

## 1. Introduction

Our approach to static contract-checking is to translate source code to a first-order logic theory and then use an automated theorem prover to check the consistency of the theory.

Consider:

```
data List a = Nil | Cons a (List a)
```

```
notnull x = case x of
  Nil -> False
  Cons(x,y) -> True
```

```
head :: (CF && {x | notnull x }) -> CF
head xs = case xs of
  Nil -> BAD
  Cons(x,y) -> x
```

First, we need to encode the List structure.

We start by stating that *Nil* and *Cons* can never be equal:

$$\forall a, b. \text{Cons}(a, b) \neq \text{Nil}$$

Then, we must state that *Nil* never crashes (ie cannot be evaluated to an exception) and that *Cons*(*x*, *y*) crashes iff either *x* or *y* crashes. The statement *x* crashes is encoded by the term *CF*(*x*).

$$\text{CF}(\text{Nil})$$

$$\forall a, b. \text{CF}(a) \wedge \text{CF}(b) \iff \text{CF}(\text{Cons}(a, b)) \quad (1)$$

We also say some stuff about unreachability but I can't think of a good way to explain it right now.

$$\forall y, ys. \text{Cons}(y, ys) \neq \text{UNR}$$

$$\text{Nil} \neq \text{UNR}$$

Finally, we define projections for *Cons*. It is not strictly necessary, but it will be handy:

$$\forall xs, y, ys. \text{sel}_{1, \text{Cons}}(\text{Cons}(y, ys)) = xs \implies xs = y$$

$$\forall xs, y, ys. \text{sel}_{2, \text{Cons}}(\text{Cons}(y, ys)) = xs \implies xs = ys$$

Now we translate the *null* function. Note that the symbols *true* and *false* are the representation of the data constructors *True* and *False* in Haskell, not the boolean values  $\top$  and  $\perp$  in our logic.

$$\forall xs. xs = \text{Nil} \implies \text{notnull}(xs) = \text{false}$$

$$\forall xs, y, ys. x = \text{Cons}(y, ys) \implies \text{notnull}(xs) = \text{true} \quad (2)$$

We also need to specify the translation of calls to *notnull* with BAD values (to encode the fact that *notnull*(BAD) = BAD).

$$\forall xs. xs = \text{BAD} \rightarrow \text{notnull}(xs) = \text{BAD}$$

Finally, we say that one call *notnull* with a an argument which is not *Nil* or *Cons* or BAD then the result is UNR.

$$\begin{aligned} &\forall xs. xs \neq \text{BAD} \wedge xs \neq \text{Nil} \\ &\wedge xs \neq \text{Cons}(\text{sel}_{1, \text{Cons}}(xs), \text{sel}_{2, \text{Cons}}(xs)) \rightarrow \\ &\quad \text{notnull}(X) = \text{UNR} \end{aligned} \quad (3)$$

The translation of *head* follows the same pattern:

$$\begin{aligned} &\forall xs. x = \text{Nil} \implies \text{head}(xs) = \text{BAD} \\ &\forall xs, y, ys. x = \text{Cons}(y, ys) \implies \text{head}(xs) = y \\ &\forall xs. xs = \text{BAD} \rightarrow \text{head}(xs) = \text{BAD} \end{aligned} \quad (4)$$

$$\begin{aligned} &\forall xs. xs \neq \text{BAD} \wedge xs \neq \text{Nil} \\ &\wedge xs \neq \text{Cons}(\text{sel}_{1, \text{Cons}}(xs), \text{sel}_{2, \text{Cons}}(xs)) \rightarrow \\ &\quad \text{head}(X) = \text{UNR} \end{aligned}$$

We now have translated the source code. Let us call all those formulae the theory *T*. We translate separately the contract:

$$\phi := \forall xs. \text{CF}(xs) \wedge \text{notnull}(xs) = \text{true} \rightarrow \text{CF}(\text{head}(xs))$$

Now that we translated everything to first-order logic, we can ask the theorem prover if the theory formed by those formulae is consistent, ie if  $T \vdash \phi$ .

Intuitively, *T* is consistent (ie  $T \not\vdash \perp$ ), because each formula serves a specific purpose. Now, assume that *xs* satisfies *CF*(*xs*) and *notnull*(*xs*) = *true*. We can derive that *xs*  $\neq$  BAD because we have *CF*(*xs*) and  $\neg \text{CF}(\text{BAD})$ . The constraint *notnull*(*xs*) = *true* doesn't directly imply that *xs* = *Cons*(*y*, *ys*) for some *y* and *ys*. But *notnull* is totally defined, because of (3). This implies (by (2)) that there exist *y* and *ys* such that *xs* = *Cons*(*y*, *ys*). Recalling *CF*(*xs*), we can now derive *CF*(*y*) and *CF*(*ys*) (by (1)). But *head*(*xs*) = *y* because of (4), and *y* is crash-free, so we can finally derive *CF*(*head*(*xs*)). QED.

## 2. Languages

### 2.1 $\mathcal{H}'$ : $\lambda$ -calculus variant

The syntax of  $\mathcal{H}$  is defined in 1. A module is a list of toplevel definitions, claims that functions satisfy contracts and data definitions.

- There's no  $\lambda$ -abstraction, because we can always lift them to toplevel declaration.
- We do not allow nested case expressions, because once again, we can always lift them to the toplevel.
- Until 6 we will only consider full application of functions (*f*(*x*, *y*)), in order to remove clutter. Dealing with partial application is not hard but a bit cumbersome.

```
mod := def1, ..., defn
def  ∈ Definition
def  := data T = K1 | ... | Kn
      | f ∈ c
      | f  $\vec{x}$  = e
      | f  $\vec{x}$  = case e of
                  K1( $\vec{x}$ ) → e1 | ... | Kn( $\vec{x}$ ) → en
```

$x, y, f, g, a, b \in$  Variables  
 $T \in$  Type Constructors  
 $K \in$  Data Constructors

$e \in$  Expressions  
 $e ::= x$   
 $\quad | \text{BAD}$   
 $\quad | e \ e$   
 $\quad | f(e, \dots, e)$   
 $\quad | K(e, \dots, e)$

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

### 2.2 Contracts

Contract syntax is described in figure 2.2. The predicates we use in our contracts can be any boolean  $\mathcal{H}'$  expression. We only consider

pairs of contract for simplicity, although there is no issue with generalisation to arbitrary tuples.

$$c := \begin{array}{l} x : c \rightarrow c \\ (c, c) \\ c \wedge c \\ c \vee c \\ \{x \mid p\} \\ \text{CF} \end{array}$$

**Figure 2.** Contract syntax

We give the semantics of contract by defining “ $e$  satisfies  $t$ ”, written  $e \in t$  in figure 2.2. Note that this definition doesn’t yield any operative way to check that an expression actually meets the specification given by its contract.

$$\begin{array}{ll} e \in \{x \mid p\} & \iff e \text{ diverges or } p[e/x] \not\vdash^* \{\text{BAD}, \text{False}\} \\ e \in x : c_1 \rightarrow c_2 & \iff \forall e_1 \in c_1, (e \ e_1) \in c_2[e_1/x] \\ e \in (c_1, c_2) & \iff e \text{ diverges or} \\ & (e \rightarrow^* (e_1, e_2) \text{ and } e_1 \in c_1, e_2 \in c_2) \\ e \in c_1 \wedge c_2 & \iff e \in c_1 \text{ and } e \in c_2 \\ e \in c_1 \vee c_2 & \iff e \in c_1 \text{ or } e \in c_2 \\ e \in \text{CF} & \iff e \text{ is crash-free} \end{array}$$

**Figure 3.** Semantics of contract satisfaction

### 2.3 Crash-freeness

Note that CF represents two things: it can be a contract, as in  $f \in \text{CF}$  or a special formula in first-order logic  $\text{CF}(f)$ .

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

**Definition 1 (Crash).** A closed term  $e$  crashed iff  $e \rightarrow^* \text{BAD}$ .

**Definition 2 (Diverges).** A closed expression  $e$  diverges iff either  $e \rightarrow^* \text{UNR}$  or there is no value  $val$  such that  $e \rightarrow^* val$ .

**Definition 3 (Syntactic safety).** A (possibly open) expression  $e$  is syntactically safe iff  $\text{BAD} \not\in_s e$ . Similarly a context  $C$  is syntactically safe iff  $\text{BAD} \not\in_s C$ .

The notation  $\text{BAD} \not\in e$  means that BAD does not appear anywhere in  $e$ , similarly for  $\text{BAD} \not\in_s C$ . For example, *Just3* is syntactically safe whereas *JustBAD* is not.

**Definition 4 (Crash-free).** An expression  $e$  is said to be crash-free iff

$$\forall C. \text{BAD} \not\in_s C \text{ and } \vdash C[e] :: () \not\vdash^* \text{BAD}$$

The notation  $C[e] :: ()$  means that  $C[e]$  is closed and well-typed. Note that there are crash-free expression that are not syntactically safe, for example *fst* (1, BAD).

### 2.4 BAD and UNR

Consider the following piece of code:

**a** = 0 + True

**b** :: CF  
**b** = undefined

**c** = error "foo"

- $a$  is ill-typed
- $b$ ’s implementation is not correct wrt its contract

$$\begin{array}{lcl} v, w, s, t & := & x \mid K(t, \dots, t) \mid f(t, \dots, t) \mid \text{app}(t, t) \\ & & \mid \text{BAD} \mid \text{UNR} \\ \phi & := & \forall x. \phi \mid \neg \phi \mid \phi \vee \phi \mid \top \mid \perp \mid t = t \mid \text{CF}(t) \\ & & \mid \phi \wedge \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi \\ \Phi & := & \epsilon \mid \phi \mid \Phi \cup \Phi \end{array}$$

**Figure 4.** First-order logic syntax

$$\begin{array}{ll} \mathcal{E}[\![x]\!] & = x \\ \mathcal{E}[\![f(e_1, \dots, e_n)]\!] & = f(\mathcal{E}[\![e_1]\!], \dots, \mathcal{E}[\![e_n]\!]) \\ \mathcal{E}[\![K(e_1, \dots, e_n)]\!] & = K(\mathcal{E}[\![e_1]\!], \dots, \mathcal{E}[\![e_n]\!]) \\ \mathcal{E}[\![\text{BAD}]\!] & = \text{BAD} \end{array}$$

**Figure 5.**  $\mathcal{E}[\![\cdot]\!]$  – Expression translation

- $c$  goes through the whole toolchain (compiler, typechecker, contractchecker)

One thing to notice is that  $a$  and  $b$  are things that “sould not happen” but are caught statically whereas  $c$  should not happen but can only be dealt with dynamically.

We can now define two types of problematic expressions: those that cannot happen during a run of the program and those that can. Expressions of the first type are called unreachable (and equated to the special value UNR in our first-order theory), whilst expressions of the latter type are called bad (and equated to the special value BAD).

We said earlier that we only considered syntactically correct and well-typed programs as input. That implies that the “ $a$ ” case cannot happen. But given that our first-order logic is not typed, the theorem prover may decide to instantiate a variable with an ill-typed value! In order to prevent this, we will need to encode some basic type-checking mechanism directly in our first-order theory.

### 2.5 First-order logic with equality

We use first-order logic with equality, defined in figure 2.5.

## 3. Translations

### 3.1 $\mathcal{E}[\![\cdot]\!]$ – Expressions

Our most basic translation is from expressions in  $\mathcal{H}'$  to terms in first-order logic. Given this translation we will be able to translate definitions, data types and contracts to first-order formulae. It is described in 3.1.

### 3.2 $\mathcal{D}[\![\cdot]\!]$ – Definitions

We give in 3.2 and 3.2 the two translations of function definitions.

3.2 gives the translation of function not defined by pattern matching, which is really easy: we just have to state the equality between le left-hand side and the right-hand side.

Translating definitions that use pattern-matching is more challenging and is described in 3.2.

The first line says that when applied to an argument that matches a pattern of the case expression, we should equate the function call to the corresponding expression.

The second line states that if the pattern-matching failed or if we pattern-matched on BAD then the result should be UNR.

$$\mathcal{D}[f(x_1, \dots, x_n) = \text{case } e \text{ of } [K_1(\vec{x}_1) = e_1, \dots, K_n(\vec{x}_n) = e_n]]$$

$$\begin{array}{lcl} \forall \vec{a}, y, \vec{x}_{1 \leq i \leq n}. & \min(y) \wedge \mathcal{E}[e] = K_1(\vec{x}_1) \wedge f \vec{a} = y & \rightarrow y = \mathcal{E}[e_1] \\ & \dots & \\ & \min(y) \wedge \mathcal{E}[e] = K_n(\vec{x}_n) \wedge f \vec{a} = y & \rightarrow y = \mathcal{E}[e_n] \\ \min(y) \wedge \mathcal{E}[e] = \text{BAD} & \wedge f \vec{a} = y & \rightarrow y = \text{BAD} \\ \min(y) \wedge & f \vec{a} = y & \rightarrow \min(e) \wedge (e = \text{BAD} \vee \bigvee_i e = \text{\$weak}(K_i(\vec{x})) \vee y = \text{UNR}) \end{array}$$

**Figure 7.**  $\mathcal{D}[\![\ ]\!]$  – Case definitions

$$\begin{array}{l} \mathcal{D}[f(\vec{x}) = e] \\ \forall \vec{x}, y. \min(y) \wedge y = f(\vec{x}) \rightarrow y = \mathcal{E}[e] \end{array}$$

**Figure 6.**  $\mathcal{D}[\![\ ]\!]$  – Regular definitions

### 3.3 $\mathcal{T}[\![\ ]\!]$ – Datatypes

We break down the translation for datatypes in four parts, described in 3.3

$$\mathcal{T}[\![\text{data } T = K_1, \dots, K_n]\!] = S_1 \cup S_2 \cup S_3 \cup S_4$$

- ( $S_1$ ) For each  $K_i$  of arity  $a_i$  we introduce selectors  $\text{sel}_{k, K_i}$ , which are the projection of  $K_i(x_1, \dots, x_{n_i})$  on its  $k$ -th component.
- ( $S_2$ ) For each pair of constructors  $K_i, K_j$ , we state that they can never map to the same value.
- ( $S_3$ ) Then, we have to give crash-freeness conditions for each  $K_i$ : Notice that we have a equivalence.
  - $\leftarrow$ : if we pack crash-free values in a data constructor, the resulting value is crash-free.
  - $\rightarrow$ : a value  $t$  of type  $T$  is crash-free implies that every value packed in it is crash-free. Recall that one can define projection on any argument of a value of type  $t$ . So if the  $k$ -th argument of  $t$  is not crash-free, then the  $k$ -th projection is a crash-free context that throws an expression that is not crash-free.

Note that this is not true for functions: a function is not required to use all of its arguments. `fst` is crash-free if and only if the first argument of the pair is crash-free. The second argument being crash-free or not doesn't matter.
- ( $S_4$ ) None of the  $K_i$  is unreachable.
- One may want to also state that if  $\vec{x} \neq \text{BAD}$  then  $K_i(\vec{x}) \neq \text{BAD}$ . It is already implied by the fact that  $\text{CF}(\vec{x}) \rightarrow \text{CF}(K_i(\vec{x}))$ .

### 3.4 $\mathcal{S}[\![\ ]\!]$ – Contracts

We give in 9 the translation of contract satisfaction. *true* refers to the translation to a term of the data constructor `True` in  $\mathcal{H}'$ , not to the actual true value.

## 4. $\mathbb{M}$ – Checking a module

### 4.1 Prelude

There are some formulae that should always be included in our FO theory.

We need to state that `BAD` is not crash-free with the formula:  $\neg \text{CF}(\text{BAD})$ .

Plus we need to give formulae for the boolean datatype and for unreachability. Strictly speaking, we can omit them and just add the following lines to source files:

```
data UNR = UNR
data Bool = True | False
```

But given that those datatypes are used by our translation, we can just directly include their translation every time we translate a module.

### 4.2 Contract checking – Non-recursive case

Input: a module  $M$  that consists of a list of definitions, datatypes, contracts and a contract  $c$  for a non-recursive function  $f$  this is defined in  $M$ .

We say that the function implementation is correct wrt to its contract iff

$$\mathbb{M} \vdash \mathcal{S}[f \in c]$$

### 4.3 Contract checking – Recursive case

If the function  $f = e$  is recursive, then we ask the theorem prover the following:

$$\mathbb{M} - f, \mathcal{D}[f = e[f/f_p]], \mathcal{S}[f_p \in c] \vdash \mathcal{S}[f \in c]$$

Where  $M - f$  means the content of the module  $M$  without  $f$ 's definition and  $f$ 's contract. TODO Stress that it's not always enough and that we may have to unroll several times!

### 4.4 Module checking

A module is a collection of function definitions, data definitions and contracts. What we want to do is to check that functions satisfy their contract(s).

#### 4.4.1 Naive example

Here is a little example showing that we should be careful about which formulae should belong to a theory.

Assume that we have a module that contains two functions definition  $f$  and  $g$  and two contracts :  $f \in \text{CF}$  and  $g \in \text{CF}$ . We assume that those contracts do not hold, for example if  $f$  is *head* and  $g$  is *last*.

First, we want to check  $f$ 's contract. So we ask the theorem prover if

$$\mathcal{D}[f], \mathcal{D}[g], \mathcal{S}[g] \vdash \mathcal{S}[f]$$

But, given that  $g$ 's contract does not hold, we can derive  $\perp$  and then prove that  $f$ 's contract hold.

For the same reason, we can prove that  $g$  contract's holds, when in fact it doesn't.

Finally, the user thinks he's done, but in fact he has proven nothing.

#### 4.4.2 The proper way to check a module

Consider the following situation, where  $a$ 's definition relies on  $f$  and  $g$ .

$$\begin{aligned}
S_1 &:= \forall \vec{x}, a. (\min(a) \wedge K_i(\vec{x}) = a) \rightarrow \bigwedge_{1 \leq j \leq k} x_j = \$weak(sel_{j, K_i}(a)) \mid 1 \leq i \leq n \\
S_2 &:= \forall \vec{x}, \vec{y}, a. \neg(\min(a) \wedge K_i(\vec{x}) = a \wedge K_j(\vec{y}) = a) \mid 1 \leq i < j \leq n \\
S_3 &:= \forall \vec{x}, a. \min(a) \wedge a = K_i(\vec{x}) \rightarrow ((\bigwedge_{1 \leq j \leq k} \mathbf{CF}(x_j)) \leftrightarrow \mathbf{CF}(K_i(\vec{x}))) \mid 1 \leq i \leq n \\
S_4 &:= \forall \vec{x}, a. (\min(a) \wedge a = K_i(\vec{x})) \rightarrow a \neq \mathbf{UNR} \wedge a \neq \mathbf{BAD} \mid 1 \leq i \leq n
\end{aligned}$$

**Figure 8.**  $\mathcal{T}[\![\cdot]\!]$  – Data type translation

$$\begin{aligned}
\mathcal{S}[e \in \{x \mid b(x)\}] &= \min(\mathcal{E}[b(e)]) \wedge (\mathcal{E}[b(e)] = \mathbf{true} \vee \mathcal{E}[b(e)] = \mathbf{UNR}) \\
\mathcal{S}[e \in x : c_1 \rightarrow c_2(x)] &= \forall x. \min(\mathcal{E}[f(x)]) \rightarrow (\mathcal{S}[x \notin c_1] \vee \mathcal{S}[\mathcal{E}[e(x)] \in c_2(x)]) \\
\mathcal{S}[e \in \mathbf{CF}] &= \mathbf{CF}(e)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[e \notin \{x \mid b(x)\}] &= \min(\mathcal{E}[b(e)]) \wedge (\mathcal{E}[b(e)] = \mathbf{false} \vee \mathcal{E}[b(e)] = \mathbf{BAD}) \\
\mathcal{S}[e \notin x : c_1 \rightarrow c_2(x)] &= \exists x. \mathcal{S}[x \in c_1] \wedge \mathcal{S}[\mathcal{E}[e(x)] \notin c_2(x)] \\
\mathcal{S}[e \notin \mathbf{CF}] &= \neg \mathbf{CF}(e)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[e \in c_1 \&\&c_2] &= \mathcal{S}[e \in c_1] \wedge \mathcal{S}[e \in c_2] \\
\mathcal{S}[e \notin c_1 \&\&c_2] &= \mathcal{S}[e \notin c_1] \vee \mathcal{S}[e \notin c_2] \\
\mathcal{S}[e \in c_1 | c_2] &= \mathcal{S}[e \in c_1] \vee \mathcal{S}[e \in c_2] \\
\mathcal{S}[e \notin c_1 | c_2] &= \mathcal{S}[e \notin c_1] \wedge \mathcal{S}[e \notin c_2] \\
\mathcal{S}[(a, b) \in (c_1, c_2)] &= \mathcal{S}[a \in c_1] \wedge \mathcal{S}[b \in c_2] \\
\mathcal{S}[(a, b) \notin (c_1, c_2)] &= \mathcal{S}[a \notin c_1] \vee \mathcal{S}[b \notin c_2]
\end{aligned}$$

**Figure 9.**  $\mathcal{S}[\![\cdot]\!]$  – Contract translation

## 5. Correctness of the translation

For the translation to be useful,  $\llbracket M \rrbracket \vdash \mathcal{S}[f \in c]$  should imply that  $f \in c$ .

## 6. Higher-orderness

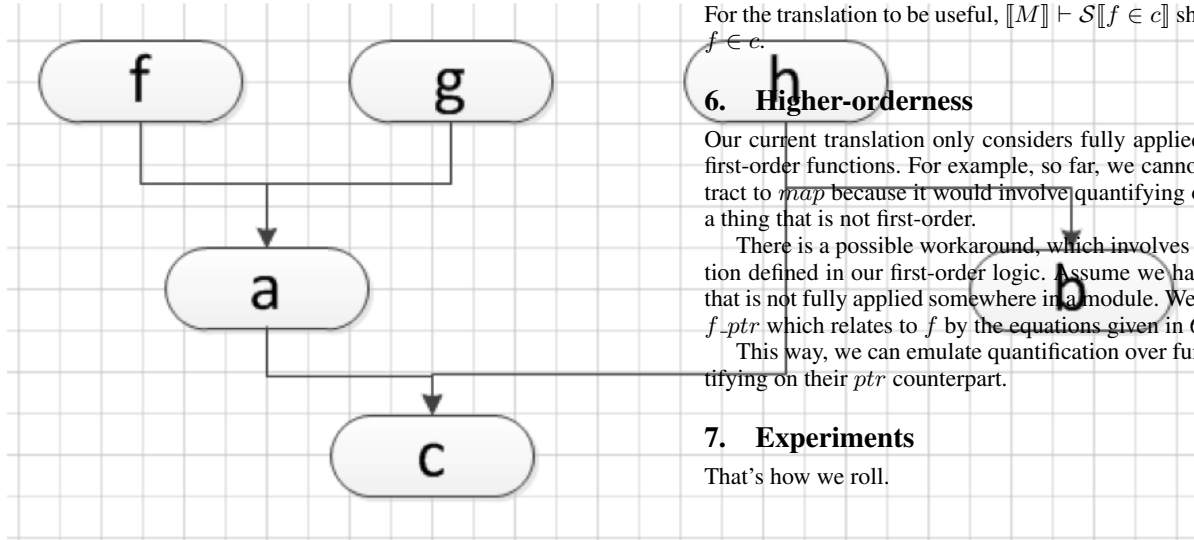
Our current translation only considers fully applied functions and first-order functions. For example, so far, we cannot give any contract to *map* because it would involve quantifying over a function, a thing that is not first-order.

There is a possible workaround, which involves the “app” function defined in our first-order logic. Assume we have a function *f* that is not fully applied somewhere in a module. We create the term *f\_ptr* which relates to *f* by the equations given in 6

This way, we can emulate quantification over function by quantifying on their *ptr* counterpart.

## 7. Experiments

That’s how we roll.



We should only include formulae that belongs to functions that are actually used. For example, to prove *a*’s contract, we should only include *f* and *g* translations, and we would ask equinox:

$$\mathcal{D}[f], \mathcal{D}[g], \mathcal{D}[a], \mathcal{S}[f], \mathcal{S}[g] \vdash \mathcal{S}[a]$$

$$\begin{aligned}
\mathcal{E}[\![e_1\ e_2]\!] &= app(e_1, e_2) \\
\forall x_1, \dots, x_n. f(x_1, \dots, x_n) &= app(app(\dots app(f\_ptr, x_1), x_2), \dots, x_n) \\
\mathbf{CF}(f\_ptr) &\leftrightarrow \forall x_1, \dots, x_n. \mathbf{CF}(x_1) \wedge \dots \wedge \mathbf{CF}(x_n) \rightarrow \mathbf{CF}(f(x_1, \dots, x_n)) \\
\forall f\_ptr, x. \mathbf{CF}(f\_ptr) \wedge \mathbf{CF}(x) &\rightarrow \mathbf{CF}(app(f\_ptr, x))
\end{aligned}$$

---

**Figure 10.** Encoding of higher-orderness

Problem	Equinox	Equinox (+ weak)	SPASS	Vampire	E
Add.hs	0.25	0.08	0.04	0.12	0.05
BinaryTree.hs	0.45	0.2	0.04	0.01	0.04
Branch.hs	0.27	0.40	0.04	0.01	0.03
Copy.hs	0.86	0.09	0.03	0.01	184.3
Head.hs	0.32	0.29	0.03	0.03	4.2
Implies.hs	3.24	0.32	0.06	0.02	0.11
Map.hs	2.47	0.14	0.92	1.02	>300
Mult.hs	>300	0.41	0.05	0.22	11.71
Multgt.hs	>300	1.24	0.62	1.31	>300
NatEq.hs	203.12	0.33	0.02	0.03	0.343
Odd.hs	0.42	1.17	0.06	0.03	>300
Reverse.hs	72.32	0.12	0.05	0.02	0.038
Simple.hs	0.07	0.04	0.01	0.01	0.022
Test.hs	7.76	2.86	0.08	0.05	>300
Test2.hs	5.63	0.09	0.07	0.01	1.02

---

**Figure 11.** Comparison (in seconds) with other theorem provers