Exploring the Regular Tree Types

Peter Morris, Thorsten Altenkirch, and Conor McBride

School of Computer Science and Information Technology,
University of Nottingham

Abstract. In this paper we use the Epigram language to define the universe of regular tree types—closed under empty, unit, sum, product and least fixpoint. We then present a generic decision procedure for Epigram's in-built equality at each type, taking a complementary approach to that of Benke, Dybjer and Jansson [7]. We also give a generic definition of map, taking our inspiration from Jansson and Jeuring [21]. Finally, we equip the regular universe with the partial derivative which can be interpreted functionally as Huet's notion of 'zipper', as suggested by McBride in [27] and implemented (without the fixpoint case) in Generic Haskell by Hinze, Jeuring and Löh [18]. We aim to show through these examples that generic programming can be ordinary programming in a dependently typed language.

1 Introduction

This paper is about generic programming [6] in the dependently typed functional language Epigram [29, 30]. Generic programming allows programmers to explain how a single algorithm can be instantiated for a variety of datatypes, by computation over each datatype's structure. In particular, we construct the universe of regular tree types—the datatypes closed under empty, unit, sum, product and least fixpoint. We define a de Bruijn indexed syntax [14] for these types, but we do not interpret this syntax via a recursive function: rather we give the elements for a given type as an *inductive family* [16]. It is Epigram's support for dependent pattern matching [13] which makes this approach practicable.

The universe of regular tree types is small compared to others we might imagine [4, 7], but it is rich in structure. We exploit some of that structure in our programs: Epigram's standard equality is decidable for every regular tree type; every regular tree type constructor has a notion of functorial 'map'; we also give the formal derivative of each type expression, including fixpoints, and acquire the related notion of one-hole context or 'zipper' [20]. In the last example McBride's observation [27], given its explanation in [3], has finally become a program.

1.1 What Is a Universe?

The notion of a *universe* in Type Theory was introduced by Per Martin-Löf [26, 34] as a means to abstract over specific collections of types. A universe is given by a type $U : \star$ of *codes* representing just the types in the collection, and a function $T : U \to \star$ which interprets each code as a type. A standard example

J.-C. Filliâtre et al. (Eds.): TYPES 2004, LNCS 3839, pp. 252–267, 2006.

is a universe of *finite* types—each type may be coded by a natural number representing its size. We can declare the natural numbers in Epigram as follows

$$\underline{\text{data}}\ \frac{}{\mathsf{Nat} : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{zero} : \mathsf{Nat}}\ \frac{n : \mathsf{Nat}}{\mathsf{suc}\ n : \mathsf{Nat}}$$

One way to interpret each Nat as a finite type is to write a recursive function which calculates a type of the right size, given an empty type Zero, a unit type One and disjoint unions $S + T$

$$\begin{array}{l} \underline{\text{let}}\ \dfrac{n : \mathsf{Nat}}{\mathbf{fin}\ n : \star}\quad \mathbf{fin}\ n \Leftarrow \underline{\text{rec}}\ n \\ \qquad\qquad\quad \mathbf{fin}\ \ \mathsf{zero} \Rightarrow \mathsf{Zero} \\ \qquad\qquad\quad \mathbf{fin}\ (\mathsf{suc}\ n) \Rightarrow \mathsf{One} + \mathbf{fin}\ n \end{array}$$

Another way is to define directly an *inductive family* [16] of finite types:

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{fz} : \mathsf{Fin}\ (\mathsf{suc}\ n)}\ \frac{i : \mathsf{Fin}\ n}{\mathsf{fs}\ i : \mathsf{Fin}\ (\mathsf{suc}\ n)}$$

Fin n gives a coding of the set $\{0, \ldots, n-1\}$. Fin zero is uninhabited because no constructor targets it; Fin (suc n) has one more element than Fin n. Below we tabulate the first few types in this family: we show the 'n' arguments to fz and fs—usually left implicit—as subscripts, and write in decimal to save space.

Fin 0	Fin 1	Fin 2	Fin 3	Fin 4	⋯
	fz_0	fz_1	fz_2	fz_3	⋯
		$\mathsf{fs}_1\ \mathsf{fz}_0$	$\mathsf{fs}_2\ \mathsf{fz}_1$	$\mathsf{fs}_3\ \mathsf{fz}_2$	⋱
			$\mathsf{fs}_2\ (\mathsf{fs}_1\ \mathsf{fz}_0)$	$\mathsf{fs}_3\ (\mathsf{fs}_2\ \mathsf{fz}_1)$	⋱
				$\mathsf{fs}_3\ (\mathsf{fs}_2\ (\mathsf{fs}_1\ \mathsf{fz}_0))$	⋱
					⋱

In either presentation, Nat acts as a syntax for the finite types which we then equip with a semantics via **fin** or Fin. Let us emphasize that Nat is an ordinary datatype, and hence operations such as **plus** can be used to equip the finite universe with structure: Fin (**plus** $m\ n$) is isomorphic to Fin m + Fin n. Universe constructions express generic programming for collections of datatypes [6, 18, 21] in terms of ordinary programming with their codes.

The notion of universe brings an extra degree of freedom and of precision to the business of generic programming. By their nature compiler extensions such as Generic Haskell [10] support the extension of generic operations to the whole of Haskell's type system, but we are free to construct a continuum, from large universes which support basic functionality to small, highly structured universes which support highly advanced functionality. Benke, Dybjer and Jansson provide a good introduction to this continuum in [7]. In fact every family of types, whether inductive like Fin or computational like **fin**, yields a universe.