Uniform Traversal Combinators: 
Definition, Use and Properties *

Leonidas Fegaras\textsuperscript{1}, Tim Sheard\textsuperscript{2}, and David Stemple\textsuperscript{1}

\textsuperscript{1} Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003. \textit{(email: \{fegaras, stemple\}@cs.umass.edu)}
\textsuperscript{2} Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR 97006. \textit{(email: sheard@cse.ogi.edu)}

Abstract

In this paper we explore ways of capturing well-formed patterns of recursion in the form of generic reductions. These reductions, called \textit{uniform traversal combinators}, can substantially help the theorem proving process by eliminating the need for induction and can also be an aid in achieving effective program synthesis.

1 Introduction

Recursive structures, such as lists and trees, can be defined inductively in most functional languages [6]. The recursive types of these structures can be formalised using axiom sets generated automatically from their type definition, which are basically equivalent to Hoare's axioms for recursive data structures [5]. Programs that operate on instances of these types can be expressed as recursive functions in a pure applicative language. Theorems about these functions can be proved using induction principles on the structure of the parameter types of these functions. The Boyer-Moore theorem prover [3], for example, proves theorems about recursive functions mechanically by using axioms, definitions, and previously proved theorems, along with powerful induction mechanisms on recursive structures. Program synthesis needs techniques similar to those used in theorem proving. However it is more difficult, partially because induction methods cannot be applied directly for synthesizing the recursive definition of a function.

Proving theorems about computations over recursive structures can be made easier by requiring that functions be expressed in stereotyped ways. One kind of stereotyping is the systematic use of higher order functions that carry out all the traversal of recursive structures. Such traversal functions can capture common patterns of recursion that occur often during programming and, therefore, minimize the explicit use of recursion, which now becomes encapsulated by these functions. The well-known map function that applies a function to each element of a list is an example of a higher order function that encapsulates a traversal. By proving theorems about such traversal functions, some properties of functions using them can be proven by shallower reasoning than would be required

* This paper is based on work supported by the National Science Foundation under grants IRI-8606424 and IRI 8822121 and by the Office of Naval Research University Research Initiative contract, number N00014-86-K-0764
if the traversals were not “pre-analyzed” in isolation. In particular, the use of induction proofs can be substantially diminished.

Reductions [12, 11] are convenient abstractions for expressing manipulations of bulk data types represented by recursive structures. They accumulate results as they traverse a structure and can be used for more computations than are expressible using a mapping traversal. Reductions tailored to particular recursive types can be generated automatically by a compiler by examining the type details. Reductions over lists and finite sets are expressive enough to directly simulate all primitive recursive functions [7]. The work reported here extends these reductions to cover a larger set of recursion patterns and is motivated by the desire to use them as an aid both in theorem proving and program synthesis.

In this paper we explore a broad class of traversal functions and prove their fundamental properties. We introduce a family of generic functions, called traversal combinators, that capture a large family of type-safe primitive recursive functions. Most functions expressed as recursive programs where only one parameter becomes smaller at each recursive call are members of this family. This restriction excludes some valid functions, such as structural equalities and ordering, because they require their two input structures to be traversed simultaneously.

Our generic functions are combinators as each takes functions as inputs and return a new function as output (the one that performs the actual reduction). The most important contribution of this paper is our treatment of the class of traversal combinators resulting from restricting their input functions to be themselves traversal combinators. We call these functions uniform traversal combinators. This offers a disciplined and uniform treatment of functions. This uniformity introduces some nice properties, such as these combinators being closed under composition, that aid in theorem proving and program synthesis. In order to prove equality theorems it was necessary to extend our language to include structural equality as a special primitive. Programs expressed in this algebra can be tested for functional equivalence in a systematic and complete way, based on the fact that there is a unique way for expressing a function as a traversal.

Our algebra is at least equivalent to the first order logic. It cannot capture some interesting functions, such as transitive closure and integer exponentiation. Nevertheless, our system can express and prove complex theorems. We envision a system where all theorems expressible in our algebra are proved using the efficient algorithms presented in this paper, while the rest are tested by a theorem prover based on heuristics, such as the Boyer-Moore theorem prover.

2 Related Work

The work reported here is a new method for theorem proving with structural induction. Even though there is some research on analyzing the properties of some highly stereotyped recursions similar to our traversals, there is no work reported on defining a systematic method for applying these properties to theorem proving.

The most influential work on realizing the importance of capturing recursions into a few powerful patterns was by Richard Bird [1]. Even though his work was focused on specific types, such as lists, it suggested ways of extending these methods to other