HALO: Haskell to Logic through Denotational Semantics

As submitted to POPL'13, July 11, 2012

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research

Dan Rosén
Koen Claessen
Chalmers University

Abstract

Even well-typed programs can go wrong, by encountering a pattern-match failure, or simply returning the wrong answer. An increasingly-popular response is to allow programmers to write *contracts* that express semantic properties, such as crash-freedom or some useful post-condition. We study the *static verification* of such contracts. Our main contribution is a novel translation to first-order logic of both Haskell programs, and contracts written in Haskell, all justified by denotational semantics. This translation enables us to prove that functions satisfy their contracts using an off-the-shelf first-order logic theorem prover.

1. Introduction

Haskell programmers enjoy the benefits of strong static types and purity: static types eliminate many bugs early on in the development cycle, and purity simplifies equational reasoning about programs. Despite these benefits, however, bugs may still remain in purely functional code and programs often crash if applied to the wrong arguments. For example, consider these Haskell definitions:

```
f xs = head (reverse (True : xs))
g xs = head (reverse xs)
```

Both `f` and `g` are well typed (and hence do not "go wrong" in Milner's sense), but `g` will crash when applied to the empty list, whereas `f` cannot crash regardless of its arguments. To distinguish the two we need reasoning that goes well beyond that typically embodied in a standard type system.

Many variations of *dependent type systems* (Norell 2007; Swamy et al. 2011; Xi 2007) or *refinement type systems* (Knowles and Flanagan 2010; Rondon et al. 2008) have been proposed to address this problem, each offering different degrees of expressiveness or automation. Another line of work aiming to address this challenge, studied by many researchers as well (Blume and McAllester 2006; Findler and Felleisen 2002; Knowles and Flanagan 2010; Siek and Taha 2006; Wadler and Findler 2009), allows programmers to annotate functions with *contracts*, which are forms of behavioural specifications. For instance, we might write the following contract for `reverse`:

$$\texttt{reverse} \in (xs : \texttt{CF}) \to \{ys \mid \texttt{null}\ xs\ \texttt{<=>}\ \texttt{null}\ ys\}$$

This contract annotation asserts that if `reverse` is applied to a crash-free (`CF`) argument list `xs` then the result `ys` will be empty (`null`) if and only if `xs` is empty. What is a crash-free argument? Since we are using lazy semantics, a list could contain cons-cells that yield errors when evaluated, and the `CF` precondition asserts that the input list is not one of those. Notice also that `null` and `<=>` are just ordinary Haskell functions, perhaps written by the programmer, even though they appear inside contracts.

With this property of `reverse` in hand, we might hope to prove that `f` satisfies the contract

$$\texttt{f} \in \texttt{CF} \to \texttt{CF}$$

But how do we verify that `reverse` and `f` satisfy the claimed contracts? Contracts are often tested dynamically, but our plan here is different: we want to verify contracts *statically* and *automatically*.

It should be clear that there is a good deal of logical reasoning to do, and a now-popular approach is to delegate the task to an off-the-shelf theorem prover such as Z3 (De Moura and Bjørner 2008) or Vampire (Hoder et al. 2010), or search for counterexamples with a finite model finder (Claessen and Sörensson 2003). With that in mind, we make the following new contributions:

- We give a translation of Haskell programs to first-order logic (FOL) theories. It turns out that lazy programs (as opposed to strict ones!) have a very natural translation into first-order logic (Section 3).
- We give a translation of contracts to FOL formulae, and an axiomatisation of the language semantics in FOL (Section 3.5).
- Our main contribution is to show that if we can prove the formula that arises from a contract translation for a given program, then the program does indeed satisfy this contract. Our proof uses the novel idea of employing the denotational semantics as a first-order model (Section 4).
- We show how to use this translation in practice for static contract checking with a FOL theorem prover (Section 4.5), and how to prove goals by induction (Section 5).

This work is a first step towards practical contract checking for Haskell programs, laying out the theoretical foundations for further engineering and experimentation. Nevertheless, we have already implemented a prototype for Haskell programs that uses GHC as a front-end. We have evaluated the practicality of our approach on many examples, including lazy and higher-order programs, and goals that require induction. We report this initial encouraging evaluation in Section 6.

To our knowledge no one has previously presented a translation of lazy higher-order programs to first-order logic, in a provably sound way with respect to a denotational semantics. Furthermore, our approach to static contract checking is distinctly different to previous work: instead of wrapping and symbolic execution (Xu 2012; Xu et al. 2009), we harness purity and laziness to directly use the denotational semantics of programs and contracts and discharge the obligations with a FOL theorem prover, side-stepping the wrapping process. Instead of generating verification conditions by pushing pre- and post- conditions through a program, we directly ask a theorem prover to prove a contract for the FOL encoding of a program. We discuss related work in Section 8.

Programs, definitions, and expressions

$$\begin{array}{lcll} P & ::= & d_1 \ldots d_n & \\ d & ::= & f\ \overline{a}\ \overline{(x{:}\tau)} = u & \\ u & ::= & e \mid \texttt{case}\ e\ \texttt{of}\ \overline{K\ \overline{y} \to e} & \\ e & ::= & x & \text{Variables} \\ & \mid & f[\overline{\tau}] & \text{Function variables} \\ & \mid & K[\overline{\tau}](\overline{e}) & \text{Data constructors (saturated)} \\ & \mid & e\ e & \text{Applications} \\ & \mid & \texttt{BAD} & \text{Runtime error} \end{array}$$

Syntax of closed values

$$v, w \quad ::= \quad K^n[\overline{\tau}](\overline{e}^n) \ \mid\ f^n[\overline{\tau}]\ \overline{e}^{m<n} \ \mid\ \texttt{BAD}$$

Contracts

$$\begin{array}{lcll} \texttt{C} & ::= & \{x \mid e\} & \text{Base contracts} \\ & \mid & (x : \texttt{C}_1) \to \texttt{C}_2 & \text{Arrow contracts} \\ & \mid & \texttt{C}_1 \& \texttt{C}_2 & \text{Conjunctions} \\ & \mid & \texttt{CF} & \text{Crash-freedom} \end{array}$$

Types

$$\begin{array}{lcll} \tau, \sigma & ::= & T\ \overline{\tau} & \text{Datatypes} \\ & \mid & a \mid \tau \to \tau & \end{array}$$

Type environments and signatures

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, x \\ \Delta & ::= & \cdot \mid \Delta, a \mid \Delta, x{:}\tau \\ \Sigma & ::= & \cdot \mid \Sigma, T{:}n \mid \Sigma, f{:}\forall \overline{a}\,.\,\tau \mid \Sigma, K^n{:}\forall \overline{a}\,.\,\overline{\tau}^n \to \texttt{T}\ \overline{a} \end{array}$$

Auxiliary functions

$$\begin{array}{lcl} (\cdot)^- & = & \cdot \\ (\Delta, a)^- & = & \Delta^- \\ (\Delta, (x{:}\tau))^- & = & \Delta^-, x \end{array}$$

Figure 1: Syntax of λ_{HALO} and its contracts

2. A higher-order lazy language and its contracts

To formalise the ideas behind our implementation, we define a tiny source language λ_{HALO}: a polymorphic, higher-order, call-by-name λ-calculus with algebraic datatypes, pattern matching, and recursion. Our actual implementation treats all of Haskell, by using GHC as a front end to translate Haskell into language λ_{HALO}.

2.1 Syntax of λ_{HALO}

Figure 1 presents the syntax of λ_{HALO}. A program P consists of a set of recursive function definitions $d_1 \ldots d_n$. Each definition has a left hand side that binds its type-variable and term-variable parameters; if f has n term-variable parameters we say that it has arity n, and sometimes write it f^n. The right hand side u of a definition is either a `case` expression or a `case`-free expression e. A `case`-free expression consists of variables x, function variables $f[\overline{\tau}]$ fully applied to their type arguments, applications $e_1\ e_2$, data constructor applications $K[\overline{\tau}](\overline{e})$. As a notation, we use $\overline{x}^n$ for sequences of elements of size n. When n is omitted $\overline{x}$ has a size which is implied by the context or is not interesting.

A program *crashes* if it evaluates the special value BAD. For example, we assume that the standard Haskell function `error` simply invokes BAD, thus:

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

Moreover, we assume that all incomplete pattern-matches are completed, with the missing case yielding BAD. For example:

```
head :: [a] -> [a]
head (x:xs) = x
head []     = BAD
```

In our context, BAD is our way to saying what it means for a program to "go wrong", and verification amounts to proving that a program cannot invoke BAD.

Our language embodies several convenient syntactic constraints: (i) λ abstractions occur only at the top-level, (ii) `case`-expressions can only immediately follow a function definition, and (iii) constructors are fully applied. Any Haskell program can be transformed into this restricted form by lambda-lifting, `case`-lifting, and eta-expansion respectively, and our working prototype does just this. However this simpler language is extremely convenient for the translation of programs to first-order logic.

λ_{HALO} is an explicitly-typed language, and we assume the existence of a typing relation $\Sigma \vdash P$, which checks that a program conforms to the definitions in the signature Σ. A signature Σ (Figure 1) records the declared data types, data constructors and types of functions in the program P. The well-formedness of expressions is checked with a typing relation $\Sigma\ ;\ \Delta \vdash u : \tau$, where Δ is a typing environment, also in Figure 1. We do not give the details of the typing relation since it is standard. Our technical development and analysis in the following sections assume that the program has been checked for type errors. The typing judgement should check that all pattern matches are exhaustive; as mentioned above, missing cases should return BAD.

The syntax of closed values is also given in Figure 1. Since we do not have arbitrary λ-abstractions, values can only be partial function applications $f^n[\overline{\tau}]\ \ \overline{e}^{m<n}$, data constructor applications $K[\tau](\overline{e})$, and the error term BAD.

The operational semantics of λ_{HALO} is entirely standard, and we do not give it here. We write $P \vdash u \Downarrow v$ to mean "in program P, right-hand side u evaluates to value v",

2.2 Contracts

We now turn our attention to contracts. The syntax of contracts is given in Figure 1 and includes base contracts $\{x \mid e\}$, arrow contracts $(x : \texttt{C}_1) \to \texttt{C}_2$, conjunctions $\texttt{C}_1 \& \texttt{C}_2$ and crash-freedom CF. Previous work (Xu et al. 2009) includes other constructs as well, but the constructs we give here are enough to verify many programs and exhibit the interesting theoretical and practical problems.

We write $e \in C$ to mean "the expression e satisfies the contract C", and similarly for functions f. We will say what contracts mean formally in Section 4.3. However, here is their informal meaning:

- $e \in \{x \mid e'\}$ means that e does not evaluate to a value or $e[e'/x]$ evaluates to `True` or does not evaluate to a value. Notice that e' is an arbitrary expression (in our implementation, arbitrary Haskell expressions), rather than being restricted to some well behaved meta-language. This is great for the programmer because the language and its library functions is familiar, but it poses a challenge for verification because these expressions in contracts may themselves diverge or crash.
- $e \in (x : \texttt{C}_1) \to \texttt{C}_2$ means that whenever e' satisfies $\texttt{C}_1$, it is the case that $(e\, e')$ satisfies $\texttt{C}_2[e'/x]$.
- $e \in \texttt{C}_1 \& \texttt{C}_2$ means that e satisfies both $\texttt{C}_1$ and $\texttt{C}_2$.
- $e \in$ CF means that e is *crash free*; that is e does not crash regardless of what context it is plugged into (see Section 3.6).

$$
\begin{array}{llll}
\text{Terms} & & & \\
s, t & ::= & x & \text{Variables} \\
 & | & f(\overline{t}) & \text{Function applications} \\
 & | & K(\overline{t}) & \text{Constructor applications} \\
 & | & \mathit{sel\_K_i}(t) & \text{Constructor selectors} \\
 & | & f_{ptr} \mid \mathit{app}(t, s) & \text{Pointers and application} \\
 & | & \mathit{unr} \mid \mathit{bad} & \text{Unreachable, bad} \\
 & & & \\
\text{Formulae} & & & \\
\varphi & ::= & \mathsf{cf}(t) & \text{Crash-freedom} \\
 & | & t_1 = t_2 & \text{Equality} \\
 & | & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi & \\
 & | & \forall x \,.\, \varphi \mid \exists x \,.\, \varphi & \\
 & & & \\
\text{Abbreviations} & & & \\
 & \mathit{app}(t, \overline{s}^n) & = & (\ldots(\mathit{app}(t, s_1), \ldots s_n) \ldots) \\
 & \varphi_1 \Rightarrow \varphi_2 & = & \neg\varphi_1 \vee \varphi_2
\end{array}
$$

Figure 2: Syntax of FOL

3. Translating λ_{HALO} to first-order logic

Our goal is to answer the question “does expression e satisfy contract C?”. Our plan is to translate the expression and the contract into a first-order logic (FOL) term and formula respectively, and get a standard FOL prover to do the heavy lifting. In this section we formalise our translation, and describe how we use it to verify contracts.

3.1 The FOL language

We begin with the syntax of the FOL language, which is given in Figure 2. There are two syntactic forms, *terms* and *formulae*. Terms include saturated function applications $f(\overline{t})$, saturated constructor applications $K(\overline{t})$, and variables. They also include, for each data constructor K^n in the signature Σ with arity n a set of *selector functions* $\mathit{sel\_K_i}(t)$ for $i \in 1 \ldots n$. The terms $\mathit{app}(t, s)$ and f_{ptr} concern the higher-order aspects of λ_{HALO} (namely un-saturated applications), which we discuss in Section 3.3. Finally we introduce two new syntactic constructs unr and bad. As an abbreviation we often use $\mathit{app}(t, \overline{s})$ for the sequence of applications to each s_i, as Figure 2 shows.

A formula φ in Figure 2 is just a first-order logic formula, augmented with a predicate $\mathsf{cf}(t)$ for crash-freedom, which we discuss in Section 3.6.

3.2 Translation of expressions to FOL

What exactly does it mean to translate an expression to first-order logic? We are primarily interested in reasoning about equality, so we might hope for this informal guiding principle:

> If we can prove[1] in FOL that $\mathcal{E}\{\!\{e_1\}\!\} = \mathcal{E}\{\!\{e_2\}\!\}$ then e_1 and e_2 are semantically equivalent.

where $\mathcal{E}\{\!\{e\}\!\}$ is the translation of e to a FOL term. That is, we can reason about the equality of Haskell terms by translating them into FOL, and then using a FOL theorem prover. The formal statement of this property is Corollary 4.5

The translation of programs, definitions, and expressions to FOL is given in Figure 3. The function $\mathcal{P}\{\!\{P\}\!\}$ translates a program to a conjunction of formulae, one for each definition d, using $\mathcal{D}\{\!\{d\}\!\}$ to

[1] From an appropriate axiomatisation of the semantics of programs.

$$
\boxed{\mathcal{P}\{\!\{P\}\!\} = \varphi} \quad \mathcal{P}\{\!\{\overline{d}\}\!\} = \bigwedge \overline{\mathcal{D}\{\!\{d\}\!\}}
$$

$$
\boxed{\mathcal{D}\{\!\{d\}\!\} = \varphi}
$$

$$
\begin{array}{rcl}
\mathcal{D}\{\!\{f\ \overline{a}\ \overline{(x{:}\tau)} = u\}\!\} & = & \forall \overline{x} \,.\, \mathcal{U}(f(\overline{x}))\{\!\{u\}\!\} \\
 & \wedge & \forall \overline{x} \,.\, f(\overline{x}) = \mathit{app}(f_{ptr}, \overline{x})
\end{array}
$$

$$
\boxed{\mathcal{U}(s)\{\!\{u\}\!\} = \varphi}
$$

$$
\begin{array}{ll}
\mathcal{U}(s)\{\!\{e\}\!\} \ = & (s = \mathcal{E}\{\!\{e\}\!\}) \\
\multicolumn{2}{l}{\mathcal{U}(s)\{\!\{\texttt{case}\ e\ \texttt{of}\ \overline{K\ \overline{y} \to e'}\}\!\}} \\
\quad = & (t = \mathit{bad} \Rightarrow s = \mathit{bad}) \\
\quad \wedge & (\forall \overline{y} \,.\, t = K_1(\overline{y}) \Rightarrow s = \mathcal{E}\{\!\{e_1'\}\!\}) \ \wedge \ldots \\
\quad \wedge & (t \neq \mathit{bad}\ \wedge\ t \neq K_1(\overline{\mathit{sel\_K_{1i}}(t)})\ \wedge\ \ldots \Rightarrow s = \mathit{unr}) \\
\quad \text{where} & t = \mathcal{E}\{\!\{e\}\!\}
\end{array}
$$

$$
\boxed{\mathcal{E}\{\!\{e\}\!\} = t}
$$

$$
\begin{array}{rcl}
\mathcal{E}\{\!\{x\}\!\} & = & x \\
\mathcal{E}\{\!\{f[\overline{\tau}]\}\!\} & = & f_{ptr} \\
\mathcal{E}\{\!\{K[\overline{\tau}](\overline{e})\}\!\} & = & K(\overline{\mathcal{E}\{\!\{e\}\!\}}) \\
\mathcal{E}\{\!\{e_1\ e_2\}\!\} & = & \mathit{app}(\mathcal{E}\{\!\{e_1\}\!\}, \mathcal{E}\{\!\{e_2\}\!\}) \\
\mathcal{E}\{\!\{\mathtt{BAD}\}\!\} & = & \mathit{bad}
\end{array}
$$

$$
\boxed{\mathcal{C}\{\!\{e \in \mathtt{C}\}\!\} = \varphi}
$$

$$
\begin{array}{rcl}
\mathcal{C}\{\!\{e \in \{(x{:}\tau) \mid e'\}\}\!\} & = & t = \mathit{unr} \\
 & \vee & t'[t/x] = \mathit{unr} \\
 & \vee & t'[t/x] = \mathit{True} \\
 & \text{where} & t\ =\ \mathcal{E}\{\!\{e\}\!\} \text{ and } t' = \mathcal{E}\{\!\{e'\}\!\} \\
\mathcal{C}\{\!\{e \in (x{:}\mathtt{C}_1) \to \mathtt{C}_2\}\!\} & = & \forall x \,.\, \mathcal{C}\{\!\{x \in \mathtt{C}_1\}\!\} \Rightarrow \mathcal{C}\{\!\{e\ x \in \mathtt{C}_2\}\!\} \\
\mathcal{C}\{\!\{e \in \mathtt{C}_1 \& \mathtt{C}_2\}\!\} & = & \mathcal{C}\{\!\{e \in \mathtt{C}_1\}\!\} \wedge \mathcal{C}\{\!\{e \in \mathtt{C}_2\}\!\} \\
\mathcal{C}\{\!\{e \in \mathtt{CF}\}\!\} & = & \mathsf{cf}(\mathcal{E}\{\!\{e\}\!\})
\end{array}
$$

Figure 3: Translation of programs and contracts to logic

translate each definition. The first clause in $\mathcal{D}$s right-hand side uses $\mathcal{U}$ to translate the right hand side u, and quantifies over the $\overline{x}$. We will deal with the second clause of $\mathcal{D}$ in Section 3.3.

Ignoring `case` for now (which we discuss in Section 3.4), the formula $\mathcal{U}(f(\overline{x}))\{\!\{e\}\!\}$ simply asserts the equality $f(\overline{x}) = \mathcal{E}\{\!\{e\}\!\}$. That is, we use a new function f in the logic for each function definition in the program, and assert that any application of f is equal to (the logical translation of) f's right hand side. Notice that we erase type arguments in the translation since they do not affect the semantics. You might think that the translation $f(\overline{x}) = \mathcal{E}\{\!\{e\}\!\}$ is entirely obvious but, surprisingly, it is only correct because we are in a call-by-name setting. The same equation is problematic in a call-by-value setting – an issue we return to towards the end of Section 4.4.

Lastly $\mathcal{E}\{\!\{e\}\!\}$ deals with expressions. We will deal with functions and application shortly (Section 3.3), but the other equations for $\mathcal{E}\{\!\{e\}\!\}$ are straightforward. Notice that $\mathcal{E}\{\!\{\mathtt{BAD}\}\!\} = \mathit{bad}$, and recall that `BAD` is the λ_{HALO}-term used for an inexhaustive `case` or a call

$\boxed{\text{Theory } \mathcal{T}}$

Axioms for bad and unr

AxAppBad	$\forall x\,.\,app(bad, x) = bad$
AxAppUnr	$\forall x\,.\,app(unr, x) = unr$
AxDisjBU	$bad \neq unr$

Axioms for data constructors

AxDisjC	$\forall \overline{x}^n \overline{y}^m\,.\,K(\overline{x}) \neq J(\overline{y})$
	for every $(K{:}\forall \overline{a}\,.\,\overline{\tau}^n \to T\ \overline{a}) \in \Sigma$ and $(J{:}\forall \overline{a}\,.\,\overline{\tau}^m \to S\ \overline{a}) \in \Sigma$
AxDisjCBU	$(\forall \overline{x}^n\,.\,K(\overline{x}) \neq unr\ \wedge\ K(\overline{x}) \neq bad)$
	for every $(K{:}\forall \overline{a}\,.\,\overline{\tau}^n \to T\ \overline{a}) \in \Sigma$
AxInj	$\forall \overline{y}^n\,.\ sel\_K_i(K(\overline{y})) = y_i$
	for every $K^n \in \Sigma$ and $i \in 1..n$

Axioms for crash-freedom

AxCfC	$\forall \overline{x}^n\,.\,\mathsf{cf}(K(\overline{x})) \Leftrightarrow \bigwedge \mathsf{cf}(\overline{x})$
	for every $(K{:}\forall \overline{a}\,.\,\overline{\tau}^n \to T\ \overline{a}) \in \Sigma$
AxCfBU	$\mathsf{cf}(unr) \wedge \neg\mathsf{cf}(bad)$

Figure 4: Theory $\mathcal{T}$: axioms of the FOL constants

to `error`. It follows from our guiding principle that for any e, if we manage to prove in FOL that $\mathcal{E}\{\!\{e\}\!\} = bad$, then the source program e must be semantically equivalent to `BAD`, meaning that it definitely crashes.

3.3 Translating higher-order functions

If λ_{HALO} was a first-order language, the translation of function calls would be easy:

$$\mathcal{E}\{\!\{f[\overline{\tau}]\ \overline{e}\}\!\} = f(\overline{\mathcal{E}\{\!\{e\}\!\}})$$

At first it might be surprising that we can also translate a *higher-order* language λ_{HALO} into first order logic, but in fact it is easy to do so, as Figure 3 shows. We introduce into the logic (a) a single new function *app*, and (b) a nullary constant f_{ptr} for each function f (see Figure 2). Then, the equations for $\mathcal{E}\{\!\{e\}\!\}$ translate application in λ_{HALO} to a use of *app* in FOL, and any mention of function f in λ_{HALO} to a use of f_{ptr} in the logic. For example:

$$\mathcal{E}\{\!\{\texttt{map f xs}\}\!\} = app(app(\texttt{map}_{ptr}, \texttt{f}_{ptr}), \texttt{xs})$$

assuming that `map` and `f` are top-level functions in the λ_{HALO}-program, and `xs` is a local variable. Once enough *app* applications stack up, so that $\texttt{map}_{ptr}$ is applied to two arguments, we can invoke the `map` function directly in the logic, an idea we express with the following axiom:

$$\forall xy\ .\ app(app(\texttt{map}_{ptr}, x), y) = \texttt{map}(x, y)$$

These axioms, one for each function f, are generated by the second clause of the rules for $\mathcal{D}\{\!\{d\}\!\}$ in Figure 3. (The notation $app(f, \overline{x})$ is defined in Figure 2.) You can think of $\texttt{map}_{ptr}$ as a "pointer to", or "name of" of, `map`. The *app* axiom for `map` translates a saturated use of `map`'s pointer into a call of `map` itself.

This translation of higher-order functions to first-order logic may be easy, but it is not complete. In particular, in first-order logic we can only reason about functions with a concrete first-order representation (i.e. the functions that we already have and their partial applications) but, for example, lambda expressions cannot be created during proof time. Luckily, the class of properties we reason about (contracts) never require us to do so.

3.4 Data types and `case` expressions

The second equation for $\mathcal{U}(s)\{\!\{u\}\!\}$ in Figure 3 deals with `case` expressions, by generating a conjunction of formulae, as follows:

- If the scrutinee t is bad (meaning that evaluating it invokes `BAD`) then the result s of the `case` expression is also bad. That is, `case` is strict in its scrutinee.
- If the scrutinee is an application of one of the constructors K_i mentioned in one of the `case` alternatives, then the result s is equal to the corresponding right-hand side, e'_i, after quantifying the variables $\overline{y}$ bound by the `case` alternative.
- Otherwise the result is unr. The bit before the implication $\Rightarrow$ is just the negation of the previous preconditions; the formula $t{\neq}K_1(\overline{sel\_K_1(t)})$ is the clumsy FOL way to say "t is not built with constructor K_1".

Why do we need the last clause? Consider the function `not`:

```
not :: Bool -> Bool
not True = False
not False = True
```

Suppose we claim that $\texttt{not} \in \texttt{CF} \to \texttt{CF}$, which is patently true. But if we lack the last clause above, the claim is *not* true in every model; for example `not` might crash when given the (ill-typed but crash-free) argument 3. The third clause above excludes this possibility by asserting that the result of `not` is the special crash-free constant unr if the scrutinee is ill-typed (i.e. not bad and not built with the constructors of the type). This is the whole reason we need unr in the first place. In general, if $\mathcal{E}\{\!\{e\}\!\} = unr$ is provable in the logic, then e is ill-typed, or divergent.

Of course, we also need to axiomatise the behaviour of data constructors and selectors, which is done in Figure 4:

- AxDisjCBU explains that a term headed by a data constructor cannot also be bad or unr.
- AxInj explains how the selectors sel_K_i work.
- AxDisjC tells the prover that all data constructors are pairwise disjoint. There are a quadratic number of such axioms, which presents a scaling problem. For this reason FOL provers sometimes offer a built-in notion of data constructors, so this is not a problem in practice, but we ignore this pragmatic issue here.

3.5 Translation of contracts to FOL

Now that we know how to translate *programs* to first order logic, we turn our attention to translating *contracts*. We do not translate a contract *per se*; rather we translate the claim $e \in \texttt{C}$. Once we have translated $e \in \texttt{C}$ to a first-order logic formula, we can ask a prover to prove it using axioms from the translation of the program, or axioms from Figure 4. If successful, we can claim that indeed e does satisfy C. Of course that needs proof, which we address in Section 4.4.

Figure 3 presents the translation $\mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$; there are four equations corresponding to the syntax of contracts in Figure 1. The last three cases are delightfully simple and direct. Conjunction of contracts turns into conjunction in the logic; a dependent function contract turns into universal quantification and implication; and the claim that e is crash-free turns into a use of the special term $\mathsf{cf}(t)$ in the logic. We discuss crash-freedom in Section 3.6.

The first equation, for predicate contracts $e \in \{x \mid e'\}$, is sightly more complicated. The first clause $t = unr$, together with the axioms for unr in Figure 4, ensures that unr satisfies every contract. The second and third say that the contract holds if e' diverges or is semantically equal to `True`. The choices embodied in this rule were discussed at length in earlier work (Xu et al. 2009) and we do not rehearse it here.

3.6 Crash-freedom

The claim $e \in$ `CF`, pronounced "e is crash-free", means that e cannot crash *regardless of context*. So, for example `(BAD, True)` is not crash-free because it can crash if evaluated in the context `fst (BAD, True)`. Of course, the context itself should not be the source of the crash; for example `(True,False)` is crash-free even though `BAD (True,False)` will crash.

We use the FOL term $\mathsf{cf}(t)$ to assert that t is crash-free. The axioms for cf are given in Figure 4. AXCFC says that a data constructor application is crash-free if and only iff its arguments are crash-free. AXCFBU says that *unr* is crash-free, and that *bad* is not. That turns out to be all that we need.

3.7 Summary

That completes our formally-described — but so far only informally-justified — translation from a λ_{HALO} program and a set of contract claims, into first-order logic. To a first approximation, we can now hand the generated axioms from the program and our axiomatisation to an FOL theorem prover and ask it to use them to prove the translation of the contract claims.

4. Soundness through denotational semantics

Our account so far has been largely informal. How can we be sure that if the FOL prover says "Yes! This FOL formula is provable", then the corresponding λ_{HALO} program indeed satisfies the claimed contract?

To prove this claim we take a denotational approach. Most of what follows is an adaptation of well-known techniques to our setting and there are no surprises — we refer the reader to (Winskel 1993) or (Benton et al. 2009) for a short and modern exposition of the standard methodology.

4.1 Technical preliminaries

We will assume a program P, well-formed in a signature Σ, so that $\Sigma \vdash P$. Given a signature Σ we define a strict bi-functor F on complete partial orders (cpos), below:

$$\begin{array}{rcll} F(D^-, D^+) & = & (\;\prod_{n_1} D^+ & K_1^{n_1} \in \Sigma \\ & + & \;\ldots & \ldots \\ & + & \;\prod_{n_k} D^+ & K_k^{n_k} \in \Sigma \\ & + & \;(D^- \Rightarrow_c D^+) & \\ & + & \;\mathbf{1}_{bad} \quad)_\perp & \end{array}$$

The bi-functor F is the lifting of a big sum: that sum consists of (i) products, one for each possible constructor (even across different data types), (ii) the continuous function space from D^- to D^+, and (iii) a unit cpo to denote `BAD` values. The notation $\prod_n D$ abbreviates n-ary products of cpos (the unit cpo $\mathbf{1}$ if $n = 0$). The product and sum constructions are standard, but note that we use their non-strict versions. The notation $C \Rightarrow_c D$ denotes the cpo induced by the space of continuous functions from the cpo C to the cpo D. We use the notation $\mathbf{1}_{bad}$ to denote a single-element cpo – the *bad* subscript is just there for readability. The notation $D_\perp$ is *lifting*.

Observe that we have dropped all type information from the source language. The elements of the products corresponding to data constructors are simply D^+ (instead of more a precise description from type information) and the return types of data constructors are similarly ignored. This is not to say that a more type-rich denotational semantics is not possible (or desirable even) but this simple denotational semantics turns out to be sufficient for formalisation and verification.

$$\begin{array}{rcl} \multicolumn{3}{l}{[\![e]\!]_{\langle\cdot,\cdot\rangle} : (\mathit{FunVar} \Rightarrow_c D_\infty) \times (\mathit{Var} \Rightarrow_c D_\infty) \Rightarrow_c D_\infty} \\ [\![x]\!]_{\langle\sigma,\rho\rangle} & = & \rho(x) \\ [\![f\ [\overline{\tau}]]\!]_{\langle\sigma,\rho\rangle} & = & \sigma(f) \\ [\![K\ [\overline{\tau}]\ (\overline{e})]\!]_{\langle\sigma,\rho\rangle} & = & \mathsf{K}(\overline{[\![e]\!]_{\langle\sigma,\rho\rangle}}) \\ [\![e_1\ e_2]\!]_{\langle\sigma,\rho\rangle} & = & \mathsf{app}([\![e_1]\!]_{\langle\sigma,\rho\rangle}, [\![e_2]\!]_{\langle\sigma,\rho\rangle}) \\ [\![\mathtt{BAD}]\!]_{\langle\sigma,\rho\rangle} & = & \mathsf{Bad} \end{array}$$

$$\begin{array}{rcl} \multicolumn{3}{l}{[\![u]\!]_{\langle\cdot,\cdot\rangle} : (\mathit{FunVar} \Rightarrow_c D_\infty) \times (\mathit{Var} \Rightarrow_c D_\infty) \Rightarrow_c D_\infty} \\ [\![e]\!]_{\langle\sigma,\rho\rangle} & = & [\![e]\!]_{\langle\sigma,\rho\rangle} \\ [\![\mathtt{case}\ e\ \mathtt{of}\ \overline{K\ \overline{y} \to e_K}]\!]_{\langle\sigma,\rho\rangle} & = & [\![e_K]\!]_{\langle\sigma,\rho,\overline{y \mapsto d}\rangle} \\ & & \quad \text{if } [\![e]\!]_{\langle\sigma,\rho\rangle} = \mathsf{K}(\overline{d}) \\ & & \quad \text{and } K \text{ is a } \mathtt{case} \text{ branch} \\ & = & \mathsf{Bad} \quad \text{if } [\![e]\!]_{\langle\sigma,\rho\rangle} = \mathsf{Bad} \\ & = & \perp \quad \text{otherwise} \end{array}$$

$$\begin{array}{l} [\![P]\!] : (\mathit{FunVar} \Rightarrow_c D_\infty) \Rightarrow_c (\mathit{FunVar} \Rightarrow_c D_\infty) \\ [\![P]\!]_\sigma f \;=\; \mathsf{Fun}(\lambda d_1 \ldots.\, \mathsf{Fun}(\lambda d_n \,.\, [\![u]\!]_{\langle\sigma,\overline{x \mapsto d}\rangle}) \ldots) \\ \qquad\qquad \text{if } (f\ \overline{a}\ \overline{x} = u) \in P \\ \qquad\quad = \perp \quad \text{otherwise} \end{array}$$

Figure 5: Denotational semantics of λ_{HALO}

Now we can define D_∞ as the solution to this recursive domain equation

$$D_\infty \approx F(D_\infty, D_\infty)$$

We can show that D_∞ exists using the standard *embedding-projection* pairs methodology. Moreover, we define the value domain V_∞ thus:

$$\begin{array}{rlll} V_\infty = & & \prod_{n_1} D_\infty & K_1^{n_1} \in \Sigma \\ & + & \ldots & \ldots \\ & + & \prod_{n_k} D_\infty & K_k^{n_k} \in \Sigma \\ & + & (D_\infty \Rightarrow_c D_\infty) & \\ & + & \mathbf{1}_{bad} & \end{array}$$

The following continuous functions also exist:

$$\begin{array}{rcl} \mathsf{ret} & : & D \Rightarrow_c D_\perp \\ \mathsf{bind}_{f:D \Rightarrow_c E_\perp} & : & D_\perp \Rightarrow_c E_\perp \\ \mathsf{roll} & : & (V_\infty)_\perp \Rightarrow_c D_\infty \\ \mathsf{unroll} & : & D_\infty \Rightarrow_c (V_\infty)_\perp \end{array}$$

However in what follows we will always elide these functions to reduce clutter.

To denote elements of V_∞ we use the following notation.

- $\mathsf{K}(d_1, \ldots, d_n)$ denotes the injection of the n-ary product of D_∞ into the component of the sum V_∞ corresponding to the n-ary constructor K.
- $\mathsf{Fun}(d)$ is the injection of an element of $D_\infty \Rightarrow_c D_\infty$ into the function component of V_∞
- Bad is the unit injection into V_∞.

4.2 Denotational semantics of expressions and programs

Figure 5 gives the denotational interpretations of expressions e, right hand sides u, and programs P, in terms of the domain-theoretic language and combinators we have defined.

$$\boxed{[\![\mathtt{C}]\!]_\rho \subseteq D_\infty}$$

$$[\![x \mid e]\!]_\rho = \{d \mid d = \bot \vee [\![e]\!]_{\rho,x\mapsto d} \in \{\mathsf{True}, \bot\}\}$$

$$[\![(x{:}\mathtt{C}_1) \to \mathtt{C}_2]\!]_\rho = \{d \mid \forall d' \in [\![\mathtt{C}_1]\!]_\rho \,.\, \mathsf{app}(d, d') \in [\![\mathtt{C}_2]\!]_{\rho,x\mapsto d'}\}$$

$$[\![\mathtt{C}_1 \& \mathtt{C}_2]\!]_\rho = \{d \mid d \in [\![\mathtt{C}_1]\!]_\rho \wedge d \in [\![\mathtt{C}_2]\!]_\rho\}$$

$$[\![\mathtt{CF}]\!]_\rho = F^\infty_{\mathsf{cf}}$$

where

$$\begin{aligned} F^\infty_{\mathsf{cf}} &= \{\bot\} \\ &\cup \{\, \mathsf{K}(\overline{d}) \mid K^n \in \Sigma,\ d_i \in F^\infty_{\mathsf{cf}}\} \\ &\cup \{\, \mathsf{Fun}(d) \mid \forall d' \in F^\infty_{\mathsf{cf}} \,.\ d(d') \in F^\infty_{\mathsf{cf}}\} \end{aligned}$$

Figure 6: Denotations of contracts

First, the denumerable set of term variable names $x_1, \ldots$ induces a discrete cpo *Var* and the denumerable set of function variable names $f_1, \ldots$ induces a discrete cpo *FunVar*. We define, *semantic term environments* to be the cpo ($Var \Rightarrow_c D_\infty$), and *semantic function environments* to be the cpo ($FunVar \Rightarrow_c D_\infty$).

Figure 5 defines the denotational semantics of expressions $[\![e]\!]$ as a continuous map from these two environments to D_∞. It is entirely straightforward except for application, which depends on the continuous function $\mathsf{app} : D_\infty \times D_\infty \Rightarrow_c D_\infty$, defined thus[2]:

$$\begin{aligned} \mathsf{app}(d, a) &= d_f(a) && \text{if } d = \mathsf{Fun}(d_f) \\ &= \mathsf{Bad} && \text{if } d = \mathsf{Bad} \\ &= \bot && \text{otherwise} \end{aligned}$$

That is, application applies the payload d_f if the function d comes from the appropriate component of V_∞, propagates Bad, and otherwise returns $\bot$.

The semantics of right-hand sides $[\![u]\!]$ is defined similarly. The semantics of a `case` expression is the semantics of the matching branch, if one exists. Otherwise, like application, it propagates Bad. In all other cases we return $\bot$, not Bad; all the missing cases can only be constructors of different datatypes than the datatype that K belongs to, because all `case` expressions are complete (Section 2.1). This treatment corresponds directly to our treatment of unr in Section 3.4.

Finally, Figure 5 gives the semantics of a program P, which should be read recalling its syntax in Figure 1. Since $[\![P]\!]$) is continuous, its limit exists and is an element of the cpo $FunVar \Rightarrow_c D_\infty$.

Definition 4.1. *We will refer to the limit of the $[\![P]\!]$ as $[\![P]\!]^\infty$ in what follows. Moreover, to reduce notational overhead below, for a program with no free variables we will use notation $[\![e]\!]$ to mean $[\![e]\!]_{\langle [\![P]\!]^\infty, \cdot \rangle}$, and $[\![e]\!]_\rho$ to mean $[\![e]\!]_{\langle [\![P]\!]^\infty, \rho \rangle}$*

Although we have not presented a formal operational semantics, we state the usual soundness and adequacy results:

Theorem 4.1 (Soundness and adequacy)**.** *Assume $\Sigma \vdash P$ and u with no free term variables. Then (i) if $P \vdash u \Downarrow v$ then $[\![u]\!] = [\![v]\!]$; and (ii) if $[\![e]\!] \neq \bot$, then $\exists v$ such that $P \vdash e \Downarrow v$.*

4.3 Denotational semantics of contracts

Now we are ready to say formally what it means for a function to satisfy a contract. We define the semantics of a contract as the set of denotations that satisfy it:

$$[\![\mathtt{C}]\!]_\rho \subseteq D_\infty$$

where `C` is a contract with free term variables in the semantic environment ρ. Figure 6 gives the definition of this function. A base contract $\{x \mid e\}$ is satisfied by $\bot$ or or by a computation that causes the predicate e to become $\bot$ or return *True*[3]. The denotation of an arrow contract, and of conjunction, are both straightforward.

The `CF` contract is a little harder. Intuitively an expression is crash-free iff it cannot crash if plugged into an arbitrary crash-free context. Of course this is a self-referential definition so how do we know it makes sense? The original paper (Xu et al. 2009) specified that an expression is crash free iff it cannot crash when plugged into a context that *syntactically* does not contain the `BAD` value. This is a reasonable definition in the operational semantics world, but here we can do better because we are working with elements of D_∞. Using techniques developed by Pitts (Pitts 1996) we can define crash-freedom denotationally as the greatest solution to the recursive equation for F^∞_{cf} in Figure 6. Technically, since the equation involves mixed-variance recursion, to show that such a fixpoint exists we have to use minimal invariance. In addition we get the following, which will be useful later on for induction:

Lemma 4.2. *$\bot \in F_{\mathsf{cf}}$ and F^∞_{cf} is admissible, that is if all elements of a chain are in F^∞_{cf} then so is its limit.*

4.4 Soundness of the logic translation

We have developed a formal semantics for expressions as well as contracts, so it is time we see how we can use this semantics to show that our translation to first-order logic is sound with respect to this semantics.

Our plan is to give an interpretation (in the FOL sense of the term) to our translated FOL terms, using the carrier set D_∞ as our model. Happily this is straightforward to do:

$$\begin{aligned} \mathcal{I}(f(\overline{t})) &= \mathsf{app}([\![f]\!], \overline{\mathcal{I}(t)}) \\ \mathcal{I}(app(t_1, t_2)) &= \mathsf{app}(\mathcal{I}(t_1), \mathcal{I}(t_2)) \\ \mathcal{I}(f_{ptr}) &= [\![f]\!] \\ \mathcal{I}(K(\overline{t})) &= \mathsf{K}(\overline{\mathcal{I}(t)}) \\ \mathcal{I}(sel\_K_i(t)) &= d_i \quad \text{if } \mathcal{I}(t) = \mathsf{K}(\overline{d}) \\ &= \bot \quad \text{otherwise} \\ \mathcal{I}(unr) &= \bot \\ \mathcal{I}(bad) &= \mathsf{Bad} \end{aligned}$$

The essential soundness theorem that states that our interpretation makes sense is the following.

Theorem 4.3 (Interpretation respects denotations)**.** *Assume that $\Sigma \vdash P$ and expression e does not contain any free variables. Then, if $\mathcal{E}\{\!\{e\}\!\} = t$ then $\mathcal{I}(t) = [\![e]\!]$.*

The proof is an easy induction on the size of the term e.

Our soundness results are expressed with the following theorem

Theorem 4.4. *If $\Sigma \vdash P$ then $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{T} \wedge \mathcal{P}\{\!\{P\}\!\}$*

As a corollary we get our "guiding principle" from the introduction.

Corollary 4.5. *Assume that $\Sigma \vdash P$ and e_1 and e_2 contain no free term variables. The following are true:*

- *$[\![e_1]\!] = [\![e_2]\!]$ iff $\mathcal{I}(\mathcal{E}\{\!\{e_1\}\!\}) = \mathcal{I}(\mathcal{E}\{\!\{e_2\}\!\})$.*
- *If $\mathcal{T} \wedge \mathcal{P}\{\!\{P\}\!\} \vdash \mathcal{E}\{\!\{e_1\}\!\} = \mathcal{E}\{\!\{e_2\}\!\}$ then $[\![e_1]\!] = [\![e_2]\!]$.*

[2] A small technical remark: we write the definition with pattern matching notation $\mathsf{app}(d, a)$ (instead of using π_1 for projecting out d and π_2 for projecting out a) but that is fine, since $\times$ is not a lifted construction. Also note that we are, as advertised, suppressing uses of bind, roll, etc.

[3] In previous work (Xu et al. 2009) the base contract also required *crash-freedom*. We changed this choice only for reasons of taste; both choices are equally straightforward technically.

Proof. The first part follows directly from Theorem 4.3. For the second part the left-hand side implies that $\mathcal{E}\{\!\{e_1\}\!\}$ and $\mathcal{E}\{\!\{e_2\}\!\}$ are equal in all models of $\mathcal{T} \wedge \mathcal{P}\{\!\{P\}\!\}$, in particular (using Theorem 4.4) by $\langle D_\infty, \mathcal{I}\rangle$ and by the first part the case is finished. $\square$

Theorem 4.6. *Assume that* e *and* C *contain no free term variables. Then the FOL translation of the claim* $e \in$ C *holds in the model if and only if the denotation of* e *is in the semantics of* C*. Formally:*

$$\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{e \in \texttt{C}\}\!\} \quad \Leftrightarrow \quad [\![e]\!] \in [\![\texttt{C}]\!]$$

Completeness of axiomatisation The D_∞ domain has a complex structure and there are many more facts that hold about elements of D_∞ that are not reflected in any of our axioms in $\mathcal{T}$. For instance, here are some admissible axioms that are valid:

$$\forall \overline{x}^n \,.\, app(f_{ptr}, \overline{x}) \neq unr \;\; \wedge app(f_{ptr}, \overline{x}) \neq bad$$
$$\wedge\; \forall \overline{y}^k \,.\, app(f_{ptr}, \overline{x}) \neq K(\overline{y})$$

for every $(f\ \overline{a}\ \overline{x}^m = u) \in P$ and $K \in \Sigma$ with $m > n$

These axioms assert that partial applications cannot be equated to any constructor, $\perp$ nor Bad. If the reader is worried that without a complete formalisation of all equalities of D_∞ it is impossible to prove any programs correct, we would like to reassure them that that is not the case as we shall see in the next section.

Lazy semantics simplifies the translation We have mentioned previously (Section 3.2) that the laziness of λ_{HALO} helps in keeping the translation simple, and here we explain why.

Whenever we use universal quantification in the logic, we really quantify over *any* denotation, including $\perp$ and Bad. In a call-by-name language, given a function `f x = True`, the axiom $\forall x \,.\, f(x) = \mathsf{True}$ is true in the intended denotational model. However, in a call-by-value setting, x is allowed to be interpreted as $\perp$. That means that the unguarded axiom is actually not true, because $f \perp \neq \mathsf{True}$. Instead we need the following annoying variation:

$$\forall x \,.\, x \neq bad \wedge x \neq unr \Rightarrow f(x) = t$$

Moreover, the axioms for the $app(\cdot,\cdot)$ combinator have to be modified to perform checks that the argument is not $\perp$ or Bad before actually calling a function. In a call-by-name language these guards are needed only for `case` and the function part of *app*.

These complications lead to a more complex first-order theory in the call-by-value case.

4.5 Contract checking as satisfiability

Having established the soundness of our translation, it is time we see in this section how we can use this sound translation to verify a program. The following theorem is then true:

Theorem 4.7 (Soundness)**.** *Assume that* e *and* C *contain only function symbols from* P *and no free term variables. Let* $\mathcal{T}_{all} = \mathcal{T} \wedge \mathcal{P}\{\!\{P\}\!\}$*. If* $\mathcal{T}_{all} \wedge \neg\mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$ *is unsatisfiable then* $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$ *and consequently* $[\![e]\!] \in [\![\texttt{C}]\!]$.

Proof. If there is no model for this formula then its negation must be valid (true in all models), that is $\neg\mathcal{T}_{all} \vee \mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$ is valid. By completeness of first-order logic $\mathcal{T}_{all} \vdash \mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$. This means that all models of $\mathcal{T}_{all}$ validate $\mathcal{C}\{\!\{f \in \texttt{C}\}\!\}$. In particular, for the denotational model we have that $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{T}_{all}$ and hence $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$. Theorem 4.6 finishes the proof. $\square$

Hence, to verify a program e satisfies a contract C we need to do the following:

- Generate formulae for the theory $\mathcal{T} \wedge \mathcal{P}\{\!\{P\}\!\}$
- Generate the negation of a contract translation: $\neg\mathcal{C}\{\!\{e \in \texttt{C}\}\!\}$
- Ask a SAT solver for a model for the conjunction of the above formulae

Incremental verification Theorem 4.7 gives us a way to check that an expression satisfies a contract. Assume that we are given a program P with a function $f \in dom(P)$, for which we have already shown that $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{f \in \texttt{C}_f\}\!\}$. Suppose next that we are presented with a "next" goal, to prove that $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{h \in \texttt{C}_h\}\!\}$. We may consider the following three variations of how to do this:

- Ask for the unsatisfiability of:
  $$\mathcal{T} \;\wedge\; \mathcal{P}\{\!\{P\}\!\} \;\wedge\; \neg\mathcal{C}\{\!\{h \in \texttt{C}_h\}\!\}$$
  The soundness of this query follows from Theorem 4.7 above.
- Ask for the unsatisfiability of:
  $$\mathcal{T} \;\wedge\; \mathcal{P}\{\!\{P\}\!\} \;\wedge\; \mathcal{C}\{\!\{f \in \texttt{C}_f\}\!\} \wedge \neg\mathcal{C}\{\!\{h \in \texttt{C}_h\}\!\}$$
  This query adds the *already proven* contract for f to the theory. If this formula is unsatisfiable, then its negation is valid, and we know that the denotational model is a model of the theory *and* of $\mathcal{C}\{\!\{f \in \texttt{C}_f\}\!\}$ and hence it must also be a model of $\mathcal{C}\{\!\{h \in \texttt{C}_h\}\!\}$.
- Ask for the unsatisfiability of:
  $$\mathcal{T} \;\wedge\; \mathcal{P}\{\!\{P \setminus f\}\!\} \;\wedge\; \mathcal{C}\{\!\{f \in \texttt{C}_f\}\!\} \;\wedge\; \neg\mathcal{C}\{\!\{h \in \texttt{C}_h\}\!\}$$
  This query removes the axioms associated with the *definition* of f, leaving only its *contract* available. This makes the proof of h's contract insensitive to changes in f's implementation. Via a similar reasoning as before, such an invocation is sound as well.

5. Induction

An important practical extension is the ability to prove contracts about recursive functions using induction. For instance, we might want to prove that `length` satisfies CF $\to$ CF.

```
length []     = Z
length (x:xs) = S (length xs)
```

In the second case we need to show that the result of `length xs` is crash-free but we do not have this information so the proof gets stuck, often resulting in the FOL-solver looping.

A naive approach would be to perform induction over the list argument of `length` – however in Haskell datatypes may be lazy infinite streams and ordinary induction is not necessarily a valid proof principle. Fortunately, we can still appeal to *fixpoint induction*. The fixpoint induction sheme that we use for `length` above would be to *assume* that the contract holds for the occurence some function `length_rec` inside the body of its definition, and then try to prove it for the function:

```
length []     = Z
length (x:xs) = S (length_rec xs)
```

Formally, our induction scheme is:

Definition 5.1 (Induction sheme)**.** *To prove that* $[\![g]\!] \in [\![\texttt{C}]\!]$ *for a function* $g\ \overline{a}\ \overline{x{:}\tau} = e[g]$ *(meaning* e *contains some occurrences of* g*), we perform the following steps:*

- *Generate function symbols* g°, $g^\bullet$
- *Generate the theory formula*
  $$\varphi = \mathcal{T} \wedge \mathcal{P}\{\!\{P \cup g^\bullet\ \overline{a}\ \overline{x{:}\tau} = e[g^\circ]\}\!\}$$
- *Prove that the query* $\varphi \wedge \mathcal{C}\{\!\{g^\circ \in \texttt{C}\}\!\} \wedge \neg\mathcal{C}\{\!\{g^\bullet \in \texttt{C}\}\!\}$ *is unsatisfiable.*

Why is this approach sound? The crucial step here is the fact that contracts are admissible predicates.

Theorem 5.1 (Contract admissibility)**.** *If $d_i \in [\![\mathtt{C}]\!]$ for all elements of a chain $d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ then the limit of the chain $\sqcup d_i \in [\![\mathtt{C}]\!]$. Moreover, $\bot \in [\![\mathtt{C}]\!]$.*

Proof. By induction on the contract `C`; for the `CF` case we get the result from Lemma 4.2. For the predicate case we get the result from the fact that the denotations of programs are continuous in D_∞. The arrow case follows by induction. □

We can then prove the soundness of our induction scheme.

Theorem 5.2. *The induction scheme in Definition 5.1 is correct.*

Proof. We need to show that: $[\![P]\!]^\infty(g) \in [\![\mathtt{C}]\!]$ and hence, by admissibility it is enough to find a chain whose limit is $[\![P]\!]^\infty(g)$ and such that every element is in $[\![\mathtt{C}]\!]$. Let us consider the chain $[\![P]\!]^k(g)$ so that $[\![P]\!]^0(g) = \bot$ and $[\![P]\!]^{k+1}(g) = [\![P]\!]_{([\![P]\!]^k)}(g)$ whose limit is $[\![P]\!]^\infty(g)$. We know that $\bot \in [\![\mathtt{C}]\!]$ so, by using contract admissiblity, all we need to show is that if $[\![P]\!]^k(g) \in [\![\mathtt{C}]\!]$ then $[\![P]\!]^{k+1}(g) \in [\![\mathtt{C}]\!]$.

To show this, we can assume a model where the denotational interpretation $\mathcal{I}$ has been extended so that $\mathcal{I}(g^\circ) = [\![P]\!]^k(g)$ and $\mathcal{I}(g^\bullet) = [\![P]\!]^{k+1}(g)$. By proving that the formula

$$\varphi \wedge \mathcal{C}\{\!\{g^\circ \in \mathtt{C}\}\!\} \wedge \neg\mathcal{C}\{\!\{g^\bullet \in \mathtt{C}\}\!\}$$

is unsatisfiable, since $\langle D_\infty, \mathcal{I}\rangle \models \varphi$ and $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{g^\circ \in \mathtt{C}\}\!\}$, we learn that $\langle D_\infty, \mathcal{I}\rangle \models \mathcal{C}\{\!\{g^\bullet \in \mathtt{C}\}\!\}$, and hence $[\![P]\!]^{k+1}(g) \in [\![\mathtt{C}]\!]$. □

Note that contract admissibility is *absolutely essential* for the soundness of our induction scheme, and is not a property that holds of every predicate on denotations. For example, consider the following Haskell definition:

```
ones = 1 : ones
f (S x) = 1 : f x
f Z     = [0]
```

Let us try to check if the $\forall x.\mathtt{f}(x) \neq \mathtt{ones}$ is true in the denotational model, using fixpoint induction. The case for $\bot$ holds, and so does the case for the `Z` constructor. For the `S` x case, we can assume that $\mathtt{f}(x) \neq \mathtt{ones}$ and we can easily prove that this implies that $\mathtt{f}(\mathtt{S}\ x) \neq \mathtt{ones}$. Nevertheless, the property is *not true* – just pick a counterexample $[\![\mathtt{s}]\!]$ where `s = S s`. What happened here is that the property is denotationally true of all the elements of the following chain

$$\bot \sqsubseteq \mathsf{S}(\bot) \sqsubseteq \mathsf{S}(\mathsf{S}(\bot)) \sqsubseteq \ldots$$

but is false in the limit of this chain. In other words $\neq$ is not admissible and our induction scheme is plain nonsense for non-admissible predicates.

Finally, we have observed that for many practical cases, a straightforward generalization of our lemma above for mutually recursive definitions is required. Indeed, our tool performs mutual fixpoint induction when a recursive group of functions is encountered. We leave it as future work to develop more advanced techniques such as strengthening of induction hypotheses or identifying more sophisticated induction schemes.

6. Implementation and practical experience

Our prototype contract checker is called **Halo**. It uses GHC to parse, typecheck, and desugar a Haskell program, translates it into first order logic (exactly as in Section 3), and invokes a FOL theorem prover (Equinox, Z3, Vampire, etc) on the FOL formula. The desugared Haskell program is expressed in GHC's intermediate language called Core (Sulzmann et al. 2007), an explicitly-typed variant of System F. It is straightforward to translate Core into our language λ_{HALO}.

6.1 Expressing contracts in Haskell

How does the user express contracts? We write contracts in Haskell itself, using higher-order abstract syntax and a GADT, in a manner reminiscent of the work on *typed contracts* for functional programming (Hinze et al. 2006):

```
data Contract t where
  (:->) :: Contract a
        -> (a -> Contract b)
        -> Contract (a -> b)
  Pred  :: (a -> Bool) -> Contract a
  CF    :: Contract a
  (:&:) :: Contract a -> Contract a -> Contract a
```

A value of type `Contract t` is a a contract for a function of type `t`. The connectives are `:->` for dependent contract function space, `CF` for crash-freedom, `Pred` for predication, and `:&:` for conjunction. One advantage of writing contracts as Haskell terms is that we can use Haskell itself to build new contract combinators. For example, a useful derived connective is non-dependent function space:

```
(-->) :: Contract a -> Contract b -> Contract (a -> b)
c1 --> c2 = c1 :-> (\_ -> c2)
```

A contract is always associated with a function, so we pair the two in a `Statement`:

```
  data Statement where
      (:::) :: a -> Contract a -> Statement
```

In our previous mathematical notation we might write the following contract for `head`:

$$\mathtt{head} \in \mathtt{CF}\ \&\ \{\mathtt{xs} \mid \mathtt{not\ (null\ xs)}\} \to \mathtt{CF}$$

Here is how we express the contract as a Haskell definition:

```
c_head :: Statement
c_head = head ::: CF :&: Pred (not . null) --> CF
```

If we put this definition in a file `Head.hs`, together with the supporting definitions of `head`, `not`, and `null`, then we can run `halo Head.hs`. The `halo` program translates the contract and the supporting function definitions into FOL, generates a TPTP file, and invokes a theorem prover. And indeed `c_head` is verified by all theorem provers we tried.

For recursive functions `halo` uses fixpoint induction, as described in Section 5.

6.2 Practical considerations

To make the theorem prover work as fast as possible we trim the theories to include only what is needed to prove a property. Unnecessary function pointers, data types and definitions for the current goal are not generated.

When proving a series of contracts, it is natural to do so in dependency order. For example:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

reverse_cf :: Statement
reverse_cf = reverse ::: CF --> CF
```

To prove this contract we must first prove that `(++)` $\in$ CF $\rightarrow$ CF $\rightarrow$ CF; then we can prove `reverse`'s contract assuming the one for `(++)`. At the moment, `halo` asks the programmer to specify which auxiliary contracts are useful, via a second constructor in the `Statement` type:

```
reverse_cf = reverse ::: CF --> CF
                       `Using` append_cf
```

6.3 Dependent contracts

`halo` can prove dependent contracts. For example:

$$\texttt{filter} \in (\texttt{p} : \texttt{CF} \rightarrow \texttt{CF}) \rightarrow \texttt{CF} \rightarrow \texttt{CF} \;\&\; \{\texttt{ys} \mid \texttt{all p ys}\}$$

This contract says that under suitable assumptions of crash-freedom, the result of `filter` is both crash-free and satisfies `all p`. Here `all` is a standard Haskell function, and `p` is the functional argument itself.

In our source-file syntax we use `(:->)` to bind `p`.

```
filter_all :: Statement
filter_all =
  filter ::: (CF --> CF) :-> \p ->
               CF --> (CF :&: Pred (all p))
```

The contract looks slightly confusing since it uses two "arrows", one from `:->`, and one from the `->` in the lambda. This contract is proved by applying fixed point induction.

6.4 Higher order functions

Our tool also deals with (very) higher order functions. Consider this function `withMany`, taken from the library `Foreign.Util.Marshal`:

```
withMany :: (a -> (b -> res) -> res)
         -> [a] -> ([b] -> res) -> res
withMany _       []     f = f []
withMany withFoo (x:xs) f = withFoo x (\x' ->
      withMany withFoo xs (\xs' -> f (x':xs')))
```

For `withMany`, our tool proves

$$\begin{aligned}\texttt{withMany} \in \;& (\texttt{CF} \rightarrow (\texttt{CF} \rightarrow \texttt{CF}) \rightarrow \texttt{CF}) \rightarrow \\ & (\texttt{CF} \rightarrow (\texttt{CF} \rightarrow \texttt{CF}) \rightarrow \texttt{CF})\end{aligned}$$

6.5 Experimental Results

We have run `halo` on a collection of mostly-small tests, some of which can be viewed in Figure 7. Our full testsuite and tables can be downloaded from `https://github.com/danr/contracts/blob/master/tests/BigTestResults.md`. The test cases include:

- Crash-freedom of standard functions (`(++)`, `foldr1`, `iterate`, `concatMap`).
- Crash-freedom of functions with more complex recursive patterns (Ackermann's function, functions with accumulators).
- Partial functions given appropriate preconditions (`foldr1`, `head`, `fromJust`).
- The `risers` example from Catch (Mitchell and Runciman 2008).

Description	equinox	Z3	vampire	E
`ack` CF	-	0.04	0.03	-
`all` CF	-	0.00	3.36	0.04
`(++)` CF	-	0.03	3.30	0.38
`concatMap` CF	-	0.03	6.60	-
`length` CF	0.87	0.00	0.80	0.01
`(+)` CF	44.33	0.00	3.32	0.10
`(*)` CF	6.44	0.03	3.36	-
`factorial` CF	6.69	0.02	4.18	31.04
`exp` CF	-	0.03	3.36	-
`(*)` accum CF	-	0.03	3.32	-
`exp` accum CF	-	0.04	4.20	0.12
`factorial` accum CF	-	0.03	3.32	-
`reverse` CF	13.40	0.03	28.77	-
`(++)/any` morphism	-	0.03	-	-
`filter` satisfies `all`	-	0.03	-	-
`iterate` CF	5.54	0.00	0.00	0.00
`repeat` CF	0.06	0.00	0.00	0.01
`foldr1`	-	0.01	1.04	24.78
`head`	18.62	0.00	0.00	0.01
`fromJust`	0.05	0.00	0.00	0.00
`risersBy`	-	-	1.53	-
`shrink`	-	0.04	-	-
`withMany` CF	-	0.00	-	-

Figure 7: Theorem prover running time in seconds on some of the problems in the test suite on contracts that hold.

- Some non-trivial post-conditions, such as the example above with `filter` and `all`, and also `any p xs || any p ys` = `any p (xs ++ ys)`.

We tried four theorem provers, Equinox, Z3, Vampire and E, and gave them 60 seconds for each problem. For our problems, Z3 seems to be the most successful theorem prover.

7. Discussion

Contracts that do not hold In practice, programmers will often propose contracts that do not hold. For example, consider the following definitions:

```
length []     = Z
length (x:xs) = S (length xs)

isZero Z = True
isZero _ = False
```

Suppose that we would like to check the (false) contract:

$$\texttt{length} \in \texttt{CF} \rightarrow \{x \mid \texttt{isZero}\ x\}$$

A satisfiability-based checker will simply diverge trying to construct a counter model for the negation of the above query; we have confirmed that this is indeed the behaviour of several tools (Z3, Equinox, Eprover). Why? When a counter-model exists, it will include tables for the function symbols in the formula. Recall that functions in FOL are total over the domain of the terms in the model. This means that function tables may be *infinite* if the terms in the model are infinite. Several (very useful!) axioms such as the discrimination axioms AXDISJC may in fact force the models to be infinite.

In our example, the table for `length` is indeed infinite since `[]` is always disjoint from `Cons x xs` for any `x` and `xs`. Even if there is a finitely-representable infinite model, the theorem prover may search forever in the "wrong corner" of the model for a counterexample.

From a practical point of view this is unfortunate; it is not acceptable for the checker to loop when the programmer writes an erroneous contract. Tantalisingly, there exists a very simple counterexample, e.g. `[Z]`, and that single small example is all the programmer needs to see the falsity of the contract.

Addressing this problem is a challenging (but essential) direction for future work, and we are currently working on a modification of our theory that admits the denotational model, but also permits *finite models* generated from counterexample traces. If the theory can guarantee the existence of a finite model in case of a counterexample, a finite model checker such as Paradox (Claessen and Sörensson 2003) will be able find it.

A tighter correspondence to operational semantics? Earlier work gave a declarative specfication of contracts using *operational semantics* (Xu et al. 2009). In this paper we have instead used a *denotational semantics* for contracts (Figure 6). It is natural to ask whether or not the two semantics are identical.

From computational adequacy, Theorem 4.1 we can easily state the following theorem:

Corollary 7.1. *Assume that* e *and* $\mathtt{C}$ *contain no term variables and assume that* $\mathcal{C}\{\!\{e \in \{x \mid e_p\}\}\!\} = \varphi$. *It is the case that* $\langle D_\infty, \mathcal{I}\rangle \models \varphi$ *if and only iff either* $P \not\vdash e \Downarrow$ *or* $P \not\vdash e_p[e/x] \Downarrow$ *or* $P \vdash e_p[e/x] \Downarrow True$.

Hence, the operational and denotational semantics of *predicate contracts* coincide. However, the correspondence is not precise for *dependent function contracts*. Recall the operational definition of contract satisfaction for a function contract:

$e \in (x{:}\mathtt{C}_1) \to \mathtt{C}_2$ iff
for all e' such that $(e' \in \mathtt{C}_1)$ it is $e\ e' \in \mathtt{C}_2[e'/x]$

The denotational specification (Figure 6) says that for all denotations d' such that $d' \in [\![\mathtt{C}_1]\!]$, it is the case that $\mathsf{app}([\![e]\!], d') \in [\![\mathtt{C}_2]\!]_{x \mapsto d'}$.

Alas there are *more* denotations than images of terms in D_∞, and that breaks the correspondence. Consider the program:

```
loop = loop

f :: (Bool -> Bool -> Bool) -> Bool
f h = if (h True True) && (not (h False False))
      then if (h True loop) && (h loop True)
           then BAD else True
      else True
```

Also consider now this candidate contract for f:

$$\mathtt{f} \in \mathtt{CF} \to (\mathtt{CF} \to \mathtt{CF} \to \mathtt{CF}) \to \mathtt{CF}$$

Under the *operational* definition of contract satisfaction, `f` indeed satisfies the contract. To reach `BAD` we have to pass both conditionals. The first ensures that `h` evaluates at least one of its arguments, while the second will diverge if either argument is evaluated. Hence `BAD` cannot be reached, and the contract is satisfied.

However, *denotationally* it is possible to have the classic parallel-or function, *por*, defined as follows:

$$\begin{array}{lcl} por\ \bot\ \bot & = & \bot \\ por\ \bot\ \mathsf{True} & = & \mathsf{True} \\ por\ \mathsf{True}\ \bot & = & \mathsf{True} \\ por\ \mathsf{False}\ \mathsf{False} & = & \mathsf{False} \end{array}$$

We have to define *por* in the language of denotational semantics, because we cannot write it in Haskell — that is the point! For convenience, though, we use pattern matching notation instead of our language of domain theory combinators. The rest of the equations (for `BAD` arguments) are induced by monotonicity and we may pick whatever boolean value we like when both arguments are `BAD`.

Now, this is denotationally a $\mathtt{CF} \to \mathtt{CF} \to \mathtt{CF}$ function, but it will pass *both* conditionals, yielding `BAD`. Hence $app(\mathtt{f}, por) = \mathsf{Bad}$, and `f`'s contract does not hold. So we have a concrete case where an expression may satisfy its contract operationally but not denotationally, because of the usual loss of full abstraction: there are more tests than programs in the denotational world. Due to contra-variance we expect that the other inclusion will fail too.

This is not a serious problem in practice. After all the two definitions mostly coincide, and they precisely coincide in the base case. At the end of the day, we are interested in whether $\mathtt{main} \in \mathtt{CF}$, and we have proven that if is crash-free denotationally, it is definitely crash-free in any operationally-reasonable term.

Finally, is it possible to define an operational model for our FOL theory that interpreted equality as contextual equivalence? Probably this could be made to work, although we believe that the formal clutter from syntactic manipulation of terms could be worse than the current denotational approach.

Polymorphic crash-freedom Observe that our axiomatisation of crash-freedom in Figure 4 includes only axioms for data constructors. In fact, our denotational interpretation F^{∞}_{cf} allows more axioms, such as:

$$\forall xy \,.\, \mathsf{cf}(x) \wedge \mathsf{cf}(y) \Rightarrow \mathsf{cf}(app(x, y))$$

This axiom is useful if we wish to give directly a `CF` contract to a value of arrow type. For instance, instead of specifying that `map` satisfies the contract $(\mathtt{CF} \to \mathtt{CF}) \to \mathtt{CF} \to \mathtt{CF}$ one may want to say that it satisfies the contract $\mathtt{CF} \to \mathtt{CF} \to \mathtt{CF}$. With the latter contract we need the previous axiom to be able to apply the function argument of `map` to a crash-free value and get a crash-free result.

In some situations, the following axiom might be beneficial as well:

$$(\forall \overline{x} \,.\, \mathsf{cf}(f(\overline{x}))) \Rightarrow \mathsf{cf}(f_{ptr})$$

If the result of applying a function to any possible argument is crash-free then so is the function pointer. This allows us to go in the inverse direction as before, and pass a function pointer to a function that expects a `CF` argument. However notice that this last axiom introduces a quantified assumption, which might lead to significant efficiency problem.

Ideally we would like to say that $[\![\mathtt{CF}]\!] = [\![\mathtt{CF} \to \mathtt{CF}]\!]$, but that is not quite true. In particular,

$$(\forall x \,.\, \mathsf{cf}(app(y, x))) \Rightarrow \mathsf{cf}(y)$$

is *not* valid in the denotational model. For instance consider the value $\mathsf{K}(\mathsf{Bad})$ for y. The left-hand side is going to always be true, because the application is ill-typed and will yield $\bot$, but y is not itself crash-free.

8. Related work

There are very few practical tools for the automatic verification of *lazy and higher-order* functional programs. Furthermore, our approach of directly translating the denotational semantics of programs does not appear to be well-explored in the literature.

Catch (Mitchell and Runciman 2008) is one of the very few tools that address the verification of lazy Haskell, and have been evaluated on real programs. Using static analysis, Catch can detect pattern match failures, and hence prove that a program cannot crash.

Some annotations that describes the set of constructors that are expected as arguments to each function may be necessary for the analysis to succeed. Our aim in this paper is to achieve similar goals, but moreover to be in a position to assert functional correctness.

Liquid Types (Rondon et al. 2008) is an influential approach to call-by-value functional program verification. Contracts are written as refinements in a fixed language of predicates (which may include recursive predicates) and the extracted conditions are discharged using an SMT-solver. Because the language of predicates is fixed, predicate abstraction can very effectively *infer* precise refinements, even for recursive functions, and hence the annotation burden is very low. In our case, since the language of predicates is, *by design*, the very same programming language with the same semantics, inference of function specifications is harder. The other important difference is that liquid types requires all *uses* of a function to satisfy its precondition, whereas in the semantics that we have chosen, bad uses are allowed but the programmer gets no guarantees back.

Rather different to Liquid Types, Dminor (Bierman et al. 2010) allows refinements to be written in the very same programming language that programs are written. Contrary to our case however, in Dminor the expressions that refine types must be pure — that is, terminating — and have a unique denotation (e.g. not depending on the store). Driven from a typing relation that includes logic entailment judgements, verification conditions are extracted and discharged automatically using Z3. Similar in spirit, other dependent type systems such as Fstar (Swamy et al. 2011) also extract verification conditions that are discharged using automated tools or interactive theorem provers. Hybrid type systems such as Sage (Knowles and Flanagan 2010) attempt to prove as many of the goals statically, and defer the rest as runtime goals.

Boogie (Barnett et al. 2005) is a verification back end that supports procedures as well as pure functions. By using Z3, Boogie verifies programs written in the BoogiePL intermediate language, which could potentially be used as the back end of our translation as well. Recent work on performing induction on top of an induction-free SMT solver proposes a "tactic" for encoding induction schemes as first-order queries, which is reminiscent of the way that we perform induction (Leino 2012).

The recent work on the Leon system (Suter et al. 2011) presents an approach to the verification of *first-order* and *call-by-value* recursive functional programs, which appears to be very efficient in practice. It works by extending SMT with recursive programs and "control literals" that guide the pattern matching search for a counter-model, and is guaranteed to find a model if one exists (whereas that is not yet the case in our system, as we discussed earlier). It treats does not include a `CF`-analogous predicate, and no special treatment of the $\perp$ value or pattern match failures seems to be in the scope of that project. However, it gives a very fast verification framework for partial functional correctness.

The tool Zeno (Sonnex et al. 2011) verifies equational properties of functional programs using Haskell as a front end. Its proof search is based on induction, equality reasoning and operational semantics. While guaranteeing termination, it can also start new induction proofs driven by syntactic heuristics. However, it only considers the finite and total subset of values, and we want to reason about Haskell programs as they appear in the wild: possibly non-terminating, with lazy infinite values, and run time crashes.

First-order logic has been used as a target for higher-order languages in other verification contexts as well. Users of the interactive theorem prover Isabelle have for many years had the opportunity to use automated first-order provers to discharge proof obligations. This work has recently culminated in the tool Sledgehammer (Blanchette et al. 2011), which not only uses first-order provers, but also SMT solvers as back ends. There has also been a version of the dependently typed programming language Agda in which proof obligations could be sent to an automatic first-order prover (Abel et al. 2005). Both of these use a translation from a typed higher-order language of well-founded definitions to first-order logic. The work in this area that perhaps comes closest to ours, in that they deal with a lazy, general recursive language with partial functions, is by Bove et al. (2012), who use Agda as a logical framework to reason about general recursive functional programs, and combine interaction in Agda with automated proofs in first-order logic.

The previous work on static contract checking for Haskell (Xu et al. 2009) was based on *wrapping*. A term was effectively wrapped with an appropriately nested contract test, and symbolic execution or aggressive inlining was used to show that `BAD` values could never be reached in this wrapped term. In follow-up work, Xu (Xu 2012) proposes a variation, this time for a *call-by-value* language, which performs symbolic execution along with a "logicization" of the program that can be used (via a theorem prover) to eliminate paths that can provably not generate `BAD` value,. The "logicization" of a program has a similar spirit to our translation to logic,a but it is not clear which model is intended to prove the soundness of this translation and justify its axiomatisation. Furthermore, the logicization of programs is dependent on whether the resulting formula is going to be used as a goal or assumption in a proof. We believe that the direct approach proposed in this paper, which is to directly encode the semantics of programs and contracts, might be simpler. That said, symbolic execution as proposed in (Xu 2012) has the significant advantage of querying a theorem prover on many small goals as symbolic execution proceeds, instead of a single verification goal in the end. We have some ideas about how to break large contract negation queries to smaller ones, guided by the symbolic evaluation of a function, and we plan to integrate this methodology in our tool.

9. Conclusions and future work

Static verification for functional programming languages seems an under-studied (compared to the imperative world) and very promising area of research. In practical terms, our most immediate goal is to ensure that we can find finite counter-examples quickly, and present them comprehensibly to the user, rather allowing the theorem prover to diverge. As mentioned in Section 7 we have well-developed ideas for how to do this. It would also be interesting to see if *triggers* in SMT 2.0 could also be used to support that goal.

We would like to add support for primitive data types, such as `Integer`, using theorem provers such as `T-SPASS` to deal with the `tff` (typed first-order arithmetic) part of TPTP. Another approach might be to generate theories in the SMT 2.0 format, understood by Z3, which has support for integer arithmetic and more. Another important direction is finding ways to split our big verification goals into smaller ones that can be proven significantly faster. Finally, we would like to investigate whether we can automatically strengthen contracts to be used as induction hypotheses in inductive proofs, deriving information from failed attempts.

Acknowledgements Thanks to Richard Eisenberg for helpful feedback and Nathan Collins for early work on a prototype of our translation.

References

Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. In *5th International Workshop on Frontiers of Combining Systems (FroCoS)*, LNCS. Springer Verlag, 2005.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387, 2005.

Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 115–130, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_10. URL `http://dx.doi.org/10.1007/978-3-642-03359-9_10`.

Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional programming*, ICFP '10, pages 105–116, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863560. URL `http://doi.acm.org/10.1145/1863543.1863560`.

Jasmin Blanchette, Sascha Böhme, and Lawrence Paulson. Extending Sledgehammer with SMT solvers. In *Conference on Automated Deduction (CADE)*, LNCS. Springer Verlag, 2011.

Matthias Blume and David McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, July 2006. ISSN 0956-7968. doi: 10.1017/S0956796806005971. URL `http://dx.doi.org/10.1017/S0956796806005971`.

Ana Bove, Peter Dybjer, and Andrés Sicard-Ramírez. Combining interactive and automatic reasoning in first order theories of functional programs. In Lars Birkedal, editor, *15th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS 2012*, volume 7213 of *LNCS*, pages 104–118, March 2012.

Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581484. URL `http://doi.acm.org/10.1145/581478.581484`.

Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, pages 208–225, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33438-6, 978-3-540-33438-5. doi: 10.1007/11737414_15. URL `http://dx.doi.org/10.1007/11737414_15`.

Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Interpolation and symbol elimination in Vampire. In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, pages 188–195, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14202-8, 978-3-642-14202-4. doi: 10.1007/978-3-642-14203-1_16. URL `http://dx.doi.org/10.1007/978-3-642-14203-1_16`.

Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010. ISSN 0164-0925. doi: 10.1145/1667048.1667051. URL `http://doi.acm.org/10.1145/1667048.1667051`.

K. Rustan M. Leino. Automating induction with an SMT solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 315–331, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27939-3. doi: 10.1007/978-3-642-27940-9_21. URL `http://dx.doi.org/10.1007/978-3-642-27940-9_21`.

Neil Mitchell and Colin Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: 10.1145/1411286.1411293. URL `http://doi.acm.org/10.1145/1411286.1411293`.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Andrew M. Pitts. Relational properties of domains. *Inf. Comput.*, 127(2): 66–90, 1996.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375602. URL `http://doi.acm.org/10.1145/1375581.1375602`.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

Willam Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, February 2011. URL `http://pubs.doc.ic.ac.uk/zeno/`.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190324. URL `http://doi.acm.org/10.1145/1190315.1190324`.

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Proceedings of the 18th International Conference on Static analysis*, SAS'11, pages 298–315, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23701-0. URL `http://dl.acm.org/citation.cfm?id=2041552.2041575`.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming*, pages 266–278, 2011.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9_1. URL `http://dx.doi.org/10.1007/978-3-642-00590-9_1`.

Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993. ISBN 978-0-262-23169-5.

Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, March 2007. ISSN 0956-7968. doi: 10.1017/S0956796806006216. URL `http://dx.doi.org/10.1017/S0956796806006216`.

Dana N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 107–116, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2. doi: 10.1145/2103746.2103767. URL `http://doi.acm.org/10.1145/2103746.2103767`.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 41–52, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480889. URL `http://doi.acm.org/10.1145/1480881.1480889`.