Extended Static Checking for Haskell

Dana N. Xu
University of Cambridge
nx200@cam.ac.uk

Abstract

Program errors are hard to detect and are costly both to programmers who spend significant efforts in debugging, and to systems that are guarded by runtime checks. Extended static checking can reduce these costs by helping to detect bugs at compile-time, where possible. Extended static checking has been applied to object-oriented languages, like Java and C#, but it has not been applied to a lazy functional language, like Haskell. In this paper, we describe an extended static checking tool for Haskell, named ESC/Haskell, that is based on symbolic computation and assisted by a few novel strategies. One novelty is our use of Haskell as the specification language itself for pre/post conditions. Any Haskell function (including *recursive* and *higher order* functions) can be used in our specification which allows sophisticated properties to be expressed. To perform automatic verification, we rely on a novel technique based on symbolic computation that is augmented by counter-example guided unrolling. This technique can automate our verification process and be efficiently implemented.

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

General Terms verification, functional language

Keywords pre/postcondition, symbolic simplification, counter-example guided unrolling

1. Introduction

Program errors are common in software systems, including those that are constructed from advanced programming languages, such as Haskell. For greater software reliability, such errors should be reported accurately and detected early during program development. This paper describes an Extended Static Checker for Haskell, named ESC/Haskell (in homage to ESC/Modular-3 [14] and ESC/Java [8]), which is a tool that allows potential errors in Haskell programs, that are not normally detected until run-time to be accurately and quickly reported at compile-time.

Consider a simple example:

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

where `head` is defined in the module Prelude as follows:

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

If we have a call `f []` in our program, its execution will result in the following error message from GHC's runtime system:

```
Exception: Prelude.head: empty list
```

This gives no information on which part of the program is wrong except that `head` has been wrongly called with an empty list. This lack of information is compounded by the fact that it is hard to trace function calling sequence at run-time for lazy languages, such as Haskell.

In general, programmers need a way to assign blame, so that the specific function that is supposedly at fault can be better examined. In the above case, the programmer's intention is that `head` should not be called with an empty list. This effectively means the programmer wants to blame the caller of `head` instead of the `head` function itself. In our system, programmers can achieve this by providing a precondition for the `head` function.

```
head xs @ requires { not (null xs) }
head (x:xs) = x

null :: [a] -> Bool
null [] = True
null xs = False

not True = False
not False = True
```

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

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

Suppose we change `f`'s definition to the following:

```
f xs = if null xs then 0
         else head xs `max` 0
```

With this correction, our tool will not give any more warning as the precondition of `head` is now fulfilled.

Basically, the goal of our system is to detect crashes in a program where a *crash* is informally defined as an unexpected termination of a program (i.e. a call to `error`). Divergence (i.e. non-termination) is not considered to be a crash.

In this paper, we develop ESC/Haskell as a compile-time checker to highlight a variety of program errors, such as pattern matching failure and integer-related violations (e.g. division by zero, array bound checks), that are common in Haskell programs. ESC/Haskell checks each program in a modular fashion on a per function basis. We check the claims (i.e. pre/post-conditions) about

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'06 September 17, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-489-8/06/0009…$5.00.

a function f using mostly the *specifications* of functions that f calls, rather then by looking at their actual *definitions*. This modularity property is essential for the system to scale. We make the following key contributions:

- Pre/postcondition annotations are written in Haskell itself so that programmers do not need to learn a new language. Moreover, arbitrary functions (including higher order and recursive functions) can be used in the pre/postcondition annotations. This allows sophisticated properties to be conveniently expressed. (§2).
- Unlike the traditional verification condition generation approach that solely relies on a theorem prover to verify it, we treat pre/postconditions as boolean-valued functions (§4) and check safety properties using symbolic simplification that adheres closely to Haskell's semantics instead (§5).
- We exploit a counter-example guided (CEG) unrolling technique to assist the symbolic simplification. (§6). CEG approach is used widely for abstraction refinement in the model checking community. However, to the best of our knowledge, this is the first time CEG is used in determining which call to be unrolled.
- We give a trace of calls that may lead to crash at compile-time, whilst such traces are usually offered by debugging tools at run-time. A counter-example is generated and reported together with its function calling trace as a warning message for each potential bug (§7).
- Our prototype system currently works on a significant subset of Haskell that includes user defined data types, higher-order functions, nested recursion, etc (§8).

2. Overview

In a type-safe language, a well-typed program is guaranteed not to crash during run-time due to type errors. In the same spirit, we allow programmers to specify more safety properties (through supplying pre/postconditions for a function) to be checked at compile-time in addition to types. This section gives an informal overview, leaving the details in §3

2.1 Pre/Postcondition Specification

We have seen the precondition annotation for `head`:

```
head xs @ requires { not (null xs) }
```

Such annotations in a program allow ESC/Haskell to check our programs in a modular fashion on a per function basis. At the definition of each function, if there is a precondition specified, our system checks if the precondition can ensure the safety of its function body. If so, when the function is called with crash-free arguments, the call will not lead to any crash. A *crash-free* argument is an expression whose evaluation may diverge, but will not invoke `error`. In other words, whenever a function is called, its caller can assume at the call site that, there will not be a crash resulting from that function call if the arguments satisfy its specified precondition.

Besides the precondition annotation mentioned above, our system also allows the programmer to specify a postcondition of a function. Here is an example:

```
rev xs @ ensures { null $res ==> null xs }
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

where the symbol `$res` denotes the result of the function and the `++` and `==>` are just functions used in an infix manner. They are defined as follows:

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

(==>) :: Bool -> Bool -> Bool
(==>) True x = x
(==>) False x = True
```

The annotated postcondition will be checked in our system to make sure that it is a correct assertion for the function body. With this confirmation, the function's postcondition can be used directly at each of its call sites without re-examining its concrete definition. For example, consider:

```
... case (rev xs) of
      [] -> head xs
      (x:xs') -> ...
```

From the postcondition of `rev`, we know `xs` is `[]` in the first branch of the `case` construct. This situation would definitely fail `head`'s precondition. With the help of pre/postcondition annotations, we can detect such potential bugs in our program.

However, some properties that our ESC/Haskell may attempt to check can either be undecidable or difficult to verify at compile-time. An example is the following:

```
g1 x :: requires { True }
g1 x = case (prime x > square x) of
         True -> x
         False -> error "urk"
```

where `prime` gives the `x`th prime number and `square` gives x^2. Most theorem provers including ours are unable to verify the condition `prime x > square x`, so we report a potential crash. For another example:

```
g2 xs ys :: requires { True }
g2 xs ys
= case (rev (xs ++ ys) == rev ys ++ rev xs) of
    True -> xs
    False -> error "urk"
```

Some theorem provers may be able to prove the validity of the theorem: `(rev (xs++ys) == rev ys ++ rev xs)` for all well-defined `xs` and `ys`. However, this is often at high cost and may require extra lemmas from programmers such as the associativity of the append operator `++`.

As it is known to be expensive to catch all errors in a program, our ESC/Haskell chooses only to provide meaningful messages to programmers based on three possible outcomes after checking for potential crashes in each function definition (say f). They are:

(a) **Definitely safe**. If the precondition of f is satisfied, any call to f with crash-free arguments will not crash.
(b) **Definite bug**. Any call to f with crash-free arguments, satisfying the declared precondition of f, crashes.
(c) **Possible bug**. The system cannot decide it is (a) or (b).

For the last two cases, a trace of function calls that leads to a (potential) crash together with a counter-example[1] will be generated and reported to the programmer. We make a distinction between definite and possible bugs, in order to show the urgency of the former and also because the latter may not be a real bug.

2.2 Expressiveness of the Specification Language

Programmers often find that they use a data type with many constructors, but at some specialised contexts in the program expect only a subset of these constructors to occur. Sometimes, such a data

[1] Programmers can set the number of counter-examples they would like to view.

type is also recursive. For example, in a software module of the Glasgow Haskell Compiler (GHC) that is used after type checking, we may expect that types would not contain mutable type variables. Under such a scenario, certain constructor patterns may be safely ignored. For example, we define a datatype `T` and a predicate `noT1` as follows:

```
data T = T1 Bool | T2 Int | T3 T T

noT1 :: T -> Bool
noT1 (T1 _) = False
noT1 (T2 _) = True
noT1 (T3 t1 t2) = noT1 t1 && noT1 t2
```

The function `noT1` returns `True` when given any data structure of type `T` in which there is no data node with a `T1` constructor. We may have a consumer:

```
sumT :: T -> Int
sumT x @ requires { noT1 x }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
```

which requires that the input data structure does not contain any `T1` node. We may also have a producer like:

```
rmT1 :: T -> T
rmT1 x @ ensures { noT1 $res }
rmT1 (T1 a) = case a of
               True -> T2 1
               False -> T2 0
rmT1 (T2 a) = T2 a
rmT1 (T3 t1 t2) = T3 (rmT1 t1) (rmT1 t2)
```

we know that for all crash-free `t` of type `T`, a call `(sumT (rmT1 t))` will not crash. Thus, by allowing a recursive predicate (e.g. `noT1`) to be used in the pre/postcondition specification, we can achieve such goal.

In fact, any Haskell function can be called in the pre/postcondition specification (though we strongly recommend a *total* function to be used). Here we show a higher-order function `filter` whose result is asserted with the help of another higher-order function `all`.

```
filter f xs @ ensures { all f $res }
filter f [] = []
filter f (x:xs') = case (f x) of
                   True -> x : filter f xs'
                   False -> filter f xs'

all f [] = True
all f (x:xs) = f x &&  all f xs

(&&) True x = x
(&&) False x = False
```

Allowing arbitrary functions to be used in the pre/postcondition specification does not increase the complication of our verification which is based on symbolic simplification. Sometimes, it makes the simplification process easier as all the known information can be re-used. In the case of the postcondition checking for `filter`, we have the following fragment during the symbolic simplification process:

```
case xs of
 [] -> True
 (x:xs') -> case all f (filter f xs') of
            True -> ... all f (filter f xs')  ...
```

All the occurrences of the scrutinee `all f (filter f xs')` in the `True` branch can be replaced by `True`. This simplification process is based on the syntactic transformation that can be very efficiently implemented.

2.3 Functions without Pre/Post Annotation

A special feature of our system is that it is not necessary for programmers to annotate all the functions. There are two reasons why a programmer may choose not to annotate a function with pre/postconditions:

1. The programmer is lazy.
2. There is no pre/postcondition that is more compact than the function definition itself.

Examples of the second case are the function `(==>)`, `null` and even a recursive function like the `noT1` function in §2.2.

If a function (including recursive function) does not have pre/post-condition annotation, one way is to assume both its precondition and postcondition to be `True`. It is always safe to assign `True` as the postcondition to any function, this weak assertion effectively causes the result of the function to be unknown. However, assuming `True` as a function's precondition may lead to unsoundness. Our approach is to *inline* the function definition at each of its call sites.

We introduce a special strategy, called *counter-example guided unrolling*, which only unroll (i.e. inline) a function call on demand and the details are described in §6. We guarantee termination in our checking by only unrolling a recursive function for a fixed number of times - a number that can be pre-set in advance. Normally, if a structural recursive function is used as a predicate in the pre/postcondition of another structural recursive function, the recursive calls in both functions may not need to be unrolled at all. An example of this is elaborated in §6 where a recursive `sumT` function makes use of a similar structurally recursive predicate `noT1` in its precondition. But we still recommend programmers to provide annotations for functions with big code size.

Inlining also helps to reduce false alarms created due to laziness. For example:

```
fst (a,b) = a
f3 xs = (null xs, head xs)
f4 xs = fst (f3 xs)
```

A conservative precondition for `f3` is `not (null xs)`. Without inlining (i.e. treating both the pre/post condition of `fst` and `snd` to be `True`), our system will report spurious warnings

```
(f4 []) may fail f3's precondition
```

when checking the definition of `f4`. However, by inlining `f3`, `fst` and `snd`, we have `f4 xs = null xs` and our system will not give the spurious warning mentioned before.

3. The Language

In this section, we set the scene for ESC/Haskell by giving the syntax and semantics of the language and necessary definitions. The language $\mathcal{H}$, whose syntax is shown in Figure 1, is a subset of Haskell augmented with a few special constructs, namely `BAD`, `UNR`, `OK` and `Inside`. These language constructs are for ESC/Haskell to use internally and hidden from Haskell programmers.

3.1 Language Syntax and Features

We assume a program is a module that contains a set of function definitions. Programmers can give multiple preconditions and postconditions with key words `requires` and `ensures` respectively. These pre/postconditions are type-checked by a preprocessor.

The `let` in the language $\mathcal{H}$ is simply a non-recursive `let`. In this paper, we allow top-level recursive functions and do not

pgm	$\in$	**Program**	
pgm	$:=$	$def_1, \ldots, def_n$	
def	$\in$	**Definition**	
def	$:=$	$f\ \vec{x} = e$	
	$\mid$	$f\ \vec{x}$ `@ requires` $\{\ e\ \}$	
	$\mid$	$f\ \vec{x}$ `@ ensures` $\{\ e\ \}$	
a, e	$\in$	**Expression**	
a, e	$::=$	`BAD` lbl	A crash
	$\mid$	`OK` e	Safe expression
	$\mid$	`UNR`	Unreachable
	$\mid$	`Inside` $lbl\ loc\ e$	A call trace
	$\mid$	$\lambda x.e$	
	$\mid$	$e_1\ e_2$	An application
	$\mid$	`case` e_0 `of` $alts$	
	$\mid$	`let` x=e_1 `in` e_2	
	$\mid$	$C\ e_1 \ldots e_n$	Constructor application
	$\mid$	x	Variable
	$\mid$	n	Constant
$alts$	$::=$	$alt_1 \ldots alt_n$	
alt	$::=$	$p\ \rightarrow\ e$	Case alternative
p	$::=$	$C\ x_1 \ldots x_n$	Pattern
val	$\in$	**Value**	
val	$::=$	$n \mid C\ e_1 \ldots e_n \mid \lambda x.e$	

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

support nested `letrec` while a version that supports `letrec` can be found in our technical report [21].

The (`OK` e) indicates that the evaluation of e will never crash. The constructor `Inside` is for tracing the calling path that leads to `BAD` where *lbl* and *loc* give the name and the location of the function being called respectively.

The (`BAD` *lbl*) indicates a point where a program definitely crashes. A program crashes if and only if it calls `BAD`. The label *lbl* is a message of type `String`. For example, a user-defined function `error` can be explicitly defined as:

```
error :: String -> a
error s = BAD ("user error:"++ s)
```

We shall ensure that source programs with missing cases of pattern matching are explicitly replaced by the corresponding equations with `BAD` constructs. This is carried out by the preprocessor as well. For example, if a programmer writes:

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

after the preprocessing, it becomes:

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

In the ESC/Haskell system, we construct a checking code named $f_{\texttt{Chk}}$ for each function f. The $f_{\texttt{Chk}}$ denotes a piece of Haskell code whose simplified version determines the three outcomes mentioned at the end of §2.1. One fragment of $f_{\texttt{Chk}}$ may look like this:

```
case f.pre x of
 True -> ...
 False -> UNR
```

where `f.pre` denotes the precondition of f and similar notation applies in the rest of the paper. If the precondition of a function is not satisfied, we assume the function body will not be evaluated. So we use `UNR` to indicate that the `False` branch is unreachable. In order not to keep a large number of unreachable branches during the simplification process, we choose to omit them. This is achieved by one of the simplification rules which tells the simplifier to remove all the unreachable branches. For example, the above fragment will become:

```
case f.pre x of
 True -> ...
```

Thus, in our language $\mathcal{H}$ if there should be any cases of missing patterns (e.g. during the symbolic simplification of $f_{\texttt{Chk}}$), they will effectively denote unreachable states.

3.2 Operational Semantics

The call-by-need operational semantics of the language is given in Figure 2 and is based on work by Moran and Sands [16]. The transitions are over machine configurations consisting of a heap Γ (which contains bindings), the expression currently being evaluated e, and a stack S.

$$\Gamma := \{x_1 = e_1, \ldots, x_n = e_n\}$$
$$S := \epsilon \mid e : S \mid alts : S \mid \sharp x : S \mid (\texttt{OK}\ \bullet) : S \mid (\texttt{Inside}\ f\ l\ \bullet) : S$$

The heap is a partial function from variable to terms. The stack S is a stack of continuations that says what to do when the current expression is evaluated. A continuation can be an expression e which is a function's argument, case alternatives, *update markers* denoted by $\sharp x$ for some variable x or constructors `OK` and `Inside`. When the stack is empty, the current expression is returned as the final result. Transition rules for `Inside` are similar to those of `OK` except for INSIDEBAD which is as follows.

$$\langle \Gamma,\ \texttt{BAD}\ lbl,\ (\texttt{Inside}\ f\ l\ \bullet) : S \rangle \quad \rightarrow \quad \langle \Gamma,\ \texttt{Inside}\ f\ l\ \texttt{BAD},\ \epsilon \rangle$$

3.3 Definitions

Before we describe the algorithm for pre/postcondition checkings, we need to give a few formal definitions. Given a function $f\ \vec{x} = e$, we wish to check under all contexts whether e will crash. If f is given an argument (say a) that contains `BAD` *lbl*, the call $(f\ a)$ may crash but this may not be f's fault. Thus, what we would like to check is whether e will crash when f takes a crash-free argument whose definition is given below.

DEFINITION 1 (Crash-free Expression). *For all heap* Γ*, an expression* e *is crash-free in* Γ *iff for all totally safe* S. $\langle \Gamma,\ e,\ S \rangle \not\rightarrow^* \langle \Gamma,\ \texttt{BAD}\ lbl,\ \epsilon \rangle$.

DEFINITION 2 (Totally Safe Stack). *A stack* S *is totally safe iff* $\forall s \in S$, $s = e$ *and* e *is a totally safe expression*
or $s = \{C_i\ \vec{x} \rightarrow e_i\}$ *and* $\lambda \vec{x}.e_i$ *is an totally safe expression.*

DEFINITION 3 (Totally Safe Expression). *An expression* e *is a totally safe expression iff* e *is closed and* `noBAD`(e) *returns* `True`.

We define a function named `noBAD :: Exp -> Bool` which syntactically checks whether there is any `BAD` appearing in an expression e. The definition of `noBAD` is shown in Appendix B.1.

Note that a crash-free expression is allowed to diverge. For example:

```
repeat x = x : repeat x
one = repeat 1
```

$$
\begin{array}{rcll}
\langle \Gamma,\ \texttt{OK}\, e,\ S\rangle & \rightarrow & \langle \Gamma,\ e,\ (\texttt{OK}\,\bullet)\!:\!S\rangle & (\text{OK})\\
\langle \Gamma,\ \texttt{BAD}\, lbl,\ (\texttt{OK}\,\bullet)\!:\!S\rangle & \rightarrow & \langle \Gamma,\ \texttt{UNR},\ [\,]\rangle & (\text{OKBAD})\\
\langle \Gamma,\ n,\ (\texttt{OK}\,\bullet)\!:\!S\rangle & \rightarrow & \langle \Gamma,\ n,\ S\rangle & (\text{OKCONSTANT})\\
\langle \Gamma,\ C\, e_1 \ldots e_n,\ (\texttt{OK}\,\bullet)\!:\!S\rangle & \rightarrow & \langle \Gamma,\ C\, (\texttt{OK}\, e_1)\ldots(\texttt{OK}\, e_n),\ S\rangle & (\text{OKCONSTRUCT})\\
\langle \Gamma,\ \lambda x.e,\ (\texttt{OK}\,\bullet)\!:\!S\rangle & \rightarrow & \langle \Gamma,\ \lambda x.\texttt{OK}\, e,\ S\rangle & (\text{OKLAMBDA1})\\
\\
\langle \Gamma,\ \texttt{UNR}\, lbl,\ S\rangle & \rightarrow & \langle \Gamma,\ \texttt{UNR},\ [\,]\rangle & (\text{UNREACHABLE})\\
\langle \Gamma,\ \texttt{BAD}\, lbl,\ S\rangle & \rightarrow & \langle \Gamma,\ \texttt{BAD}\, lbl,\ [\,]\rangle \quad (\texttt{OK}\,\bullet)\notin S & (\text{BAD})\\
\langle \Gamma\{x=e\},\ x,\ S\rangle & \rightarrow & \langle \Gamma,\ e,\ \sharp x\!:\!S\rangle & (\text{LOOKUP})\\
\langle \Gamma,\ val,\ \sharp x\!:\!S\rangle & \rightarrow & \langle \Gamma\{x=val\},\ val,\ S\rangle & (\text{UPDATE})\\
\langle \Gamma,\ \lambda x.e_1,\ e_2\!:\!S\rangle & \rightarrow & \langle \Gamma\{x=e_2\},\ e_1,\ S\rangle & (\text{LAMBDA})\\
\langle \Gamma,\ e_1\, e_2,\ S\rangle & \rightarrow & \langle \Gamma,\ e_1,\ e_2\!:\!S\rangle & (\text{UNWIND})\\
\langle \Gamma,\ \texttt{case}\ e\ \texttt{of}\ alts,\ S\rangle & \rightarrow & \langle \Gamma,\ e,\ alts\!:\!S\rangle & (\text{CASE})\\
\langle \Gamma,\ C_j\, \vec{y},\ \{C_i\, \vec{x_i} \rightarrow e_i\}\!:\!S\rangle & \rightarrow & \langle \Gamma,\ e_j[\vec{y}/\vec{x_j}],\ S\rangle & (\text{BRANCH})\\
\langle \Gamma,\ \texttt{let}\ \{\vec{x}=\vec{e}\}\ \texttt{in}\ e_0,\ S\rangle & \rightarrow & \langle \Gamma\{\vec{x}=\vec{e}\},\ e_0,\ S\rangle \quad \vec{x}\notin dom(\Gamma,S) & (\text{LET})
\end{array}
$$

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

where `one` is an infinite list of `1`s. The expression `(repeat 1)` is crash-free, despite its potential for divergence.

Now we can formally define valid pre/postconditions of a function, as follows.

DEFINITION 4 (Precondition). f.`pre` *is a precondition of a function* f *iff for all heap* Γ *and crash-free expressions* $\vec{a}$ *in* Γ*, if* `ok` $(f.\texttt{pre}\ \vec{a})$ *is crash-free in* Γ*, then* $(f\ \vec{a})$ *is crash-free in* Γ.

The definition of the function `ok` is defined as follows.

```
ok :: Bool -> ()
ok True = ()
ok False = BAD "ok"
```

The definition of precondition says that f's arguments $\vec{a}$ are crash-free (but allowed to diverge), if $f.\texttt{pre}\ \vec{a}$ does not evaluate to `False` or `BAD`, then $f\ \vec{a}$ will not crash.

As we allow recursive predicates to be used in the precondition specification, the precondition may diverge. If the precondition itself diverges, it is still considered as a valid precondition because any call satisfying the precondition will diverge *before* the call is invoked. For example:

```
bot :: a -> a
bot x = bot x

p :: [Int] -> Int
p xs @ requires { bot xs == 5 && not (null xs) }
p [] = BAD "p"
p (x:xs') = x + 1

q :: [Int] -> Int
q [] = 0
q xs = case bot xs == 5 of
         True -> p xs
         False -> 0
```

We can see that p's precondition is satisfied in the definition of `q`. When `q` is called, the program diverges and thus, the call to `(p xs)` will never be invoked and `(q xs)` is crash-free.

DEFINITION 5 (Postcondition). f.`post` *is a postcondition of a function* f *iff for all heap* Γ *and crash-free expressions* $\vec{a}$ *in* Γ*. if* `ok`$(f.\texttt{pre}\ \vec{a})$ *is crash-free in* Γ *and then* `ok` $(f.\texttt{post}\ \vec{e}\ (f\ \vec{a}))$ *is crash-free in* Γ.

As we allow recursive predicates to be used in the postcondition specification, the postcondition may diverge as well. For example:

```
and :: [Bool] -> Bool
and [] = True
and (b:bs) = b && (and bs)

ts @ ensures { and $res }
ts = repeat True

h1 xs @ requires { and xs }
h1 xs = ...

h2 xs = take 5 (h1 ts)
```

The postcondition of `ts` diverges, but this postcondition can be useful at its call site, for example, in `h2`.

4. Symbolic Pre/Post Checking for ESC/Haskell

At the definition of each function f, we shall assume that its given precondition holds, and proceed to check three aspects, namely:

(1) No pattern matching failure

(2) Precondition of all calls in the body of f holds

(3) Postcondition holds for f itself.

Given $f\ \vec{x} =\ e$ with precondition `f.pre` and postcondition `f.post`, we can specify the above checkings by the following symbolic *checking code*, named $f_{\texttt{Chk}}$:

$$
\begin{array}{l}
f_{\texttt{Chk}}\ \vec{x} = \texttt{case f.pre}\ \vec{x}\ \texttt{of}\\
\quad\quad \texttt{True} \rightarrow \texttt{let \$res} = e[f_1\texttt{\#}/f_1,\ldots,f_n\texttt{\#}/f_n]\\
\quad\quad\quad\quad \texttt{in case f.post}\ \vec{x}\ \texttt{\$res}\\
\quad\quad\quad\quad\quad \texttt{True} \rightarrow \texttt{\$res}\\
\quad\quad\quad\quad\quad \texttt{False} \rightarrow \texttt{BAD "}post\texttt{"}
\end{array}
$$

where $f_1 \ldots f_n$ refer to top-level functions that are called in e, including f itself in the self-recursive calls. In our system, for each function f in a program, we compute a representative function for it, named f`#`. The representative function f`#` is computed solely based on the pre/postcondition of f (if they are given) as follows:

$$
\begin{array}{lcl}
f\texttt{\#}\ \vec{x} & = & \texttt{case f.pre}\ \vec{x}\ \texttt{of}\\
& & \quad \texttt{False} \rightarrow \texttt{BAD "}f\texttt{"}\\
& & \quad \texttt{True} \rightarrow \texttt{let \$res} = (\texttt{OK}\ f)\ \vec{x}\\
& & \quad\quad\quad \texttt{in case f.post}\ \vec{x}\ \texttt{\$res of}\\
& & \quad\quad\quad\quad \texttt{True} \rightarrow \texttt{\$res}
\end{array}
$$

where $(\texttt{OK}\ f)$ means given a crash-free argument $\vec{a}$, $(f\ \vec{a})$ will not crash. The f`#` basically says that, if the precondition of f is satisfied, there will not be a crash from a call to f. Moreover, if the postcondition is satisfied, we return the function's symbolic result which is ((OK f) $\vec{x}$). If the precondition of f is not satisfied, it indicates a potential bug by BAD "f". That means all crashes from f are exposed in f# (i.e. the BAD in the False branch) as (OK f) turns all BAD in f to UNR according the operational semantics in Figure 2. and this justifies the substitution $[f_1\#/f_1 \ldots f_n\#/f_n]$ in the $f_{\texttt{Chk}}$. We claim that $f_{\texttt{Chk}}$ satisfies the following theorem.

THEOREM 1 (Soundness of Pre/Postcondition Checking). *For all e such that e is crash-free in Γ, if $f_{\texttt{Chk}}$ e is crash-free in Γ, then* `f.pre` *is a precondition of f and* `f.post` *is a postcondition of f.*

To show the soundness, we need to answer the following two questions:

(a) How to show $f_{\texttt{Chk}}$ is crash-free?

(b) If $f_{\texttt{Chk}}$ is crash-free, why does it help in checking the three aspects (1), (2) and (3)?

To show $f_{\texttt{Chk}}$ is crash-free, we symbolically simplify the RHS of $f_{\texttt{Chk}}$ and check for the existence of BAD in the simplified version. The check for the existence of BAD in e is achieved by invoking a (noBAD e) function call. That means we hope that all or some of the BADs could be eliminated during the simplification process. If the BAD "$post$" remains after simplification, we know the postcondition has failed. A residual BAD lbl indicates a precondition has failed. Furthermore, from the label lbl, we can also determine which function call's precondition has failed. Details of the simplification process are described in §5.

To check (1), we just need to check whether there is any BAD in e because a preprocessing algorithm converts each missing pattern matching of a function from the source program to a case-branch that leads to a BAD in e. If there is no BAD in e, we know that when the function f is called, the program will not crash due to any pattern matching failure in f.

To check (2), we need to check whether there is any BAD in

$$e[f_1\#/f_1, \ldots, f_n\#/f_n]$$

If the BAD in each f_i# is removed, by the definition of f#, $\forall i.$ the precondition of f_i is satisfied. If f is a recursive function, it means we assume the precondition is True at the entry of the definition and try to show that the precondition at each recursive call is satisfied.

To check (3), we want to check whether (f.post $\vec{x}$ \$res) gives True where \$res $= e[f_1\#/f_1, \ldots, f_n\#/f_n]$. So if the BAD "$post$" remains after simplification, it indicates that the postcondition does not hold. Note that in the definition of f#, we assume the postcondition holds for each recursive call. In other words, with this assumption, we try to show the postcondition holds for the RHS of f as well.

For a function without pre/postcondition annotations, it is always safe to assume f.post is True. But for precondition, we first assume f.pre is True and use the same checking code $f_{\texttt{Chk}}$ to determine if there are any BADs after simplification. If there is no BAD, we know it is safe to assign f.pre to be True and can use: f# $\vec{x}$ = (OK f) $\vec{x}$. Otherwise, we have: f# $\vec{x}$ = (f $\vec{x}$). Our use of direct calls to f is meant to allow its concrete definition e to be inlined, where necessary. Our strategy for inlining (also called unrolling) is discussed later in §6.

5. Simplifier

As there is no automatic theorem prover that handles arbitrary user defined data types and higher-order functions, we need to write our own specialised solver which we call the *simplifier*. The simplifier is based on symbolic evaluation and attempts to simplify our checking code to some normal form. A set of deterministic simplification rules is shown in Figure 3 (where $fv(e)$ returns free variables of e). Each rule is a theorem which has been proven to be sound (see [21]). That means for each rule $e_1 \Longrightarrow e_2$, we prove $e_1 \equiv_s e_2$. In the DEFINITION 7, as usual, we restrict the result type to be a single observable type, here Boolean.

DEFINITION 6 (Convergence). *For closed configurations $\langle\Gamma,\ e,\ S\rangle$, $\langle\Gamma,\ e,\ S\rangle \Downarrow val$ iff $\exists\Gamma'.\langle\Gamma,\ e,\ S\rangle \rightarrow^* \langle\Gamma',\ val,\ \epsilon\rangle$.*

DEFINITION 7 (Semantically Equivalent). *Two expressions e_1 and e_2 are semantically equivalent, namely $e_1 \equiv_s e_2$, iff $\forall\Gamma, S$. $(\langle\Gamma,\ e_1,\ S\rangle\Downarrow \texttt{True}) \Leftrightarrow (\langle\Gamma,\ e_2,\ S\rangle\Downarrow \texttt{True})$.*

5.1 Simplification Rules

Many simplification rules are adopted from the literature [17]. For example, the INLINE rule removes all let bindings, the beta-reduction rule BETA and the rule CASECASE which floats out the scrutinee. The short-hand $\{C_i\ \vec{x_i} \rightarrow e_i\}$ stands for $\forall i, 1 \leq i \leq n.\{C_1\ \vec{x_1} \rightarrow e_1; \ldots; C_n\ \vec{x_n} \rightarrow e_n\}$ where $\vec{x_i}$ refers to a vector of fields of a constructor C_i. The rule CASEOUT pushes an application into each branch. The rest of the rules are elaborated as follows.

Unreachable In the rule NOMATCH, it says that if the scrutinee does not match any branch, we replace the case-expression by UNR. Due to the unreachable False branch of the test of the f.pre in the $f_{\texttt{Chk}}$, we may have the following derived code fragment during the simplification process:

```
... case False of
      True -> ...
```

The inner case expression contains only one pattern matching branch, and we assume the other branch (i.e. the missing case) is unreachable as mentioned in §3. So the fragment actually represents this:

```
... case False of
      True -> ...
      False -> UNR
```

which means the scrutinee matches the False branch which is an unreachable branch and this justifies our simplification rule NOMATCH.

As explained earlier in §3, in order to reduce the size of the expression during the simplification process, we remove all branches that are unreachable and this is achieved by the rule UNREACHABLE.

Match The rule MATCH follows directly from the transition rule BRANCH in Figure 2 which selects the matched branch and remove the unmatched branches. This rule seems to be able to replace the two rules NOMATCH and UNREACHABLE, but this is not the case. Consider:

```
... case xs of
     True -> case False of
               True -> ...
     False -> ...
```

The rule MATCH only deals with the situation when the scrutinee matches one of the branches. So in the above case, we need to apply the rule NOMATCH and UNREACHABLE respectively to get:

```
... case xs of
     False -> ...
```

Common Branches During the simplification process, we often encounter code fragment like this:

```
... case xs of
     C1 -> True
     C2 -> True
```

$$\texttt{let}\ x = r\ \texttt{in}\ b \implies b[r/x] \qquad \text{(INLINE)}$$

$$(\lambda x.e_1)\ e_2 \implies e_1[e_2/x] \qquad \text{(BETA)}$$

$$(\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\})\ a \implies \texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to (e_i\ a)\} \quad fv(a) \cap \vec{x_i} = \emptyset \qquad \text{(CASEOUT)}$$

$$\texttt{case}\ (\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\})\ \texttt{of}\ alts \implies \texttt{case}\ e_o\ \texttt{of}\ \{C_i\ \vec{x_i} \to \texttt{case}\ e_i\ \texttt{of}\ alts\} \quad fv(alts) \cap \vec{x_i} = \emptyset \qquad \text{(CASECASE)}$$

$$\texttt{case}\ C_j \vec{e_j}\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\} \implies \texttt{UNR} \quad \forall i. C_j \neq C_i \qquad \text{(NOMATCH)}$$

$$\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i; C_j\ \vec{x_j} \to \texttt{UNR}\} \implies \texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\} \qquad \text{(UNREACHABLE)}$$

$$\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\} \implies e_1 \quad \text{patterns are exhaustive and for all } i, fv(e_i) \cap \vec{x_i} = \emptyset \text{ and } e_1 = e_i \qquad \text{(SAMEBRANCH)}$$

$$\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to e\} \implies e_0 \quad e_0 \in \{\texttt{BAD}\ lbl, \texttt{UNR}\} \qquad \text{(STOP)}$$

$$\texttt{case}\ C_i\ \vec{y_i}\ \texttt{of}\ \{C_i\ \vec{x_i} \to e_i\} \implies e_i[y_i/x_i] \qquad \text{(MATCH)}$$

$$\texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to \ldots \texttt{case}\ e_0\ \texttt{of} \{C_i\ \vec{x_i} \to e_i\} \ldots\} \implies \texttt{case}\ e_0\ \texttt{of}\ \{C_i\ \vec{x_i} \to \ldots e_i \ldots\} \qquad \text{(SCRUT)}$$

Figure 3. Simplification Rules

In the rule SAMEBRANCH if all branches are identical (w.r.t. α-conversion), the scrutinee is redundant. However, we need to be careful as we should do this *only if*

(a) all patterns are exhaustive (i.e. all constructors of a data type are tested) and

(b) no free variables in e_i are bound in $c_i\ \vec{x_i}$.

For example, consider:

```
rev xs @ ensures { null $res ==> null xs }
```

During the simplification of its checking code `revChk`, we may have:

```
... case $res of
    [] -> case xs of
           [] -> $res
    (x:xs') -> ...
```

The inner `case` has only one branch (the other branch is understood to be unreachable). It might be believed that we would replace the expression (`case xs of {[] -> $res }`) by `$res` as there is only one branch that is reachable and the resulting expression does not rely on any substructure of `xs`. However, this makes us lose a critical piece of information, namely:

$$\textit{if}\ \texttt{(rev xs) == []}, \textit{then}\ \texttt{xs == []}.$$

On the other hand, given this information we can perform more aggressive simplification. For example, suppose we have another function `g` that calls `rev`:

```
g xs = case (rev xs) of
         [] -> ... case xs of
                     [] -> True
                     (x:xs) -> False
         (x:xs) -> ...
```

we may use the above information to simplify the inner `case` to `True` which may allow more aggressive symbolic checking.

Termination The rule STOP follows from the transitions:

$$\begin{array}{rcl}\langle \Gamma,\ \texttt{case BAD}\ lbl\ \texttt{of}\ alts,\ S\rangle & \to & \langle \Gamma,\ \texttt{BAD}\ lbl,\ alts : S\rangle \\ & \to & \langle \Gamma,\ \texttt{BAD}\ lbl,\ [\,]\rangle\end{array}$$

Similar reasoning applies when the scrutinee is `UNR`.

Static Memoization As mentioned at the end of §2.2, all known information should be used in simplifying an expression. In order for the rule SCRUT to work, we need to keep a table which captures all the information we know when we traverse the syntax tree of an expression. As the scrutinee of a case-expression is an expression, the key of the table is an expression rather than a variable. The value of the table is the information that is true for the corresponding scrutinee. For example, when we encounter:

```
case (noT1 x) of
 True -> e1
```

we extend the information table like this:

:	:
noT1 x	True

When we symbolically evaluate `e1` and encounter `(noT1 x)` a second time in `e1`, we look up its corresponding value in the information table for substitution.

5.2 Arithmetic

Our simplification rules are mainly to handle pattern matchings. For expressions involving arithmetic, we need to consult a theorem prover. Suppose we have:

```
foo :: Int -> Int -> Int
foo i j @ requires {i > j}
```

Its representative function `foo#` looks like this:

```
foo# i j = case (i > j) of
             False -> BAD "foo"
             True  -> ...
```

Now, suppose we have a call to `foo`:

```
goo i = foo (i+8) i
```

After inlining `foo#`, we may have such symbolic checking code:

```
gooChk i = case (i+8 > i) of
              False -> BAD "foo"
              True  -> ...
```

A key question to ask is if `BAD` can be reached? To reach `BAD`, we need `i+8 > i` to return `False`. Now we can pass this off to a theorem prover that is good at arithmetic and see if we can prove that this case is unreachable. If so, we can safely remove the branch leading to BAD.

In theory, we can use any theorem prover that can perform arithmetics. Currently, we choose a free theorem prover named *Simplify* [5] to perform arithmetic checking in an incremental manner. For each case scrutinee such that

- it is an expression involving solely primitive operators, or
- it returns a boolean data constructor

we invoke *Simplify* prover to determine if this scrutinee evaluates to definitely *true*, definitely *false* or *DontKnow*. If the answer is either *true* or *false*, the simplification rule of MATCH is applied as well as adding this to our information table. Otherwise, we just keep the scrutinee and continue to symbolically evaluate the branches.

Each time we query the theorem prover *Simplify*, we pass the knowledge accumulated in our information table as well. For example, we have the following fragment during the simplification process:

```
... case i > j of
     True -> case j < 0 of
              False -> case i > 0 of    -- (*)
                         False -> BAD
```

When we reach the line marked by `(*)` and before query `i > 0`, we send information `i > j == True` and `j < 0 == False` to the *Simplify*. Such querying can be efficiently implemented through the push/pop commands supplied by the theorem prover which allow truth information to be pushed to a global (truth) stack and popped out when it is no longer needed.

6. Counter-Example Guided Unrolling

If every function is annotated with a pre/postcondition that is succinct and precise enough to capture the gist of the function and no recursive function is used in the pre/postcondition, the *simplifier* alone is good enough to determine whether the checking code is crash-free or not. However, real life programs may not fit into the above scenario and we need to introduce new strategies. Consider:

```
sumT :: T -> Int
sumT x @ requires { noT1 x }
sumT (T2 a) = a
sumT (T3 t1 t2) = sumT t1 + sumT t2
```

where `noT1` is the recursive predicate mentioned in §2.3. After simplifying the RHS of its checking code `sumTChk`, we may have:

```
case ((OK noT1) x) of
 True ->case x of
       T1 a -> BAD
       T2 a -> a
       T3 t1 t2 ->case ((OK noT1) t1) of
                  False -> BAD
                  True ->case ((OK noT1) t2) of
                        False -> BAD
                        True -> (OK sumT) t1
                              + (OK sumT) t2
```

Program Slicing To focus on our goal (i.e. removing BADs) as well as to make the checking process more efficient, we slice the program by collecting only the paths that lead to BAD. A function named `slice`, which does the job, is defined in Appendix B.2. A call to `slice` gives the following sliced program:

```
case ((OK noT1) x) of
 True ->case x of
       T1 a -> BAD
       T3 t1 t2 ->case ((OK noT1) t1) of
                  False -> BAD
                  True ->case ((OK noT1) t2) of
                        False -> BAD
```

The Unrolling Itself We know we need to unroll one or all of the call(s) to `noT1` in order to proceed. Let us unroll them one by one. The unrolling is done by a function named `unroll` which is defined in Appendix B.3. This function unrolls calls *on demand*, for example, $\texttt{unroll}(f\ (g\ x))$ will only inline the definition of f and leaves the call $(g\ x)$ untouched. When `unroll` is given an expression wrapped with OK, besides unrolling the call, it wraps all functions in each call with OK. Thus, the unrolling of the topmost `(OK noT1)` gives:

```
case (\x -> case x of
            T1 a' -> False
            T2 a' -> True
            T3 t1' t2' -> (OK noT1) t1' &&
                          (OK noT1) t2') x) of
   True ->case x of
         T1 a -> BAD
         T3 t1 t2 ->case ((OK noT1) t1) of
                    False -> BAD
                    True ->case ((OK noT1) t2) of
                          False -> BAD
```

Keeping Known Information Note that the new information `(OK noT1) t1' && (OK noT1) t2'` after the unrolling is what we need to prove `((OK noT1) t1)` and `((OK noT1) t2)` cannot be `False` at the branches. However, if we continue unrolling the calls `((OK noT1) t1)` and `((OK noT1) t2)` at the branches, we lose the information `(noT1 t1) == False` and `(noT1 t2) == False`. To solve this problem (i.e. to keep this information), we add one extra case-expression after each unrolling. So unrolling the call of `(noT1 x)` actually yields:

```
case (case (NoInline ((OK noT1) x)) of
     True ->(\x -> case x of
                T1 a' -> False
                T2 a' -> a'
                T3 t1' t2'->((OK noT1) t1' &&
                         ((OK noT1) t2'))) x) of
 True ->case x of
         T1 a -> BAD
         T3 t1 t2 ->case ((OK noT1) t1) of
                    False -> BAD
                    True ->case ((OK noT1) t2) of
                          False -> BAD
```

But to avoid unrolling the same call more than once, we wrap `(noT1 x)` with `NoInline` constructor which prevents the function `unroll` from unrolling it again.

Counter-Example Guided Unrolling - The Algorithm Given a checking code $f_{\texttt{Chk}}\ \vec{x} = rhs$, as we have seen that in order to remove BADs, we may have to unroll some function calls in the rhs. One possible approach is to pre-set a fixed number of unrolling (either by system or by programmers) and we unroll all function calls a fixed number of times before we proceed further. A better alternative is to use a counter-example guided unrolling technique which can be summarised by the pseudo-code algorithm `escH` defined below:

```
escH rhs 0 = "Counter-example :" ++ report rhs
escH rhs n =
let rhs' = simplifier rhs
    b = noBAD rhs'
in case b of
  True → "No Bug."
  False → let s = slice rhs'
          in case noFunCall s of
            True → let eg = oneEg s
                   in "Definite Bug :" ++ report eg
            False → let s' = unrollCalls s
                    in escH s' (n − 1)
```

Basically, the `escH` function takes the RHS of $f_{\texttt{Chk}}$ to simplify it and hope all `BAD`s will be removed by the simplification process. If there is any residual `BAD`, it will report to the programmer by generating a warning message. To guarantee termination, `escH` takes a pre-set number which indicates the maximum unrolling that should be performed. Before this number decreases to 0, it simplifies the `rhs` once and calls `noBAD` to check for the absence of `BAD`. If there is any `BAD` left, we slice rhs' and obtain an expression which contains all paths that lead to `BAD`. If there is no function calls in the sliced expression which can be checked by a function named `noFunCalls`, we know the existence of a definite bug and report it to programmers. In our system, programmers can pre-set an upper bound on the number of counter-examples that will be generated for the pre/post checking of each function. By default, it gives one counter-example. If there are function calls, we unroll each of them by calling `unroll`.

This procedure is repeated until either all `BAD`s are removed or the pre-set number of unrollings has decreased to 0. When `escH` terminates, there are three possible outcomes:

- No `BAD` in the resulting expression (which implies definitely safe);
- `BAD` *lbl* (where *lbl* is not "*post*") appears and there is no function calls in the resulting expression (where each such `BAD` implies a definite bug);
- `BAD` *lbl* (where *lbl* is not "*post*") appears and there are function calls in the resulting expression (where each such `BAD` implies a possible bug).

These are essentially the three types of messages we suggest to report to programmers in §2.1.

From our experience, unrolling is mainly used in the following two situations:

1. A recursive predicate (say `noT1`) is used in the pre/postcondition of another function (say `sumT1`). During the checking process, only the recursive predicates are unrolled. We do not need to unroll `sumT1` at all as its recursive call is represented by its pre/postcondition whose information is enough for the checking to be done. Thus, we recommend programmers to use only recursive predicate of small code size.
2. A recursive function is used without pre/postcondition annotation. In such a case, we may unroll its recursive call to obtain more information during checking. An example is illustrated in §8.3.

7. Tracing and Counter-Example Generation

After trying hard to simplify all `BAD`s in a checking code, if there is still any `BAD` left, we will report it to programmers by generating a meaningful message which contains a counter-example that shows the path that leads to the potential bug.

As claimed in §1, our static checker can give more meaningful warnings. We achieve this by putting a label in front of each representative function. The real `f#` used in our system is of this form:

```
f# x⃗  =  Inside "f" loc
            (case f.pre x⃗ of
             False → BAD "f"
             True → let $res = (OK f) x⃗
                    in case f.post x⃗ $res of
                       True → $res)
```

where the *loc* indicates the location (e.g. (row,column)) of the definition of f in the source code file. For example, we have:

```
f1 x z @ requires { x < z }
f2 x z = 1 + f1 x z

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

After simplification of the checking code of `f3`, we may have:

```
f3Chk xs z = case xs of
             [] -> 0
             (x:y) -> case x > z of
                      True -> Inside "f2" <l2>
                          (Inside "f1" <l1> (BAD "f1"))
                      False -> ...
```

This residual fragment enables us to give one counter-example with the following meaningful message at compile-time:

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

where `<l3>` is a pseudo symbol which indicates the location of the definition of `f3` in the source file.

Simplification rules related to `Inside` follow directly from the transition rules for `Inside`, the details can be found in [21].

8. Implementation and Challenging Examples

We have implemented a prototype system based on the ideas described in previous sections and experimented with various examples. The checking time for each of them is within a second or a few seconds. Besides the ability to check pre/postconditions involving *recursive predicates* and predicates involving *higher-order functions*, here, we present a few more challenging examples which can be classified into the following categories.

8.1 Sorting

As our approach gives the flexibility of asserting properties about components of a data structure, it can verify sorting algorithms. Here we give examples on list sorting. In general, our system should be able to verify sorting algorithms for other kinds of data structures, provided that appropriate predicates are given.

```
sorted [] = True
sorted (x:[]) = True
sorted (x:y:xs) = x <= y && sorted (y : xs)

insert i xs @ ensures { sorted xs ==> sorted $res }
insert item []  = [item]
insert item (h:t) = case item <= h of
                      True -> cons item (cons h t)
                      False -> cons h (insert item t)
```

```
insertsort xs @ ensures { sorted $res }
insertsort []    = []
insertsort (h:t) = insert h (insertsort t)
```

Other sorting algorithms that can be successfully checked include `mergesort` and `bubblesort` whose definitions and corresponding annotations are shown in [21].

8.2 Nested Recursion

The McCarthy's `f91` function always returns 91 when its given input is less than or equal to 101. We can specify this by the following pre/post annotations that can be automatically checked.

```
f91 n @ requires { n <= 101 }
f91 n @ ensures { $res == 91 }
f91 n = case (n <= 100) of
          True -> f91 (f91 (n + 11))
          False -> n - 10
```

This example shows how pre/post conditions can be exploited to give succinct and precise abstraction for functions with complex recursion.

8.3 Quasi-Inference

Our checking algorithm sometimes can verify a function without programmer supplying specifications. This can be done with the help of the counter-example guided unrolling technique. While the utility of unrolling may be apparent for non-recursive functions, our technique is also useful for recursive functions. Let us examine a recursive function named `risers` [15] which takes a list and breaks it into sublists that are sorted. For example, `risers [1,4,2,5,6,3,7]` gives `[[1,4],[2,5,6],[3,7]]`. The key property of `risers` is that when it takes a non-empty list, it returns a non-empty list. Based on this property, the calls to both `head` and `tail` (with the non-empty list arguments) can be guaranteed not to crash. We can automatically exploit this property by using counter-example guided unrolling without the need to provide pre/post annotations for the `risers` function. Consider:

```
risers [] = []
risers [x] = [[x]]
risers (x:y:etc) =
   let ss = risers (y : etc)
   in case x <= y of
       True -> (x : (head ss)) : (tail ss)
       False -> ([x]) : ss

head (s:ss) = s
tail (s:ss) = ss
```

By assuming `risers.pre == True` for its precondition, we can define the following symbolic checking code for `risers`, namely:

```
risersChk =
 case xs of
 [] -> []
 [x] -> [[x]]
 (x:y:etc) -> let ss = (OK risers) (y : etc)
              in case x <= y of
                 True -> (x:(head_1 ss)):(tail_1 ss)
                 False -> ([x]):ss
```

We use the label `_i` to indicate different calls to `head` and `tail`. As the pattern-matching for the parameter of `risers` is exhaustive and the recursive call will not crash, what we need to prove is that the function calls `(head_1 ss)` and `(tail_1 ss)` will not crash. Here, we only show the key part of the checking process due to space limitation. Unrolling the call `(head_1 ((OK risers) (y:etc)))` gives:

```
case (case (y:etc) of
      [] -> []
      [x'] -> [[x']]
      (x':y':etc')->let ss' = (OK risers) (y':etc')
                 in case x' <= y' of
                    True ->(x':((OK head_2) ss')):
                              ((OK tail_2) ss')
                    False -> [x']:ss') of
 [] -> BAD "risers"
 (z:zs) -> x:z:zs
```

The branch `[]->[]` will be removed by the simplifier according to the rule *match* because `[]` does not match the pattern `(y:etc)`. For the rest of the branches, each of them returns a non-empty list. This information is sufficient for our simplifier to assert that `ss` is non-empty. Thus, the calls `(head_1 ss)` and `(tail_1 ss)` are safe from pattern-matching failure. Note that when we unroll a function call wrapped with `OK` (e.g. `OK risers`), we push `OK` to all function calls in the unrolled definition by a function named `pushOK` which is defined in Appendix B.3. This is why `head_2` and `tail_2` are wrapped with `OK`.

In essence, our system checks whether `True` is the precondition of a function when no annotation is supplied from programmers. We refer to this simple technique as quasi-inference. Note that we do not claim that we can infer pre/postconditions for arbitrary functions, which is an undecidable problem, in general.

9. Related Work

In an inspiring piece of work [9, 8], Flanagan et al, showed the feasibility of applying an extended static checker (named ESC/Java) to Java. Since then, several other similar systems have been further developed, including Spec#'s and its automatic verifier Boogie [3] that is applicable to the C# language. We adopt the same idea of allowing programmers to specify properties about each function (in the Haskell language) with pre/post annotations, but also allow pre/post annotations to be selectively omitted where desired. Furthermore, unlike previous approaches based on verification condition (VC) generation which rely solely on a theorem prover to verify, we use an approach based on symbolic evaluation that can better capture the intended semantics of a more advanced lazy functional language. With this, our reliance on the use of theorem provers is limited to smaller fragments that involve the arithmetical parts of expressions. Symbolic evaluation gives us much better control over the process of the verification where we have customised sound and effective simplification rules that are augmented with counter-example guided unrolling. More importantly, we are able to handle specifications involving recursive functions and/or higher-order functions which are not supported by either ESC/Java or Spec#.

In the functional language community, type systems have played significant roles in guaranteeing better software safety. Advanced type systems, such as dependent types, have been advocated to capture stronger properties. While full dependent type system (such as Cayenne [1]) is undecidable in general, Xi and Pfenning [20] have designed a smaller fragment based on indexed objects drawn from a constraint domain $\mathcal{C}$ whose decidability closely follows that of the constraint domain. Typical examples of objects in $\mathcal{C}$ include linear inequalities over integers, boolean constraints, or finite sets. In a more recent Omega project [18], Sheard shows how extensible kinds can be built to provide a more expressive dependent-style system. In comparison, our approach is much more expressive and programmer friendly as we allow arbitrary functions to be used in the pre/post annotations without the need to encode them as types. It is also easier for programmers to add properties incrementally. Moreover, our symbolic evaluation is formulated to adhere to lazy semantics and is guaranteed to terminate when code safety is detected or when a preset bound on the unrollings of each recursive function is reached.

Counter-example guided heuristics have been used in many projects (in which we can only cite a few) [2, 10] primarily for abstraction refinement. To the best of our knowledge, this is the first time it is used to guide unrolling which is different from abstraction refinement.

In [12], a compositional assertion checking framework has been proposed with a set of logical rules for handling higher order functions. Their assertion checking technique is primarily for postcondition checking and is currently used for manual proofs. Apart from our focus on automatic verification, we also support precondition checking that seems not to be addressed in [12].

Contracts checking for higher-order functional programs have been advocated in [7, 11]. However, their work is based on dynamic assertions that are applied at run-time, while ours is on static checking to find potential bugs at compile-time.

Amongst the Haskell community, there have been several works that are aimed at providing high assurance software through validation (testing) [4], program verification [13] or a combination of the two [6]. Our work is based on program verification. Compared to the Programatica project which attempts to define a P-Logic for verifying Haskell programs, we use Haskell itself as the specification language and rely on sound symbolic evaluation for its reasoning. Our approach eliminates the effort of inventing and learning a new logic together with its theorem prover. Furthermore, our verification approach does not conflict with the validation assisted approach used by [4, 6] and can play complementary roles.

10. Conclusion and Future Work

We have presented an extended static checker for an advanced functional programming language, Haskell. With ESC/Haskell, more bugs can be detected at compile-time. We have demonstrated via examples the expressiveness of the specification language and highlighted the effectiveness of our verification techniques. Apart from the fact that ESC/Haskell is good at finding bugs, it also has good potential for optimisation to remove redundant runtime tests and unreachable dead code.

Our system is designed mainly for checking pattern matching failures as well as other potential bugs. Being able to verify the postcondition of a function is also for the goal of detecting more bugs at the call sites of the function.

Our extended static checking is sound as our symbolic evaluation follows closely the semantics of Haskell. We have proven the soundness of each simplification rule and given a proof of the soundness of pre/postcondition checking in the technical report [21].

In the near future, we shall extend our methodology to accommodate parametric polymorphism. That means to extend the language $\mathcal{H}$ to GHC Core Language [19] which the full Haskell (including type classes, IO Monad, etc) can be transformed to. We plan to integrate it into the Glasgow Haskell Compiler and test it on large programs so as to confirm its scalability and usefulness for dealing with real life programs.

Acknowledgments

I would like to thank my advisor Simon Peyton Jones for spending tremendous time in discussing the detailed design of the ESC/Haskell system. I would also like to thank Koen Claessen and John Hughes for their earlier discussions and Byron Cook for teaching me counter-example guided abstraction refinement. I greatly appreciate the careful comments and valuable feedback from my advisor Alan Mycroft and the anonymous referees. This work was partially supported by a studentship from Microsoft Research, Cambridge.

References

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

[2] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

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

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

[5] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

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

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

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

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

[10] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. *Automata, Languages and Programming: 30th International Colloquium, (ICALP03)*, 2719:886–902, 2003.

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

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

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

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

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

[16] Andrew Moran and David Sands. Improvement in a lazy context: an operational theory for call-by-need. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–56, New York, NY, USA, 1999. ACM Press.

[17] Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc European Symposium on Programming (ESOP)*, pages 18–44, 1996.

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

[19] The GHC Team. *The Glasgow Haskell Compiler User's Guide*. www.haskell.org/ghc/documentation.html, 1998.

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

[21] Dana N. Xu. Extended static checking for Haskell - technical report. *http://www.cl.cam.ac.uk/users/nx200/research/escH-tr.ps*, 2006.

A. Free Variables

```
fv :: Exp → [Var]
fv(BAD lbl)                   =  ∅
fv(UNR)                       =  ∅
fv(OK e)                      =  ∅
fv(Inside lbl loc (e))        =  fv(e)
fv(λx.e)                      =  fv(e) − {x}
fv(e1 e2)                     =  fv(e1) ∪ fv(e2)
fv(case e0 {ci x⃗i → ei})      =  fv(e0) ∪ ⋃_{i=0}^{n} (fv(ei) − x⃗i)
fv(let x = e1 in e2)          =  fv(e1) ∪ fv(e2) − {x}
fv(C e1 ... en)               =  ⋃_{i=0}^{n} fv(ei)
fv(x)                         =  {x}
fv(n)                         =  ∅
```

B. Auxiliary Functions

The two auxiliary functions `noBAD` and `slice` are combined into one algorithm in our real implementation. But for the clarity of presentation, we leave them as two separate functions.

B.1 A Totally Safe Expression

The function `noBAD` checks syntactically the existence of `BAD` in an expression. So when it encounters a free variable (i.e. a variable not in ρ) which may refer to `BAD` in the heap, in such case, it simply return $False$. However, for an application wrapped with `OK`, it returns $True$ by the semantics of `OK`.

```
noBAD :: Exp → Bool
noBAD e = noBAD' e [ ]

noBAD' :: Exp → [Var] → Bool
noBAD' (BAD lbl) ρ            =  False
noBAD' (v) ρ                  =  v ∈ ρ
noBAD' (n) ρ                  =  True
noBAD' (OK e) ρ               =  True
noBAD' (e1 e2) ρ              =  noBAD' e1 ρ &&
                                 noBAD' e2 ρ
noBAD' (λx.e) ρ               =  noBAD' e (x:ρ)
noBAD' (C e⃗) ρ                =  and (map noBAD' e⃗ ρ))
noBAD' (case e0 of alts) ρ    =  noBAD' e0 ρ &&
        and (map (λ(C x⃗ e) → noBAD e (x⃗ ++ρ)) alts)
noBAD' (let x = e1 in e2)     =  let ρ' = x : ρ
                                 in noBAD' e1 ρ &&
                                    noBAD' e2 ρ'
noBAD' (Inside n e)           =  noBAD' e ρ
noBAD' (NoInline e)           =  noBAD' e ρ
```

B.2 An Algorithm for Slicing

The expression slicing is always done after the simplification of the expression. During the simplification process, all `let` bindings are inlined so we do not need to deal with let-expression during slicing.

```
slice :: Exp → Exp
slice (BAD lbl)          =  BAD
slice (OK e)             =  UNR
slice (n)                =  UNR
slice (v)                =  v
slice (e1 e2)            =  (e1 e2)
slice (λx.e)             =  let s = λx.(slice e)
                            in case s of
                                UNR → UNR
                                _ → s
slice (C e⃗)              =  let s = (map slice e⃗))
                            in if all (map (== UNR) s)
                                then UNR
                                else C s
slice (Inside n e)       =  let s = (slice e)
                            in case s of
                                UNR → UNR
                                _ → Inside n s
slice (case e0 of alts)  =
   case e0 of (filter (λ(C x⃗ e) → slice (e) ≠ UNR) alts)
```

B.3 Unrolling

The function `unroll` takes an expression, two environments as inputs. The environment ρ`#` is a mapping from a function name to its representative function while the environment ρ is a mapping from a function name to its representative function, an its concrete definition. The function `unroll` returns a new expression in which all function calls are unrolled. By all function call, we mean, for example, given a call $(f\ (g\ x))$, the f is unrolled while the g is untouched as $(g\ x)$ is an argument to f. All function calls in arguments are untouched. Remark: as the unrolling is always done after the simplification, we do not encounter a let-expression as an input.

```
unroll :: Exp → [(Name, Exp)]
              → [(Name, Exp)] → Exp
unroll (e1 e2) ρ# ρ          =  ((unroll e1 ρ# ρ) e2)
unroll (v) ρ# ρ              =  ρ#(v)
unroll (OK v) ρ# ρ           =  let ns = map fst ρ
                                in pushOK ρ(v) ns
unroll (NoInline e) ρ# ρ     =  NoInline e
unroll (case e0 of {ci x⃗i → ei}) ρ# ρ =
  case (case (unroll e0 ρ# ρ) of {ci x⃗i → NoInline e0}) of
      {ci x⃗i → unroll ei ρ# ρ}}
unroll (λx.e) ρ# ρ           =  λx.(unroll e)
unroll (C x1..xn) ρ# ρ       =  C (unroll x1)..(unroll xn)
unroll Inside lbl loc e      =  Inside lbl loc (unroll e)
unroll others                =  others
```

The `pushOK` function make sure that if there is any top-level function is called in the input expression, it will indicate the call is safe by wrapping the function with `OK`. So `pushOK` takes an expression and a list of top-level function names and return a new safe expression.

```
pushOK :: Exp → [Name] → Exp
pushOK e ρ                          =  if fv(e) ∉ ρ then e
                                       else pOK e ρ

pOK (e1 e2) ρ                       =  (pOK e1 ρ) e2
pOK v ρ                             =  if v ∈ ρ then OK v
                                       else v
pOK (λx.e) ρ                        =  λx.(pOK e ρ)
pOK (case e0 of {ci x⃗i → ei}) ρ     =
                           case pOK e0 ρ of {ci x⃗i → pOK ei ρ})
pOK (C x1 ... xn) ρ   =   C (pOK x1 ρ) ... (pOK xn ρ)
```