Static contract checking for Haskell

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:

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

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 $\texttt{CF}(x)$.

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

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

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

$$Nil \neq \texttt{UNR}$$

Finally, we define projections for $Cons$. It is not strictly necessary, but it will be handy: $\forall xs, y, ys.\ sel_{1,Cons}(Cons(y, ys)) = xs \implies xs = y$

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

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 $\bot$ in our logic.

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

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

We also need to specify the translation of calls to $notnull$ with $\texttt{BAD}$ values (to encode the fact that $notnull(\texttt{BAD}) = \texttt{BAD}$.

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

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

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

The translation of $head$ follows the same pattern:

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

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

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

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

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 $T \nvdash \bot$), because each formula serves a specific purpose. Now, assume that xs satisfies $\texttt{CF}(xs)$ and $notnull(xs) = true$. We can derive that $xs \neq \texttt{BAD}$ because we have $\texttt{CF}(xs)$ and $\neg\texttt{CF}(\texttt{BAD})$. 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 (**??**). This implies (by (**??**)) that there exist y and ys such that $xs = Cons(y, ys)$. Recalling $\texttt{CF}(xs)$, we can now derive $\texttt{CF}(y)$ and $\texttt{CF}(ys)$ (by (**??**)). But $head(xs) = y$ because of (**??**), and y is crash-free, so we can finally derive $\texttt{CF}(head(xs))$. QED.

2. Languages

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

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

- There's no λ-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 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.

2.2 Contracts

Contract syntax is described in figure **??**. 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.

We give the semantics of contract by defining "e satisfies t", written $e \in t$ in figure **??**. Note that this definition doesn't yield

$$
\begin{array}{rcl}
mod & := & def_1, \ldots, def_n \\
def & \in & \text{Definition} \\
def & := & \texttt{data } T = K_1 \mid \cdots \mid K_n \\
 & \mid & f \in c \\
 & \mid & f\ \vec{x}\ =\ e \\
 & \mid & f\ \vec{x}\ =\ \texttt{case } e \texttt{ of} \\
 & & \quad K_1(\vec{x}) \to e_1 \mid \cdots \mid K_n(\vec{x}) \to e_n \\
x, y, f, g, a, b & \in & \text{Variables} \\
T & \in & \text{Type Constructors} \\
K & \in & \text{Data Constructors} \\
e & \in & \text{Expressions} \\
e & ::= & x \\
 & \mid & \texttt{BAD} \\
 & \mid & e\ e \\
 & \mid & f(e, \ldots, e) \\
 & \mid & K(e, \ldots, e)
\end{array}
$$

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

$$
\begin{array}{rcl}
c & := & x : c \to c \\
 & \mid & (c, c) \\
 & \mid & c \wedge c \\
 & \mid & c \vee c \\
 & \mid & \{x \mid p\} \\
 & \mid & \texttt{CF}
\end{array}
$$

Figure 2. Contract syntax

any operative way to check that an expression actually meets the specification given by its contract.

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

Figure 3. Semantics of contract satisfaction

2.3 Crash-freeness

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

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

[Crash] A closed term e crashed iff $e \to^* \texttt{BAD}$.

[Diverges] A closed expression e diverges iff either $e \to^* \texttt{UNR}$ or there is no value val such that $e \to^* val$

[Syntactic safety] A (possibly open) expression e is syntactically safe iff $\texttt{BAD} \notin_s e$. Similarly a context C is syntactically safe iff $\texttt{BAD} \notin_s \mathcal{C}$.

The notation $\texttt{BAD} \notin e$ means that BAD does not appear anywhere in e, similarly for $\texttt{BAD} \notin_s \mathcal{C}$. For example, $Just3$ is syntactically safe whereas $Just\texttt{BAD}$ is not.

[Crash-free] An expression e is said to be crash-free iff

$$\forall \mathcal{C}.\ \texttt{BAD} \notin_s \mathcal{C} \text{ and } \vdash \mathcal{C}[\![e]\!] :: ()\mathcal{C} \not\to^* \texttt{BAD}$$

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

2.4 BAD and UNR

Consider the following piece of code:

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

Figure 4. First-order logic syntax

$$
\begin{array}{rcl}
\mathcal{E}[\![\text{expression}]\!] & \to & t \text{ (Term)} \\
\mathcal{D}[\![f\ \vec{x} = \ldots]\!] & \to & \Phi \text{ (Set of formulae)} \\
\mathcal{K}[\![\text{data } T = \ldots]\!] & \to & \Phi \text{ (Set of formulae)} \\
\mathcal{C}[\![f \in c]\!] & \to & \Phi \text{ (Set of formulae)}
\end{array}
$$

Figure 5. Translations

```
a = 0 + True

b ::: CF
b = undefined

c = error "foo"
```

- a is ill-typed
- b's implementation is not correct wrt its contract
- 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 instanciate a variable with an ill-typed value! In order to prevent this, we will need to encode some basic type-checking mecanism directly in our first-order theory.

2.5 First-order logic with equality

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

3. Translations

For an overview of the different translations we define, see **??**

3.1 $\mathcal{E}[\![]\!]$ – 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 **??**.

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

We give in **??** and **??** the two translations of function definitions.

?? 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.

$$
\begin{aligned}
\mathcal{E}[\![x]\!] &= x \\
\mathcal{E}[\![f(e_1,\ldots,e_n)]\!] &= f(\mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![K(e_1,\ldots,e_n)]\!] &= K(\mathcal{E}[\![e_1]\!],\ldots,\mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![\mathtt{BAD}]\!] &= \mathtt{BAD}
\end{aligned}
$$

Figure 6. $\mathcal{E}[\![]\!]$ – Expression translation

$$\mathcal{D}[\![f(\vec{x}) = e]\!]$$

$$\forall \vec{x}, y.\ \min(y) \wedge y = f(\vec{x}) \rightarrow y = \mathcal{E}[\![e]\!]$$

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

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

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`.

3.3 $\mathcal{K}[\![]\!]$ – Datatypes

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

$$\mathcal{K}[\![\text{data } T = K_1,\ldots,K_n]\!] = \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$$

- (Φ_1) For each K_i of arity a_i we introduce selectors sel_{k,K_i}, which are the projection of $K_i(x_1,\ldots,x_{n_i)})$ on its k-th component.
- (Φ_2) For each pair of constructors K_i, K_j, we state that they can never map to the same value.
- (Φ_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.
- (Φ_4) None of the K_i is unreachable.
- One may want to also state that if $\vec{x} \neq \mathtt{BAD}$ then $K_i(\vec{x}) \neq \mathtt{BAD}$. It is already implied by the fact that $\mathtt{CF}(\vec{x}) \rightarrow \mathtt{CF}(K_i(\vec{x}))$.

3.4 $\mathcal{C}[\![]\!]$ – Contracts

We give in **??** 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.

Note that we define the translation of $f \in c$ and of $f \notin c$. We have to do that because one is not the negation of the other, even though $\neg(f \notin c)$ implies $f \in c$.

4. $[\![]\!]$ – 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: $\neg\mathtt{CF}(\mathtt{BAD})$.

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

$$[\![M]\!] \vdash \mathcal{C}[\![f \in c]\!]$$

4.3 Contract checking – Recursive case

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

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

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 defintion f and g and two contracts : $f \in \mathtt{CF}$ and $g \in \mathtt{CF}$. 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

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

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.

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

$$\begin{array}{lllll}
\forall \vec{a}, y.\ \min(y) \wedge f(\vec{a}) = y & \rightarrow & (& \min(e) \wedge (e = \mathtt{BAD} \vee \bigvee_i \forall \vec{x}.\ e = K_i(\vec{x}) \vee y = \mathtt{UNR}) & \\
 & \wedge & \forall \vec{x_1}.\ \mathcal{E}[\![e]\!] = K_1(\vec{x_1}) & \rightarrow & y = \mathcal{E}[\![e_1]\!] \\
 & \wedge & & \ldots & \\
 & \wedge & \forall \vec{x_n}.\ \mathcal{E}[\![e]\!] = K_n(\vec{x_n}) & \rightarrow & y = \mathcal{E}[\![e_n]\!])
\end{array}$$

Figure 8. $\mathcal{D}[\![]\!]$ – Case definitions

$$\mathcal{K}[\![\text{data } T = K_1,\ldots,K_n]\!] = \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$$

where

$$\begin{array}{lcl}
\Phi_1 & = & \bigcup_{1 \le i \le n} \forall \vec{x}, a.\ (\min(a) \wedge K_i(\vec{x}) = a) \rightarrow \bigwedge_{1 \le j \le k} x_j = sel_{j,K_i}(a) \\
\Phi_2 & = & \bigcup_{1 \le i < j \le n} \forall \vec{x}, \vec{y}, a.\ \neg(\min(a) \wedge K_i(\vec{x}) = a \wedge K_j(\vec{x}) = a) \\
\Phi_3 & = & \bigcup_{1 \le i \le n} \forall \vec{x}, a.\ \min(a) \wedge a = K_i(\vec{x}) \rightarrow ((\bigwedge_{1 \le j \le k} \mathtt{CF}(x_j)) \leftrightarrow \mathtt{CF}(K_i(\vec{x}))) \\
\Phi_4 & = & \bigcup_{1 \le i \le n} \forall \vec{x}, a.\ (\min(a) \wedge a = K_i(\vec{x})) \rightarrow a \neq \mathtt{UNR} \wedge a \neq \mathtt{BAD}
\end{array}$$

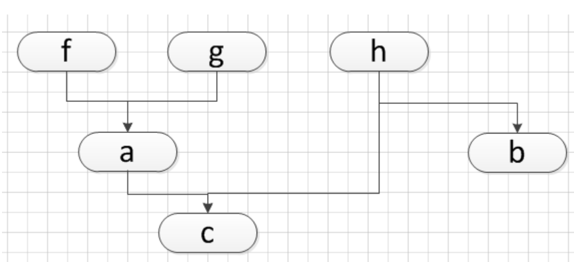
Figure 9. $\mathcal{K}[\![]\!]$ – Data type translation

$$\begin{array}{rcl}
\mathcal{C}[\![e \in \{x \mid b(x)\}]\!] & = & \min(\mathcal{E}[\![b(e)]\!]) \wedge (\mathcal{E}[\![b(e)]\!] = true \vee \mathcal{E}[\![e]\!] = \mathtt{UNR}) \\
\mathcal{C}[\![e \in x : c_1 \rightarrow c_2(x)]\!] & = & \forall x.\ \min(\mathcal{E}[\![e(x)]\!]) \rightarrow (\mathcal{C}[\![x \notin c_1]\!] \vee \mathcal{C}[\![\mathcal{E}[\![e(x)]\!] \in c_2(x)]\!]) \\
\mathcal{C}[\![e \in \mathtt{CF}]\!] & = & \mathtt{CF}(\mathcal{E}[\![e]\!])
\end{array}$$

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

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

Figure 10. $\mathcal{C}[\![]\!]$ – Contract translation



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:

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

5. Correctness of the translation

For the translation to be useful, $[\![M]\!] \vdash \mathcal{C}[\![f \in c]\!]$ 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 **??**

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

7. Experiments

That's how we roll.

$$\mathcal{E}[\![e_1\ e_2]\!] = app(e_1, e_2)$$
$$\forall x_1, \ldots, x_n.\ f(x_1, \ldots, x_n) = app(app(\ldots app(f\_ptr, x_1), x_2), \ldots, x_n)$$
$$\mathtt{CF}(f\_ptr) \leftrightarrow \forall x_1, \ldots, x_n.\ \mathtt{CF}(x_1) \wedge \cdots \wedge \mathtt{CF}(x_n) \rightarrow \mathtt{CF}(f(x_1, \ldots, x_n))$$
$$\forall f\_ptr, x.\ \mathtt{CF}(f\_ptr) \wedge \mathtt{CF}(x) \rightarrow \mathtt{CF}(app(f\_ptr, x))$$

Figure 11. 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 12. Comparison (in seconds) with other theorem provers