Computer Languages, Systems & Structures 000 (2017) 1–19

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Type generic observation of intermediate data structures for debugging lazy functional programs

Maarten Faddegon*, Olaf Chitil

University of Kent, UK

ARTICLE INFO

Article history:
Received 25 May 2016
Revised 23 March 2017
Accepted 1 May 2017
Available online xxx

Keywords:
Tracing
Debugging
Lazy evaluation
Haskell

ABSTRACT

Observing intermediate values helps to understand what is going on when your program runs. For lazy functional languages Gill presented an observation method that preserves the program's semantics. However, users need to define for each type how its values are observed: a laborious task and a mistake can yield misleading observations and even cause non-termination. Here we define how any value can be observed based on the structure of its type by applying generic programming. Furthermore, we present an extension to specify per observation point how much to observe of a value. We discuss how to obtain behaviour dependent on class membership with type-generic programming and with meta programming.

1. Introduction

Many algorithms can be modularized into a generator part that constructs a large number of possible answers and a selector part that selects the right answer.

Consider the Haskell program

```
main = sel (gen 3)
```

where the function `gen` computes intermediate values that are used as inputs for the function `sel`. Code modularised into a generator and a selector is simpler and promotes reuse.

Most programming languages would first run `gen`, store all resulting intermediate values in memory, and finally apply `sel` to the values. However, `gen` might construct many intermediate values, which could require an unacceptable amount of space, possibly not even fitting into memory. Lazy evaluation avoids this problem by interleaving the execution of the generator and the selector. Therefore many programs in lazy languages like Haskell use this modularisation [1].

Programmers make mistakes. When a program misbehaves, the programmer cannot tell from the output whether the selector or the generator part is defective. In this situation the programmer needs to know the otherwise hidden intermediate values produced by the generator part and demanded by the selector part.

To observe these otherwise hidden intermediate values, Gill developed a library called HOOD [2]. The programmer imports the library and just marks in the program certain expressions for *observation*. HOOD leaves the semantics of the program unchanged except that during execution it additionally records the values of all marked expressions. After termination of the program the programmer can inspect these observed values.

* Corresponding author.
E-mail addresses: mf357@kent.ac.uk (M. Faddegon), oc@kent.ac.uk (O. Chitil).

http://dx.doi.org/10.1016/j.cl.2017.05.001
1477-8424/

However, *how* HOOD observes intermediate values has to be stated for each type in a specific definition. Because the HOOD library comes with many of these definitions, observing values of common types works well. In contrast, when users define their own data types, they also need to define how these shall be observed. There are two reasons why it is undesirable that users define observation for types:

1. Writing these definitions is a laborious task. It also requires some otherwise unnecessary knowledge of how HOOD works. Thus HOOD is less accessible.
2. The programmer may make a mistake in writing these definitions. Such a mistake can yield misleading observations. Changing the strictness of observation functions can even cause non-termination.

While we worked on generic definitions for observing values, we realised that HOOD lacks a mechanism for *not* observing values of a certain type. Partial observation can be beneficial for two reasons:

1. When observing a value that is a large data structure, the programmer may not need to see many of its components; these superfluous components may even clutter the formatted output such that it becomes harder for the programmer to find the information that they are looking for.
2. Fully observing a value of a parameterised type requires the addition of class predicates to the type of the function in which the observation is made. The change then may require further type changes or additional instance declarations wherever the function is called. If such changes are required beyond the boundaries of one module, then they are too intrusive and thus practically infeasible.

In this paper we make the following main contributions:

- We extend Launchbury's semantics for lazy evaluation [3] to describe precisely how HOOD works (Section 3).
- We design an extension of HOOD that generates for any type, based on the structure of this type, definitions for observing values of the type. Our implementation is based on an established type generic programming framework, the Generic Deriving Mechanism GDM (Section 5).
- We explain why users of a library like HOOD also need partial observations (Section 6).
- We design a second extension of HOOD that also allows partial observations. To provide the additional flexibility this implementation is based on meta-programming, namely Template Haskell (Section 7).

Our second extension provides all the features of the first one plus partial observations. However, it has a disadvantage: Type generic programming is type-safe after compiling our library with the type generic definitions; meta-programming is type-safe after compiling the program that uses the meta-language annotations. Therefore typechecking our meta-programming code gives no guarantee that correct code is produced under all circumstances. An error will be caught when the user of our library typechecks their code, but this is a much weaker guarantee compared to using a type generic programming framework [4].

We developed Hoed, an improved version of the HOOD tracing library that includes both our extensions. Hoed is available on Hackage[1] and can be installed with `cabal install Hoed`. Some of the features described in this paper have also been merged into HOOD version 0.3.

This is an extended and substantially modified version of our earlier paper on type generic observing [5]. To clarify how HOOD works, we also include definitions from our paper on using HOOD traces for algorithmic debugging [6].

We start by demonstrating how HOOD is used (Section 2) and how value observations have to be defined for each type (Section 4).

2. Finding a defect with value observation tracing

The generator function `mkTreeRat` from Fig. 1 constructs a sorted infinite binary tree of rational numbers (the Stern-Brocot tree) and the selector function `toRational` uses the tree for finding the rational number that represents a floating point number. For completeness my `TreeRat` type also has a `Leaf` constructor, even though the tree we define is infinite and has no leafs. Floating point number 0.75 can for example be represented by the rational number $\frac{3}{4}$ (for which we use the notation `3 :% 4` in the program). However, when we run the program it prints the unexpected result `1 :% 2` instead. How do we find out if the defect is in the definition of the generator function or in the definition of the selector function?

If we inspected the intermediate values that are used as inputs for the selector function, then we could see whether these are right (the defect is in the selector) or wrong (the defect is in the generator).

In a naive attempt to inspect the intermediate values, we derive an instance of the class `Show` for the `TreeRat` type and try to reveal the intermediate values with

```
main = print (mkTreeRat 0 1 1 0)
```

Unsurprisingly, `print` tries to `show` all values in the infinite tree constructed by `mkTreeRat` and hence this program never terminates.

[1] https://hackage.haskell.org/package/Hoed.

Please cite this article as: M. Faddegon, O. Chitil, Type generic observation of intermediate data structures for debugging lazy functional programs, Computer Languages, Systems & Structures (2017), http://dx.doi.org/10.1016/j.cl.2017.05.001