Program errors are common in software systems, including those that are constructed from functional languages. For greater software reliability, such errors should be reported accurately and detected early during program development. Contract checking (both static and dynamic) has been widely used in procedural and object-oriented languages [12, 7, 3, 1]. The difficulty of contract checking in functional languages lies in the use of higher-order functions. However, dynamic checking of contracts for higher-order functions has been studied in [5, 2, 4, 9]. Recently, static pre/postcondition checking [13] as well as hybrid contract checking [6, 11, 10, 8] for functional languages have also been proposed.

In this paper, we combine the idea of the contract semantics [2] and the idea of the static verification through symbolic execution [13] to propose a sound automatic static contract checking framework for a higher-order lazy functional language, Haskell. Consider:

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

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 the runtime system of the Glasgow Haskell Compiler (GHC):

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

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

The programmer’s intention is that \( \text{head} \) should not be called with an empty list. To achieve this, programmers can give a contract to the function \( \text{head} \). Contracts are implemented as pragmas with notation {-# CONTRACT <contract> #-}.

```haskell
{-# CONTRACT head :: {s | not (null s)} -> {r | True} #-}
```

where \( \text{not} \) and \( \text{null} \) are just ordinary Haskell functions.

\[1\] a static contract checking followed by a dynamic contract checking

This places the onus on callers to ensure that the argument to \( \text{head} \) satisfies the expected precondition. With this contract, our compiler would generate the following warning (with 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:

```haskell
f xs = if null xs then 0 else head xs ‘max’ 0
```

With this correction, our compiler will not give any more warning as the precondition of \( \text{head} \) is now fulfilled.

Note that programmers do not need to learn a new language of predicates; instead they can use an arbitrary Haskell expression in a contract including higher-order functions and recursive functions.

Our goal 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 \( \text{error} \)). Divergence (i.e. non-termination) is not a crash. We make the following contributions:

- Compared with the dynamic contract checking work in [5, 2, 9], our system can detect contract violations early at compile-time.
- Compared with the hybrid contract checking [6, 11], we deal with a lazy language instead of a strict one.
- We allow data constructors to be used in constructing contracts so that properties of the subcomponents of a data type can be specified. This is not addressed in [5, 2, 4, 9], but in [9].
- Compared with our earlier ESC/Haskell system [13], we can now
  - detect and locate bugs more precisely by giving contracts to higher order function’s parameters which themselves may be functions;
– reduce false alarms caused by laziness in an efficient way;
– specify pre- and post-condition in a type-like way and allow contract synonyms to be defined easily. For example:

{-# TYPE Ok = {x | True} #-}
{-# TYPE Nat = {x | x >= 0} #-}
{-# TYPE NotNull = {xs | not (null xs)} #-}
{-# CONTRACT head :: NotNull -> Ok #-}

• We develop a concise notation ($<$ and $>$) for describing contract checking, which enjoys many useful properties. Thus equipped, we give a complete proof of the soundness and completeness of dynamic contract checking that takes care of laziness; this proof is much trickier than it looks [14]!

We implement the idea in one branch of the GHC, which can accept full Haskell and also support separate compilation as the verification is modular.

References