Static Contract Checking for Haskell

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

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

Koen Claessen
Chalmers University of Technology
koen@chalmers.se

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[1] 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:

```
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 `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 `head` should not be called with an empty list. To achieve this, programmers can give a contract to the function `head`. Contracts are implemented as pragamas with notation `{-# CONTRACT <contract> #-}`.

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

where `not` and `null` are just ordinary Haskell functions.

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

not :: Bool -> Bool
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 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:

```
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 `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 `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, 6, 11], 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;

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

  - 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 ($\triangleright$ and $\triangleleft$) 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

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

[2] Matthias Blume and David McAllester. A sound (and complete) model of contracts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 189–200, New York, NY, USA, 2004.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of jml tools and applications, 2003.

[4] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Functional and Logic Programming*, pages 226–241. Springer Berlin / Heidelberg, 2006.

[5] 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.

[6] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM Press.

[7] 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.

[8] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Eighth Symposium on Trends in Functional Programming*, April 2007.

[9] 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.

[10] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*. Springer-Verlag, April 2007.

[11] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report. *http://sage.soe.ucsc.edu/sage-tr.pdf*, 2006.

[12] 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.

[13] Dana N. Xu. Extended static checking for haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59, New York, NY, USA, 2006. ACM Press.

[14] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Sound Haskell - Draft. *www.cl.cam.ac.uk/users/nx200/research/hvs.ps*, 2007.