Static Contract Checking via First-Order Logic

Nathan Collins[⋆]

Portland State University

Abstract. We enrich the static semantics of Haskell in order to give stronger static guarantees about the input/output behavior of programs. Our approach has two parts: a contract system for Haskell, and a novel strategy for statically checking that a term satisfies a contract.
The contract system includes refinement types, which refine Haskell types by arbitrary Boolean-valued Haskell expressions, and a "crash free" predicate, which is true of expressions that can't cause a run-time exception in any "safe" context. Our novel contract-checking strategy is to translate a contract-annotated source program into a theorem in first-order logic, and then invoke an automatic theorem prover to prove (or refute) the theorem.

1 Introduction

The goal of statically-checked Haskell contracts is to statically rule out crashes [15, Section 2.5]. *Statically checking a contract* means proving that it accurately describes program behavior. *Crashes* are uncaught exceptions, due to pattern match failures and calls to `error`; the Haskell type system does not rule out crashes.

Earlier work on contracts focused on designing the contract language and its semantics, and on proving that checking contracts in general could be reduced, by a source-to-source translation, to checking crash-freeness [15] [14]. However, that earlier work did not result in a practically effective way to check contract satisfaction. Indeed, an earlier contract paper says [15, Section 5]:

> But how can we prove that [a translated Haskell expression] is crash free? There are many ways to do this, and doing so is not the focus of this paper.

Their approach was to symbolically evaluate the translated expression, succeeding when no syntactic crashes remained.

The goal of our work is a practically-effective system for statically-checking Haskell contracts. We take the existing contract language of Xu, Peyton-Jones, and Claessen, and develop a new way of checking contracts, using a novel translation to first-order logic (FOL) and automatic theorem proving. Unlike the earlier work, we do not focus on proving soundness properties. Indeed, a

[⋆] This RPE paper reports joint work with Charles-Pierre Astolfi, Koen Claessen, Simon Peyton-Jones, and Dimitrios Vytiniotis. My work was performed mostly while I was an intern at Microsoft Research Cambridge in Fall 2011.

soundness theorem is not of much use if it proves soundness of an ineffective technique. Instead, once effectiveness is established empirically, we will then worry more about soundness.

The contract language includes a predicate CF, for “crash free”. This predicate is obviously not computable. Moreover, it can’t be expressed in a type system like Haskell’s, because checking it depends on a path-sensitive analysis, whereas HM-style type inference is path-insensitive. Consider:

```
tail  []      = error "The sky is falling !"
tail  (_:xs) = xs

contrivedTail  [] = []
contrivedTail  xs = tail  xs
```

The contrivedTail function takes CF arguments to CF results, even though it calls the unsafe tail function. We can express this with a contract annotation:

```
contrivedTail  :::  CF → CF.
```

Establishing that contrivedTail satisfies this contract depends on a path-sensitive analysis that checks that there is no valid execution path from a call to contrivedTail xs to a (crashing) call to tail [].

But the CF predicate is not enough, because some functions, such as the function tail above, may only be safe on a subset of their arguments. So, the contract language also includes refinements ($\{x|p\}$) by arbitrary Bool-valued Haskell expressions (p). For example, the tail function returns a crash-free result when given a crash-free argument list from the subset of lists that are not null:

```
tail  :::  (CF&&{xs |not (null xs)}) →  CF .
```

Contributions. Because this is an RPE paper, I list my individual contributions:

- rewrote the contract checker and added many features, including the first implementation of a Min-translation, the change to a Haskell compatible syntax, and additions to the syntax, such as nested case expressions;
- wrote many examples, including the first non-trivial use of lemmas;
- designed and implemented a new Min-translation, which is more intuitive to me and Dimitrios, and identified points of freedom in the translation. My translation turned out to be mostly equivalent to the original Min-translation, up to bugs in the original;
- designed and implemented a type checker for contracts, by translating contracts to Haskell expressions which are then type checked by GHC.

2 The Languages

We translate modules in a Haskell-like source language into formulas in a FOL target language. The source language is an expressive subset of Haskell extended with contract annotations. The target language is a FOL with equality whose term language is similar to the Haskell-like source expression language

2.1 The Haskell-like Source Language

Our source language, presented in Figure 1, evolved as the smallest subset of Haskell in which we did not find it too painful to program examples. A source program is a module ($\mathcal{M}$) consisting of data-type definitions ($data$), function definitions (fun), and contract annotations (ann). A data type is a parametrized type name (T) and a list of term constructors (K) applied to argument types (τ). A function definition consists of a function name (f), zero of more arguments ($\overline{x}$), and a body expression (b). Expressions (e) consist of variables (x), functions (f), term constructors (K), applications ($e\ e$), and crashes (BAD).

There are three closely-related grammars for expression-like things. We call these grammars function bodies (b), expressions (e), and terms (t). Both b and t are extensions of e. The differences are that b adds case expressions and t adds the constant UNR.[1] Note that b and e, but not t, occur in input programs.

The source language is not as syntactically rich as Haskell, but nearly as expressive. We support top level definitions, but the expression language does not include let binding or λ-abstraction. The expression language includes nested case expressions, but restricts patterns to be flat.[2] We support plain, but not generalized, algebraic data types. We don't support type classes.

2.2 The Contract Language

The contract language is presented in Figure 1. It's taken from the earlier work of Xu, Peyton-Jones, and Claessen [15]. There are two base contracts, the crash-free contract (CF) and the refinement contract ($\{x|p\}$), and three recursive contracts, the dependent arrow contract ($x{:}c \to c$), conjunctions ($c \wedge c$), and disjunctions ($c \vee c$). In a refinement contract $\{x|p\}$, the predicate p may be any Bool-valued expression. Unlike the earlier work, we don't support term-constructor contracts.

2.3 The FOL Target Language

The target language (ϕ), presented in Figure 1, is an FOL with equality [13]. The quantifiers and logical connectives are standard. The atomic formulas include equality of terms, the crash-free predicate, and the Min predicate (Section 3.4). The term language (t) for our FOL is the source expression language (e), extended with a constant symbol UNR.

The term language has constants f and K for each function f and constructor K in the source program. Note that the Curried application form, $t\ t$, is a binary function symbol denoted by white space. It dos not contain case expressions,

[1] The mnemonic is "unreachable", and UNR plays the role of infinite loops and ill-typed (stuck) expressions.

[2] The nesting is restricted however. Only plain expressions (e), not case expressions (b), can be scrutinized ($\mathsf{case}\ e\ \mathsf{of}\ \ldots$) and applied ($e\ e$). Case expressions ($b$) can be nested only in the targets of patterns ($\ldots\ \overline{K\ \overline{x} \to b}$).

Module	$\mathcal{M}, mod ::= \overline{decl}$
Declaration	$decl ::= fun \mid ann \mid data$
Function Definition	$fun ::= f\ \overline{x} = b$
Data Type Definition	$data ::= \mathsf{data}\ T\ \overline{\alpha} = \overline{K\ \overline{\tau}}$
Type	$\tau ::= \alpha \mid \tau \to \tau \mid T\ \overline{\tau}$
Contract Annotation	$ann ::= f ::: c$
Contract	$c ::= \mathsf{CF} \mid c \wedge c \mid c \vee c \mid x{:}c \to c \mid \{x \mid p\}$
Function Body	$b ::= e \mid \mathsf{case}\ e\ \mathsf{of}\ \overline{K\ \overline{x} \to b}$
Expression	$e, p ::= x \mid f \mid K \mid \mathsf{BAD} \mid e\ e$
Term	$t ::= x \mid f \mid K \mid \mathsf{BAD} \mid t\ t$
	$\mid \mathsf{UNR} \mid \hat{f}(t, \ldots, t) \mid \hat{K}(t, \ldots, t) \mid \pi_i^K(t)$
Formula	$\phi ::= \forall x.\phi \mid \exists x.\phi$
	$\mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \to \phi$
	$\mid t = t \mid \mathsf{CF}(t) \mid \mathsf{Min}(t)$
Variables	$\alpha \in$ Type variables
	$x \in$ Term variables
	$T \in$ Type constructors
	$K \in$ Term constructors
	$f \in$ Functions

Fig. 1. Syntax of the Haskell-like source language and FOL target language.

The highlighting in the function body (b) and term (t) grammars marks additions relative the expression grammar (e).

because we found it simpler to translate case expressions to formulas involving more primitive expressions, than to implement capture-avoiding substitution in the logic.

The π_i^K are projections for the term constructor K:

$$\pi_i^K(K\ x_1 \cdots\ x_i \cdots\ x_n) = x_i\ .$$

We use the projections to axiomatize the injectivity of term constructors, and to avoid some quantified variables in the translation of case expressions (b).

3 The Translations

3.1 The Naive Translation Into FOL

We now describe a naive translation into FOL, presented in Figure 2 and Figure 3. The translation we actually use is more sophisticated (Section 3.4), but we start by presenting a simpler versions that illustrates all the important points in representing Haskell contracts in FOL. The real translation generates more restrictive formulas, in order to constrain the prover's search space and allow for finite counter models.

We use Oxford brackets ($[\![\cdot]\!]$) to denote translation, and we decorate the brackets with a letter indicating the kind of translation, e.g. $[\![\cdot]\!]_{\mathtt{T}}$ for data-type translations. The function-definition translation $[\![f\ \overline{x} = b]\!]_{\mathtt{f}}$ asserts that the function definition is true for all possible arguments, by universally quantifying the arguments $\overline{x}$.

The function-body translation $[\![(e, b)]\!]_{\mathtt{b}}$ equates the function e with its body, when the body b is not a case expression. The case where b is a case expression (case e' of $\cdots$) is more interesting, and considers the possible values of the scrutinee e'. If e' is an application of an appropriate constructor (some K_i), then the corresponding pattern-branch (b_i) is followed. If e' is an exception (BAD), then the function crashes (returns BAD). Finally, if e' is not an appropriate constructor, or an exception, then the function loops by returning UNR.[3] The translation also asserts that these possibilities for the scrutinee are exhaustive.

The contract translation $[\![e ::: c]\!]_{\mathtt{c}}$ is straightforward in the recursive cases: dependent arrow ($x{:}c_1 \to c_2$), conjunction ($c_1 \wedge c_2$), and disjunction ($c_1 \vee c_2$). When c is CF, we simply apply the CF predicate to the expression. When c is the refinement $\{x|p\}$, we state that p must evaluate to True or UNR (loop) when e terminates. The semantics of Xu et al. [15] additionally requires e to be crash-free, but we make our semantics more permissive in order to be able to express their Any contract directly as $\{x|\mathsf{True}\}$.

The data-type translation $[\![\mathsf{data}\ T\ \overline{\alpha} = \overline{K\ \overline{\tau}}\,]\!]_{\mathtt{T}}$ is the most complex. We break it into four parts, ϕ_{Lazy}, ϕ_{CF}, $\phi_{\text{Injective}}$, and ϕ_{Disjoint}, which describe the term-constructors K_i:

$\phi_{\textbf{Lazy}}$**:** Constructors are lazy functions (don't evaluate their arguments). You can safely apply a constructor to a loop (UNR) or an exception (BAD).

$\phi_{\textbf{CF}}$**:** Constructors are crash-free functions. A constructor application can cause a crash if-and-only-if one of its arguments can.

$\phi_{\textbf{Injective}}$**:** Constructors are injective functions. Constructors have projections that extract the arguments.

$\phi_{\textbf{Disjoint}}$**:** Constructors are disjoint functions. Applications of distinct constructors are never equal.[4]

The module translation ($[\![\mathcal{M}]\!]_{\mathtt{M}}$) is the input to the theorem prover. It says to assume a "prelude" of common-axioms (ϕ_{Prelude}) and the translations ($\phi_{\mathcal{T}}$) of all data-types in the module ($\mathcal{M}$) while checking that the mutually-recursive functions ($\mathcal{F}$) satisfy their contracts. The prelude asserts that loops (UNR) are

[3] Because the program is type checked, we know that e' is never a constructor of the wrong type in an actual program run. However, the logic is untyped, and must account for all possibilities.

[4] Note that we don't axiomatize disjointness of constructors in *different* types. This helps keep the counter models small.

$\boxed{[\![f\ \overline{x} = b]\!]_{\mathtt{f}}}$

$$[\![f\ \overline{x} = b]\!]_{\mathtt{f}} = \forall \overline{x}.\ [\![(f\ \overline{x}, b)]\!]_{\mathtt{b}}$$

$\boxed{[\![(e,b)]\!]_{\mathtt{b}}}$

$$\begin{array}{l}
[\![(e,e')]\!]_{\mathtt{b}} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = (e = e') \\
[\![(e, \mathsf{case}\ e'\ \mathsf{of}\ (K_1\ \overline{x_1}^{a_1} \to b_1) \cdots (K_n\ \overline{x_n}^{a_n} \to b_n))]\!]_{\mathtt{b}} = \\
\quad \bigwedge_{1\le i\le n} \left(e' = k_i \to \left[\!\!\left[\left(e, b_i[\pi_j^{K_i}(e')/x_{ij}]_{1\le j\le a_i}\right)\right]\!\!\right]_{\mathtt{b}}\right) \\
\quad \wedge \left(e' = \mathsf{BAD} \to e = \mathsf{BAD}\right) \\
\quad \wedge \bigvee_{1\le i\le n} (e' = k_i) \\
\qquad \vee \left(e' = \mathsf{BAD}\right) \\
\qquad \vee \bigwedge_{1\le i\le n} (e' \ne k_i) \\
\qquad\quad \wedge \left(e' \ne \mathsf{BAD}\right) \\
\qquad\quad \wedge (e = \mathsf{UNR}) \\
\quad \text{where } k_i = K_i\ \left(\pi_1^{K_i}(e')\right) \cdots \left(\pi_{a_i}^{K_i}(e')\right)
\end{array}$$

$\boxed{[\![e ::: c]\!]_{\mathtt{c}}}$

$$\begin{array}{ll}
[\![e ::: \mathsf{CF}]\!]_{\mathtt{c}} & = \mathsf{CF}(e) \\
[\![e ::: \{x|p\}]\!]_{\mathtt{c}} & = (e \ne \mathsf{UNR} \to p[e/x] \in \{\mathsf{True}, \mathsf{UNR}\}) \\
[\![e ::: x{:}c_1 \to c_2]\!]_{\mathtt{c}} & = \forall x.\ [\![x ::: c_1]\!]_{\mathtt{c}} \to [\![e\ x ::: c_2]\!]_{\mathtt{c}} \\
[\![e ::: c_1 \mathbin{||} c_2]\!]_{\mathtt{c}} & = [\![e ::: c_1]\!]_{\mathtt{c}} \vee [\![e ::: c_2]\!]_{\mathtt{c}} \\
[\![e ::: c_1 \&\& c_2]\!]_{\mathtt{c}} & = [\![e ::: c_1]\!]_{\mathtt{c}} \wedge [\![e ::: c_2]\!]_{\mathtt{c}}
\end{array}$$

$\boxed{[\![\mathsf{data}\ T\ \overline{\alpha} = \overline{K\ \overline{\tau}}\,]\!]_{\mathtt{T}}}$

$$\begin{array}{l}
[\![\mathsf{data}\ T\ \overline{\alpha}^m = K_1\ \overline{\tau_1}^{a_1} \cdots K_n\ \overline{\tau_n}^{a_n}]\!]_{\mathtt{T}} = \phi_{\text{Lazy}} \wedge \phi_{\mathsf{CF}} \wedge \phi_{\text{Injective}} \wedge \phi_{\text{Disjoint}} \\
\quad \text{where } \phi_{\text{Lazy}} \quad = \bigwedge_{1\le i\le n} (\forall \overline{x}^{a_i}.\ K_i\ \overline{x} \notin \{\mathsf{UNR}, \mathsf{BAD}\}) \\
\qquad\qquad \phi_{\mathsf{CF}} \qquad = \bigwedge_{1\le i\le n} \left([\![K_i ::: \underbrace{\mathsf{CF} \to \cdots \to \mathsf{CF}}_{a_i+1}]\!]_{\mathtt{c}} \wedge \left(\forall \overline{x}^{a_i}.\ \mathsf{CF}(K_i\ \overline{x}) \to \bigwedge_{1\le j\le a_i} \mathsf{CF}(x_j)\right)\right) \\
\qquad\qquad \phi_{\text{Injective}} = \bigwedge_{1\le i\le n} \left(\forall \overline{x}^{a_i}.\ \bigwedge_{1\le j\le a_i} \left(x_j = \pi_j^{K_i}(K_i\ \overline{x})\right)\right) \\
\qquad\qquad \phi_{\text{Disjoint}} \ = \bigwedge_{1\le i<j\le n} \left(\forall \overline{x}^{a_i}, \overline{y}^{a_j}.\ K_i\ \overline{x} \ne K_j\ \overline{y}\right)
\end{array}$$

$\boxed{t \in \{\ldots\}}$

$$t \in \{t_1, \ldots, t_n\} = (t = t_1 \vee \cdots \vee t = t_n)$$

$\boxed{\overline{x}^n}$

$$\overline{x}^n = x_1 \cdots x_n$$

Fig. 2. The naive translation of functions ($[\![\cdot]\!]_{\mathtt{f}}$), contracts ($[\![\cdot]\!]_{\mathtt{c}}$), and data types ($[\![\cdot]\!]_{\mathtt{T}}$).

In the function-body translation ($[\![(e,b)]\!]_{\mathtt{b}}$) the indentation is significant and indicates the grouping of subformulas. We omit the length superscripts (n) on sequences ($\overline{x}^n$) when irrelevant, and when specified earlier.

$\boxed{[\![\mathcal{M}]\!]_{\mathtt{M}}}$

$$[\![\mathcal{M}]\!]_{\mathtt{M}} = \Big(\phi_{\text{Prelude}} \wedge \phi_{\mathcal{T}}\Big) \to \Big(\textstyle\bigwedge_{\mathcal{F}\in\mathcal{S}} [\![(\mathcal{F},\mathcal{M})]\!]_{\mathtt{F}}\Big)$$

where $\mathcal{S}$ = The collection of strongly-connected components of mutually-recursive functions in $\mathcal{M}$. There is an edge from f to g in $\mathcal{M}$ if f calls g.

$\phi_{\text{Prelude}} = \mathsf{CF}(\mathsf{UNR}) \wedge \neg\mathsf{CF}(\mathsf{BAD})$

$\mathcal{T}$ = The data types defined in $\mathcal{M}$.

$\phi_{\mathcal{T}} = \bigwedge_{data\in\mathcal{T}} [\![data]\!]_{\mathtt{T}}$

$\boxed{[\![(\mathcal{F},\mathcal{M})]\!]_{\mathtt{F}}}$

$$[\![(\mathcal{F},\mathcal{M})]\!]_{\mathtt{F}} = \Big(\phi_{\mathcal{G}} \wedge \phi_{\mathcal{C}_{\mathcal{G}}} \wedge \phi_{\mathcal{F}_{\mathsf{rec}}} \wedge \phi_{\mathcal{C}_{\mathcal{F}_{\mathsf{rec}}}}\Big) \to \phi_{\mathcal{C}_{\mathcal{F}}}$$

where $\mathcal{G}$ = The functions defined in $\mathcal{M}$ but not in $\mathcal{F}$.

$\mathcal{C}_{\mathcal{F}}$ = The contract annotations in $\mathcal{M}$ for functions in $\mathcal{F}$.

$\mathcal{C}_{\mathcal{G}}$ = The contract annotations in $\mathcal{M}$ for functions in $\mathcal{G}$.

$\phi_{\mathcal{G}} = \bigwedge_{fun\in\mathcal{G}} [\![fun]\!]_{\mathtt{f}}$

$\phi_{\mathcal{C}_{\mathcal{G}}} = \bigwedge_{ann\in\mathcal{C}_{\mathcal{G}}} [\![ann]\!]_{\mathsf{c}}$

$\phi_{\mathcal{F}_{\mathsf{rec}}} = \bigwedge_{(f\,\overline{x}=b)\in\mathcal{F}} \Big[\!\!\Big[f\,\overline{x} = \Big(b[f'_{\mathsf{rec}}/f']_{(f'\,\_=\_)\in\mathcal{F}}\Big)\Big]\!\!\Big]_{\mathtt{f}}$

$\phi_{\mathcal{C}_{\mathcal{F}_{\mathsf{rec}}}} = \bigwedge_{(f:::c)\in\mathcal{C}_{\mathcal{F}}} [\![f_{\mathsf{rec}} ::: c]\!]_{\mathsf{c}}$

$\phi_{\mathcal{C}_{\mathcal{F}}} = \bigwedge_{ann\in\mathcal{C}_{\mathcal{F}}} [\![ann]\!]_{\mathsf{c}}$

Fig. 3. Naive translation of a module ($\mathcal{M}$).

Note that when checking a strongly-connected component of functions $\mathcal{F}$, we assume (by $\phi_{\mathcal{C}_{\mathcal{F}_{\mathsf{rec}}}}$) that the contracts $\mathcal{C}_{\mathcal{F}}$ hold for the anonymous recursive versions f_{rec} of the functions f in $\mathcal{F}$, and we redefine (by $\phi_{\mathcal{F}_{\mathsf{rec}}}$) the f in $\mathcal{F}$ to call the anonymous functions when recursing. This is analogous to how recursive functions are type-checked in standard type systems, and corresponds to an ill-founded induction principle. Namely, when checking a property of a recursive function, you may assume it holds on all recursive calls.

crash free, but exceptions (BAD) are not. The data-type translations in $\phi_{\mathcal{T}}$ introduce additional crash-freeness axioms via ϕ_{CF}.

The SCC-translation ($[\![(\mathcal{F},\mathcal{M})]\!]_{\mathtt{F}}$) gives specific assumptions under which to check that an SCC of mutually-recursive functions satisfy their contracts ($\phi_{\mathcal{C}_{\mathcal{F}}}$). The assumptions include:

$\phi_{\mathcal{G}}$: All other functions ($\mathcal{G}$) are defined as expected.

$\phi_{\mathcal{C}_{\mathcal{G}}}$: All other functions satisfy their contracts.

$\phi_{\mathcal{F}_{\mathsf{rec}}}$: The mutually-recursive functions ($\mathcal{F}$) are redefined so that recursive calls are made by new functions (f_{rec}). These new functions are opaque, and have no associated definition translation.

$\phi_{\mathcal{C}_{\mathcal{F}_{\mathsf{rec}}}}$: The opaque functions satisfy the contracts of the non-opaque recursive functions they replace calls to. This corresponds to induction.

3.2 A Detailed Example

We now present the translation of a concrete example. Refer to Figures 2 and 3 for the abstract translation. Consider the tail and length functions:

```
data List a = Nil    | Cons a (List a)
data Nat    = Zero   | Succ Nat

length xs = case xs of
  Nil         → Zero
  Cons x xs' → Succ (length xs')

 tail xs = case xs of
  Nil         → BAD
  Cons x xs' → xs' .
```

We give length a simple contract: length doesn't crash when its argument is crash-free:

```
length ::: CF → CF .
```

The contract we give tail is more interesting: given a non-null crash-free argument xs, tail produces a crash-free result r, which has length one-less than the argument:

```
 tail    ::: xs:(CF&&{xs | not (null xs)})
              → (CF&&{r | length xs == Succ (length r)}) .
```

Note that r corresponds to tail xs in the result contract.

Next, consider the translation of the List data type:

$$
\begin{aligned}
&[\![\mathsf{data\ List\ a} = \mathsf{Nil} \mid \mathsf{Cons\ a\ (List\ a)}]\!]_{\mathsf{T}} \\
&\quad \left.\begin{aligned}&= \mathsf{Nil} \notin \{\mathsf{UNR},\mathsf{BAD}\}\\ &\wedge\ \forall x_1 x_2.\ \mathsf{Cons}\ x_1\ x_2 \notin \{\mathsf{UNR},\mathsf{BAD}\}\end{aligned}\right\}\phi_{\mathrm{Lazy}} \\
&\quad \left.\begin{aligned}&\wedge\ [\![\mathsf{Nil} ::: \mathsf{CF}]\!]_{\mathsf{c}} \wedge (\mathsf{CF}(\mathsf{Nil}) \to \top)\\ &\wedge\ [\![\mathsf{Cons} ::: \mathsf{CF} \to \mathsf{CF} \to \mathsf{CF}]\!]_{\mathsf{c}} \wedge (\forall x_1 x_2.\ \mathsf{CF}(\mathsf{Cons}\ x_1\ x_2) \to \mathsf{CF}(x_1) \wedge \mathsf{CF}(x_2))\end{aligned}\right\}\phi_{\mathsf{CF}} \\
&\quad \left.\begin{aligned}&\wedge\ \top\\ &\wedge\ \forall x_1 x_2.\ x_1 = \pi_1^{\mathsf{Cons}}(\mathsf{Cons}\ x_1\ x_2) \wedge x_2 = \pi_2^{\mathsf{Cons}}(\mathsf{Cons}\ x_1\ x_2)\end{aligned}\right\}\phi_{\mathrm{Injective}} \\
&\quad \wedge \forall y_1 y_2.\ \mathsf{Nil} \neq \mathsf{Cons}\ y_1\ y_2 \quad \}\phi_{\mathrm{Disjoint}}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{Nil} ::: \mathsf{CF}]\!]_{\mathsf{c}} &= \mathsf{CF}(\mathsf{Nil}) \\
[\![\mathsf{Cons} ::: \mathsf{CF} \to \mathsf{CF} \to \mathsf{CF}]\!]_{\mathsf{c}} &= \forall x_1.\ \mathsf{CF}(x_1) \to \forall x_2.\ \mathsf{CF}(x_2) \to \mathsf{CF}(\mathsf{Cons}\ x_1\ x_2)\ .
\end{aligned}
$$

The trivial tautology $\top$ appears in places where the abstract translation would produce empty formulas. The translation of the Nat data-type is similar and we omit it. We have labeled the parts of the translation with the formula names from Figure 2 which they correspond to.

Next, consider the translation of the length function:

$$
\begin{aligned}
&[\![\mathsf{length\ xs} = b[\mathsf{length_{rec}}/\mathsf{length}]]\!]_{\mathtt{f}}\\
&\quad = \forall \mathsf{xs}.\ [\![(\mathsf{length\ xs}, \mathsf{case\ xs\ of\ (Nil} \to \mathsf{Zero})\ (\mathsf{Cons\ x\ xs'} \to \mathsf{Succ\ (length_{rec}\ xs')}))]\!]_{\mathsf{b}}\\
&\quad = \forall \mathsf{xs}.\ (\mathsf{xs} = \mathsf{Nil} \to [\![(\mathsf{length\ xs}, \mathsf{Zero})]\!]_{\mathsf{b}})\\
&\qquad\qquad \wedge \Big(\mathsf{xs} = \mathsf{Cons}\ \left(\pi_1^{\mathsf{Cons}}(\mathsf{xs})\right)\ \left(\pi_2^{\mathsf{Cons}}(\mathsf{xs})\right)\\
&\qquad\qquad\quad \to \left[\!\!\left[\left(\mathsf{length\ xs}, (\mathsf{Succ\ (length_{rec}\ xs')})\ [\pi_1^{\mathsf{Cons}}(\mathsf{xs})/\mathsf{x}][\pi_2^{\mathsf{Cons}}(\mathsf{xs})/\mathsf{xs'}]\right)\right]\!\!\right]_{\mathsf{b}}\Big)\\
&\qquad\qquad \wedge (\mathsf{xs} = \mathsf{BAD} \to \mathsf{length\ xs} = \mathsf{BAD})\\
&\qquad\qquad \wedge (\mathsf{xs} = \mathsf{Nil}) \vee \left(\mathsf{xs} = \mathsf{Cons}\ \left(\pi_1^{\mathsf{Cons}}(\mathsf{xs})\right)\ \left(\pi_2^{\mathsf{Cons}}(\mathsf{xs})\right)\right)\\
&\qquad\qquad\quad \vee (\mathsf{xs} = \mathsf{BAD})\\
&\qquad\qquad\quad \vee (\mathsf{xs} \neq \mathsf{Nil}) \wedge \left(\mathsf{xs} \neq \mathsf{Cons}\ \left(\pi_1^{\mathsf{Cons}}(\mathsf{xs})\right)\ \left(\pi_2^{\mathsf{Cons}}(\mathsf{xs})\right)\right)\\
&\qquad\qquad\qquad \wedge (\mathsf{xs} \neq \mathsf{BAD})\\
&\qquad\qquad\qquad \wedge (\mathsf{length\ xs} = \mathsf{UNR})
\end{aligned}
$$

$$
\begin{aligned}
&[\![(\mathsf{length\ xs}, \mathsf{Zero})]\!]_{\mathsf{b}} = (\mathsf{length\ xs} = \mathsf{Zero})\\
&\left[\!\!\left[\left(\mathsf{length\ xs}, (\mathsf{Succ\ (length_{rec}\ xs')})\ [\pi_1^{\mathsf{Cons}}(\mathsf{xs})/\mathsf{x}][\pi_2^{\mathsf{Cons}}(\mathsf{xs})/\mathsf{xs'}]\right)\right]\!\!\right]_{\mathsf{b}}\\
&\quad = \left(\mathsf{length\ xs} = \mathsf{Succ}\ \left(\mathsf{length_{rec}}\ \left(\pi_2^{\mathsf{Cons}}(\mathsf{xs})\right)\right)\right).
\end{aligned}
$$

The translation of tail is very similar and we omit it. We chose to show the translation of length because it cases over a list, like tail, but, in addition, makes a recursive call.

Finally, consider the translation of tail's contract:

$$
\begin{aligned}
&[\![\ \mathsf{tail} ::: \mathsf{xs}{:}(\mathsf{CF}\&\&\{\mathsf{xs} \mid \mathsf{not\ (null\ xs)}\}) \to (\mathsf{CF}\&\&\{\mathsf{r} \mid \mathsf{length\ xs} == \mathsf{Succ\ (length\ r)}\})]\!]_{\mathsf{c}}\\
&\quad = \forall \mathsf{xs}.\ \mathsf{CF(xs)}\\
&\qquad\qquad \wedge \left(\mathsf{xs} \neq \mathsf{UNR} \to \mathsf{not\ (null\ xs)} \in \{\mathsf{True}, \mathsf{UNR}\}\right)\\
&\qquad \to \mathsf{CF(tail\ xs)}\\
&\qquad\quad \wedge \left(\mathsf{tail\ xs} \neq \mathsf{UNR} \to (\mathsf{length\ xs} == \mathsf{Succ\ (length\ (tail\ xs))}) \in \{\mathsf{True}, \mathsf{UNR}\}\right).
\end{aligned}
$$

We discuss the provability of this contract in the next section. Note that we only showed a few of the translations that would actually be generated by the module translation ($[\![\cdot]\!]_{\mathtt{M}}$) for this example; Figure 3 outlines the full formula we generate.

3.3 Lemmas

Our system has no special support for stating, proving, or using lemmas, but all this can be encoded. For example, it turns out that tail 's contract in the previous section is not provable from the module translation ($[\![\cdot]\!]_{\mathtt{M}}$) in Figure 3. Recall tail 's contract:

```
21  tail  :::  xs:(CF&&{xs | not (null xs)})
22            → (CF&&{r | length xs == Succ (length r)}) ,
```

We need an induction on lists to prove the contract is satisfied, but the induction hypothesis

$$[\![\mathsf{tail}_{\mathsf{rec}} ::: \mathsf{xs}{:}(\mathsf{CF\&\&}\{\mathsf{xs} \mid \mathsf{not}\ (\mathsf{null}\ \mathsf{xs})\}) \rightarrow (\mathsf{CF\&\&}\{\mathsf{r} \mid \mathsf{length}\ \mathsf{xs} == \mathsf{Succ}\ (\mathsf{length}\ \mathsf{r})\})]\!]_{\mathsf{c}}$$

for tail is no help, because tail isn't recursive. Our lemma encoding trick can help us here.

Consider the following code, which gives a lemma-encoding example:

```
23  data Proof = QED
24
25  lem xs = case xs of
26    Nil          → BAD
27    Cons _ xs' → case xs' of
28        Nil       → QED
29        Cons _ _ → lem xs'
30
31  lem  :::  xs:(CF&&{xs|not (null xs)}
32           → (CF&&{r|length xs == Succ (length ( tail  xs))}) .
```

The lemma statement is the contract (Line 31) on the lemma function lem, where we've replaced r with tail xs because r was tail xs in the contract for tail (Line 21). Here r is the dummy Proof returned by the lemma. The function lem proves the lemma by recursion/induction on xs. We introduce the Proof data type so that lem will have something to return in the non-recursive case; almost any data type would do.

How does lem's implementation correspond to a proof? First, we check if the argument list xs is null. If yes, then the argument contract CF&&{xs|not (null xs)} is not satisfied and there is nothing to prove. The exception BAD is returned in this case, but any expression would do. If xs is not null, then we consider its tail xs'. If xs' is null, then we are in the base case and computation shows that

$$(\mathsf{length}\ (\mathsf{Cons}\ \_\ \mathsf{Nil}) == \mathsf{Succ}\ (\mathsf{length}\ \mathsf{Nil})) = \mathsf{True}\ .$$

If xs' is not null, then we call lem xs' to establish the inductive hypothesis

$$(\mathsf{length}\ (\mathsf{Cons}\ \_\ \mathsf{xs'}) == \mathsf{Succ}\ (\mathsf{length}\ \mathsf{xs'})) \in \{\mathsf{True}, \mathsf{UNR}\}\ ,$$

from which the goal

$$\mathsf{length}\ (\mathsf{Cons}\ \_\ (\mathsf{Cons}\ \_\ \mathsf{xs'})) == \mathsf{Succ}\ (\mathsf{length}\ (\mathsf{Cons}\ \_\ \mathsf{xs})) \in \{\mathsf{True}, \mathsf{UNR}\}$$

follows by computation. Actually, the inductive hypothesis tells us that lem diverges (returns UNR) *or* that the equality computation returns True or UNR, and so the conclusion also includes the possibility of divergence. This makes sense, because the length function will indeed diverge on infinite lists, but these divergences have consequences which we return to shortly.

To use the lemma, we change the definition of tail so that it calls the lemma function lem:

```
withLemma p e = case p of QED → e

 tail xs = case xs of
   Nil        → BAD
   Cons _ xs' → withLemma (lem xs) xs' .
```

The way this works is rather subtle. When we call lem xs we know xs ::: CF&&{xs|not (null xs)}, and we conclude that the lemma call lem xs diverges, or that

$$\text{length (Cons \_ (Cons \_ xs'))} == \text{Succ (length (Cons \_ xs))} \in \{\text{True}, \text{UNR}\} .$$

If the lemma call diverges we have learned nothing, but then withLemma (lem xs) xs' also diverges, since withLemma scrutinizes the lemma call, and so tail itself diverges and the proof succeeds. So, it is essential that withLemma be strict in the proof argument; otherwise we'd get stuck when the proof diverged.[5]

The general idea is to write a lemma function ℓ whose recursion corresponds to the induction needed to prove the lemma. The lemma is invoked in the proof of a contract satisfaction f ::: c by adding an appropriate withLemma $(\ell\ \overline{e_1})\ e_2$ call in the definition of f.

There is a caveat, however. As we saw in the tail example, using lemmas can change the run-time behavior of programs! Because withLemma $p\ e$ scrutinizes the proof p before returning e, any crash or loop in p propagates to e. This is not as bad as it might sound though. As we saw in the example, we run into trouble when the lemma function encodes an inductive proof of a property that diverges for infinite data (the equality function (==) diverges in the example). The contract language does not allow us to distinguish between finite and infinite data, and so it's not surprising that we pay a price when giving contracts that only make sense in the finite case.

3.4 The Min-Translation Into FOL

We now describe a less-naive translation (Figures 4 and 5). Our changes to the naive translation are motivated by two objectives:

1. Restrict the search space of the prover (conservatively);
2. Allow finite models of the axioms.

The first objective is obvious: we want the theorem prover to succeed more often, and more quickly. The second objective comes into to play when the prover is destined to fail, because the goal theorem (from the module translation in Figure 3) is not provable, which can happen for two reasons:

I. some contract in the program is not satisfied, or

[5] Incidentally, in the Min-translation of Section 3.4, we also need withLemma to scrutinize the lemma call so that we can conclude Min for the lemma call and unfold the lemma function definition.

II. all contracts in the program are satisfied, but our generated theory is not strong enough to prove this, because it does not characterize our source language well enough.

When the goal theorem is not provable, there will be a counter model for our generated theory.[6] By generating axioms which permit finite models, we can hope to identify unprovable theorems by using a finite-model finder, and moreover, the finite models found can provide insight into why the goal theorem is not provable. In case (II) we can add more contracts, including lemmas (Section 3.3), to improve the situation.

To achieve these objectives, we add a new predicate to our FOL, and incorporate it into our translations. The new predicate is Min, and intuitively $\mathsf{Min}(e)$ corresponds to "*e* will be evaluated"; the mnemonic is "I'm interested".[7] Roughly speaking, we change the translations so that the formulas they generate only apply to expressions for which Min holds.

The Min predicate behaves as a sort of guard, e.g. preventing function-definition formulas ($[\![\cdot]\!]_{\mathtt{f}}$)) from being instantiated for calls that would not be evaluated. Because evaluation begets evaluation, of subterms or other related terms:

$$\begin{aligned} \mathsf{Min}(e_1\ e_2) &\to \mathsf{Min}(e_1)\ , \\ \mathsf{Min}(\mathsf{case}\ e\ \mathsf{of}\ \overline{K\ \overline{x} \to b}) &\to \mathsf{Min}(e)\ , \\ e ::: \{x|p\} \wedge \mathsf{Min}(e) &\to \mathsf{Min}(p[e/x])\ , \end{aligned}$$

some formulas introduce new Min assumptions. Such formulas are always guarded with a Min obligation.

In order to make the placement of Min's work out, we distinguish between assumptions and goals, and this is reflected by *sign* superscripts on the translation. We write $[\![\cdot]\!]^{\ominus}_{\cdot}$ for translations used as assumptions (axioms) and $[\![\cdot]\!]^{\oplus}_{\cdot}$ for translations used as goals (proof obligations). The mnemonic is that assumptions ($\ominus$) imply goals ($\oplus$), and that the left side of an arrow has negative variance while the right side has positive variance. A simple induction on the structure of contracts shows that, for all expressions e and contracts c, $[\![e ::: c]\!]^{\ominus}_{\mathsf{c}}$ implies $[\![e ::: c]\!]^{\oplus}_{\mathsf{c}}$.

Consider the contract-satisfaction translation in Figure 4. All cases but one, the refinement case ($\{x|p\}$), are uniform in the sign s. However, the meaning of a translation can be different for different signs, even when the resulting formulas are identical.

For example, consider the translation of CF contracts: $\mathsf{Min}(e) \to \mathsf{CF}(e)$. When this translation is used as an assumption ($s = \ominus$), the $\mathsf{Min}(e)$ is an obligation which guards the conclusion of $\mathsf{CF}(e)$. When this translation is used as a

[6] This is one direction of a fundamental theorem in first-order model theory: being provable and being true in all models are equivalent for FOL.

[7] This intuition is helpful in deciding where to place Mins when designing the translation, but keep in the mind the Min does not really "mean" anything, and whether it is true or not of a particular expression depends on context.

$$\boxed{[\![e ::: c]\!]^s_{\mathsf{c}}}$$

$$\begin{array}{lcl}
[\![e ::: \mathsf{CF}]\!]^s_{\mathsf{c}} & = & \mathsf{Min}(e) \to \mathsf{CF}(e) \\
[\![e ::: \{x|p\}]\!]^{\ominus}_{\mathsf{c}} & = & \mathsf{Min}(e) \to \mathsf{Min}(p[e/x]) \land (e \neq \mathsf{UNR} \to p[e/x] \in \{\mathsf{True}, \mathsf{UNR}\}) \\
[\![e ::: \{x|p\}]\!]^{\oplus}_{\mathsf{c}} & = & \mathsf{Min}(e) \land \mathsf{Min}(p[e/x]) \to (e \neq \mathsf{UNR} \to p[e/x] \in \{\mathsf{True}, \mathsf{UNR}\}) \\
[\![e ::: x{:}c_1 \to c_2]\!]^s_{\mathsf{c}} & = & \forall x. [\![x ::: c_1]\!]^{\overline{s}}_{\mathsf{c}} \to [\![e\ x ::: c_2]\!]^s_{\mathsf{c}} \\
[\![e ::: c_1 \,||\, c_2]\!]^s_{\mathsf{c}} & = & [\![e ::: c_1]\!]^s_{\mathsf{c}} \lor [\![e ::: c_2]\!]^s_{\mathsf{c}} \\
[\![e ::: c_1 \&\& c_2]\!]^s_{\mathsf{c}} & = & [\![e ::: c_1]\!]^s_{\mathsf{c}} \land [\![e ::: c_2]\!]^s_{\mathsf{c}}
\end{array}$$

$$\boxed{\overline{s}}$$

$$\overline{\oplus} = \ominus$$
$$\overline{\ominus} = \oplus$$

Fig. 4. Min-translation of contract satisfaction ($[\![\cdot]\!]_{\mathsf{c}}$). The highlighting marks changes versus the naive translation.

goal ($s = \oplus$), the $\mathsf{Min}(e)$ is an assumption to be used in proving $\mathsf{CF}(e)$. Similarly, in the arrow translation ($[\![e ::: x{:}c_1 \to c_2]\!]^s_{\mathsf{c}}$), the left and right sides of the arrow are translated with opposite signs, so that the left side becomes an assumption in a goal translation and a goal in an assumption translation. Flipping the sign here is consistent with the usual treatment of variance of arrows.[8]

The most interesting case is the translation of refinement contracts ($\{x|p\}$):

$$[\![e ::: \{x|p\}]\!]^{\ominus}_{\mathsf{c}} = \mathsf{Min}(e) \to \mathsf{Min}(p[e/x]) \land (e \neq \mathsf{UNR} \to p[e/x] \in \{\mathsf{True}, \mathsf{UNR}\})$$
$$[\![e ::: \{x|p\}]\!]^{\oplus}_{\mathsf{c}} = \mathsf{Min}(e) \land \mathsf{Min}(p[e/x]) \to (e \neq \mathsf{UNR} \to p[e/x] \in \{\mathsf{True}, \mathsf{UNR}\}) \ .$$

The $\mathsf{Min}(e)$ can be interpreted as in the translation of CF contracts, as explained in the previous paragraph. However, the $\mathsf{Min}(p[e/x])$ must *always* be an assumption, because it can't be concluded from $\mathsf{Min}(e)$, and it's needed to unfold definitions (Figure 5), both when establishing goals and when using assumptions.

The contract-satisfaction translation is the only translation that can appear in a goal (Figure 3) and so the other translations (Figure 5) only have $\ominus$-versions. In other words, we both prove and assume contract satisfaction, but we only assume the definitions of data types and functions.

The remaining translations, of functions and data types, appear in Figure 5. Generally, the Min's are placed so that you can't use the formulas unless you have Min for their subject, but there are two Min placements that don't follow this pattern. First, note the introduction of $\mathsf{Min}(e')$ in the translation of function bodies. The point is that evaluating a case-expression entails evaluating the scrutinee e'. Second, note the $\mathsf{Min}(K_i\ \overline{x})$ in $\phi_{\mathrm{Injective}}$. Naively, you might ex-

[8] E.g. arrow subtyping, and the consequence rule in Hoare logic.

$$\boxed{[\![f\ \overline{x} = b]\!]^{\ominus}_{\mathtt{f}}}$$

$$[\![f\ \overline{x} = b]\!]^{\ominus}_{\mathtt{f}} = \forall \overline{x}. \mathsf{Min}(f\ \overline{x}) \rightarrow [\![(f\ \overline{x}, b)]\!]^{\ominus}_{\mathtt{b}}$$

$$\boxed{[\![(e, b)]\!]^{\ominus}_{\mathtt{b}}}$$

$$\begin{array}{ll}
[\![(e, e')]\!]^{\ominus}_{\mathtt{b}} & = (e = e') \\
[\![(e, \mathsf{case}\ e'\ \mathsf{of}\ (K_1\ \overline{x_1}^{a_1} \rightarrow b_1) \cdots (K_n\ \overline{x_n}^{a_n} \rightarrow b_n))]\!]^{\ominus}_{\mathtt{b}} & =
\end{array}$$

$$\begin{array}{l}
\quad \mathsf{Min}(e') \wedge \\
\quad \bigwedge_{1 \le i \le n} \left(e' = k_i \rightarrow \left[\!\!\left[\left(e, b_i [\pi^{K_i}_j(e') / x_{ij}]_{1 \le j \le a_i} \right) \right]\!\!\right]^{\ominus}_{\mathtt{b}} \right) \\
\quad \wedge\ (e' = \mathsf{BAD} \rightarrow e = \mathsf{BAD}) \\
\quad \wedge\ \bigvee_{1 \le i \le n} (e' = k_i) \\
\qquad \vee\ (e' = \mathsf{BAD}) \\
\qquad \vee\ \bigwedge_{1 \le i \le n} (e' \neq k_i) \\
\qquad\quad \wedge\ (e' \neq \mathsf{BAD}) \\
\qquad\quad \wedge\ (e = \mathsf{UNR}) \\
\quad \text{where } k_i = K_i\ \left(\pi^{K_i}_1(e')\right) \cdots \left(\pi^{K_i}_{a_i}(e')\right)
\end{array}$$

$$\boxed{[\![\mathsf{data}\ T\ \overline{\alpha} = \overline{K\ \overline{\tau}}]\!]^{\ominus}_{\mathtt{T}}}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{[\![\mathsf{data}\ T\ \overline{\alpha}^m = K_1\ \overline{\tau_1}^{a_1} \cdots K_n\ \overline{\tau_n}^{a_n}]\!]^{\ominus}_{\mathtt{T}} = \phi_{\mathrm{Lazy}} \wedge \phi_{\mathsf{CF}} \wedge \phi_{\mathrm{Injective}} \wedge \phi_{\mathrm{Disjoint}}} \\
\quad \text{where } \phi_{\mathrm{Lazy}} & = & \bigwedge_{1 \le i \le n} (\forall \overline{x}^{a_i}.\ \mathsf{Min}(K_i\ \overline{x}) \rightarrow K_i\ \overline{x} \notin \{\mathsf{UNR}, \mathsf{BAD}\}) \\
\qquad\qquad \phi_{\mathsf{CF}} & = & \bigwedge_{1 \le i \le n} \left([\![K_i ::: \underbrace{\mathsf{CF} \rightarrow \cdots \rightarrow \mathsf{CF}}_{a_i + 1}]\!]^{\ominus}_{\mathtt{c}} \wedge \left(\forall \overline{x}^{a_i}.\ \mathsf{Min}(K_i\ \overline{x}) \rightarrow \mathsf{CF}(K_i\ \overline{x}) \rightarrow \bigwedge_{1 \le j \le a_i} \mathsf{CF}(x_j) \right) \right) \\
\qquad\qquad \phi_{\mathrm{Injective}} & = & \bigwedge_{1 \le i \le n} \left(\forall \overline{x}^{a_i}.\ \bigwedge_{1 \le j \le a_i} \left(\mathsf{Min}(K_i\ \overline{x}) \rightarrow x_j = \pi^{K_i}_j(K_i\ \overline{x}) \right) \right) \\
\qquad\qquad \phi_{\mathrm{Disjoint}} & = & \bigwedge_{1 \le i < j \le n} \left(\forall \overline{x}^{a_i}, \overline{y}^{a_j}.\ \left(\mathsf{Min}(K_i\ \overline{x}) \vee \mathsf{Min}(K_j\ \overline{y}) \right) \rightarrow K_i\ \overline{x} \neq K_j\ \overline{y} \right)
\end{array}$$

Fig. 5. The Min-translation of functions ($[\![\cdot]\!]^{\ominus}_{\mathtt{f}}$) and data types ($[\![\cdot]\!]^{\ominus}_{\mathtt{T}}$). The highlighting marks changes versus the naive translation.

pect $\mathsf{Min}(\pi_j^{K_i}(K_i\ \overline{x}))$, but that would not work. The problem is that it is $K_i\ \overline{x}$, and not $\pi_j^{K_i}(K_i\ \overline{x})$, which occurs in input programs, and so we only expect to have Min for the former.

The Min-translation of modules is as in the naive translation in Figure 3, except for two changes. First, we must replace all naive translations with signed translations. The contract-satisfaction translation in $\phi_{\mathcal{C}_\mathcal{F}}$ becomes a $\oplus$-translation, and all other translations become $\ominus$-translations. Second, we extend ϕ_{Prelude} to with an axiom related to Min and higher order functions:

$$\forall e, x.\ \mathsf{Min}(e\ x) \rightarrow \mathsf{Min}(e)\ .$$

This helps when e is a function application and we want to unfold the corresponding function definition.

4 Experimental Results

We implemented the FOL translation in a Haskell contract checker called `hcc`. The source code for `hcc`, and our contract examples, are available on-line [2]. The results of experiments comparing the naive (no Min) and Min-translations are reported in Figure 6.

The `hcc` tool is implemented in GHC Haskell, and can output the generated FOL theory in TPTP format, SMTLIB format, and Coq format. In practice we targeted Koen Claessen's Equinox theorem prover, which uses the TPTP format. Our TPTP files are valid for any TPTP prover, but the Equinox prover provides special support for our Min predicate.

In our experiments, reported in Figure 6, the Min-translation fared much better than the naive translation with no Mins. The sub-second run-times are not meant to be precise — each experiment was run only once and on a partially loaded machine — but the overall trend is striking.

The example files are divided into two sets:

./yes contains tests for which all contracts are or could be satisfied in our Haskell-like language;

./no contains tests for which some contract is not satisfied in our Haskell-like language.

When all contracts are satisfied, the desired outcome is "Pass", meaning the prover proved the generated theorem. When some contract is not satisfied the desired outcome is "Fail", meaning the prover found a counter model. The "or could be satisfied" corresponds to the case where the satisfaction of some contract is underspecified. For example, the test `./yes/id-is-cf.hs` asserts that the identity function, $\mathsf{id\ x = x}$, is crash free ($\mathsf{id\ ::: CF}$). This is not provable without an axiom like

$$\forall e.\ (\forall x. \mathsf{CF}(x) \rightarrow \mathsf{CF}(e\ x)) \rightarrow \mathsf{CF}(e)\ ,$$

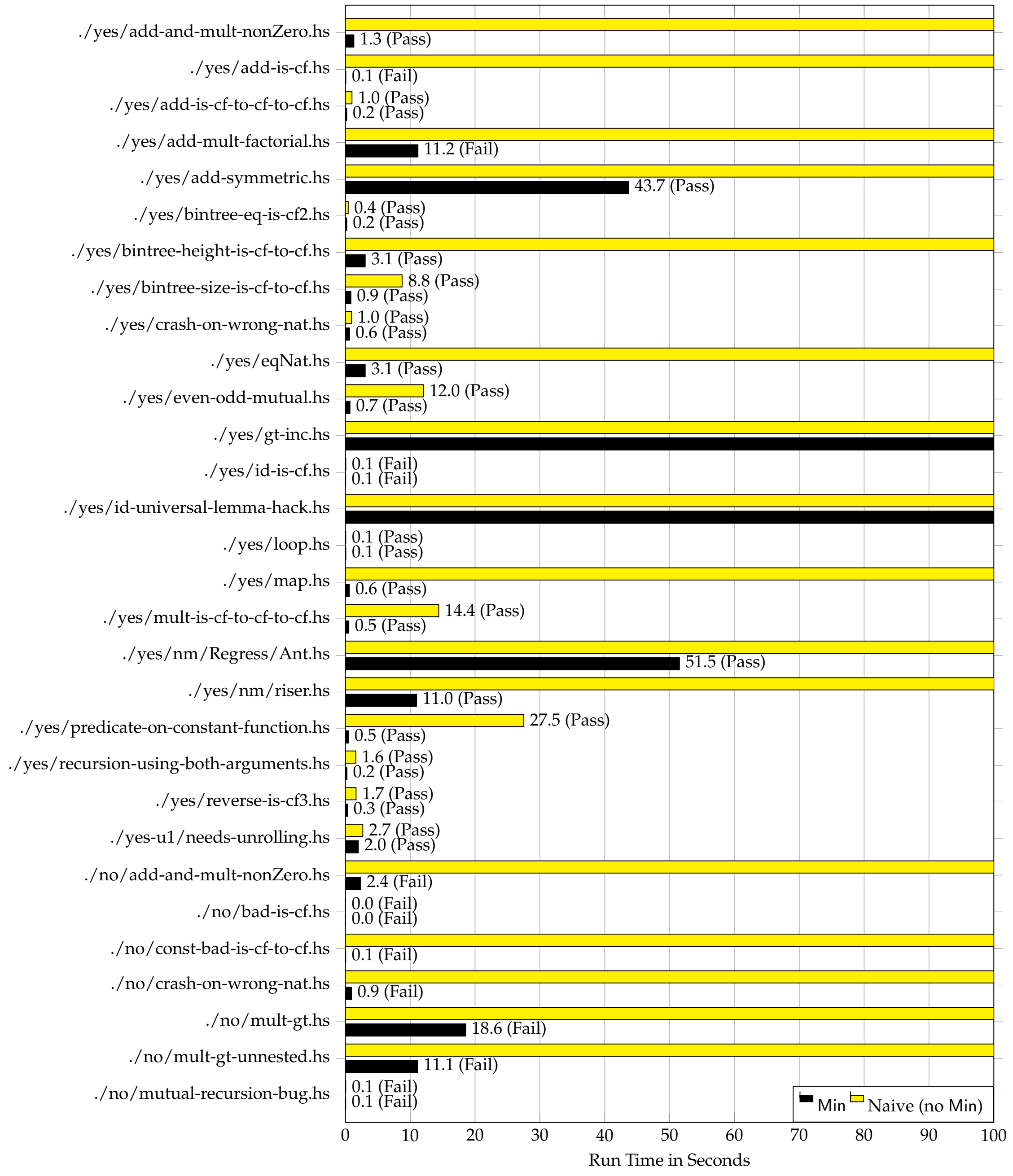

Fig. 6. Run time and outcome summary for naive and Min translations.

The test files are listed on the left side. The bars measure run time, and are labeled with run time and outcome (“Pass” meaning “proof found” and “Fail” meaning “counter model found”). Bars extending all the way to the right (100 seconds) correspond to time out. All test files in `./no` contain contracts that aren’t satisfied, and all files in `./yes` contain contracts that are satisfied or are not provable from the generated axioms.

but we found such axioms to cause significant performance degradation, and so we don't include one.[9] When some contract is underspecified, the desired outcome is "Fail".

Figure 6 shows that the Min-translation is effective on our example programs, but it doesn't tell us anything about the development process of adding and modifying contracts until the prover is finally able to verify the program. In practice, we find it is difficult to identify the cause, when the prover fails by timing out or by finding a counter model. The difficulty is that the system provides little feedback. When the prover times out we get no feedback at all. When a counter model is found, it must be the case that one of the contracts is violated in the counter model, but it can be hard to identify the violation. When the contracts are satisfied in Haskell, but our generated theory is not strong enough to prove this, it is usually because we did not axiomatize an induction needed to prove an unknown lemma. In the counter model, this will be manifested by an infinite term (e.g. xs s.t. $xs = \mathsf{Cons}\ \mathsf{Zero}\ xs$) for which the unknown lemma is violated. Our task then is to identify the violated unknown lemma and add an appropriate contract. This can be very difficult, because the counter model may be complex and violated lemma is unknown!

The counter models found by Equinox are always finite, but not usually minimal. We had some luck using another tool, Paradox, which finds minimal counter models. In some cases Paradox could find counter models with a domain of size three or four, which we could inspect manually. In fact, Paradox was helpful in getting the `./yes/add-symmetric.hs` example working, because it allowed us to identify some missing CF annotations. But, these are the simplest kind of contract, and it's not clear how to use Paradox to identify more complex missing contracts.

5 Related Work

In this paper we explored statically-checked higher-order contracts for a lazy functional language. Contracts have a long and rich history in computer science, with variations on all these parameters: static or dynamic contract checking, only first or also higher order functions, lazy (call-by-name) or strict (call-by-value) evaluation, and functional or other language paradigm. In this section we briefly mention some of the important but not-closely-related work, and then discuss the closely-related work in more detail.

Dynamically-checked first-order contracts appeared as early as the 1980s, in the object-oriented language Eiffel [8]. Statically-checked first-order contracts appeared in the *Extended Static Checking* system for the object-oriented language Modula-3 [4]. Dynamically-checked higher-order contracts appeared in the strict functional language Scheme [5]. Dynamically-checked [7] and statically-chekced [14] higher-order contracts appeared in the lazy language Haskell. Our

[9] And without an axiom like this, ϕ_{CF} is not adequate for term constructors with higher order arguments.

goal of ruling out run-time exceptions is an inherently static problem, because the failure of a dynamic contract check results in a run-time exception.

Our present system, and all of systems above, are fully automatic in the sense that the user tells the system *what* contracts to check but not *how* to check them.[10] A contrasting approach is interactive theorem proving, which combines automation with manual proof specification. Generic interactive theorem proving systems, such as Isabelle [9], Coq [3], and Agda [10], support the verification of arbitrary properties of programs. In such systems the object language (for us Haskell) must be modeled [1], rather than reasoned about directly. Alternatively, one can design a special logic specific to the object language, such as P-logic, which is designed for Haskell and uses Haskell as its term language [6]. Although our term language t is Haskell-like, FOL knows nothing about Haskell, and so our approach is closer to the encoding approach than to P-logic.

Systems with dynamically-checked higher-order contracts must track *blame*, because higher-order arguments and return values are checked when they are used, not when they are produced. The Scheme work [5] developed blame for Scheme contracts, and later work [12] developed blame more generally, relating it to subtyping, and hybrid (static and dynamic) type checking.

William Sonnex et al. developed an automatic equational-theorem prover for a strict variant of Haskell [11]. Because of the strictness, it's really more like an automatic prover for ML, but it's very effective in practice, proving many properties fully automatically using novel induction heuristics. In conversation William said that he had abandoned attempts to incorporate laziness into this system.

Our present work is an extension of earlier work by Dana Xu, Simon Peyton-Jones, and Koen Claessen [15] [14]. That earlier work designed the contract system for Haskell, and a technique for statically checked contracts by symbolic evaluation. By a source-to-source translation, they reduced the general problem of contract checking to crash-freeness checking. Their frustration with the ineffectiveness of the symbolic-evaluation approach led to our present effort.

In parallel with our present work, Xu developed a hybrid-checked higher-order contract system for the strict functional language OCaml [16]. Of all the work reported in this section, this most recent work of Xu is probably the closest to our present work. The static-checking part of Xu's system uses a combination of symbolic evaluation and automatic theorem proving. The automatic theorem proving part uses an SMT solver and translation to a polymorphic typed FOL. This system is still based on reduction of general contract checking to crash-freeness checking; the theorem prover is used to aid the symbolic simplification, by proving that branches in case statements are unreachable. Unlike our present work and Xu's earlier work, Xu's most recent system requires all refinement predicates to be terminating, and diverging predicates are unsound. Towards this end she employs an automatic termination checker.

[10] Our lemma encoding trick could be considered a form of interactivity, but we still consider our system to be fully automatic.

6 Future Work

To improve the usability of `hcc` we need it to give more feedback when it fails. There is some hope in the counter models, but we're not yet sure how to utilize them in a systematic way. In our examples, the majority of the contract annotations we wrote were of the form $\mathsf{CF} \to \cdots \to \mathsf{CF}$, and we believe these simple contracts could be suggested automatically. Also, the structure of many lemma functions is completely determined by their contract (our lemma example in Section 3.3 is unusual in this respect), and so the lemma functions could probably be generated automatically.

Another way to improve feedback is make the proof obligations small and numerous. Xu's recent system [16] can report each path to a syntactic crash which the symbolic simplification phase was unable to eliminate. Following Xu and making a separate proof obligation for each path through a function, we could report back to the user a precise execution path for which we weren't able to verify a contract.

By adding polymorphic contracts, which quantify contract variables, we could improve the expressivity of the contract system. The natural contract for the identity function, $\mathsf{id}\ \mathsf{x} = \mathsf{x}$, is $\mathsf{id} ::: \forall c.c \to c$. We don't currently support quantification over contracts, and neither does the earlier Haskell contracts work [16] [15] [14], but we believe we've got the details mostly worked out on paper.

7 Conclusion

We set out to create an effective system for statically checking Haskell contracts. Our approach was to translate to first-order logic, and then to invoke an automatic theorem prover on the translation. We designed two translations: a more naive translation, which performs poorly, and a more sophisticated translation, based on the naive translation, which performs much better. Our system is moderately effective at checking contracts, but the feedback it provides on failure leaves much to be desired.

Acknowledgments. Thanks to Max Orhai, Caylee Hogg, and Garrett Morris for providing feedback on drafts of this paper. Special thanks to Tim Sheard for providing feedback, support, and encouragement throughout the entire RPE paper writing process.

Bibliography

[1] Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying haskell programs using constructive type theory. In: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell. pp. 62–73. ACM (2005)

[2] Astolfi, C.P., Collins, N., Vytiniotis, D.: Haskell contract checker (hcc) tool source code repository (2012), http://github.com/cpa/haskellcontracts

[3] Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer-Verlag New York Inc (2004)

[4] Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. In: SRC RESEARCH REPORT 159, COMPAQ SYSTEMS RESEARCH CENTER (1998)

[5] Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. pp. 48–59. ICFP '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/581478.581484

[6] Harrison, W., Kieburtz, R.: The logic of demand in haskell. Journal of Functional Programming 15(06), 837–891 (2005)

[7] Hinze, R., Jeuring, J., Löh, A.: Typed contracts for functional programming. Functional and Logic Programming pp. 208–225 (2006)

[8] Meyer, B.: Eiffel: the language. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)

[9] Nipkow, T., Wenzel, M., Paulson, L.: Isabelle/HOL: a proof assistant for higher-order logic. Springer-Verlag (2002)

[10] Norell, U.: Dependently typed programming in agda. Advanced Functional Programming pp. 230–266 (2009)

[11] Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: TACAS. Lecture Notes in Computer Science (March 2012), http://pubs.doc.ic.ac.uk/zenoTwo/

[12] Wadler, P., Findler, R.: Well-typed programs cant be blamed. Programming Languages and Systems pp. 1–16 (2009)

[13] Wikipedia: First-order logic: Equality and its axioms — Wikipedia, the free encyclopedia (2012), https://en.wikipedia.org/w/index.php?title=First-order_logic&oldid=488767337#Equality_and_its_axioms, [Online; accessed 29-April-2012]

[14] Xu, D.N.: Extended static checking for haskell. In: Löh, A. (ed.) Haskell. pp. 48–59. ACM (2006)

[15] Xu, D.N., Peyton-Jones, S.L., Claessen, K.: Static contract checking for haskell. In: Shao, Z., Pierce, B.C. (eds.) POPL. pp. 41–52. ACM (2009)

[16] Xu, D.: Hybrid contract checking via symbolic simplification. In: Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation. pp. 107–116. ACM (2012)

8 Appendix

8.1 The Full-Application Optimization

The careful reader may have noticed that we have not yet used the function symbols $\hat{f}$ and $\hat{K}$ from the term grammar (t) in Figure 1. We use these to simplify our formulas and hence improve the performance of the theorem prover. Wherever a function f or constructor K appears *fully applied*, i.e., applied to as many arguments as appear on the left side of its definition, we replace the full application with a function call, to $\hat{f}$ or $\hat{K}$ respectively, and add an axiom relating the two.

More precisely, for each function f defined by $f\ \overline{x}^m = b$ and each term constructor K defined by $\mathsf{data}\ T\ \overline{\alpha} = \cdots K\ \overline{x}^n \cdots$, we replace all occurrences of $f\ x_1 \cdots\ x_m$ with $\hat{f}(x_1, \ldots, x_m)$ and all occurrences of $K\ x_1 \cdots\ x_n$ with $\hat{K}(x_1, \ldots, x_n)$ in the theory, and then add axioms

$$\forall x_1, \ldots, x_m.\ \Big(\mathsf{Min}(f\ x_1 \cdots\ x_m) \vee \mathsf{Min}(\hat{f}(x_1, \ldots, x_m))\Big) \rightarrow f\ x_1 \cdots\ x_m = \hat{f}(x_1, \ldots, x_m)$$

and

$$\forall x_1, \ldots, x_n.\ \big(\mathsf{Min}(K\ x_1 \cdots\ x_n) \vee \mathsf{Min}(\hat{K}(x_1, \ldots, x_n))\big) \rightarrow\ K\ x_1 \cdots\ x_n = \hat{K}(x_1, \ldots, x_n)$$

to the prelude (ϕ_{Prelude}).

8.2 Multiple Unrollings

Recall (Figure 3) that when attempting to prove contract satisfaction ($[\![e ::: c]\!]_{\mathsf{c}}$) in $\phi_{\mathcal{C}_{\mathcal{F}}}$, we replace all recursive calls to $(f'\ \overline{x} = e) \in \mathcal{F}$ with the opaque functions f'_{rec} for which we assume the corresponding contracts are satisfied ($\phi_{\mathcal{C}_{\mathcal{F}_{\mathsf{rec}}}}$). This gives us proof by induction, but limits us to one unrolling of function definitions, which is sometimes problematic. For example, suppose we wanted to prove that

$$\mathsf{length}\ (\mathsf{Cons\ Zero\ Nil}) = \mathsf{Succ\ Zero}\ .$$

We would get as far as

$$\mathsf{length}\ (\mathsf{Cons\ Zero\ Nil}) = \mathsf{Succ}\ (\mathsf{length_{rec}\ Nil})\ ,$$

and then be stuck, because we have no evaluation rules for opaque function $\mathsf{length_{rec}}$.

To workaround this problem, we allow the user to specify the number of unrollings to support. For example, if the user asks for one level of unrolling, then instead of redefining length to call the opaque $\mathsf{length_{rec}}$ recursively, we define length to call a new function $\mathsf{length_1}$ recursively, and then define $\mathsf{length_1}$ the same as length, except to call the opaque $\mathsf{length_{rec}}$ recursively. Hence our earlier example becomes

$$\mathsf{length}\ (\mathsf{Cons\ Zero\ Nil}) = \mathsf{Succ}\ (\mathsf{length_1\ Nil}) = \mathsf{Succ\ Zero}\ ,$$

and we succeed.

More precisely, suppose the user requests N unrollings. Recall that the insertion of opaque functions f'_{rec} only applies to the checked functions in the SCC $\mathcal{F}$, and not to the remaining functions $\mathcal{G}$. So, we replace $\phi_{\mathcal{F}_{\mathsf{rec}}}$ in Figure 3 with

$$\phi_{\mathcal{F}_{\mathsf{rec}}} = \bigwedge_{(f\,\overline{x}=b)\in\mathcal{F}} \bigwedge_{i=0}^{N+1} \left[\!\!\left[f_i\;\overline{x} = \left(b[f'_{i+i}/f']_{(f'\,\_=\_)\in\mathcal{F}} \right) \right]\!\!\right]_{\mathtt{f}} ,$$

where $f'_0 := f$, $f'_{N+1} := f'_{\mathsf{rec}}$, and $\{f'_i\}_{i=1}^{N}$ are new symbols, for each $(f'\;\_ = \_) \in \mathcal{F}$.