsto (\xi, t))(x) & = & (\xi, t) & \\
(\Gamma, x \mapsto (\xi, t))(y) & = & \Gamma(y) & \forall y \neq x
\end{array}
$$

Fig. 3. Translation from external to internal language.

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—trying to apply an integer, passing a non-integer or the wrong number of arguments to one of the built-in operations. (One can think of this as having contract wrappers on those primitive operations and even implement it that way).

$$
\begin{array}{rcl}
v, w, \ldots & ::= & \underline{i} \mid \lambda x.e \\
r & ::= & v \mid \xi \\
c_{\mathrm{e}} & ::= & \{\cdot\} \mid (c_{\mathrm{e}}\ e)_\xi \mid (v\ c_{\mathrm{e}})_\xi \mid f_\xi(v_1, \ldots, v_{i-1}, c_{\mathrm{e}}, e_{i+1}, \ldots, e_k) \mid (\mathcal{W}_t^{\xi',\xi}\ c_{\mathrm{e}}) \mid \\
 & & c_{\mathrm{e}}?_\xi e \mid v?_\xi c_{\mathrm{e}} \\
c & ::= & [\cdot] \mid \lambda x.c \mid (c\ e)_\xi \mid (e\ c)_\xi \mid f_\xi(e_1, \ldots, e_{i-1}, c, e_{i+1}, \ldots, e_k) \mid (\mathcal{W}_t^{\xi',\xi}\ c) \mid \\
 & & c?_\xi e \mid e?_\xi c
\end{array}
$$

Fig. 4. Semantic domains for values (v), results (r), evaluation contexts (c_{e}), and contexts (c). As a notational simplification, we will sometimes drop the subscript from evaluation contexts.

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

1. in $\mathcal{W}_{t_i}^{\top^k,\top^i}$ which is wrapped around uses of x_i in e_k^{e} for $k > i$,
2. in $\mathcal{W}_{t_j}^{\top^i,\top^j}$ which is wrapped around uses of x_j within e_i^{e} for $j < i$,
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: $\mathcal{C}_\xi(e^{\mathrm{e}}; \Gamma)$ 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; $\mathcal{C}(t^{\mathrm{e}})$ translates external contracts to internal ones; $\mathcal{C}(p; \Gamma)$ 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 $\mathcal{C}(p; \emptyset)$.

The statement of our central lemma (Lemma 7) requires that predicates within contracts do not raise contract exceptions of their own. This property, formally captured by the notion of *safe contracts*, while not decidable in general, is guaranteed by the way the operational semantics substitutes guarded values into contracts. This relies on the fact that $\mathcal{C}(\cdot)$ never inserts wrapper expressions into the code of contract predicates while artificially using $\bot$ for all language contracts. (This means that the program will be sent into an infinite loop—as opposed to having it raise an unaccounted-for contract exception—should a contract predicate "misbehave.") Thus, contracts start out safe and then stay safe during evaluation.

In a practical implementation it makes sense to use a separate $\top^{\mathrm{contract}}$ instead of $\bot$ for reporting contract violations caused by predicate code, effectively putting contracts on these contract predicates. To account for $\top^{\mathrm{contract}}$, 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 use *evaluation contexts* (Felleisen & Hieb, 1992) to specify the operational semantics of the internal language (see Figures 4 and 5). Every closed expression e

$$\frac{e = c_{\mathrm{e}}\{e'\} \qquad e' \hookrightarrow e'' \qquad c_{\mathrm{e}}[e'']\Downarrow_n v}{e\Downarrow_{n+1} v}$$

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

$$\begin{array}{rcl}
v & \Downarrow_0 & v \\
c_{\mathrm{e}}\{(\underline{i}\ v)_{\top^j}\} & \Downarrow_0 & \top^j \\
c_{\mathrm{e}}\{f_{\top^j}(v_1,\ldots,\lambda x.e,\ldots,v_k)\} & \Downarrow_0 & \top^j \\
c_{\mathrm{e}}\{(\mathcal{W}^{\xi,\top^j}_{\underline{\mathrm{int}}}\ \lambda x.e)\} & \Downarrow_0 & \top^j \\
c_{\mathrm{e}}\{(\mathcal{W}^{\xi,\top^j}_{t_1 \overset{x}{\to} t_2}\ \underline{i})\} & \Downarrow_0 & \top^j \\
c_{\mathrm{e}}\{v?_{\top^j} v'\} & \Downarrow_0 & \top^j \quad ; v \neq \underline{1}
\end{array}
\qquad
\begin{array}{rcl}
(\underline{i}\ v)_\perp & \hookrightarrow & \Omega \\
f_\perp(v_1,\ldots,\lambda x.e,\ldots,v_k) & \hookrightarrow & \Omega \\
(\mathcal{W}^{\xi,\perp}_{\underline{\mathrm{int}}}\ \lambda x.e) & \hookrightarrow & \Omega \\
(\mathcal{W}^{\xi,\perp}_{t_1 \overset{x}{\to} t_2}\ \underline{i}) & \hookrightarrow & \Omega \\
v?_\perp v' & \hookrightarrow & \Omega \quad ; v \neq \underline{1}
\end{array}$$

where $\Omega \equiv ((\lambda x.(x\ x)_\perp)\ \lambda x.(x\ x)_\perp)_\perp$

Fig. 5. Operational semantics of the internal language.

that is not a value v has a unique decomposition into an evaluation context c_{e} and a current β_v-redex e'; we write $e = c_{\mathrm{e}}\{e'\}$ 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.[4] The meaning of built-in primitives is assumed to be given by the semantic function $\mathcal{A}$.

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 a value 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:

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

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.

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

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 $(\mathcal{W}_t^{\xi',\xi}\ v)$. These rules are directed by the syntax of the contract t. If t is $\underline{\text{safe}}$, then the wrapper acts as an identity function; if t is $\underline{\text{int}}$, 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 (†) 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.[5]

If t is a restricted predicate contract $\langle t' \mid \phi \rangle$, then v is wrapped using t' and checked for satisfying the predicate ϕ. Substitution of values into predicate code happens when such a predicate is applied or when a value is substituted into a dependent type. Notice that we never substitute *arbitrary* values into predicate code. The value v to be substituted is always guarded, and the context exception on the guard is always $\bot$. This trick is what guarantees that safe contracts stay safe during the course of evaluation. As hinted in Section 2.2, a practical implementation should use some $\top^{\text{contract}}$ instead of $\bot$ in order to be able to help track down predicate code that is not behaving correctly.

We give an operational semantics to external programs by way of their translation into the internal language:

$$[\![p]\!]^{\text{e}} = [\![\mathcal{C}(p;\emptyset)]\!]$$

In the following examples, it is instructive to verify how the rule marked (†) in Figure 5 produces the results in cases (2) and (3):

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

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

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

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

[5] Contract wrappers can be viewed as retractions. Unfortunately, the corresponding retracts do not have the right properties to be used for interpreting contracts (although we have pursued this direction elsewhere (Findler *et al.*, 2004)). For example, we want $\lambda x.x$ to satisfy $\underline{\text{int}} \to \underline{\text{int}}$, but no term equivalent to $\lambda x.x$ is in the image of $\mathcal{W}^{\xi,\xi'}_{\underline{\text{int}}\to\underline{\text{int}}}$.

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

Fig. 6. Semantics of contracts.

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 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' (or $t \leq t'$ for t, t') if e' (or t') can be obtained from e (or t) by replacing zero or more occurrences of $\top^i$ (for any i) 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 $\top^i$ (for all i) with $\bot$ (implying $\lfloor e \rfloor \leq e$).

Lemma 1
If $e' \leq e$ then the following is true:

$$
\begin{array}{rcl}
[\![e']\!] = \top^i & \Rightarrow & [\![e]\!] = \top^i \\
[\![e]\!] = \bot & \Rightarrow & [\![e']\!] = \bot \\
[\![e]\!] = \underline{i} & \Leftrightarrow & [\![e']\!] = \underline{i}
\end{array}
$$

Proof sketch
By showing that the rules of the operational semantics preserve the $\leq$ relation on terms until an exception is raised. $\square$

2.6 Semantic interpretation of contracts

The interpretation of a contract t is some set of values $[\![t]\!]$. 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 6 define $[\![t]\!]$ for contracts t. The semantics $[\![\cdot]\!]^{\mathrm{e}}$ for the external contract language is handled by viewing it as a refinement (i.e., a syntactically sugared subset) of the internal contract language. (This means that external contracts are interpreted as sets of internal values. See Appendix A 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. It is easy to check that $[\![t]\!]$ is closed under semantic equivalence.

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 $[\![\cdot]\!]_{\mathrm{FF}}$ (see definition 1). If e satisfies t under $[\![\cdot]\!]_{\mathrm{FF}}$ then $(\mathcal{W}_t^{\xi',\xi}\ e)$ should not raise ξ in any context 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. But membership in $[\![t_1 \to t_2]\!]$—as pointed out before—is 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 merely 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 $\neg((f\ v)\ :\ t_3)$. But once again, 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 detected. 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

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

in the program fragment:

$$\begin{array}{l}\textbf{let } x_1 : t_1 = \lambda i.\lambda k.k - i \textbf{ in}\\ \quad \textbf{let } x_2 : \underline{\mathrm{int}} = ((x_1\ \underline{4})\ \underline{3}) \textbf{ in}\\ \quad\quad x_2\end{array}$$

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 $x_1\ :\ t_1$.

Now take a look at this variant of the program:

$$\begin{array}{l}\textbf{let } x_1 : t_1 = \lambda i.\lambda k.i - k \textbf{ in}\\ \quad \textbf{let } x_2 : \underline{\mathrm{int}} = ((x_1\ \underline{3})\ \underline{4}) \textbf{ in}\\ \quad\quad x_2\end{array}$$

Here the body of x_1 produces a positive number, as promised, if the arguments satisfy their contracts. The order of the arguments, however, has been inverted so that they no longer satisfy x_1's contract. The contract checker now raises $\top^2$ because x_2 fails to meet the preconditions on 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

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

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 since, clearly, $x - 1$ is not non-negative for all integers x. But this failure to meet the precondition escapes discovery. The code of x_1 applies its argument only once—to the number 1, and $1 - 1 = 0$ is indeed non-negative, so the problem with the argument value is never witnessed. Not knowing about the argument's failure to meet the precondition, the Findler-Felleisen algorithm detects a violation of the range contract of x_1 (because $-1 \not\geq 0$), 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! Nevertheless, blaming x_1 is not wrong here since there exist other values, for example

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

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 $(\mathcal{W}_t^{\xi',\xi}\ e)$:

Lemma 2
Let $e' = (\mathcal{W}_t^{\xi',\xi}\ e)$. If $[\![c[e']]\!] = \underline{i}$ then $[\![c[e]]\!] = \underline{i}$. Also, if $[\![c[e]]\!] = \underline{i}$ and $[\![c[e']]\!]$ is a value, then $[\![c[e']]\!] = \underline{i}$.

Proof sketch
Using a bi-simulation between expressions that contain instances of $\mathcal{W}$ 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 in the soundness proof and in the interpretation of $\underline{\text{safe}}$—the contract whose guard is a no-op—formalizes the familiar practice of coding *as defensively as possible*. If a program is safe, then before attempting any operation that could "go wrong" it 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.[6] 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 2 (*Safety, take 1*)

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

Let $\mathbf{Safe}^{\text{syn}} = \{\lfloor v \rfloor \mid v \text{ is a value}\}$ be the set of *syntactically safe* values, i.e., values that do not contain syntactic occurrences of $\top^i$. From definition 2 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 3 (*Safety, take 2*)

$$\mathbf{Safe}_2 = \{v \mid \forall c.[\![\lfloor c \rfloor[v]]\!] \text{ is a value or } \bot\}$$

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 $\nu\mathcal{S}$ of the monotonic operator $\mathcal{S}$:

Definition 4 (*Safety, take 3*)

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

[6] 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.

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

Lemma 3

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

Proof
This follows from lemmas 4 and 5 (see below). □

Lemma 4
$\mathbf{Safe}_1 = \mathbf{Safe}_2$

Proof for $\subseteq$
By definition we have $v \cong \lfloor v \rfloor$, so (by the definition of semantic equality) 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 for $\supseteq$
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 $[\![c[\lfloor v \rfloor]]\!] = \top^i$.[7] But since c is the witnessing context for v and $\lfloor v \rfloor$ being different, the latter is impossible. This concludes the proof. □

Lemma 5
$\mathbf{Safe}_2 = \mathbf{Safe}_3$.

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

Proof for $\supseteq$
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 $\hat{c}_\mathrm{e}$ and corresponding expression $\hat{e}$ such that $\hat{c}_\mathrm{e}\{\hat{e}\} = \lfloor c \rfloor [v]$ where $\hat{e}$ is the next β_v-reduction to do in $\lfloor c \rfloor [v]$ (Felleisen & Hieb, 1992). 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 $\lfloor c' \rfloor [v'] \Downarrow_{n-1} \top^i$.

[7] Making this informal argument precise is not difficult but tedious.

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 $\hat{c}_\mathrm{e}$ 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 $\hat{c}_\mathrm{e}[d'] \Downarrow_{n-k-1} \top^i$, 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,$\hat{c}_\mathrm{e}$, and c_0 are syntactically safe. We know that $\hat{c}_\mathrm{e}[b[v'/x]] \Downarrow_{n-1} \top^i$. 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 $\hat{c}_\mathrm{e}[\lfloor c_1 \rfloor [c_0[v]]] \Downarrow_{n-1} \top^i$, which is the contradiction we are looking for since $\hat{c}_\mathrm{e}[c_1[c_0[\cdot]]]$ is a syntactically safe context.

□

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 $\hat{\mathcal{S}}$, defined as

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

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 6
$e, e'\ :\ \underline{\mathrm{safe}} \Rightarrow (e\ e')_\perp\ :\ \underline{\mathrm{safe}}$

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

3 Soundness and completeness

We would like to show that $[\![\cdot]\!]_\mathrm{FF}$ 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., $[\![t]\!] \subseteq [\![t]\!]_\mathrm{FF}$. This means that any difference between $[\![t]\!]_\mathrm{FF}$ and $[\![t]\!]$ can always be explained by non-terminating predicate code.[8] In any case, blame

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

assignment is sound as every blame is justified by a corresponding contract violation:

Theorem 1 ($[\![t]\!] \subseteq [\![t]\!]_{\mathrm{FF}}$)
If

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

then

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

such that

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

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

$$\mathcal{C}_{\top^i}(v_j^{\mathrm{e}}; \emptyset) \cong v_j$$

i.e.

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

Furthermore, we can get a completeness result if all base predicates ρ_0 are assumed to be total predicates. (For example, we could restrict ρ_0 to 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 expressions.) In this case every contract violation has the potential for causing corresponding blame:

Theorem 2 ($[\![t]\!] \supseteq [\![t]\!]_{\mathrm{FF}}$)
If all base predicates ρ_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

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

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

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

(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 7). 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 $\mathrm{st}(\epsilon; t)$ where ϵ denotes the empty sequence of variables.

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

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

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

Fig. 7. Definition of the set T_{safe} of safe contracts.

By slight abuse of notation, let's write $\mathcal{W}_t^{\xi',\xi}$ for $\lambda x.(\mathcal{W}_t^{\xi',\xi}\ x)$ and $\mathcal{W}_{t_1}^{\xi_1',\xi_1} \circ \mathcal{W}_{t_2}^{\xi_2',\xi_2}$ for $\lambda x.(\mathcal{W}_{t_1}^{\xi_1',\xi_1}\ (\mathcal{W}_{t_2}^{\xi_2',\xi_2}\ x))$.

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

$$\mathcal{W}_t^{\xi_1,\xi_2} \circ \mathcal{W}_t^{\xi_3,\xi_4} = \mathcal{W}_t^{\xi_1,\xi_4}$$

Thus, wrappers $\mathcal{W}_t^{\top^i,\perp}$ and $\mathcal{W}_t^{\perp,\top^j}$ can be seen as two "halves" of $\mathcal{W}_t^{\top^i,\top^j}$. 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 7 (*Central lemma*)
For any ξ and any $t \in T_{\mathrm{safe}}$:

a. $v \ : \ t \Rightarrow (\mathcal{W}_t^{\perp,\xi}\ v) \ : \ \underline{\mathrm{safe}}$
b. $v \ : \ \underline{\mathrm{safe}} \Rightarrow (\mathcal{W}_t^{\xi,\perp}\ v) \ : \ t$

Once again abusing notation, we can render this as:

$$\begin{array}{lcl} \mathcal{W}_t^{\perp,\xi} & : & t \to \underline{\mathrm{safe}} \\ \mathcal{W}_t^{\xi,\perp} & : & \underline{\mathrm{safe}} \to t \end{array}$$

3.2 Proof of soundness

The proof for Theorem 1 uses Lemma 7(b) to construct the required values $v_1, \ldots, v_{i-1}$ and then finishes by applying Lemma 7(a).

Proof of Theorem 1 (sketch)
First we define a substitution σ defined by the equation

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

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

Let e be the expression $(\mathcal{W}^{\bot,\top^n}_{\mathcal{C}(t^{\mathrm{e}}_n)}\ \sigma(x_n))$. 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

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

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

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

Pick v_k for $k = 1\ldots i-1$ to be $(\mathcal{W}^{\top^i,\bot}_{t_k}\ \lfloor\sigma(x_k)\rfloor)$. By Lemma 7(b) we find $v_k \in [\![t^{\mathrm{e}}_k]\!]^{\mathrm{e}}$ as required. Each v_k has a semantically equivalent external version v^{e}_k (see Appendix A). Substituting v_k into the above equation yields

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

which means that $e_i[v_k/x_k]_{k=1\ldots i-1}\ :\ [\![t^{\mathrm{e}}_i]\!]^{\mathrm{e}}$ would contradict Lemma 7(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 ρ_0 in a predicate contract

$$\langle t \mid \lambda x_n.(\ldots(\rho_0\ x_1)_\bot\ \ldots x_n)_\bot\rangle$$

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:

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

4.1 A simple proof of the central lemma

We now give a proof of Lemma 7 under the assumption of totality for predicates:

Proof of 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

$$(\mathcal{W}^{\xi,\bot}_{t_1}\ w)\ :\ t_1,$$

so using the contract on $\lambda x.e$ we get

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

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 $[\![t_2[(\mathcal{W}^{\xi,\perp}_{t_1}\ w)/z]]\!]$. 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 $(\mathcal{W}^{\xi,\perp}_{t_1}\ w)$. Using the induction hypothesis (part a.) we find that

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

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

b. Consider any $w\ :\ t_1$: By induction hypothesis (part a.) we know that $(\mathcal{W}^{\perp,\xi}_{t_1}\ w)$ is safe, so by Lemma 6 we find $((\lambda x.e)\ (\mathcal{W}^{\perp,\xi}_{t_1}\ w))_\perp$ to be safe as well. By induction hypothesis (part b.) this means that:

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

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

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

a. Since $v \in [\![\langle t' \mid \lambda x.e\rangle]\!]$ we also have $v \in [\![t']\!]$ and $[\![((\lambda x.e)\ v)_\perp]\!] = \underline{1}$. But $(\mathcal{W}^{\perp,\xi}_{\langle t' | \lambda x.e\rangle}\ v)$ makes a transition to

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

and finally evaluates to $[\![\mathcal{W}^{\perp,\xi}_{t'} v]\!]$, 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

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

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

□

4.2 Completeness

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

Lemma 8
If $v\ :\ t'$ but $\neg(c[v]\ :\ t)$, then $\neg(c[(\mathcal{W}^{\top^i,\top^j}_{t'}\ v)]\ :\ t)$

Proof
By induction on the structure of t:

int**:** Because of totality the extra wrapper cannot cause non-termination. But by Lemma 2, if $c[(\mathcal{W}_{t'}^{\top^i,\top^j}\ v)]$ 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

$$[\![\lfloor c' \rfloor [c[(\mathcal{W}_{t'}^{\top^i,\top^j}\ v)]]]\!] \text{ is a value or } \bot.$$

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'.)

$\langle t \mid \phi \rangle$**:** 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 $c'[(\mathcal{W}_{t'}^{\top^i,\top^j}\ v)][w/x] \ : \ t_2[w/x]$. Consider $c'' = c'[w/x]$ and use the induction hypothesis with t_2.

□

We now have the tools for proving completeness (Theorem 2):

Proof of Theorem 2

To complete the proof of Theorem 2, recall that we have

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

such that $\mathcal{C}_{\top^i}(e^{\mathrm{e}};\emptyset)[v_j/x_j]_{j=1,\ldots,i-1}$ does not satisfy $\mathcal{C}(t^{\mathrm{e}})$. We pick $e_1^{\mathrm{e}}, \ldots, e_{i-1}^{\mathrm{e}}$ equivalent to $\lfloor v_1 \rfloor \ldots, \lfloor v_{i-1} \rfloor$. (See Appendix A for how this can be done.) Now consider the **let**-expansion of e^{e} which is equivalent to

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

It is easy to see that $\lfloor v_j \rfloor \ : \ t_j$, so according to Lemma 8 this expression, let's call it $\hat{e}$, does not satisfy $\mathcal{C}(t_i^{\mathrm{e}})$.

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

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 $[\cdot]$.

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.

$\langle t \mid \phi \rangle$**:** 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 8 this means that

$$((\lambda y.e')\ (\mathcal{W}_{t_1}^{\top^i,\top^{i+1}}\ v))_\perp$$

also violates $t_2[v/x]$. Using the induction hypothesis for this contract-expression combination, we pick a c' in such a way that

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

raises $\top^i$. But then $c'[((\mathcal{W}_{t_1 \xrightarrow{x} t_2}^{\top^{i+1},\top^i}\ \lambda y.e')\ \lfloor v \rfloor)_\perp]$ 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 ($\mathcal{C}(\cdot)$, 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

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

are true. To prove Lemma 7 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 8. 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 $\mathcal{W}_{\hat{t}}^{\perp,\perp}$ where $\hat{t} \in T_{\mathrm{safe}}$.

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

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

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

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

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

$$\frac{t_1' \preceq t_1 \quad t_2' \preceq t_2}{t_1' \xrightarrow{x} t_2' \preceq t_1 \xrightarrow{x} t_2}$$

Fig. 8. A partial order on expressions and contracts.

Lemma 9
If $e' \preceq e$ and $[\![e']\!] = \underline{i}$ for some number literal $\underline{i}$, then $[\![e]\!] = \underline{i}$. Also, if $[\![e]\!] = \underline{i}$ and $[\![e']\!]$ is a value, then $[\![e']\!] = \underline{i}$.

The proof for this proceeds like that for Lemma 2 (using a bi-simulation 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 7:

Lemma 10 (*Stronger version of central lemma*)
For any ξ, any $t' \in T_{\text{safe}}$, and t such that $t' \preceq t$

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

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 $t_2'[(\mathcal{W}_{t_1'}^{\perp,\perp}\ v)/z] \preceq t_2[(\mathcal{W}_{t_1'}^{\xi,\perp}\ v)/z]$ we need to show that the result of applying $(\mathcal{W}_{t_1' \xrightarrow{z} t_2'}^{\perp,\xi}\ \lambda x.e)$ to a safe value v is safe. This can be seen as follows:

$$\underbrace{(\mathcal{W}^{\perp,\xi}_{t_2'[(\mathcal{W}^{\perp,\perp}_{t_1'}\ v)/z]}\ \underbrace{((\lambda x.e)\ (\mathcal{W}^{\xi,\perp}_{t_1'}\ \overbrace{v}^{\in \mathbf{Safe}}))}_{t_2[(\mathcal{W}^{\xi,\perp}_{t_1'}\ v)/z]}}_{\in \mathbf{Safe}} \quad \overbrace{}^{t_1}$$

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.), $(\mathcal{W}^{\perp,\xi}_{t'_1}\ v)$ is safe, so by definition of $\mathbf{Safe}_1$ it is, in fact, equal to $(\mathcal{W}^{\perp,\perp}_{t'_1}\ v)$, which means that we have:

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

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

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

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}.

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

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

$$\underbrace{(\phi'\ \underbrace{(\mathcal{W}^{\perp,\xi}_{t'}\ \overbrace{v}^{t})}_{\in\mathbf{Safe}})}_{\in\mathbf{Safe}} ?_\xi \underbrace{(\mathcal{W}^{\perp,\xi}_{t'}\ \overbrace{v}^{t})}_{\in\mathbf{Safe}}$$

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 $(\phi'\ (\mathcal{W}^{\perp,\xi}_{t'}\ v))$, which by the properties of safety is the same as $(\phi'\ (\mathcal{W}^{\perp,\perp}_{t'}\ v))$, yielding a proper value other than $\underline{1}$. By Lemma 9 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 $(\mathcal{W}^{\xi,\perp}_{\langle t' \mid \phi' \rangle}\ v)$ which expands into

$$\underbrace{(\phi'\ (\mathcal{W}^{\perp,\perp}_{t'}\ \overbrace{\overbrace{v}^{\in\mathbf{Safe}}}^{\in\mathbf{Safe}}))}_{\in\mathbf{Safe}} ?_\perp (\mathcal{W}^{\xi,\perp}_{t'}\ v)$$

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

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

and the result is $(\mathcal{W}^{\xi,\perp}_{t'}\ v)$. But in that case, since $\phi' \preceq \phi$ by Lemma 9 we also have $[\![(\phi\ (\mathcal{W}^{\xi,\perp}_{t'}\ v))]\!] = \underline{1}$, which means that the value satisfies $\langle t \mid \phi \rangle$.

Notice that the proof for $\langle t \mid \phi \rangle$ would not go through had the evaluation rule for $\mathcal{W}_{\langle t|\phi\rangle}$ failed to place t-guards on either side of the ? operator (see Figure 5).

□

Lemma 7 is implied by Lemma 10. 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. We also add a form of sum contracts $t_1 \vee_\chi t_2$ where the sets $[\![t_1]\!]$ and $[\![t_2]\!]$ are recursively separated by the computable total predicate χ on values. The predicate is false for all of $[\![t_1]\!]$ and true for all of $[\![t_2]\!]$.[10] We will sometimes simply write $t_1 \vee t_2$ for $t_1 \vee_\chi t_2$, relying on the existence of a suitable separation predicate without actually naming it. Finally, for technical convenience there is also a contract bot which is satisfied by no value.

The changes to language and semantics that account for recursive contracts are given in Figure 9. Also, in the following discussion we will silently drop any mention of safe. Its place will be taken by rsafe, a contract that can be synthesized using other contract constructors, and which, therefore, does not need to be added separately to the contract language.

Because of the rule in the operational semantics that identifies $\mu\alpha.t$ with $t[\mu\alpha.t/\alpha]$, structural induction on contract expression breaks down in the presence of recursive contracts. If we could restrict α to only occur in positive (covariant) positions within t, 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 α occurs in negative positions. 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 gives a different (but consistent!) view on the problem of how to interpret Findler and Felleisen's original unrestricted predicate contracts.

[10] Such sums are sometimes called *tidy sums*.

$$
\begin{array}{rcll}
\alpha \mid \beta \mid \ldots & : & \text{type variables} & \\
\\
t^{\mathrm{e}} & ::= & \ldots \mid \alpha \mid \mu\alpha.t^{\mathrm{e}} \mid t^{\mathrm{e}} \vee_\chi t^{\mathrm{e}} \mid \underline{\mathrm{bot}} & \\
T & ::= & \ldots \mid \alpha \mid \mu\alpha.t \mid t \vee_\chi t \mid \underline{\mathrm{bot}} & \\
\\
\mathcal{C}(\underline{\mathrm{bot}}) & = & \underline{\mathrm{bot}} & \\
\mathcal{C}(\alpha) & = & \alpha & \\
\mathcal{C}(\mu\alpha.t^{\mathrm{e}}) & = & \mu\alpha.\mathcal{C}(t^{\mathrm{e}}) & \\
\mathcal{C}(t_1^{\mathrm{e}} \vee_\chi t_2^{\mathrm{e}}) & = & \mathcal{C}(t_1^{\mathrm{e}}) \vee_\chi \mathcal{C}(t_2^{\mathrm{e}}) & \\
\\
(\mathcal{W}^{\xi,\top^j}_{\underline{\mathrm{bot}}}\ v) & \Downarrow_0 & \top^j & \\
(\mathcal{W}^{\xi,\bot}_{\underline{\mathrm{bot}}}\ v) & \hookrightarrow & \Omega & \\
(\mathcal{W}^{\xi',\xi}_{\mu\alpha.t}\ v) & \hookrightarrow & (\mathcal{W}^{\xi',\xi}_{t[\mu\alpha.t/\alpha]}\ v) & \\
(\mathcal{W}^{\xi',\xi}_{t_1 \vee_\chi t_2}\ v) & \hookrightarrow & \left\{ \begin{array}{lll} (\mathcal{W}^{\xi',\xi}_{t_1}\ v) & : & \text{if } \chi(v) \neq \underline{1} \\ (\mathcal{W}^{\xi',\xi}_{t_2}\ v) & : & \text{if } \chi(v) = \underline{1} \end{array} \right. &
\end{array}
$$

Fig. 9. Recursion-related modifications to external and internal contract language, to the translation between them, and to the operational semantics.

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 possible to adapt an indexed model of recursive types (Appel & McAllester, 2001) to the case of recursive contracts and to modify proofs accordingly.

In the indexed model, a contract t is interpreted as a set $[\![t]\!]_{\mathrm{idx}}$ 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. Each $[\![t]\!]_{\mathrm{idx}}$ is closed under decreasing index, i.e., $\langle k, v\rangle \in [\![t]\!]_{\mathrm{idx}} \wedge 0 \leq j < k \Rightarrow \langle j, v\rangle \in [\![t]\!]_{\mathrm{idx}}$.

An index-free interpretation $[\![t]\!]_\infty$ of contracts can then be recovered as:

$$[\![t]\!]_\infty = \bigcap_k \{v \mid \langle k, v\rangle \in [\![t]\!]_{\mathrm{idx}}\}$$

As we will see, $[\![\cdot]\!]_\infty$ coincides with our original $[\![\cdot]\!]$ for contracts that do not contain μ.

Along with the interpretation of contracts as sets of index-value pairs goes an indexed contract-satisfaction relation $e :_k t$, which is defined as:

$$e :_k t \Leftrightarrow \forall j. 0 \leq j < k \wedge e\Downarrow_j v \Rightarrow \langle k - j, v\rangle \in [\![t]\!]_{\mathrm{idx}}$$

The index-free version of this relation is then defined as:

$$e :_\infty t \Leftrightarrow \forall k \geq 0. e :_k t$$

To avoid additional complications arising from diverging predicates in predicate contracts, we now revert back to the totality assumption that we used earlier (see

$$
\begin{array}{rcl}
[\![\underline{\text{bot}}]\!]_{\text{idx}} & = & \{\} \\
[\![\underline{\text{int}}]\!]_{\text{idx}} & = & \{\langle k, \underline{i}\rangle \mid k \geq 0\} \\
[\![t_1 \xrightarrow{x} t_2]\!]_{\text{idx}} & = & \{\langle k, \lambda x.e\rangle \mid \forall j < k \forall v. \langle j, v\rangle \in [\![t_1]\!]_{\text{idx}} \Rightarrow e[v/x] :_j t_2[v/x]\} \\
[\![\langle t \mid \lambda x.e\rangle]\!]_{\text{idx}} & = & \{\langle k, v\rangle \mid \langle k, v\rangle \in [\![t]\!]_{\text{idx}} \wedge [\![e[v/x]]\!] = \underline{1}\} \\
[\![t_1 \vee_\chi t_2]\!]_{\text{idx}} & = & \{\langle k, v\rangle \mid \langle k, v\rangle \in [\![t_1]\!]_{\text{idx}} \wedge \chi(v) \neq \underline{1}\} \cup \\
 & & \{\langle k, v\rangle \mid \langle k, v\rangle \in [\![t_2]\!]_{\text{idx}} \wedge \chi(v) = \underline{1}\} \\
[\![\mu\alpha.t]\!]_{\text{idx}} & = & \{\langle k, v\rangle \mid \langle k, v\rangle \in [\![\text{unroll}(k+1, \alpha, t)]\!]_{\text{idx}}\} \\
 & \textit{where} & \text{unroll}(0, \alpha, t) = \underline{\text{bot}} \\
 & & \text{unroll}(i+1, \alpha, t) = t[\text{unroll}(i, \alpha, t)/\alpha]
\end{array}
$$

Fig. 10. The indexed model of recursive contracts.

Section 4). This assumption once again simplifies the operational semantics of contract guards for $t_1 \xrightarrow{x} t_2$, and we use it to construct a slightly simpler indexed model than otherwise possible.

A note of caution: Indexed models have the inherent disadvantage of being extremely sensitive to details of the underlying operational semantics in general and to the way steps are being counted in particular. As a consequence, $[\![t]\!]_{\text{idx}}$ is in general *not* closed under semantic equivalence, and two different ways of counting steps produces two different, incomparable sets $[\![t]\!]_{\text{idx}}$. In the limit, however, these differences vanish: $[\![\cdot]\!]_\infty$ does not suffer from such problems.

Like Appel and McAllester, we define $[\![t]\!]_{\text{idx}}$ by induction on *indices* and refer to them for the proof that the so-defined sets satisfy the required closure condition (closed under decreasing index) the we need.

The indexed model of contracts is shown in Figure 10. Notice that $[\![t_1 \vee_\chi t_2]\!]_{\text{idx}}$ is not simply the union of $[\![t_1]\!]_{\text{idx}}$ and $[\![t_2]\!]_{\text{idx}}$. Our construction throws out those pairs $\langle k, v\rangle$ which the separator χ classifies as belonging to t_1 (or t_2) when they do not look like a t_1- (or t_2-) value for at least k steps. (This detail only concerns values which ultimately belong to neither t_1 nor t_2. It will be important later when we prove the indexed version of the central lemma.)

All our type constructors are either *well-founded* or *nonexpansive.*[11] The following lemma is a consequence of this fact:

Lemma 11
Let $0 \leq j < i$. Then the following statement holds:

$$\langle j, v\rangle \in [\![\text{unroll}(i, \alpha, t)]\!]_{\text{idx}} \Leftrightarrow \langle j, v\rangle \in [\![\text{unroll}(j+1, \alpha, t)]\!]_{\text{idx}}$$

This says that a value v looks like a sufficiently precise approximation of a recursive type for some number of steps if and only if it also looks like an arbitrarily

[11] Let $\text{approx}(k, T) = \{\langle j, v\rangle \mid j < k \wedge \langle j, v\rangle \in T\}$. We say a type constructor f is well-founded whenever the functional F which maps $[\![t]\!]_{\text{idx}}$ to $[\![f(t)]\!]_{\text{idx}}$ has the property that for all t and all $k \geq 0$ we have: $\text{approx}(k+1, F([\![t]\!]_{\text{idx}})) = \text{approx}(k+1, F(\text{approx}(k, [\![t]\!])))$. Similarly, we call f non-expansive if $\text{approx}(k, F([\![t]\!]_{\text{idx}})) = \text{approx}(k, F(\text{approx}(k, [\![t]\!])))$.

more precise approximation of the same type for the same number of steps. We omit a detailed proof here and refer to the literature (Appel & McAllester, 2001).

Lemma 12

$$\langle k, n\rangle \in [\![t[\text{unroll}(k+1, \alpha, t)/\alpha]]\!]_{\text{idx}} \Leftrightarrow \langle k, n\rangle \in [\![t[\mu\alpha.t/\alpha]]\!]_{\text{idx}}$$

Proof
By definition of $[\![\cdot]\!]_{\text{idx}}$, the truth of the statement $\langle k, n\rangle \in [\![t[\text{unroll}(k+1, \alpha, t)/\alpha]]\!]_{\text{idx}}$ can be expressed as a function $\mathcal{F}_t$ on answers to questions of the form

$$\langle j, w\rangle \in [\![\text{unroll}(k+1, \alpha, t)]\!]_{\text{idx}}$$

where $0 \leq j \leq k$. Moreover, function $\mathcal{F}_t$ is completely determined by t.
By the same argument, the truth of the statement $\langle k, n\rangle \in [\![t[\mu\alpha.t/\alpha]]\!]_{\text{idx}}$ is expressible as the *same* function $\mathcal{F}_t$ applied to the answers to questions of the form

$$\langle j, w\rangle \in [\![\mu\alpha.t]\!]_{\text{idx}}$$

for the same set of pairs $\langle j, w\rangle$ as above. By definition, these questions reduce to:

$$\langle j, w\rangle \in [\![\text{unroll}(j+1, \alpha, t)]\!]_{\text{idx}}$$

According to Lemma 11, each argument to $\mathcal{F}_t$ in case of the first statement is equal to the corresponding argument to $\mathcal{F}_t$ in case of the second statement. □

For the purpose of comparison with $[\![\cdot]\!]_\infty$ we now extend the definition of $[\![\cdot]\!]$ to also handle sums and $\underline{\text{bot}}$:

$$\begin{aligned} [\![\underline{\text{bot}}]\!] &= \{\} \\ [\![t_1 \vee_\chi t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \end{aligned}$$

The following lemma states that, in the limit, the indexed model for non-recursive types coincides with our original model:

Lemma 13
For all t which do not contain μ we have $[\![t]\!] = [\![t]\!]_\infty$.

Proof
Straightforward by induction on the structure of t. (An inductive proof is fine since t does not contain recursion.) □

6.2 Indexed safety

We will now build up the necessary machinery to adapt our original proof of soundness to the indexed model. We use a modified version of the central lemma (Lemma 7) which in its formulation requires an indexed version of the concept of safety. It turns out that one of the major benefits of working with recursive contracts is that we can used them to characterize safety directly.

In our non-indexed analysis we spent significant effort on showing that (abusing notation)

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

where $\mathbf{Safe}^{\text{syn}} = \{\lfloor w \rfloor \mid w \text{ is a value}\}$ (see Lemmas 3 and 6). Here α occurs in positive positions only, which makes it possible to use co-induction. Using the machinery of recursive contracts we can avoid such detours and express safety directly as a contract:

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

We write $\underline{\text{rsafe}}_k$ for the k-approximation of $\underline{\text{rsafe}}$:

$$\underline{\text{rsafe}}_k = \text{unroll}(k+1, \alpha, \underline{\text{int}} \vee (\alpha \to \alpha))$$

Notice that, by definition, if $\langle k, v\rangle \in [\![\underline{\text{rsafe}}]\!]_{\text{idx}}$ then also $\langle k, v\rangle \in [\![\underline{\text{rsafe}}_k]\!]_{\text{idx}}$.

Lemma 14
$[\![\underline{\text{rsafe}}]\!]_\infty = \mathbf{Safe}$.

Proof
We show that $[\![\underline{\text{rsafe}}]\!]_\infty = \mathbf{Safe}_1$, using an indexed version of the $\cong$ relation:

$$e \cong_k e' \Leftrightarrow \forall c.c[e]\Downarrow_i v \wedge c[e']\Downarrow_{i'} v' \wedge v, v' \in [\![\underline{\text{int}}]\!] \cup X \wedge i, i' < k \Rightarrow v = v'$$

It suffices to show that $e \cong_k \lfloor e \rfloor \Leftrightarrow e :_k \underline{\text{rsafe}}$. We do this by simultaneous induction on k for both directions.

($\Rightarrow$) The statement is trivially true if e does not reduce to a value within k steps. Thus, we just need to focus on the case where e is already a value v, i.e., that $v \cong_k \lfloor v \rfloor$ implies $v :_k \underline{\text{rsafe}}$ which is equivalent to $\langle k, v\rangle \in [\![\underline{\text{rsafe}}_k]\!]_{\text{idx}}$. Indirect. Assume there is some w such that $\langle i, w\rangle \in [\![\underline{\text{rsafe}}_i]\!]_{\text{idx}}$ for some $i < k$ but $\neg((v\ w)_\perp :_i \underline{\text{rsafe}}_i)$ By induction hypothesis we know that $w \cong_i \lfloor w \rfloor$, so it must be that also $\neg((v\ \lfloor w \rfloor)_\perp :_i \underline{\text{rsafe}}_i)$ or otherwise w and $\lfloor w \rfloor$ were operationally distinguishable within i steps. We already have that $v \cong_i \lfloor v \rfloor$ since $i < k$. Therefore, it must be that $\neg((\lfloor v \rfloor\ \lfloor w \rfloor)_\perp :_i \underline{\text{rsafe}}_i)$, which contradicts the induction hypothesis.

($\Leftarrow$) As before, we only need to consider the case where e is some value v. We proceed indirectly and consider the smallest k such that $\langle k, v\rangle \in [\![\underline{\text{rsafe}}]\!]_{\text{idx}}$ but $\neg(v \cong_k \lfloor v \rfloor)$. Pick a context c which demonstrates semantic inequality in less than k steps. The only way this can happen is by having $c[v]\Downarrow_{k-1}\top$ while $c[\lfloor v \rfloor]$ does not evaluate to $\top$. In this case we also have that $\lfloor c \rfloor[v]\Downarrow_{k-1}\top$. We proceed by case analysis on the current redex in $\lfloor c \rfloor[v]$:

1. The current redex is $(v\ \lfloor w \rfloor)_\perp$ for some value w. By induction hypothesis we have $\langle k-1, \lfloor w \rfloor\rangle \in [\![\underline{\text{rsafe}}]\!]_{\text{idx}}$. Since the program must not loop forever, v has to be of the form $\lambda x.b$, implying that $\langle k, \lambda x.b\rangle \in [\![\underline{\text{rsafe}}_{k-1} \to \underline{\text{rsafe}}_{k-1}]\!]_{\text{idx}}$. Thus, the program can make one step by reducing the current redex to $b[\lfloor w \rfloor/x]$, and for this term we have $b[\lfloor w \rfloor/x] :_{k-1} \underline{\text{rsafe}}$. Plugging it back into the evaluation context yields a program that yields $\top$ in $k-2$ steps, so we have a contradiction to the assumption that k was minimal.
2. The current redex has the form $(\lambda x.b\ a)_\perp$ where v is a sub-term of a. The program can perform at least one reduction, after which in general there will exist more than one copy of v. Since the only occurrences of $\top$ are in these

v, we can replace all but one of them with $\lfloor v \rfloor$ and still have a program that reduces to $\top$ in $k-2$ steps. This again violates the assumption that k was chosen minimal.

3. In all other cases, the program can perform at least one step after which there still exists exactly one copy of v, so k could not have been minimal.

□

Notice that the definition of $\underline{\text{rsafe}}$ looks suspiciously like the equation often used to characterize the domain of untyped λ-terms (Scott, 1972). However, it should be noted that here it does not name 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.)

6.3 The indexed central lemma

We restate Lemma 7 as follows:

Lemma 15 (*Central lemma with recursive contracts*)
For any ξ, any $t \in T_{\text{safe}}$, and any $k \geq 0$:

a. $e :_k t \Rightarrow (\mathcal{W}_t^{\bot,\xi}\ e) :_k \underline{\text{rsafe}}$
b. $e :_k \underline{\text{rsafe}} \Rightarrow (\mathcal{W}_t^{\xi,\bot}\ e) :_k t$

Proof
The proof proceeds by simultaneous induction on k. Clearly, either statement of the lemma is true for $k = 0$. In the induction step we can assume e to be some value v, as otherwise the induction hypothesis applies trivially. Here is a complete description of the induction step:

a. We perform a case analysis on the outermost contract constructor:

   $\underline{\text{bot}}$**:** $[\![\underline{\text{bot}}]\!]_{\text{idx}}$ is empty, which means that no v exists.

   $\underline{\text{int}}$**:** $[\![\underline{\text{int}}]\!]_{\text{idx}}$ is a subset of $[\![\underline{\text{rsafe}}_{k-j}]\!]_{\text{idx}}$.

   $t_1 \vee_\chi t_2$**:** We consider the case that $\chi(v)$ evaluates to false. (The other case proceeds analogously, with t_1 and t_2 swapped.) We have $\langle k, v\rangle \in [\![t_1]\!]_{\text{idx}}$, so by downward closure $\langle k-1, v\rangle \in [\![t_1]\!]_{\text{idx}}$. But $(\mathcal{W}_{t_1 \vee_\chi t_2}^{\bot,\xi}\ v)$ makes one step to $(\mathcal{W}_{t_1}^{\bot,\xi}\ v)$, and by induction hypothesis we have $(\mathcal{W}_{t_1}^{\bot,\xi}\ v) :_{k-1} \underline{\text{rsafe}}$.

   $\langle t \mid \phi\rangle$**:** $(\mathcal{W}_{\langle t|\phi\rangle}^{\bot,\xi}\ v)$ makes one step yielding $\phi(v)?_\xi(\mathcal{W}_t^{\bot,\xi}\ v)$. We know that $\phi(v)$ evaluates to $\underline{1}$, so after $i \geq 1$ steps we have $(\mathcal{W}_t^{\bot,\xi}\ v)$. But by induction hypothesis and downward closure it is the case that $(\mathcal{W}_t^{\bot,\xi}\ v) :_{k-i} \underline{\text{rsafe}}$ (unless $k - i < 0$, in which case the statement of the lemma is also true).

   $\mu\alpha.t$**:** $\langle k, v\rangle \in [\![\mu\alpha.t]\!]_{\text{idx}}$, so by definition

$$\langle k, v\rangle \in [\![\text{unroll}(k+1, \alpha, t)]\!]_{\text{idx}} = [\![t[\text{unroll}(k, \alpha, t)/\alpha]]\!]_{\text{idx}}.$$

   Using downward closure we get:

$$\langle k-1, v\rangle \in [\![t[\text{unroll}(k, \alpha, t)/\alpha]]\!]_{\text{idx}}$$

By Lemma 12 this means that

$$\langle k-1, v\rangle \in [\![t[\mu\alpha.t/\alpha]]\!]_{\mathrm{idx}}.$$

But $(\mathcal{W}^{\perp,\xi}_{\mu\alpha.t}\ v)$ can perform one step yielding $(\mathcal{W}^{\perp,\xi}_{t[\mu\alpha.t/\alpha]}\ v)$. Applying the induction hypothesis gives

$$(\mathcal{W}^{\perp,\xi}_{t[\mu\alpha.t/\alpha]}\ v) :_{k-1} \underline{\mathrm{rsafe}}.$$

$t_1 \xrightarrow{x} t_2$**:** Here v must have the form $\lambda x.b$. Taking one step $(\mathcal{W}^{\perp,\xi}_{t_1 \xrightarrow{x} t_2}\ v)$ reduces to

$$\lambda y.(\mathcal{W}^{\perp,\xi}_{t_2[y/x]}\ (v\ (\mathcal{W}^{\xi,\perp}_{t_1}\ y))_\perp).$$

Let's call this expression X. We need to show that $\langle k-1, X\rangle \in [\![\underline{\mathrm{rsafe}}]\!]_{\mathrm{idx}}$, i.e., that $\langle k-1, X\rangle \in [\![\underline{\mathrm{rsafe}}_{k-1}]\!]_{\mathrm{idx}}$. Given that X is not an integer, this is the same as showing that $\langle k-1, X\rangle \in [\![\underline{\mathrm{rsafe}}_{k-2} \to \underline{\mathrm{rsafe}}_{k-2}]\!]_{\mathrm{idx}}$. Pick any $j < k-1$ and $\langle j, w\rangle \in [\![\underline{\mathrm{rsafe}}_{k-2}]\!]_{\mathrm{idx}}$. By Lemma 11 this is equivalent to $\langle j, w\rangle \in [\![\underline{\mathrm{rsafe}}_j]\!]_{\mathrm{idx}}$, which means that $w :_j \underline{\mathrm{rsafe}}$. Induction hypothesis(b.) gives us $(\mathcal{W}^{\xi,\perp}_{t_1}\ w) :_j t_1$. If $(\mathcal{W}^{\xi,\perp}_{t_1}\ w)\Downarrow_i w'$ with $0 < i \le j$, then $\langle j-i, w'\rangle \in [\![t_1]\!]_{\mathrm{idx}}$, so $b[w'/x] :_{j-i} t_2[w'/x]$. Induction hypothesis(a.) now implies that $(\mathcal{W}^{\perp,\xi}_{t_2[w'/x]}\ b[w'/x]) :_{j-i} \underline{\mathrm{rsafe}}$ and, consequently, that[12]

$$(\mathcal{W}^{\perp,\xi}_{t_2[w/x]}\ (v\ (\mathcal{W}^{\xi,\perp}_{t_1}\ w))_\perp) :_j \underline{\mathrm{rsafe}}.$$

So, indeed, we obtain the required result:

$$\langle k-1, \lambda y.(\mathcal{W}^{\perp,\xi}_{t_2[y/x]}\ (v\ (\mathcal{W}^{\xi,\perp}_{t_1}\ y))_\perp)\rangle \in [\![\underline{\mathrm{rsafe}}_{k-2} \to \underline{\mathrm{rsafe}}_{k-2}]\!]_{\mathrm{idx}}$$

b. Again, we perform a case analysis on the outermost type constructor:

$\underline{\mathrm{bot}}$**:** The expression $(\mathcal{W}^{\xi,\perp}_{\underline{\mathrm{bot}}}\ v)$ always goes into an infinite loop.

$\underline{\mathrm{int}}$**:** The expression $(\mathcal{W}^{\xi,\perp}_{\underline{\mathrm{int}}}\ v)$ either returns the integer v or goes into an infinite loop if v was not an integer.

$t_1 \vee_\chi t_2$**:** We consider the case that $\chi(v)$ evaluates to false. (As before for (a.), the other case is completely analogous.) $(\mathcal{W}^{\xi,\perp}_{t_1 \vee_\chi t_2}\ v)$ takes one step to $(\mathcal{W}^{\xi,\perp}_{t_1}\ v)$. Let's call this X. By downward closure we have that $\langle k-1, v\rangle \in [\![\underline{\mathrm{rsafe}}]\!]_{\mathrm{idx}}$, so applying the induction hypothesis yields $X :_{k-1} t_1$. Let $X\Downarrow_j w$ with $j < k-1$. (Otherwise the statement of the lemma is trivially true.) So $\langle k-j-1, w\rangle \in [\![t_1]\!]_{\mathrm{idx}}$ and therefore $\langle k-j-1, w\rangle \in [\![t_1 \vee_\chi t_2]\!]_{\mathrm{idx}}$.

$\langle t \mid \phi\rangle$**:** $(\mathcal{W}^{\xi,\perp}_{\langle t|\phi\rangle}\ v)$ makes one step resulting in $\phi(v)?_\perp(\mathcal{W}^{\xi,\perp}_t\ v)$. If $\phi(v)$ returns false, then the expression diverges and the statement of the lemma is true. Otherwise, if $\phi(v)$ returns true, then after $i \ge 1$ steps we arrive at $(\mathcal{W}^{\xi,\perp}_t\ v)$. Let's call this X. Induction hypothesis and downward closure imply that $X :_{k-i} t$. Let $X\Downarrow_j w$ with $j < k-i$ (or else the statement of the lemma becomes trivially true). Then $\langle k-i-j, w\rangle \in [\![t]\!]_{\mathrm{idx}}$ and therefore $\langle k-i-j, w\rangle \in [\![\langle t \mid \phi\rangle]\!]_{\mathrm{idx}}$.

[12] We rely here on the fact that terms substituted into contracts never play any role outside of becoming arguments to predicates – which we assume to be total. The number of steps they take to evaluate are not part of the interpretation of contracts.

$\mu\alpha.t$**:** We know that $\langle k, v\rangle \in [\![\underline{\text{rsafe}}_k]\!]_{\text{idx}}$, so by downward closure $\langle k-1, v\rangle \in [\![\underline{\text{rsafe}}_k]\!]_{\text{idx}}$, and by Lemma 11 $\langle k-1, v\rangle \in [\![\underline{\text{rsafe}}_{k-1}]\!]_{\text{idx}}$. The term $(\mathcal{W}^{\xi,\perp}_{\mu\alpha.t}\ v)$ makes one step to $(\mathcal{W}^{\xi,\perp}_{t[\mu\alpha.t/\alpha]}\ v)$. Let's call this X. By induction hypothesis:

$$X :_{k-1} t[\mu\alpha.t/\alpha]$$

and by Lemma 12 this implies

$$X :_{k-1} t[\text{unroll}(k, \alpha, t)/\alpha],$$

i.e.,

$$X :_{k-1} \text{unroll}(k+1, \alpha, t).$$

Using Lemma 11 again we arrive at the desired result, namely $X :_{k-1} \text{unroll}(k, \alpha, t)$ which means $X :_{k-1} \mu\alpha.t$.

$t_1 \xrightarrow{x} t_2$**:** Unless v has the form $\lambda z.b$, the expression $(\mathcal{W}^{\xi,\perp}_{t_1 \xrightarrow{x} t_2}\ v)$ goes into an infinite loop, and the statement of the lemma is satisfied. Let $v \equiv \lambda z.b$. Taking one step, $(\mathcal{W}^{\xi,\perp}_{t_1 \xrightarrow{x} t_2}\ v)$ reduces to

$$\lambda y.(\mathcal{W}^{\xi,\perp}_{t_2[y/x]}\ (v\ (\mathcal{W}^{\perp,\xi}_{t_1}\ y))_\perp).$$

Let's call this X. We have to show that $\langle k-1, X\rangle \in [\![t_1 \xrightarrow{x} t_2]\!]_{\text{idx}}$. Pick any $j < k-1$ and $\langle j, w\rangle \in [\![t_1]\!]_{\text{idx}}$. By induction hypothesis(a.) we get $(\mathcal{W}^{\perp,\xi}_{t_1}\ w) :_j \underline{\text{rsafe}}$. If $(\mathcal{W}^{\perp,\xi}_{t_1}\ w)\Downarrow_i w'$ with $0 < i \le j$, then $\langle j-i, w'\rangle \in [\![\underline{\text{rsafe}}]\!]_{\text{idx}}$. We know that $\langle k, \lambda z.b\rangle \in [\![\underline{\text{rsafe}}]\!]_{\text{idx}}$, so by downward closure also $\langle j-i+1, \lambda z.b\rangle \in [\![\underline{\text{rsafe}}]\!]_{\text{idx}}$, i.e., $\langle j-i+1, \lambda z.b\rangle \in [\![\underline{\text{rsafe}}_{j-1} \to \underline{\text{rsafe}}_{j-1}]\!]_{\text{idx}}$. Thus, $b[w'/z] :_{j-i} t_2[w'/x]$. Using induction hypothesis(b.) we conclude that $(\mathcal{W}^{\xi,\perp}_{t_2[w'/x]}\ b[w'/z]) :_{j-i} t_2[w'/x]$, and that therefore

$$(\mathcal{W}^{\xi,\perp}_{t_2[w/x]}\ (v\ (\mathcal{W}^{\perp,\xi}_{t_1}\ w))_\perp) :_j t_2[w/x].$$

Thus, we have proved that

$$\langle k-1, \lambda y.(\mathcal{W}^{\xi,\perp}_{t_2[y/x]}\ (v\ (\mathcal{W}^{\perp,\xi}_{t_1}\ y))_\perp)\rangle \in [\![t_1 \xrightarrow{z} t_2]\!]_{\text{idx}}.$$

□

Using Lemma 15, the following corollary is immediately obvious:

Lemma 16
For any ξ and any $t \in T_{\text{safe}}$:

a. $e :_\infty t \Rightarrow (\mathcal{W}^{\perp,\xi}_t\ e) :_\infty \underline{\text{rsafe}}$
b. $e :_\infty \underline{\text{rsafe}} \Rightarrow (\mathcal{W}^{\xi,\perp}_t\ e) :_\infty t$

6.4 Soundness of recursive contracts

The proof of soundness of contract checking in the presence of recursive contracts (replacing $[\![\cdot]\!]$ with $[\![\cdot]\!]_\infty$) proceeds exactly like the proof of the original Theorem 1, using Lemma 16 instead of Lemma 7.

6.5 Completeness of recursive contract checking

We omit the detailed proof of completeness (Theorem 2) for the case of recursive contracts (assuming total predicates) and limit ourselves to the following informal argument: If an expression e violates a potentially recursive contract t, then there is an e' derived from e by replacing every recursive contract with a sufficiently precise approximation and a t' derived from t in an analogous way such that e' violates t'. The original proof now applies to e' and t', showing that there is a context c in which the contract checker finds the problem by raising some exception $\top$. But the same $\top$ will be raised when considering the original e with the original t in context c.

6.6 Recursion, safety, and predicates

We started with a non-recursive contract language that included a primitive contract $\underline{\text{safe}}$. We interpreted it—in an ad-hoc fashion—as the set of safe values, and subsequently found that this interpretation is sound and complete (under reasonable assumptions). We could have motivated our choice by looking at the statement of Lemma 7, noting that the contract wrapper for the always-true predicate is simply the identity. If the identity maps safe values to values satisfying the contract and vice versa, every satisfying term is a safe term, and every safe term is a term satisfying $\underline{\text{safe}}$. Thus, in any model $[\![\cdot]\!]$ of contracts that has ambitions at being sound as well as complete, it must be the case that $[\![\underline{\text{safe}}]\!] = \mathbf{Safe}$. But Lemma 7 was just a means and not the end here, so it might be worth asking the question whether another interpretation would work as well—perhaps giving up on the central lemma in exchange for a different method of proving soundness and completeness.

It turns out that this is not the case, and the consideration of recursive contracts shows why. The guards for the two contracts $\underline{\text{safe}}$ and $\underline{\text{rsafe}}$ are operationally equivalent, and contracts with equivalent guards should have equal interpretations:

Lemma 17
For an arbitrary expression e and for any k:

$$(\mathcal{W}^{\xi',\xi}_{\underline{\text{rsafe}}}\ e) \cong_k (\mathcal{W}^{\xi',\xi}_{\underline{\text{safe}}}\ e)$$

Proof
The right-hand side is clearly equivalent to e, so we show by induction on k that the left-hand side is also equivalent to e. Consider the case where e is some value v. (Otherwise the induction hypothesis applies trivially.) After one step, the left-hand side yields $(\mathcal{W}^{\xi',\xi}_{\underline{\text{int}}\vee(\underline{\text{rsafe}}\to\underline{\text{rsafe}})}\ v)$ If v is an integer, it further reduces to $(\mathcal{W}^{\xi',\xi}_{\underline{\text{int}}}\ v)$ and then to v. Otherwise v must have the form $\lambda x.b$. To be operationally distinguishable, there must be some w such that $(\mathcal{W}^{\xi',\xi}_{\underline{\text{rsafe}}}\ (v\ (\mathcal{W}^{\xi,\xi'}_{\underline{\text{rsafe}}}\ w))_\perp)$ is distinguishable from $(v\ w)_\perp$ in fewer than k steps. By induction hypothesis, this is impossible. □

Therefore, $\underline{\text{safe}}$ *must* be interpreted as **Safe** because $\underline{\text{rsafe}}$ is. 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, the Findler-Felleisen system has *unrestricted* predicate contracts $\langle\phi\rangle$ whose guards are operationally equivalent to the guards for our $\langle\underline{\text{safe}} \mid \phi\rangle$ (or $\langle\underline{\text{rsafe}} \mid \phi\rangle$). 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, $\{v \mid (\phi\ v) \in \{\underline{1}, \bot\}\}$. 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 (Findler & Felleisen, 2002) because they fail to statically type-check there.

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 always assigns blame properly and is—in principle—able to discover all violations: for every violation there is a context in which the checker finds it. The contract language can be extended to include a recursion operator without compromising the existence of a sound and complete model.

The main technical insight from our proofs is in the simple and apparently fundamental theoretical properties of contract wrappers expressed in the central lemma (Lemma 7 and its variations: Lemmas 10 and 15). The central lemma shows that there is strong interaction between the semantics of contracts and a notion of safety. Furthermore, the fact that Findler-Felleisen-style unrestricted predicate contracts $\langle\phi\rangle$ are operationally equivalent to our $\langle\underline{\text{safe}} \mid \phi\rangle$ 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 $\underline{\text{safe}}$ 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 nontermination 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 (Findler *et al.*, 2004).

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

$$
\begin{array}{rcll}
(\mathcal{W}^{+\xi}_{\underline{\text{int}}}\ \underline{i}) & \to & \underline{i} & \\
(\mathcal{W}^{+\xi}_{t_1 \to t_2}\ \lambda x.e) & \to & \lambda y.(\mathcal{W}^{+\xi}_{t_2}\ ((\lambda x.e)\ (\mathcal{W}^{-\xi}_{t_1}\ y))_{\perp}) & \\
(\mathcal{W}^{+\xi}_{t}\ v) & \to & \textbf{raise}\ \xi & ;\textit{otherwise} \\
(\mathcal{W}^{-\xi}_{t_1 \to t_2}\ \lambda x.e) & \to & \lambda y.(\mathcal{W}^{-\xi}_{t_2}\ ((\lambda x.e)\ (\mathcal{W}^{+\xi}_{t_1}\ y))_{\perp}) & \\
(\mathcal{W}^{-\xi}_{t}\ v) & \to & v & ;\textit{otherwise}
\end{array}
$$

Fig. 11. Operational semantics for $\mathcal{W}^{+\xi}_t$ and $\mathcal{W}^{-\xi}_t$ for a simple contract language with only $\underline{\text{int}}$ and $\to$.

higher-order wrappers, i.e., functions from wrappers to wrappers, to treat contracts of the form $\forall\alpha.t$ interpreted as $\bigcap_{t'}[\![t[t'/\alpha]]\!]$. Similarly, we have not considered mutual recursion between modules, although given the untyped nature of our core calculus it appears easy to simulate such mutual recursion using value-level fixpoint constructions. Our soundness proof is for a language with call-by-value semantics. Since most real-world languages that are pure (e.g., Clean (Brus *et al.*, 1987) or Haskell (Jones, 2003)) 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^{e}_i satisfies t_i in the $[\![\cdot]\!]_{\text{FF}}$ 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:

$$\mathcal{W}^{\xi'_2,\xi_2}_t \circ \mathcal{W}^{\xi'_1,\xi_1}_t = \mathcal{W}^{\xi'_2,\xi_1}_t$$

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

$$\mathcal{W}^{\xi'_2,\xi_2}_{t_2} \circ \mathcal{W}^{\xi'_1,\xi_1}_{t_1}$$

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

$$\mathcal{W}^{\xi'_2,\perp}_{t_2} \circ \mathcal{W}^{\perp,\xi_1}_{t_1}$$

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

$$\mathcal{W}_t^{\xi',\xi} = \mathcal{W}_t^{-\xi'} \circ \mathcal{W}_t^{+\xi}$$

Operational rules for $\mathcal{W}_t^{-\xi'}$ and $\mathcal{W}_t^{+\xi}$ are easy to set up. The main idea is to alternate between $\mathcal{W}^-$ and $\mathcal{W}^+$ instead of swapping exception superscripts in contravariant positions. The rules (abusing notation when "raising" ξ) for a simple contract language with only $\underline{\text{int}}$ and $\rightarrow$ are shown in Figure 11.

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

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

Notice that the requirements on the context expressed, by $\mathcal{W}^-$, are *preconditions* while the requirements on the value, expressed by $\mathcal{W}^+$, are *postconditions*. Thus, the above laws precisely capture the fact that one has to keep the weakest precondition and the strongest postcondition (Hoare, 1969; Dijkstra, 1976). Using $[\![t_1]\!] \subseteq [\![t_2]\!]$ we get

$$\begin{aligned} \mathcal{W}_{t_2}^{\xi_2',\xi_2} \circ \mathcal{W}_{t_1}^{\xi_1',\xi_1} &= \mathcal{W}_{t_2}^{-\xi_2'} \circ \mathcal{W}_{t_2}^{+\xi_2} \circ \mathcal{W}_{t_1}^{-\xi_1'} \circ \mathcal{W}_{t_1}^{+\xi_1} \\ &= \mathcal{W}_{t_2}^{-\xi_2'} \circ \mathcal{W}_{t_1}^{-\xi_1'} \circ \mathcal{W}_{t_2}^{+\xi_2} \circ \mathcal{W}_{t_1}^{+\xi_1} \\ &= \mathcal{W}_{t_2}^{-\xi_2'} \circ \mathcal{W}_{t_1}^{+\xi_1} \end{aligned}$$

which—as we had hoped—no longer contains those 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. We also would like to express our appreciation to several anonymous reviewers as well as members of the audience at ICFP'04, where an earlier version of this paper was presented (Blume & McAllester, 2004).

References

Appel, Andrew W., & McAllester, David. (2001). An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, **23**(5), 657–683.

Blume, Matthias, & McAllester, David. (2004). A sound (and complete) model of contracts. *Pages 189–200 of: Proc. 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*. ACM Press.

Bruce, Kim B., Cardelli, Luca, & Pierce, Benjamin C. (1997). Comparing object encodings. *Pages 415–438 of: Theoretical aspects of computer software*.

Brus, T.H., van Eekelen, M.C., van Leer, M.O., & Plasmeijer, M.J. (1987). CLEAN: A Language for Functional Graph Rewriting. Kahn, G. (ed), *Proc. of the Conf. on*

Functional Programming Languages and Computer Architecture (FPCA'87), Portland, Oregon, USA. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, Germany.

Dijkstra, Edsger W. (1976). *A discipline of programming*. Prentice-Hall.

Felleisen, Matthias, & Hieb, Robert. (1992). A revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, **103**(2), 235–271.

Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, & Kristnamurthi, Shriram. (1998). The DrScheme project: An overview. *SIGPLAN Notices*, **33**(6), 17–23.

Findler, Robert B. (2002). *Behavioral software contracts*. Ph.D. thesis, Rice University.

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Pages 48–59 of: Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

Findler, Robert Bruce, Clements, John, Cormac Flanagan, Matthew Flatt, Krishnamurthi, Shriram, Steckler, Paul, & Felleisen, Matthias. (2002). DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, **12**(2), 159–182.

Findler, Robert Bruce, Blume, Matthias, & Felleisen, Matthias. (2004). *An investigation of contracts as projections*. Tech. rept. TR-2004-02. University of Chicago Computer Science Department.

Hartley Rogers, Jr. (1987). *Theory of recursive functions and effective computability*. Cambridge, MA, USA: MIT Press.

Hoare, C A. R. (1969). An axiomatic basis for computer programming. *Communications of the acm*, **12**(10), 578–580.

Jones, Simon Peyton. (2003). *Haskell 98 language and libraries*. Cambridge University Press.

Leroy, Xavier. 1990 (Feb.). *The ZINC experiment: an economical implementation of the ML language*. Tech. rept. No. 117. INRIA.

Meyer, Bertrand. (1992). *Eiffel: The Language*. Prentice-Hall.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML (revised)*. Cambridge, MA: MIT Press.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. amer. math. soc.*, **74**, 358–366.

Scott, D. S. (1972). Continuous lattices. *Pages 97–136 of:* Lawvere, F. W. (ed), *Toposes, algebraic geometry and logic*. Lecture Notes in Mathematics, vol. 274. Springer.

A 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, e.g., as witnesses for some contract being violated. A potential problem with this is the fact that these are values of the *internal* language, while the main theorem had been stated in terms of the *external* language. If an internal witness turns out to have no external counterpart, one might argue that its existence is irrelevant since the programmer is only interested in the external language. Thus, we should convince ourselves that external witnesses can be constructed from internal ones.

The construction is possible because all our witness values v are syntactically safe. Given sufficient language support safety can be "coded up."

For the sake of simplicity we restricted the external language used in this paper

to an extremely simple one. A slightly more realistic version would certainly have some form of conditional, for example a branch on equality to $\underline{1}$. If the language also comes with a mechanism for separating functions from integers (as most dynamically typed languages do), then all witnesses $\lfloor v \rfloor$ have an operationally equivalent external counterpart. For example, the language could come with some *typecase* construct which is capable of implementing the separation function for $\underline{\text{int}} \vee \alpha \to \beta$. (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 $(\textbf{tycase}\ e_1\ e_2\ e_3)$ 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 $(e_1^{\text{e}}\ e_2^{\text{e}})$ and re-code it as

$$((\lambda f.(\textbf{tycase}\ f\ \Omega\ (f\ e_2^{\text{e}})))\ e_1^{\text{e}})$$

where—as before—Ω is a diverging term, e.g., $\Omega = ((\lambda x.(x\ x))\ \lambda x.(x\ x))$ and run that through the translator $\mathcal{C}$. The implicit exception inserted by the translation 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 $\mathcal{W}_t^{\perp,\perp}$ and $\mathcal{W}_t^{\top^i,\perp}$. Given conditionals and typecase, coding up a wrapper in a type-directed fashion is straightforward. Each wrapper becomes an ordinary value-level function; wrappers for recursive contracts are recursive functions, i.e., results of applying a value-level fixpoint operator. 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 $\mathcal{C}_{\top^i}(\cdot;\emptyset)$ (see Theorem 1), the exception annotation on $(\underline{0}\ \underline{0})$ will indeed be $\top^i$.