Static Contract Checking for Haskell

Dana N. Xu

University of Cambridge
nx200@cam.ac.uk

Simon Peyton Jones

Microsoft Research
simonpj@microsoft.com

Koen Claessen

Chalmers University of Technology
koen@chalmers.se

Abstract

Program errors are hard to detect and are costly both to programmers who spend significant efforts in debugging, and for systems that are guarded by runtime checks. Static verification techniques have been applied to imperative and object-oriented languages, like Java and C#, but few have been applied to a higher-order lazy functional language, like Haskell. In this paper, we describe a sound and automatic static verification tool for Haskell, that is based on contracts and symbolic execution. Our approach is modular and gives precise blame assignments at compile-time in the presence of higher-order functions and laziness.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms verification, functional language

Keywords contract satisfaction, static contract checking

1. Introduction

Program errors are common in software systems, including those that are constructed from functional languages. For greater software reliability, such errors should be reported accurately and detected early during program development. *Contract checking* (both static and dynamic) has been widely used in procedural and object-oriented languages [21, 12, 4, 2]. The difficulty of contract checking in functional languages lies in the presence of advanced features such as higher-order functions and laziness. However, dynamic checking of contracts for higher-order functions has been studied in [10, 3, 9, 16]. Recently, static pre/postcondition checking [34] as well as hybrid contract checking [11, 20, 19, 15] for functional languages have also been proposed.

In this paper, we combine the idea of the contract semantics [10, 3] and the idea of the static verification through symbolic execution [34] to propose a sound automatic static contract checking framework for a higher-order lazy functional language, Haskell. Consider:

```
f :: [Int] -> Int
f xs = head xs 'max' 0
```

where head is defined in the module Prelude as follows:

```
head :: [a] -> a
head (x:xs) = x
head [] = error "empty list"
```

If we have a call (f []) in our program, its execution will result in the following error message from the runtime system of the Glasgow Haskell Compiler (GHC):

> Exception: Prelude.head: empty list

This gives no information on which part of the program is wrong except that head has been wrongly called with an empty list. Presumably, the programmers' intention is that head should not be called with an empty list. To achieve this, programmers can give a contract to the function head. Contracts are implemented as pragmas:

```
{-# CONTRACT head :: {s | not (null s)} -> {r | True} #-}
```

where not and null are just ordinary Haskell functions:

```
null :: [a] -> Bool        not :: Bool -> Bool
null [] = True             not True = False
null xs = False            not False = True
```

This contract places the onus on callers of head to ensure that the argument to head satisfies the expected precondition. With this contract, our compiler would generate the following warning (by giving a counter-example) when checking the definition of f:

```
Error: f []    calls head
               which may fail head's precondition!
```

This paper makes the following specific contributions:

- We give a crisp, declarative specification for what it means for a term to satisfy a contract (§4). This is unusual, with the notable exception of Blume & McAllester [3]. Unlike Blume & McAllester (and most other related work on higher-order contracts) we focus on *static* verification, and target a *lazy* language.

- Our contracts themselves contain unrestricted Haskell terms, so we tackle head-on the question of what happens if the contract itself crashes (§6) or diverges (§7).

- Despite these complications, we are able to give a very strong theorem expressing the soundness and completeness of contract wrappers as compared to contract satisfaction (§5).

- We develop a concise notation (▷ and ◁) for describing contract checking, that enjoys many useful properties (§5.3). Thus equipped, we give a new and relatively-simple proof of the soundness and completeness of dynamic contract checking; this proof is much trickier than it looks.

- Our framework neatly accommodates some subtle points that others have encountered, including: ensuring that all contracts are inhabited (§4.2), and the Any contract (§4.4).

- We describe how to augment the contract decision procedure so that it maintains extra information that helps the programmer to localise the error (§8).

2. Overview

The type of a function constitutes a partial specification to the function. For example, `inc :: Int -> Int` says that `inc` is a function that takes an integer and returns an integer. A *contract* of a function gives more detailed specification. For example: `{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}` says that the function `inc` takes a positive value and returns a value that is greater than the input. A contract can therefore be viewed as a refinement to a type, so it is also known as refinement type in [14, 7, 11].

This paper describes a system that allows a programmer to write a contract on some (but, like type signatures, not necessarily all) definitions, and then statically checks whether the definition *satisfies* the contract. This check is undecidable, and so may give the result "definitely satisfies", "definitely does not satisfy", or "don't know". In the latter two cases we emit information that helps to localise the (possible) bug. We begin, however, by giving the flavour of contracts themselves with various examples, concentrating on aspects where our approach differs from other contract systems. §4 makes contracts precise.

2.1 Base and dependent function contracts

We adopt the basic contract notation from [10, 11]. For example, here is a contract that declares that the length of the result `rs` is the same as the length of the argument `xs`:

```
reverse :: [a] -> [a]
{-# CONTRACT
   reverse :: xs:Ok -> {rs | length xs == length rs} #-}
reverse = ...
```

The expression (`length xs == length rs`) is an arbitrary boolean-valued Haskell expression. *Programmers do not need to learn a new language of predicates; they just use Haskell.* The contract `Ok` is short for `{x | True}`.

Notice too that this is a *dependent* function contract, because the argument `xs` is used in the result contract.

The contract notation is more expressive than the `requires`, `ensures` notation used in our earlier work [34], because it scales properly to higher order functions. Consider an example adapted from [3]:

```
f1 :: (Int -> Int) -> Int
f1 g = (g 1) - 1
{-# CONTRACT f1 :: ({x | True} -> {y | y>=0})
                    -> {r | r>=0} #-}
f2 = f1 (\x -> x - 1)
```

The contract of `f1` says that if `f1` takes a function, which returns a natural number when given any integer, the function `f1` itself returns a natural number.

The Findler-Felleisen algorithm in [10] (a dynamic contract checking algorithm) can detect a violation of the contract of `f1`, however, it cannot tell that the argument of `f1` in the definition of `f2` fails `f1`'s precondition (due to the lack of witness at run-time). On the other hand, the Sage system in [20] (a hybrid contract checking system) can detect the failure in `f2` statically, and can report contract violation of `f1` at run-time. Our system can report both failures at compile-time with the following messages:

```
Error: f1's postcondition fails
```

```
       when (g 1) >= 0 holds
            (g 1) - 1 >= 0 does not hold

Error: f2 calls f1
       which fails f1's precondition
```

2.2 Laziness

Laziness can cause false alarms. For example:

```
fst (a, b) = a
f3 = fst (5, error "f")
```

Syntactically, the call `fst (5, error "f")` appears unsafe because the existence of a call to `error`, but in a lazy language like Haskell the call is perfectly safe. The only static verification tool that caters for laziness is the ESC/Haskell system [34], which can reduce false alarms due to laziness by inlining. In the above case, the function `fst` is inlined, so the call to `fst` in `f3` becomes 5 which is syntactically safe. However, if the size of the lazy function is big, or the function is recursive, the inlining strategy breaks down. In this paper we therefore introduce a special contract `Any`, which every expression satisfies. Now we can give `fst` a contract

```
{-# CONTRACT fst :: (Ok, Any) -> Ok #-}
```

which says that `fst` does not care what the second component of the argument is, as long as the first component is crash-free, the result is crash-free. Here, with the contract `Any`, without inlining any function, our system can tell that `f3` is safe.

This means we give a crash (`error "msg"`) a contract `Any`, while in [3] an expression that unconditionally crashes satisfies no contract. This is one of the key differences in designing the contract semantics.

2.3 Data constructors in contracts

In the previous section we gave `fst`'s argument the contract (`Ok, Any`); that is, the argument should be a pair whose first component satisfies `Ok`, and whose second satisfies `Any`. We generalise this form for any user-defined data constructor. For example:

```
data A = A1 Int Bool | A2 A

f4 :: A -> Int
{-# CONTRACT f4 :: A1 {x | x > 0} Ok
                 -> {r | r > 0} #-}
f4 (A1 x y) = if y then x + 1
                   else error "f4"
```

Note that, the data constructor `A1` is used in the above contract while the data type `A` is used in the type specification. Besides checking the sub-components of `A1`, a call (`f4 (A2 z)`) is a crash because the contract of `f4` says that `f4` can only be applied to data constructed by `A1`.

2.4 Partial Functions in Contracts

A partial function is one that may *crash* or *diverge*. For example, function `head`, which crashes when given an argument `[]`. Since we allow arbitrary Haskell code in contracts, what are we to say about contracts that crash or diverge? One possibility is to simply exclude all such contracts – but excluding divergence (in a statically-checkable system) requires a termination checker, and excluding functions like `head` is extremely restrictive. For example:

```
headPlus :: [Int] -> Int
{-# CONTRACT headPlus :: {xs | not (null xs)}
                         -> {r | r > head xs} #-}
headPlus []     = error "Urk"
headPlus (x:x)  = x+1
```

Here the postcondition uses `head` (which may crash), but that seems entirely reasonable in view of the precondition that `xs` is non-empty. Nevertheless, such a contract is rejected by [3], because of the call to `head`.

Our approach is to permit divergence in contracts (which avoids the requirement for a termination checker), but to restrict contracts to be *well-formed*, which is still considerably more liberal than requiring them to be crash-free. This liberality is, we believe, key to making contracts usable in practice. We discuss well-formedness in §6.1 and divergence in §7.1.

2.5 The Plan for Verification

It is all very well for programmers to *claim* that a function satisfies a contract, but how can we *verify* the claim statically (i.e. at compile time)? Our overall plan, which is similar to that of Blume & McAllester [3], is as follows.

- Our overall goal is to prove that the program does not *crash*, so we must first say what programs are, and what it means to "crash" (§3.3).

- Next, we give a semantic specification for what it means for an expression e to "satisfy a contract" t, written $e \in t$ (§4).

- From a definition $f = e$ we form a term $e \triangleright t$ pronounced "e ensures t". This term behaves just like e except that (a) if e disobeys t then the term crashes; (b) if the context uses e in a way not permitted by t then the term loops. The term $e \triangleright t$ is essentially the wrapper mechanism first described by Findler & Felleisen [10], with some important refinements (§5).

- With these pieces in place, we can write down our main theorem for well-formed contracts t (§5), namely that

$$e \in t \iff (e \triangleright t) \text{ is crash-free}$$

We must ensure that everything works properly, even if e diverges, or laziness is involved, or the contract contains divergent or crashing terms.

- Using this theorem, we may check whether $f \in t$ holds as follows: we attempt to prove that $(e \triangleright t)$ is crash-free — that is, does not crash under all contexts. We conduct this proof in a particularly straightforward way: we perform symbolic evaluation of $(e \triangleright t)$. If we can simplify the term to a new term e', where e' is syntactically safe — that is, contains no crashes everywhere in the expression — then we are done. This test is sufficient, but not necessary; of course, the general problem is undecidable.

3. The Language

The language presented in this paper, named language $\mathcal{H}$, is simply-typed lambda calculus with case-expression, constructors and integers. Language $\mathcal{H}$ is simpler than the language we use in our implementation, which is the GHC Core Language [29], which is similar to System F and includes parametric polymorphism.

3.1 Syntax

The syntax of our language $\mathcal{H}$ is shown in Figure 1. A program is a module that contains a set of data type declarations, contract specifications and function definitions. Expressions include variables, type and term abstractions, type and term applications, constructors and `case` expressions. We treat `let`-expressions as syntactic sugar:

$$\texttt{let } x = e_1 \texttt{ in } e_2 \quad \equiv_s \quad (\lambda x.e_2)\, e_1$$

We omit local `letrec`, in favour of recursive (or mutually recursive) top-level functions. The language is typed, and to save clutter

pgm	$::=$	$def_1, \ldots, def_n$	**Programs**
def	$\in$	**Definition**	
def	$::=$	$\texttt{data } T\, \vec{\alpha} = K_1\, \vec{\tau_i} \mid \cdots \mid K_n\, \vec{\tau_i}$	
	$\mid$	$f \in t$	Contract attribution
	$\mid$	$f\, \vec{x} = e$	Top-level definition
a, e, p	$\in$	**Exp**	**Expressions**
a, e, p	$::=$	n	integers
	$\mid$	r	exception
	$\mid$	$v \mid \lambda(x{:}\tau).e \mid e_1\, e_2$	
	$\mid$	$\texttt{case } e_0 \texttt{ of } \{alt_1 \ldots alt_n\}$	case-expression
	$\mid$	$K\, \vec{e}$	constructor
	$\mid$	$\texttt{fin}_n\, e$	finite evaluation
r	$::=$	$\texttt{BAD} \mid \texttt{UNR}$	**Exceptions**
alt	$::=$	$K\, (x_1{:}\tau_1) \ldots (x_n{:}\tau_n) \to e$	**Alternatives**
	$\mid$	$\texttt{DEFAULT} \to e$	
val	$::=$	$n \mid r \mid K\, \vec{e} \mid \lambda(x{:}\tau).e$	**Values**
τ	$::=$	$\texttt{Int} \mid \texttt{Bool} \mid () \mid T\, \vec{\tau} \mid \tau_1 \to \tau_2$	**Types**

Figure 1: Syntax of the language $\mathcal{H}$

in this paper we silently assume that all expressions, contexts and contracts are well-typed. Type checking for contracts can be found in [35].

There are two *exception values* adopted from [34]:

`BAD` is an expression that *crashes*. A program crashes if and only if it evaluates to `BAD`. For example, a user-defined function `error` can be explicitly defined as:

```
error :: String -> a
error s = BAD
```

A preprocessor ensures that source programs with missing cases of pattern matching are explicitly replaced by the corresponding equations with `BAD` constructs. For example, after preprocessing, function `head`'s definition becomes:

```
head (x:xs) = x
head []     = BAD
```

`UNR` (short for "unreachable") is an expression that gets stuck. This is *not* considered a crash, although execution comes to a halt without delivering a result. A program that loops forever also does not crash, and does not deliver a result, so you can think of `UNR` as a term that simply goes into an infinite loop.

3.2 Operational Semantics

The semantics of our language is given by the confluent, non-deterministic rewrite rules in Figure 2. We use a small-step reduction-rule semantics, rather than (say) a deterministic more machine-oriented semantics, because the more concrete the semantics becomes, the more involved the proofs become too.

Most of these rules are entirely conventional. The rule [E-top] deals with a top-level function call f. We fetch its definition from the environment Δ, which maps a variable to its type, contract and definition. To save clutter, we usually leave this environment implicit, rather than writing (say) $\Delta \vdash e_1 \to_M e_2$. Rules [E-exapp] and [E-excase] deal with exception values in the usual way. Rule [E-ctx] allows a reduction step to take place anywhere. The relation

Evaluation

$$\frac{(f = \lambda x.e) \in \Delta}{f \rightarrow_M \lambda x.e} \qquad \text{[E-top]}$$

$$\frac{e \rightarrow_M e' \quad n < M}{\mathtt{fin}_n\, e \rightarrow_M \mathtt{fin}_{n+1}\, e'} \qquad \text{[E-fin1]}$$

$$\mathtt{fin}_n\, \mathtt{UNR} \quad \rightarrow_M \quad \mathtt{True} \qquad \text{[E-fin2]}$$
$$\mathtt{fin}_M\, e \quad \rightarrow_M \quad \mathtt{True} \qquad \text{[E-fin3]}$$

$$(\lambda x.e_1)\, e_2 \quad \rightarrow_M \quad e_1[e_2/x] \qquad \text{[E-beta]}$$

$$\begin{array}{l}\mathtt{case}\ K_i\ \vec{a_i}\ \mathtt{of} \\ \quad \{ \ldots; \\ \quad K_i\ \vec{x_i} \rightarrow e_i; \qquad \rightarrow_M \quad e_i[a_i/x_i] \qquad \text{[E-match1]} \\ \quad \ldots\} \end{array}$$

$$\begin{array}{l}\mathtt{case}\ K\ \vec{a}\ \mathtt{of} \qquad\qquad \rightarrow_M \quad e \qquad \text{[E-match2]} \\ \quad \{pt_i \rightarrow e_i; \qquad\qquad\qquad \text{for all } i. \\ \quad \mathtt{DEFAULT} \rightarrow e\} \qquad\qquad K\ \vec{a} \not\sim pt_i \end{array}$$

$$r\, e \quad \rightarrow_M \quad r \qquad \text{[E-exapp]}$$
$$\mathtt{case}\ r\ \mathtt{of}\ alts \quad \rightarrow_M \quad r \qquad \text{[E-excase]}$$

$$\frac{e_1 \rightarrow_M e_2}{\mathcal{C}[\![e_1]\!] \rightarrow_M \mathcal{C}[\![e_2]\!]} \qquad \text{[E-ctx]}$$

$$\begin{array}{lll}\text{Contexts} & \mathcal{C} \ ::=\ & [\![\bullet]\!] \mid \mathcal{C}\, e \mid e\, \mathcal{C} \mid \lambda x.\mathcal{C} \\ & \mid & \mathtt{case}\ \mathcal{C}\ \mathtt{of}\ \{alt_1; \ldots; alt_n\} \\ & \mid & \mathtt{case}\ e\ \mathtt{of}\ \{\ldots; p_i \rightarrow \mathcal{C}; \ldots\} \end{array}$$

Figure 2: Semantics of the language $\mathcal{H}$

$e_1 \rightarrow_M e_2$ performs a single step reduction and the relation $\rightarrow_M^*$ is the reflexive-transitive closure of $\rightarrow_M$.

The unconventional features are the "M" subscript on the reduction arrow, the form $\mathtt{fin}_n\, e$, and the reduction rules [E-fin1,2,3]. These aspects all concern divergence, and are discussed in detail in §7.1, where we define $\rightarrow^*$ in terms of $\rightarrow_M^*$. For the present, simply ignore the subscripts and $\mathtt{fin}$.

Now we can give the usual definition of contextual equivalence:

DEFINITION 1 (Semantically Equivalent). *Two expressions e_1 and e_2 are semantically equivalent, namely $e_1 \equiv_s e_2$, iff*

$$\forall \mathcal{C}, r. \quad \mathcal{C}[\![e_1]\!] \rightarrow^* r \quad \Longleftrightarrow \quad \mathcal{C}[\![e_2]\!] \rightarrow^* r$$

Two expressions are said to be semantically equivalent, if under all (closing) contexts, if one evaluates to an exception r, the other also evaluates to r.

3.3 Crashing

We use BAD to signal that something has gone wrong in the program: it has *crashed.*

DEFINITION 2 (Crash). *A closed term e crashes iff $e \rightarrow^*$ BAD.*

Our technique can only guarantee *partial* correctness: a diverging program does not crash.

DEFINITION 3 (Diverges). *A closed expression e diverges, written $e \uparrow$, iff either $e \rightarrow^*$ UNR, or there is no value val such that $e \rightarrow^*$ val.*

At compile-time, one decidable way to check the safety of a program is to see whether the program is syntactically safe.

DEFINITION 4 (Syntactic safety). *A (possibly-open) expression e is* syntactically safe *iff* BAD $\notin_s e$*. Similarly, a context $\mathcal{C}$ is syntactically safe iff* BAD $\notin_s \mathcal{C}$*.*

The notation BAD $\notin_s e$ means BAD does not syntactically appear anywhere in e, similarly for BAD $\notin_s \mathcal{C}$. For example, $\lambda x.x$ is syntactically safe while $\lambda x.\,(\mathtt{BAD}, x)$ is not.

DEFINITION 5 (Crash-free Expression). *A (possibly-open) expression e is crash-free iff:*

$$\forall \mathcal{C}.\ \mathtt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![e]\!] \not\rightarrow^* \mathtt{BAD}$$

Definition 5 says that if an expression does not crash in all syntactically safe contexts, which are like "probes" for BAD, then the expression cannot crash regardless whether there is any BAD syntactically appearing in it (because all of them are unreachable). Notice that a crash-free expression may not be syntactically safe, for example:

```
\x. case x*x >= 0 of {True -> x+1; False -> BAD}
```

The tautology $x * x >= 0$ is always true, so the BAD can never be reached. On the other hand, $(\mathtt{BAD}, 3)$ is not crash-free because there exists a context, $\mathtt{fst}\, [\![\bullet]\!]$, such that:

$$\mathtt{fst}\, (\mathtt{BAD}, 3) \rightarrow \mathtt{BAD}$$

In short, crash-freeness is a *semantic* concept, and hence undecidable, while syntactic safety is *syntactic* and readily decidable. Certainly, a syntactically safe expression is crash-free:

LEMMA 1 (Syntactically Safe Expression is Crash-free).

$$e \text{ is syntactically safe} \quad \Rightarrow \quad e \text{ is crash-free}$$

4. Contract Syntax and Semantics

$$\begin{array}{llll}t & \in & \mathbf{Contract} \\ t & ::= & \{x \mid p\} & \text{Predicate Contract} \\ & \mid & x\colon t_1 \rightarrow t_2 & \text{Dependent Function Contract} \\ & \mid & (t_1, t_2) & \text{Data Constructor Contract} \\ & \mid & \mathtt{Any} & \text{Polymorphic Any Contract} \end{array}$$

Figure 3: Syntax of contracts

Having discussed the language of programs, we now discuss the language of contracts. Figure 3 gives their syntax. For reasons of notational simplicity, we only restrict data constructor contracts to pairs only, but the idea generalises readily.

4.1 Contract Satisfaction

We give the semantics of contracts by defining "e satisfies t", written $e \in t$, in Figure 4. This is a purely declarative specification of contract satisfaction, that says *which* terms satisfy a contract, without saying *how* a satisfaction check might be performed. We regard the ability to give a simple, declarative, programmer-accessible specification of contract satisfaction as very important, but it is a property that few related works share, with the notable and inspiring exception of [3]. As that paper says

> The structure of a non-compositional semantics like [the Findler-Felleisen wrapping algorithm] is difficult to understand. With just Definition 1 [which says that a term satisfies a contract if its wrapping cannot crash] to hand, an answer to the question "Does e satisfy t?" is not easy because it involves consideration of every possible context. Nor can we

For a well-typed expression e, define $e \in t$ thus:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \not\to^* \{\mathtt{BAD}, \mathtt{False}\}) \quad [A1]$$

$$e \in x{:}\, t_1 \to t_2 \iff \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \quad [A2]$$

$$e \in (t_1, t_2) \iff e\uparrow \text{ or } (e \to^* (e_1, e_2) \text{ and } e_1 \in t_1, e_2 \in t_2) \quad [A3]$$

$$e \in \mathtt{Any} \iff \mathtt{True} \quad [A4]$$

Figure 4: Contract Satisfaction

ignore this problem, since in our experience most people's intuition differs from [Definition 1].

To a first approximation, the rules in Figure 4 should be self-explanatory. For example, e satisfies $\{x \mid p\}$ if $p[e/x]$ evaluates to `True`. More interestingly, e satisfies the (non-dependent) function contract $t_1 \to t_2$ iff $(e\ e_1)$ satisfies t_2 for any term e_1 satisfying t_1. To get *dependent* function contracts we must simply remember to substitute $[e_1/x]$ in t_2. However, these definitions are carefully crafted at the edges, and we now discuss the less-obvious choices.

In Figure 4, both e and t may mention functions bound in the top-level definitions Δ. These functions are necessary for the evaluation relation of rule [A1] to make sense. To reduce clutter, we do not make these top-level bindings explicit, by writing $\Delta \vdash e \in t$, but instead allow rule [E-top] of Figure 2 to consult Δ implicitly.

4.2 Diverging terms

The definitions in Figure 4 specify that a divergent term e satisfies *every* contract. This choice is expressed directly for predicate contracts $\{x \mid p\}$ and tuple contracts (t_1, t_2), and is an easy consequence for function contracts. We made this choice because otherwise we would often have to prove termination in order to prove that $e \in t$. For example:

```
f x = if x < 10 then x else f (x/2)
```

Does $\mathtt{f} \in \mathtt{Ok} \to \{x \mid x < 10\}$? The `then` branch clearly satisfies the postcondition but what about the `else` branch? Specifying that divergence satisfies any contract allows us to answer "yes" without proving termination. Furthermore, despite divergence, a caller of `f` can still rely on `f`'s postcondition:

```
g y = if (f y > 10) then error "Urk" else True
```

Here `g` cannot crash, because `f` guarantees a result less than 10, or else diverges.

Our choice has the nice consequence that *every contract is inhabited* (by divergence). This matters. Consider whether $(\lambda x.\mathtt{BAD})$ satisfies $\{x \mid \mathtt{False}\} \to \mathtt{Ok}$. If $\{x \mid \mathtt{False}\}$ was uninhabited, the answer would be "yes". But that choice is incompatible with building a rigorous connection (sketched in §2.5) between contract satisfaction and Findler-Felleisen-style wrapping. Indeed, Findler and Blume are forced to invent an awkward (and entirely informal) predicate form "`non-empty-predicate`" [9], which we do not need.

4.3 Only crash-free terms satisfy predicate contracts

The alert reader will notice that [A1] specifies that *only crash-free terms satisfy a predicate contract* $\{x \mid p\}$. This means that the contract $\{x \mid \mathtt{True}\}$, which we abbreviate to `Ok`, is satisfied

precisely by the crash-free terms. Even the identity function only guarantees a crash-free result if it is given a crash-free argument!

Other choices are possible, but we postpone the discussion to §5.4, when we have more scaffolding in place.

4.4 The `Any` contract

If we only have [A1]-[A3], the expression `BAD` would not satisfy any contract. In a lazy language this is much too conservative, so in [A4] we introduce a special contract, named `Any`, which is satisfied by expression including `BAD`. We saw an example of its usefulness in §2.2:

```
{-# CONTRACT fst :: (Ok, Any) -> Ok #-}
fst (x,y) = x
```

`Any` is also useful in post-conditions: a function whose postcondition is `Any` is a function that always crashes. Haskell programmers often write packaged versions of Haskell's `error` function, such as

```
myError :: String -> a
{-# CONTRACT myError :: Ok -> Any #-}
myError s = error ("Fatal error: " ++ s)
```

So `BAD` satisfies `Any`. In fact, `BAD` satisfies *only* the contract `Any` because it fails the constraints stated in [A1]-[A3]:

$$\mathtt{BAD} \notin (\mathtt{Any}, \mathtt{Any})$$
$$\mathtt{BAD} \notin \mathtt{Any} \to \mathtt{Any}$$

In [9], Findler and Blume point out that it is desirable to have a contract such as `Any` which is not equivalent to $\{x \mid \mathtt{True}\}$, but there is no detailed formalization on `Any` contract in any of their work.

4.5 Open Expressions

We have mentioned that e and t may mention functions bound in the top-level environment. These functions participate in the evaluation of rule [A1]. But suppose that the programmer declares

```
{-# CONTRACT f :: {x | x>0} -> Ok #-}
f = ...
```

When checking the contracts of a function `g` that calls `f`, we should presumably assume only `f`'s declared contract, *without looking at its actual definition*. Doing so is more modular, and allows the programmer to leave room for future changes by specifying a contract that is more restrictive than the current implementation.

This goal is easily achieved. Suppose the declared contracts for `f` and `g` are t_f, t_g respectively, and the definition of `g` is $\mathtt{g} = e_g$ where `f` is called in e_g. Then, instead of checking that $e_g \in t_g$, we check that

$$(\lambda f. e_g) \in t_f \to t_g$$

That is, simply lambda-abstract over any variables free in e_g that have declared contracts. This approach also allows the programmer to omit a contract specification (just as type signatures are often omitted), in which case the contract checker can "look inside" the definition when proving the correctness of calls to that function. The exact details are a software engineering matter; our point here is that the underlying infrastructure allows a variety of choices.

The same technique simplifies the problem of checking satisfaction for recursive functions. If the programmer specifies the contract t_f for a definition $\mathtt{f} = e$, then it suffices to check that

$$\lambda f. e \in t_f \to t_f$$

which is easier because $\lambda f.e$ does not call `f` recursively. There is nothing new here – it is just the standard technique of loop invariants in another guise – but it is packaged very conveniently.

$$r_1, r_2 \in \{\texttt{BAD}, \texttt{UNR}\} \qquad\qquad e \triangleright t = e \overset{\texttt{BAD}}{\underset{\texttt{UNR}}{\bowtie}} t \qquad\qquad e \triangleleft t = e \overset{\texttt{UNR}}{\underset{\texttt{BAD}}{\bowtie}} t$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \;=\; e \; \text{`seq`} \; \texttt{case} \; (\texttt{fin}_0 \, p[e/x]) \; \texttt{of} \; \{\texttt{True} \to e; \; \texttt{False} \to r_1\} \qquad \text{[P1]}$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x : t_1 \to t_2 \;=\; e \; \text{`seq`} \; \lambda v. ((e \, (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \; \overset{r_1}{\underset{r_2}{\bowtie}} \; t_2 [v \overset{r_2}{\underset{r_1}{\bowtie}} t_1 / x]) \qquad \text{[P2]}$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} (t_1, t_2) \;=\; \texttt{case} \; e \; \texttt{of} \; (x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1, \; x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2) \qquad \text{[P3]}$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any} \;=\; r_2 \qquad \text{[P4]}$$

Figure 5: Projection Definition

5. Contract Checking

So far we have a nice declarative specification of when a term satisfies a contract. If we could statically check such claims, we would have a powerful tool. For example, if we could show that $\texttt{main} \in \texttt{Ok}$, then we would have proved that the entire program is crash-free.

So far we have no algorithm for that can *check* a claim that $e \in t$. In their ground-breaking paper, Findler & Felleisen describe how to "wrap" a term in a contract-checking wrapper, that reports checks (a) that the term obeys its contract, and (b) that the context of the term respects the contract [10]. One route to checking contract satisfaction statically, was first proposed in [34] and sketched in §2.5, is to wrap the term in the Findler-Felleisen way, and check that the resulting term cannot crash.

This leads to our main theorem:

THEOREM 1 (Soundness and Completeness of Contract Checking). *For all expressions e, and well-formed contracts t,*

$$(e \triangleright t) \text{ is crash-free} \quad \Longleftrightarrow \quad e \in t$$

The free variables of e and t must all be bound by the top-level environment Δ, which we leave implicit as before. The form $(e \triangleright t)$ wraps e in a Findler-Felleisen-style contract checker, specified in Figure 5. As in the case of contract satisfaction, there are tricky details, as we discuss in §5.1. Another subtle but important point is the requirement that the contract t be "well-formed"; this deals with contracts that crash and is discussed in §6.1.

Theorem 1 is very strong. It states that the wrapped term $e \triangleright t$ is crash-free *when and only when* $e \in t$. Certainly, then, if we can prove that $e \triangleright t$ is crash-free, we have proved that $e \in t$. But how can we prove that $e \triangleright t$ is crash-free? There are many ways to do this, and doing so is not the focus of this paper. Briefly, the approach we take is to perform meaning-preserving transformations on $e \triangleright t$, of precisely the kind that an optimising compiler might perform (inlining, β-reduction, constant folding, etc). If we can "optimise' (i.e. symbolically simplify) the term to a form that is syntactically safe (§3.3), then we are done. The better the optimiser, the more contract satisfaction checks will succeed – but none of that affects Theorem 1. For details see [34].

5.1 Wrapping

Our goal is to define $e \triangleright t$ such that Theorem 1 holds. Figure 5 gives the definition of a single, general combinator $\bowtie$, and two notational abbreviations $\triangleright$ and $\triangleleft$. None of these operators are part of the syntax of expressions (Figure 1); rather they are thought of as macros, which expand to a particular expression. Informally:

$$e \overset{r_1}{\underset{r_2}{\bowtie}} t$$

is a term that behaves just like e, except that it throws exception r_1 if e does not respect t, and throws exception r_2 if the wrapped term is used in a way that does not respect t. In the vocabulary of "blame", r_1 means "blame e" while r_2 means "blame the context". Figure 5 defines the convenient abbreviations

$$e \triangleright t = e \overset{\texttt{BAD}}{\underset{\texttt{UNR}}{\bowtie}} t \qquad\qquad e \triangleleft t = e \overset{\texttt{UNR}}{\underset{\texttt{BAD}}{\bowtie}} t$$

So $e \triangleright t$ crashes (with BAD) if e does not satisfy t, and diverges (with UNR) if the context does not respect t.

Temporarily ignoring the occurrences of 'seq' and 'fin', the main structure of Figure 5 is standard from earlier works [10], and we do not belabour it here. In particular, note the inversion of r_1 and r_2 in the expansion of function contracts. The wrapping of Any, while new, is also obvious after a moment's thought. For example,

```
fst ▷ (Ok, Any) → Ok
= λv.((fst (v ◁ (Ok, Any))) ▷ Ok)
= λv.((fst (v ◁ (Ok, Any))))
= λv.((fst (case v of (a, b) → (a ◁ Ok, b ◁ Any))))
= λv.((fst (case v of (a, b) → (a, BAD))))
```

Here we have used the fact that $e \bowtie \texttt{Ok} = e \bowtie \{x \mid \texttt{True}\} = e$. That is, considered as a wrapper Ok does nothing at all. In this example we see that the wrapper replaces the second component of the argument to fst with BAD, so that if fst should ever look at it, the program will crash. That is exactly right, because the contract says that the second component can be anything, with contract Any.

5.2 The use of seq

The reader may wonder about the uses of seq in [P1] and [P2] of Figure 5. The Haskell function seq (short for "sequence") is defined as follows:

$$e_1 \; \text{`seq`} \; e_2 \;=\; \texttt{case} \; e_1 \; \texttt{of} \; \{ \texttt{DEFAULT} \to e_2 \}$$

It is necessary in the definition of $\bowtie$ to ensure that Theorem 1 holds for (a) divergent and (b) crashing terms. For example, if bot is a diverging term (defined by bot = bot), then Figure 4 says that $\texttt{bot} \in \{x \mid \texttt{False}\}$. But if [P1] lacked the seq, we would have

```
bot ▷ {x | False}
= case False of { True -> bot; False -> BAD }
= BAD,   which is not crash-free
```

thus contradicting Theorem 1.

Dually, we must ensure that $\texttt{BAD} \notin \texttt{Ok} \to \texttt{Any}$. Without the seq in [P2] we would get

```
BAD ▷ Ok → Any = λv. ((BAD (v ◁ Ok)) ▷ Any)
               = λv. UNR,   which is crash-free
```

again contradicting Theorem 1. Have we covered all the cases? A quick check shows that for any contract t, $\texttt{BAD} \notin t$ and $\texttt{UNR} \in t$,

which is reassuring. More solidly, Theorem 1 goes through with the definitions of Figure 5.

5.3 Properties of Contracts

Our contract combinators possess many beautiful properties. We summarise these results in Figure 6. It took us some while to evolve a set of definitions for $\in$, $\bowtie$, etc that validated such crisp results. These lemmas form a basis for proving our main result: Theorem 1. Theorem 1 is similar to the soundness and completeness theorems in [3]. We have a complete proof of the theorem in [35]. The proving technique is different from that in [3].

Figure 6 employs a useful ordering over expressions, called *crashes more often*:

DEFINITION 6 (Crashes more often). e_1 *crashes more often than* e_2, *written* $e_1 \preceq e_2$, *iff for all closing contexts* $\mathcal{C}$

$$\mathcal{C}[\![e_2]\!] \to^* \text{BAD} \quad \Rightarrow \quad \mathcal{C}[\![e_1]\!] \to^* \text{BAD}$$

Informally e_1 crashes more often than e_2 if they behave in exactly the same way except that e_1 may crash when e_2 does not. (The definition does not look as strong as that, but as every theorist knows, it is. For example, could e_1 produce True while e_2 produces False? No, because we could find a context $\mathcal{C}$ that would make $\mathcal{C}[\![e_2]\!]$ crash while $\mathcal{C}[\![e_1]\!]$ does not.)

The proof of our main Theorem 1 is by induction on the size of the contract t. To give a flavour of the proof we show the only interesting case here, that for function contracts when $e \to^* val \notin \{\text{BAD}, \text{UNR}\}$. We give only the proof for *non-dependent* function contracts because the dependent case is more intricate and it is harder to see the wood for the trees. Full proofs can be found in [35]. The **cf** stands for "crash-free".

$$e \rhd t_1 \to t_2 \text{ is } \mathbf{cf}$$
$$\Longleftrightarrow \quad \text{(By definition of } \rhd\text{)}$$
$$e \text{ `seq` } \lambda v.(e\,(v \lhd t_1)) \rhd t_2 \text{ is } \mathbf{cf}$$
$$\Longleftrightarrow \quad \text{(Since } e \to^* val \notin \{\text{BAD}, \text{UNR}\}\text{)}$$
$$\lambda v.(e\,(v \lhd t_1)) \rhd t_2 \text{ is } \mathbf{cf}$$
$$\Longleftrightarrow \quad \text{(Property of crash-freeness)}$$
$$(\dagger) \quad \forall \text{ closed, } \mathbf{cf} \; e'.\; (e\,(e' \lhd t_1)) \rhd t_2 \text{ is } \mathbf{cf}$$

Now the proof splits into two. In the reverse direction, we start with the assumption $e \in t_1 \to t_2$:

$$e \in t_1 \to t_2$$
$$\Rightarrow \quad \text{(By defn of } e \in t\text{)}$$
$$\forall e_1 \in t_1.\; (e\,e_1) \in t_2$$
$$\Rightarrow \quad \text{(By Key Lemma (Figure 6))}$$
$$\forall \text{ closed, } \mathbf{cf} \; e'.\; (e\,(e' \lhd t_1)) \in t_2$$
$$\Longleftrightarrow \quad \text{(By induction)}$$
$$(\dagger) \quad \forall \text{ closed, } \mathbf{cf} \; e'.\; (e\,(e' \lhd t_1)) \rhd t_2 \text{ is } \mathbf{cf}$$

and now we have reached the desired conclusion ($\dagger$). In the forward direction, we start with ($\dagger$):

$$(\dagger) \quad \forall \text{ closed, } \mathbf{cf} \; e'.\; (e\,(e' \lhd t_1)) \rhd t_2 \text{ is } \mathbf{cf}$$
$$\Rightarrow \quad \text{(By induction, } e_1 \in t_1 \Rightarrow e_1 \rhd t_1 \text{ is } \mathbf{cf}\text{)}$$
$$\forall \text{ closed } e_1 \in t_1.\; (e\,((e_1 \rhd t_1) \lhd t_1)) \rhd t_2 \text{ is } \mathbf{cf}$$
$$\Rightarrow \quad \text{(By Projection Pair and Congruence (Figure 6))}$$
$$\forall e_1 \in t_1.\; (e\,e_1) \rhd t_2 \text{ is } \mathbf{cf}$$
$$\Rightarrow \quad \text{(By induction)}$$
$$\forall e_1 \in t_1.\; (e\,e_1) \in t_2$$
$$\Longleftrightarrow \quad \text{(by definition of } \in\text{)}$$
$$e \in t_1 \to t_2$$

There are two key steps in this short sequence. First, we choose a *particular* crash-free e', namely $(e_1 \rhd t_1)$ where $e_1 \in t_1$. The second step is the appeal to the Projection Pair lemma, which itself is non-trivial.

5.4 Why only crash-free terms satisfy predicate contracts

In §4.3 we promised to explain why we chose to allow only *crash-free* terms to satisfy a predicate contract, regardless of the predicate. An obvious alternative design choice for contract satisfaction would be to drop the "e is crash-free" condition in the predicate contract case:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } p[e/x] \not\to^* \{\text{BAD}, \text{False}\} \quad \text{[B1]}$$

Then we could get rid of Any, because $\{x \mid \text{True}\}$ would do instead. On the other hand, a polymorphic contract meaning "crash-free" is extremely useful in practice, so we would probably need a new contract Ok (now not an abbreviation) defined thus:

$$e \in \text{Ok} \iff e \text{ is crash-free} \quad \text{[B2]}$$

This all seems quite plausible, but it has a fatal flaw: *we could not find a definition for $\rhd$ that validates our main theorem*. That is, our chosen definition for $\in$ makes Figure 5 work out, whereas the otherwise-plausible alternative appears to prevent it doing so.

Suppose we have [B1] instead of [A1], that means $(K\,\text{BAD}) \in \{x \mid \text{True}\}$. However, according to [P1], we have $(K\,\text{BAD}) \rhd \{x \mid \text{True}\} = (K\,\text{BAD})$ which is not crash-free. This means Theorem 1 fails. In order to make Theorem 1 hold, we might define $e \rhd \{x \mid p\}$ as follows.

$$e \rhd \{x \mid p\} \;=\; e \text{ `seq` case fin } p[e/x] \text{ of} \quad \text{[Q1]}$$
$$\text{True} \to ()$$
$$\text{False} \to \text{BAD}$$

so that $(K\,\text{BAD}) \rhd \{x \mid \text{True}\} = ()$ which is crash-free. But we know $\text{BAD} \in \{x \mid \text{True}\}$ and we have $\text{BAD} \rhd \{x \mid \text{True}\} = \text{BAD}$ which is not crash-free. We might change [Q1] by removing the `seq`. However, without the `seq`, we have $\text{UNR} \in \{x \mid \text{False}\}$, but $\text{UNR} \rhd \{x \mid \text{False}\} = \text{BAD}$, which is not crash-free. So if we use [B1], we do not know how to define $\rhd$ such that Theorem 1 holds.

6. Contracts that crash

Our goal is to detect crashes in a *program* with the help of contracts, we do not expect contracts themselves to introduce crashes. One approach, taken by Blume & McAllester [3], is to prohibit a contract from mentioning any function that might crash. But that is an onerous restriction, as we argued in §2.4.

It is attractive simply to allow arbitrary crashes in contracts; after all, Figure 4 specifies exactly which terms inhabit even crashing contracts. Alas, if we drop the (still-to-be-defined) condition "well-formed contract" from Theorem 1, the ($\Rightarrow$) direction still holds, but the ($\Leftarrow$) direction fails. Here is a counter-example involving a crashing contract. We know that:

$$\lambda x.x \in \{x \mid \text{BAD}\} \to \text{Ok}$$

because the only expression that satisfies $\{x \mid \text{BAD}\}$ is an expression that diverges and a diverging expression satisfies Ok. But we have:

$$\lambda x.x \rhd \{x \mid \text{BAD}\} \to \text{Ok} = \lambda v.(\lambda x.x\,(v \lhd \{x \mid \text{BAD}\}))$$
$$= \lambda v.(v \lhd \{x \mid \text{BAD}\})$$
$$= \lambda v.(v \text{ `seq` BAD})$$
$$\text{which is not crash-free}$$

Congruence	$\forall e_1, e_2.\ e_1 \preceq e_2 \iff \forall \mathcal{C}.\ \mathcal{C}[\![e_1]\!] \preceq \mathcal{C}[\![e_2]\!]$
Conditional Projection (w.r.t. $\preceq, \succeq$)	For all e and well-formed t, if $e \in t$, then (a) $e \triangleleft t \preceq e$; (b) $e \triangleright t \succeq e$.
Key Lemma	For all crash-free e, well-formed t, $e \triangleleft t \in t$.
Monotonicity of $\in$	If $e_1 \in t$ and $e_1 \preceq e_2$, then $e_2 \in t$
Idempotence	$\forall e, t.$ (a) $e \triangleright t \triangleright t \equiv e \triangleright t$ (b) $e \triangleleft t \triangleleft t \equiv e \triangleleft t$
Projection Pair	$\forall e, t.\ e \triangleright t \triangleleft t \preceq e$
Closure Pair	$\forall e, t.\ e \preceq e \triangleleft t \triangleright t$
Telescoping Property	For all e, well-formed t. $(e \underset{r_2}{\overset{r_1}{\bowtie}} t) \underset{r_4}{\overset{r_3}{\bowtie}} t = e \underset{r_4}{\overset{r_1}{\bowtie}} t$

Figure 6: Properties of $\triangleright$ and $\triangleleft$

6.1 Well formed contracts

So unrestricted crashes in contracts invalidates (one direction of) Theorem 1. But no one is asking for unrestricted crashes! For example, this contract doesn't make much sense:

$$t_{bad} = xs : \mathtt{Ok} \to \{r \mid r > \mathtt{head}\ xs\}$$

What is it supposed to do if the argument list is empty? Much more plausible is a contract like this (see §2.4):

$$t_{good} = xs : \{xs \mid \mathtt{not}\ (\mathtt{null}\ xs)\} \to \{r \mid r > \mathtt{head}\ xs\}$$

which specifies that the argument list is non-empty, and guarantees to return a result bigger than head of the argument. Thus motivated, we define a notation of a "well-formed" contract:

DEFINITION 7 (Well-formed contract). *A contract t is well-formed iff*

> *t is $\{x \mid p\}$ and p is crash-free*
> *or t is $x : t_1 \to t_2$ and t_1 is well-formed and*
> *for all $e_1 \in t_1, t_2[e_1/x]$ is well-formed*
> *or t is (t_1, t_2) and both t_1 and t_2 are well-formed*
> *or t is* Any

The definition is essentially the same as that of T_{safe} in [3], although perhaps a little more straightforward. It simply asks that the predicates in a contract are crash-free *under the assumption that the dependent function arguments satisfy their contracts*. So, under this definition, t_{bad} is ill-formed while t_{good} is well-formed. The latter is well-formed because head xs is crash-free for every xs that satisfies not (null xs).

6.2 Wrapping dependent function contracts

Recall [P2] from Figure 5:

$$e \underset{r_2}{\overset{r_1}{\bowtie}} x : t_1 \to t_2 = e\ \mathtt{`seq`}\ \lambda v.\ ((e\ (v \underset{r_1}{\overset{r_2}{\bowtie}} t_1)) \underset{r_2}{\overset{r_1}{\bowtie}} t_2[(v \underset{r_1}{\overset{r_2}{\bowtie}} t_1)/x])$$

Notice that v is wrapped by $v \underset{r_1}{\overset{r_2}{\bowtie}} t_1$ even in the contract t_2, as well as in the argument to e. Could we simplify [P2] by omitting this wrapping, thus?

$$e \underset{r_2}{\overset{r_1}{\bowtie}} x : t_1 \to t_2 = e\ \mathtt{`seq`}\ \lambda v.\ ((e\ (v \underset{r_1}{\overset{r_2}{\bowtie}} t_1)) \underset{r_2}{\overset{r_1}{\bowtie}} t_2[v/x])$$

No, we could not: Theorem 1 would fail again. Here is a counter-example.

```
{-# CONTRACT h :: {x | not (null x)}
              -> {r | head x == r} #-}
h (y:ys) = y
```

Now h satisfies its contract t_h, but $h \triangleright t_h$ is not crash-free, as the reader may verify.

We remarked earlier that Blume & McAllester require that contracts only call crash-free functions. But the wrapping of v inside t_2 in rule [P2] might itself introduce crashes, at least if t_2 uses x in

a way that does not respect t_1. They therefore use another variant of [P2], as follows:

$$e \underset{r_2}{\overset{r_1}{\bowtie}} x : t_1 \to t_2 = e\ \mathtt{`seq`}\ \lambda v.\ ((e\ (v \underset{r_1}{\overset{r_2}{\bowtie}} t_1)) \underset{r_2}{\overset{r_1}{\bowtie}} t_2[(v \underset{\mathtt{UNR}}{\overset{r_2}{\bowtie}} t_1)/x])$$

Notice the "UNR" introduced out of thin air in the wrapping of v in t_2, which is enough to maintain their no-crashing invariant. Happily, if the contracts are well-formed (which we need anyway, so that it is possible to call head) there is no need for this somewhat ad-hoc fix.

6.3 Practical consequences

One might worry that the well-formedness condition in Theorem 1 makes the verification task more onerous: perhaps to prove $e \in t$ now we must check two things (a) that t is well formed and (b) that $e \triangleright t$ is crash-free. Happily, this is not necessary, because the ($\Rightarrow$) of Theorem 1 still holds:

THEOREM 2. *For all closed expression e, for all contract t,*

$$(e \triangleright t)\ \text{is crash-free} \quad \Rightarrow \quad \begin{array}{ll} (a) & e \in t \\ (b) & t\ \text{is well-formed} \end{array}$$

So if we can prove that $e \triangleright t$ is crash-free, the contract must be well-formed.

7. Contracts that diverge

Our system allows non-termination both in the *programs* we verify, and in their *specifications* (contracts), which is most unusual for a system supporting static verification.

As we discussed in §4.2, we allow non-termination for *programs* because we work with a real-life programming language, in which many functions actually do not terminate. We do not want to exclude non-termination in general, even for specifications, because we do not want to be forced to perform termination proofs, which are often tedious to do. Since the current advances in automatic termination proofs are still limited, especially for lazy programs, requiring termination would put a substantial extra burden on the user of our system.

What about divergent *contracts*? Many program verification systems for functional programming, such as HOL, systems based on dependent types (Coq, Agda), and ACL2, do not allow *any* non-terminating definitions. The main reason is that this introduces an immediate unsoundness in these systems. For example, by an (unsound) induction proof, a constant defined as let x = x could be proven equal to both 1 and 2, concluding that 1=2.

But it would be an onerous burden to insist that all contracts terminate, because the programmer can write arbitrary Haskell in contracts, and proving termination of arbitrary Haskell programs is hard. Furthermore, allowing non-termination in specifications is of direct benefit. Consider a function zipE which requires two inputs to have the same length:

```
{-# CONTRACT zipE :: xs:Ok
                  -> {ys | sameLen xs ys} -> Ok #-}
 zipE []      []     = []
 zipE (x:xs) (y:ys) = (x,y) : zipE xs ys
 zipE _       _      = error ''unequal lengths''

 sameLen []      []     = True
 sameLen (x:xs) (y:ys) = sameLen xs ys
 sameLen _       _      = False
```

Here, two infinite lists satisfy the contract for ys, since (sameLen xs ys) diverges, and indeed zipE does not crash for such arguments. Care is necessary in writing the contract: if we had instead said length xs == length ys, the contract would diverge if only *one* argument was infinite, but zipE would crash for such arguments, so it would not satisfy this alternative contract. In this way, *safety properties* over infinite structures are allowed. (Safety properties are properties that always have finite counter-examples whenever there exists any counter-example.)

Why is our approach sound? First, any contract we verify for a program only deals with partial correctness. In other words, all contracts are inhibited by non-terminating programs as well. Second, our system does not include reasoning mechanisms like equational reasoning. In fact, our system has no means of expressing equality at all! Everything is expressed in terms of Haskell expressions evaluating to boolean values, crashing, or not-terminating. Third, any specification that does not terminate is semantically the same as a True contract. Why? Because of the mysterious fin construct, as we discuss next.

7.1 Using fin in contract wrappers

Suppose we have the top-level definition bot = bot; that is, bot diverges. Now consider $e = $ (BAD,BAD) and $t = \{x \mid \text{bot}\}$. Then $e \notin t$ (since e is not crash-free). If we did not use fin in the definition of $\bowtie$ (Figure 5), $e \triangleright t$ would reduce to this term:

```
case bot of { True -> (BAD, BAD); False -> BAD }
```

This term is contextually equivalent to bot itself, and so it is crash-free, contradicting our main theorem (§5).

What to do? Execution has gotten stuck evaluating the diverging contract, and has thereby missed crashes in the term itself. Our solution is to limit the work that can be spent on contract evaluation. The *actual* definition of $\bowtie$ makes $e \triangleright t$ equal to

```
case (fin₀ bot) of
  { True  -> (BAD, BAD); False -> BAD }
```

The operational semantics of fin (Figure 2) gives a finite M units of "fuel" to each fin. Each reduction under a fin increases the subscript on the fin until it reaches the maximum M (rule [E-fin1]). When the fin subscript n reaches the limit M, fin gives up and returns True (rule [E-fin3]). Now, in the above example, *for any finite M*, we have

$$e \triangleright t \rightarrow_M^* \text{(BAD,BAD)}$$

So we define our full-scale reduction relation $\rightarrow^*$ in terms of $\rightarrow_M^*$:

DEFINITION 8. *We say that $e \rightarrow^* val$ iff there exists N such that for any $M \geq N$ we have $e \rightarrow^* val$.*

Under this definition, $e \triangleright t \rightarrow^*$ (BAD,BAD), and Theorem 1 holds.

7.2 Practical consequences

This may all seem a bit complicated or artificial, but it is very straightforward to implement. First, remember that we are concerned with static verification, not dynamic checking. Uses of fin

are introduced only to check contract satisfaction, and are never executed in the running program. Second, our technique to check that $e \triangleright t$ is crash-free is to optimise it and check for syntactic safety. To be faithful to the $\rightarrow^*$ semantics, we need only *refrain* from "optimising" (case (fin bot) of <alts>) to bot, thereby retaining any BADs lurking in <alts>. Since this particular optimisation is a tricky one anyway, it is quite easy to omit! In other words, in our static contract checking, we can safely omit fin and the rules [E-fin1,2,3].

7.3 Summary

By being careful with our definition of $\rightarrow^*$, we can retain Theorem 1 in its full, bi-directional form. This approach is, of course, only available to us because we are taking a static approach to verification. A dynamic checker cannot avoid divergence in contracts (since it must evaluate them), and hence must lose the ($\Rightarrow$) direction of Theorem 1, as indeed is the case in [3].

8. Error Reporting

Given $f \in t$, we have shown that to check $f \in t$, we check "$f \triangleright t$ is crash-free" instead. What happens if $f \triangleright t$ is not crash-free? That means when we try to optimize the term $f \triangleright t$ to some e' and there are some residual BADs in e', what we can report to the programmer from the contract violation?

To know which function to blame [10], we need to give each BAD a tag. That is BAD *lbl* where the label *lbl* is a function name. For a function f with contract t, we check $f \triangleright t$ is crash-free or not, where

$$f \triangleright t = f \overset{\text{BAD "f"}}{\underset{\text{UNR}}{\bowtie}} t$$

A residual BAD "f" tells us that we should blame f. That means assuming f takes arguments satisfying their corresponding preconditions, f fails to produce a result that meets its postcondition. For example:

```
{-# CONTRACT inc :: {x | x > 0} -> {r | r > x} #-}
inc x = x - 1
```

after optimizing (i.e. simplifying) inc $\triangleright$ t_{inc}, we have:

```
\v -> (case v > 0 of
        True -> case v - 1 > v of
                  True -> case v > 0 of
                            True -> v - 1
                            False -> UNR
                  False -> BAD "inc"
        False -> UNR)
```

We can report to programmers during compile-time:

```
Error: inc fails its postcondition
       when v > 0 holds
            v - 1 > v does not hold
```

This error message is generated directly from the path that leads to the BAD "inc".

```
case v > 0 of
  True -> case v - 1 > v of
            False -> BAD "inc"
```

How about precondition violation? Suppose we know $f \in t_f$. If f is called in a function g with contract t_g, recalling the reasoning in §4.5, we shall check $\lambda f.e_g \in t_f \rightarrow t_g$. That means we check: $\lambda f.e_g \triangleright t_f \rightarrow t_g$ is crash-free or not. By the definition of $\triangleright$, we have: $\lambda f. ((e_g \ (f \triangleleft t_f)) \triangleright t_g)$.

In order to trace "which function calls which function that fails which function's precondition", instead of using $(f \triangleleft t_f)$, we use:

$$\text{Inside } lbl \; loc \; (f \triangleleft t_f)$$

The lbl is the function name ("f" in this case) and the loc indicates the location (e.g. (row,column)) of the definition of f in the source file. Note that

$$f \triangleleft t = f \underset{\text{BAD "f"}}{\overset{\text{UNR}}{\bowtie}} t$$

For example, we have:

```
{-# CONTRACT f1 :: {x | True} -> {z | x < z}
                -> {r | True} #-}
f1 x z = if x < z then 5 else error "Urk"
f2 x z = 1 + f1 x z

f3 [] z = 0
f3 (x:xs) z = case x > z of
                True -> f2 x z
                False -> ...
```

After optimizing $\mathtt{f3} \triangleright t_{\mathtt{f3}}$, we have:

```
\xs -> \z ->
 case xs of
 [] -> 0
 (x:y) -> case x > z of
         True -> Inside "f2" <l2>
                   (Inside "f1" <l1> (BAD "f1"))
         False -> ...
```

Note that, the "f1" in (BAD "f1") indicates which function's precondition is not fulfilled. Thus, the residual fragment enables us to give one counter-example with the following meaningful message at compile-time:

```
Error <l3>: f3 (x:y) z
            when x > z holds
            calls f2
            which calls f1
            which may fail f1's precondition!
```

where the location <l3> indicates the location of the definition of f3 in the source file.

This error tracing technique is adapted from ESC/Haskell [34], which achieves the same goal as that in [22] but in a much simpler way.

9. Related Work

Static verification of software is a field dense with related work, of which we can only summarise a limited fraction here.

9.1 Type systems

In the functional language community, type systems have played significant roles in guaranteeing better software safety. Advanced type systems, such as dependent types, have been advocated to capture stronger properties. While full dependent type system (such as Cayenne [1]) is undecidable in general, Xi and Pfenning [33] have designed a smaller fragment based on indexed objects drawn from a constraint domain $\mathcal{C}$ whose decidability closely follows that of the constraint domain. Typical examples of objects in $\mathcal{C}$ include linear inequalities over integers, boolean constraints, or finite sets. In a more recent Omega project [28], Sheard shows how extensible kinds can be built to provide a more expressive dependent-style system. In comparison, our approach is much more expressive and programmer friendly as we allow arbitrary functions to be used in the pre/post annotations without the need to encode

them as types. It is also easier for programmers to add properties incrementally. Moreover, our symbolic evaluation is formulated to adhere to lazy semantics and is guaranteed to terminate when code safety is detected or when a preset bound on the unrollings of each recursive function is reached. Compared with the dependent type approaches [33, 6, 28, 32] in general, we separate type and contract declarations so that type related work (e.g. type inference) and contract related techniques can be developed independently.

In Hoare Type Theory (HTT) [26, 25], higher-order predicates and recursive predicates can be used in specifications. Another work along this line is [27]. We allow higher-order functions and recursive functions to be used in contracts so in terms of these two aspects, we share the same expressiveness. As we use symbolic execution, inlining and induction make the job of verifying such contracts much easier compared with theorem proving higher-order predicates and recursive predicates. But our contracts do not contain quantifiers while quantifiers are supported in [26, 25, 27].

9.2 Extended static checking

In an inspiring piece of work [21, 13, 12, 4], Flanagan et al, showed the feasibility of applying an extended static checker (named ESC/Java) to Java. Since then, several other similar systems have been further developed, including Spec#'s and its automatic verifier Boogie [2] that is applicable to the C# language. We adapt the same idea of allowing programmers to specify properties about each function (in the Haskell language) with pre/post annotations, but also allow pre/post annotations to be selectively omitted where desired. Furthermore, unlike previous approaches based on verification condition (VC) generation which rely solely on a theorem prover to verify, we use an approach based on symbolic evaluation that can better capture the intended semantics of a more advanced lazy functional language. With this, our reliance on the use of theorem provers is limited to smaller fragments that involve the arithmetical parts of expressions. Symbolic evaluation gives us much better control over the process of the verification where we have customised sound and effective simplification rules that are augmented with counter-example guided unrolling. More importantly, we are able to handle specifications involving recursive functions and/or higher-order functions which are not supported by either ESC/Java or Spec#.

9.3 Contracts

Meyer first established the term "contract" in Eiffel [23]. More recently, Findler and Felleisen introduced the notion of higher-order contracts, including a careful treatment of "blame" [10]. This paper unleashed a new wave of papers about contract checking in higher order languages, including [3, 9, 3, 16, 31, 11, 20, 19, 15]. Although they share a common foundation, these papers differ in their notation and approach, which makes like-for-like comparisons difficult.

Of these papers, the work of Blume and McAllester [3, 3] is by far the most closely related because they give a declarative semantics for contract satisfaction, and prove a connection with the dynamic wrappers of Findler and Felleisen. Here is a brief summary of the differences between some of this work, especially [3], and our own:

- We aim at static contract checking, for a statically typed language, whereas most of the related work deals with dynamic checks, or a a hybrid checking strategy for a dynamically typed language.

- We deal with a lazy language; all other related work is for a strict languages. In particular, we give a crashing expression a contract Any while a contract is only given to non-crashing expressions in [10, 3, 9].

- We deal with dependent function contracts which [10, 9] do not.

- We lay great emphasis on crashing and diverging contracts, which are either not the focus of these other works, or are explicitly excluded. Our solution appears both less restrictive than [3], by allowing crashing functions to be called within contracts, and supports Theorem 1 in both directions, rather than the ($\Leftarrow$) direction only.

- The telescoping property (Figure 6) is first discovered in [3], but it does not seem to be used in any of the proofs in [3] while we use it intensively to make many proofs of our lemmas much simpler.

- Findler and Blume discover that contracts are pairs of projections in [9]. That means given a contract t, $\lambda e.\mathcal{W}_t(e)$ is a projection where $\mathcal{W}_t$ is a wrapper function. To be a projection w.r.t. $\sqsubseteq$, a function p must satisfy these two properties:

  1. $p \circ p = p$ (idempotence)

  2. $p \sqsubseteq 1$ (result of projection contains no more information then its input)

  Our ($\bullet \rhd t$) and ($\bullet \lhd t$) (i.e. $\lambda x.(x \rhd t)$ and $\lambda x.(x \lhd t)$) satisfy the idempotence property as shown in Figure 6, but does not satisfy (2). They only satisfy (2) under the condition that the input of the projection satisfies its contract t as shown in Figure 6. Moreover, we discover the *projection pair* property (in Figure 6), which plays a crucial role in our proof.

- Blume & McAllester deal with recursive contracts, which we do not.

Inspired by [10, 3], Hinze et. cl. [16] implement contracts as a library in Haskell and contracts are checked at run-time. The framework also support contract constructors such as pairs, lists, etc. Another dynamic contract checking work is the Camila project [30] which use monads to encapsulate the pre/post-conditions checking behaviour.

The hybrid contract checking framework [11, 20, 19, 15], in theory, can be as powerful as our system. (Hybrid checking means a combination of static and dynamic contract checking.) But in practice, our symbolic execution strategy adopted from [34] gives more flexibility to the verification as illustrated in §2.1. In [31], Wadler and Findler show how contracts fit with hybrid types and gradual types by requiring casts in the source code. The casts are similar to the job of our $\rhd$ and $\lhd$.

9.4 Other work

In [17], a compositional assertion checking framework has been proposed with a set of logical rules for handling higher order functions. Given arguments satisfying their precondition, they check whether function definition satisfies its postcondition and the checking is currently a manual proof based on the logical rules. Apart from our focus on automatic verification, we can give precise blame when a contract violation is detected. The work in [17] has the strength in verification, but not in assigning blames.

Amongst the Haskell community, there have been several works that are aimed at providing high assurance software through validation (testing) [5], program verification [18] or a combination of the two [8]. In the Programatica project, P-Logic has been introduced to specify properties for Haskell programs. P-Logic allows programmers to express relations among multiple functions and it is more comprehensive then contracts, which only allow programmers to specify pre/postcondition of one function. The design of P-Logic focuses on proof construction while our focus is to find the right function to blame based on contract violation. Certainly, the more properties we can prove, the more precise the contract vi-

olation checking will be. Some design of P-Logic could be adopted in future.

Our approach eliminates the effort of inventing and learning a new logic together with its theorem prover. Furthermore, our verification approach does not conflict with the validation assisted approach used by [5, 8] and can play complementary roles.

In [24], pattern matching failures can be inferred during compile time without specifications from programmers. Notably they use a language of regular expressions for contracts, which is incomparable with ours, and the technical details are very different. Moreover, high-order functions are inlined before analysis is applied in [24] while we analyse higher-order functions directly and do not lose modularity.

10. Conclusion and Future Work

We have presented a sound and automatic static verification tool for a functional language, Haskell. Based on contracts and symbolic execution, our approach gives precise blame assignments at compile-time in the presence of higher-order functions and laziness. We lay particular emphasis on allowing the programmer to use all of Haskell in contracts, including functions that may crash or diverge.

We have developed a prototype implementation in the context of the Glasgow Haskell Compiler. It can prove some simple contract satisfaction checks, but is still incomplete for more sophisticated examples involving the use of recursive functions in predicates, such as sorting or AVL trees. Such predicates are also outside the reach of other static checkers, but they are important in Haskell programs. The shortcoming lies in the optimiser (§5) and, in particular, the choice of precisely when to inline a function. We have manual proofs of many of our motivating examples (including sorting and AVL trees), and are working on heuristics to enable the optimiser to prove them too.

In future, we may introduce conjunctive/disjunctive contracts, recursive contracts, and contracts with quantifiers. Another important direction is to offer ways in which the programmer can help the system by suggesting suitable lemmas.

References

[1] Lennart Augustsson. Cayenne - language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *CASSIS*, LNCS 3362, 2004.

[3] Matthias Blume and David McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of jml tools and applications, 2003.

[5] Koen Claessen and John Hughes. *Specification-based testing with QuickCheck*, volume Fun of Programming of *Cornerstones of Computing*, chapter 2, pages 17–40. Palgrave, March 2003.

[6] Sa Cui, Kevin Donnelly, and Hongwei Xi. Ats: A language that combines programming with theorem proving. In Bernhard Gramlich, editor, *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 310–320. Springer, 2005.

[7] Rowan Davies. Refinement-type checker for standard ml. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 565–566, London, UK, 1997. Springer-Verlag.

[8] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell programs by combining testing and proving. In *Proceedings of Third International Conference on Quality Software*, pages 272–279. IEEE Press, 2003.

[9] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Functional and Logic Programming*, pages 226–241. Springer Berlin / Heidelberg, 2006.

[10] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.

[11] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM Press.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[13] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205, New York, NY, USA, 2001. ACM Press.

[14] Tim Freeman and Frank Pfenning. Refinement types for ml. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.

[15] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Eighth Symposium on Trends in Functional Programming*, April 2007.

[16] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS '06: Functional and Logic Programming: 8th International Symposium*, pages 208–225, 2006.

[17] Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 191–202, New York, NY, USA, 2004. ACM Press.

[18] James Hook, Mark Jones, Richard Kieburtz, John Matthews, Peter White, Thomas Hallgren, and Iavor Diatchki. Programatica. `http://www.cse.ogi.edu/PacSoft/projects/programatica/bodynew.htm`, 2005.

[19] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*. Springer-Verlag, April 2007.

[20] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). Technical report, UC Santa Cruz, 2006. `http://sage.soe.ucsc.edu/sage-tr.pdf`.

[21] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modular-3. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 302–305, London, UK, 1998. Springer-Verlag.

[22] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 218–231. ACM, 2006.

[23] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, London, 1992.

[24] Neil Mitchell and Colin Runciman. Unfailing Haskell: A static checker for pattern matching. In *TFP '05: The 6th Symposium on Trends in Functional Programming*, pages 313–328, 2005.

[25] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2007.

[26] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In John H. Reppy and Julia L. Lawall, editors, *ICFP*, pages 62–73. ACM, 2006.

[27] Franois Pottier and Yann Rgis-Gianas. A hoare logic for call-by-value functional programs. In *Mathematics of Program Construction*, July 2008.

[28] Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.

[29] The GHC Team. The Glasgow Haskell Compiler user's guide. `www.haskell.org/ghc/documentation.html`, 1998.

[30] J Visser, J. N. Oliveira, Barbosa L. S., J. F. Ferreira, and A. Mendes. CAMILA revival: VDM meets haskell. In Nico Plat and Peter Gorm Larsen, editors, *Overture Workshop (co-located with FM'05)*, 2005.

[31] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, Sept 2007.

[32] Edwin M. Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 268–279. ACM, 2005.

[33] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.

[34] Dana N. Xu. Extended static checking for haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59, New York, NY, USA, 2006. ACM Press.

[35] Dana N. Xu. Ph.D. Thesis submitted in July 2008. `www.cl.cam.ac.uk/users/nx200/research/dana-phd.ps`, 2008.