Assignments for Applicative Languages

Vipin Swarup*  Uday S. Reddy†  Evan Ireland‡
The MITRE Corporation  University of Illinois at Urbana-Champaign  Massey University

Abstract

We propose a theoretical framework for adding assignments and dynamic data to functional languages without violating their semantic properties. This differs from semi-functional languages like Scheme and ML in that values of expressions remain static and side-effect-free. A new form of abstraction called observer is designed to encapsulate state-oriented computation from the remaining purely applicative computation. The type system ensures that observers are combined linearly, allowing an implementation in terms of a global store. The utility of this extension is in manipulating shared dynamic data embedded in data structures. Evaluation of well-typed programs is Church-Rosser. Thus, programs produce the same results whether an eager or lazy evaluation order is used (assuming termination). A simple, sound logic permits reasoning about well-typed programs. The benefits of this work include greater expressive power and efficiency (compared to applicative languages), while retaining simplicity of reasoning.

Keywords  Functional languages, imperative programming, lambda calculus, type systems, strong normalization, Church-Rosser property, referential transparency, continuation-passing style.

1 Introduction

Functional languages are popular among computer scientists because of their strong support of modularity. They possess two powerful glues, higher-order functions and laziness, that permit programs to be modularized in new, useful ways. Hughes [Hug90] convincingly argues that "...lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess. One should not obstruct access to such a vital tool." However, side-effects are incompatible with laziness: programming with them requires knowledge of global context, defeating the very modularity that lazy evaluation is designed to enhance.

* Address: The MITRE Corporation, Burlington Road, Bedford, MA 01730; E-mail: swarup@mitre.org; Supported by NASA grant NAG-I-613 (while at the University of Illinois at Urbana-Champaign).
† Address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; E-mail: reddy@cs.uiuc.edu; Supported by a grant from Motorola Corporation.
‡ Address: School of Information Sciences, Massey University, Palmerston North, New Zealand; E-mail: E.Ireland@massey.ac.nz
Pure functional languages have nice properties that make them easy to reason about. For instance, + is commutative, = is reflexive, and most other familiar mathematical properties hold of the computational operators. This is a consequence of expressions representing static values: values that do not change over time. Thus, an expression's value is independent of the order in which its subexpressions are evaluated. Side-effects are incompatible with these properties, as side-effects change the values of other expressions, making the order of evaluation important.

Assignments are a means of describing dynamic data: data whose values change over time. In their conventional form, assignments have side-effects on their environment, making their order of evaluation important. Not only are such assignments incompatible with laziness, but they also destroy the nice mathematical properties of pure languages. Hence lazy functional languages shun assignments.

However, since assignments directly model the dynamic behavior of a physical computer's store, they yield efficient implementations of dynamic data. In contrast, one models dynamic data in functional languages by representing the state explicitly or, possibly, by creating streams of states. Compilation techniques and language notations have been proposed to permit explicit state manipulation to be implemented efficiently [HB85, GH90, Wad90b, Wad90a]. Unfortunately, these methods do not achieve all the effects of true dynamic data. For instance, dynamic data may be "shared", i.e., embedded in data structures and accessed via different access paths. When shared dynamic data are updated using assignments, the change is visible to all program points that have access to the data. In contrast, when state is being manipulated explicitly, updating shared data involves constructing a fresh copy of the entire data structure in which the data are embedded, and explicitly passing the copy to all program points that need access to the data. This tends to be tedious and error-prone, and results in poor modularity. One particularly faces this difficulty while encoding graph traversal algorithms such as topological sort, unification and the graph reduction execution model of lazy functional languages.

In this paper, we propose a theoretical framework for extending functional languages with dynamic data and assignments while retaining the desirable properties of static values. The resulting language has the following key properties:

- Expressions have static values. State-dependent and state-independent expressions are distinguished via a type system. The former are viewed as functions from states to values and the functions themselves are static. (Such functions are called observers and resemble classical continuations [Sto77, SW74]). The type system ensures that this view can be consistently maintained, and limits the interaction between observers in such a way that expressions do not have side-effects.\(^1\)

- The language is a strict extension of lambda calculus. Function abstraction and application have precisely the same meaning as in

---

\(^1\)In the contemporary functional programming community, the terms "assignment" and "side-effect" are sometimes used synonymously. We use the term "side-effect" in its original meaning: an expression has a side-effect if, in addition to yielding a value, it changes the state in a manner that affects the values of other expressions in the context. Assignments in our proposed language do not have such side-effects. Similar comments apply to terms like "procedure" and "object".