Asymptotic Genetic Improvement Programming via Type Functors and Catamorphisms

Zoltan A. Kocsis and Jerry Swan
Computing Science and Mathematics, University of Stirling.
{zak,jsw}@cs.stir.ac.uk

1 Motivation

Genetic Improvement Programming (GIP) is an increasingly important technique for software maintenance. It employs Genetic Programming (GP) to optimize human-generated source code for a variety of functional and non-functional properties [1]. For example, it has been successfully used to improve runtime performance [2] of a 50,000 line C++ program. As observed in [3], exploring the space of datatypes is likely to be useful, since it allows (for example) memory to be traded for execution speed. We present a novel semantics-preserving point mutation that operates on algebraic data types (ADTs). The mutation operator replaces a source data structure with a target data structure that is functionally-equivalent but which can have asymptotically superior performance. An ADT [4] is comprised (possibly recursively) of a choice of constructors (denoted by sum), tupling (denoted by product) and higher-order function types (denoted by exponentiation). A simple example of an ADT is the singly-linked list, having two constructors, `Nil` for the empty list and `Cons` for the combination of a new element with a list to create a new list. In Scala this is expressed as `CList` as shown in Fig. 1. The method we employ is general and can in principle be applied to most ADTs.

2 Method

The method of datatype substitution is as follows: we obtain a *type functor* (described in more detail below) [4] corresponding to the source datatype S. Using the type functor, we then obtain a mapping between S and the target datatype T via proof search within the framework of *sequent calculus* [5]. This proof yields a 'wrapper function' $f : S \to T$ which allows all instances of the source datatype s to be replaced by instances of the target datatype $f(s)$ [6]. The type functor for an ADT is built up from constants, products and sums in the manner described above by factoring out the recursion [4]. In this manner, it can be shown that the type functor associated with the `CList` type defined above is $F(r) = 1 + Int * r$.

3 Experiment

We describe how the method can be used to automatically replace `CList` with a more efficient alternative. The *difference list* [7] is a functional representation

```
sealed trait CList
case class Nil extends CList
case class Cons( x: Int, xs: CList ) extends CList
sealed case class DList = DList(f : CList => CList)
def fromCListToDList(x : CList) : DList = x match {
  case x : Nil => DList((b : CList) => b))
  case Cons(c,d) => DList(b => Cons(c,fromList(d).f(b)))
}
```

Listing 1: Example Scala ADTs and a synthesized mapping between them

of a list that enables concatenation as an $\mathcal{O}(1)$ operation (as opposed to $\mathcal{O}(n)$ for `CList`). The Scala signature for a difference list `DList` is given in the listing above. By extending the approach of Djinn [8] to support a bigger class of types (via catamorphisms) [9], we were able to use proof search to obtain mappings between lists and difference lists. By the Curry-Howard Isomorphism [5], these mappings are expressible as code that can be dynamically generated by Scala's native reflection mechanism in the manner of [3]. Using reflection, we can search for all ADTs (expressed as a fixed hierarchy of **`case`** classes, as in Fig. 1) and determine if a semantics-preserving transformation between them is possible. The resulting synthesised code (`fromCListToDList`) appears at the bottom of Listing 1, and the corresponding point mutation replaces an AST node containing a `CList c` with the invocation `fromCListToDList(c)`. This mutation process will be described in full detail at the workshop.

References

[1] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *ASE*, pages 1–14, 2012.

[2] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 2013. Accepted.

[3] Jerry Swan, Michael G. Epitropakis, and John R. Woodward. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report CSM-195, University of Stirling, January 2014.

[4] Richard S. Bird and Oege de Moor. *Algebra of programming.* Prentice Hall International series in computer science. Prentice Hall, 1997.

[5] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.

[6] Andy Gill and Graham Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[7] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[8] Lennart Augustsson. Djinn, a theorem prover in Haskell, for Haskell. http://hackage.haskell.org/package/djinn, 2005.

[9] Graham Hutton, Mauro Jaskelioff, and Andy Gill. Factorising Folds for Faster Functions. *Journal of Functional Programming Special Issue on Generic Programming*, 20(3&4):353–373, June 2010.