Lightweight Extensible Records for Haskell

Mark P. Jones
Oregon Graduate Institute
mpj@cse.ogi.edu

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

September 6, 1999

Abstract

Early versions of Haskell provied only a *positional* notation to build and take apart user-defined datatypes. This positional notation is awkward and error-prone when dealing with datatypes that have more than a couple of components, so later versions of Haskell introduced a mechanism for *labeled fields* that allows components to be set and extracted by *name*. While this has been useful in practice, it also has several significant problems; for example, no field name can be used in more than one datatype.

In this paper, we present a concrete proposal for replacing the labeled-field mechanisms of Haskell 98 with a more flexible system of records that avoids the problems mentioned above. With a theoretical foundation in the earlier work of Gaster and Jones, our system offers lightweight, extensible records and a complement of polymorphic operations for manipulating them. On a more concrete level, our proposal is a direct descendent of the Trex implementation ("typed records with extensibility") in Hugs, but freed from the constraints of that setting, where compatibility with Haskell 98 was a major concern.

1 Introduction

Records are one of the basic building blocks for data structures. Like a simple tuple or product, a record gathers together some fixed number of components, each of a potentially different type, into a single unit of data. Unlike a tuple however, the individual components are accessed by *name* rather than *position*. This is particularly important in languages where there is no support for pattern matching, or in cases where the number of different components would make pattern matching unwieldy and error-prone.

Haskell allows programmers to define a wide range of algebraic datatypes, but early versions of the language (up to and including Haskell 1.2) did not provide any support for records. As Haskell became an increasingly attractive platform for general purpose program development, the need for some form of records became more pressing: it is quite common to find data structures with many components in real-world applications, and it is very awkward to deal with such datatypes in a language that supports only the traditional constructor and pattern matching syntax for algebraic datatypes. Motivated by such practical concerns, the mechanisms for defining algebraic datatypes were extended in Haskell 1.3 (and carried over into Haskell 1.4 and Haskell 98) to support *labeled fields*, which allows the components of a datatype to be accessed by name. In essence, the syntax of Haskell was extended so that the definition of a particular algebraic datatype could include not just the names of its constructors, as in earlier versions of the language, but also the names for its selectors. We review the Haskell 98 record system in Section 2.

The Haskell 98 system has the merit of being explicable by translation into a simpler positional notation, and has no impact at all on the type system. However, this simplicity comes at the price of expressiveness, as we discuss in Section 2.1. This paper presents a concrete proposal for an alternative record system, designed to replace the current one (Section 3).

Our proposal is based closely on the extensible records of Gaster and Jones [4]. An implementation of their system has been available for a couple of years, in the form of the "Trex" extension to Hugs, so quite a bit of experience has accumulated of the advantages and shortcomings of the system. Trex is designed to be compatible with Haskell 98, and that in turn leads to some notational clumsiness. This proposal instead takes a fresh look at the whole language.

The resulting design is incompatible with Haskell 98 in minor but pervasive ways; most Haskell 98 programs would require modification. However, we regard this as a price worth paying in exchange for a coherent overall design. We review the differences between our proposal and Trex, Gaster's design, and other system in Section 4.

The paper contains essentially no new technical material; the design issues are mainly notational. So why have we written it? Firstly, there seems to be a consensus that some kind of full-blown record system is desirable, but debate is hampered by the lack of a concrete proposal to serve as a basis for discussion. We hope that this paper may serve that role. Second, it is not easy to see what the *disadvantages* of a new feature might be until a concrete design is presented. Our system does have disadvantages, as we discuss in Section 5. We hope that articulating these problems may spark off some new ideas.