Haskell Programming with Nested Types: A Principled Approach†

Patricia Johann and Neil Ghani‡
({patricia,ng}@cis.strath.ac.uk)
University of Strathclyde, Glasgow, G1 1XH, Scotland

Abstract. Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. For each inductive data type, it provides a Church encoding for that type, a `build` combinator which constructs data of that type, a `fold` combinator which encapsulates structured recursion over data of that type, and a `fold`/`build` rule which optimises modular programs by eliminating from them data constructed using the `build` combinator, and immediately consumed using the `fold` combinator, for that type. It has long been thought that initial algebra semantics is not expressive enough to provide a similar foundation for programming with nested types in Haskell. Specifically, the standard `fold`s derived from initial algebra semantics have been considered too weak to capture commonly occurring patterns of recursion over data of nested types in Haskell, and no `build` combinators or `fold`/`build` rules have until now been defined for nested types. This paper shows that standard `fold`s are, in fact, sufficiently expressive for programming with nested types in Haskell. It also defines `build` combinators and `fold`/`build` fusion rules for nested types. It thus shows how initial algebra semantics provides a principled, expressive, and elegant foundation for programming with nested types in Haskell.

1. Introduction

Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. It provides support for `fold` combinators which encapsulate structured recursion over data structures, thereby making it possible to write, reason about, and transform programs in principled ways. Recently, [15] extended the usual initial algebra semantics for inductive types to support not only standard `fold` combinators, but also Church encodings and `build` combinators for them as well. In addition to being theoretically useful in ensuring that `build` is seen as a fundamental part of the basic infrastructure for programming with inductive types, this development has practical merit: the `fold` and `build` combinators can be used to define, for each inductive type, a `fold`/`build` rule which optimises modular programs by eliminating from them data of that type constructed using its `build` combinator and immediately consumed using its `fold` combinator.

† This is a revised and extended version of the conference paper [23].
‡ Supported in part by EPSRC grant EP/C511964/2.