A REDUCTION SEMANTICS FOR IMPERATIVE HIGHER-ORDER LANGUAGES

Matthias Felleisen, Daniel P. Friedman

Computer Science Department, Indiana University, Bloomington, IN 47405, USA

Abstract

Imperative higher-order languages are a highly expressive programming medium. Compared to functional programming languages, they permit the construction of safe and modular programs. However, functional languages have a simple reduction semantics, which makes it easy to evaluate program pieces in parallel. In order to overcome this dilemma, we construct a conservative extension of the λ-calculus that can deal with control and assignment facilities. This calculus simultaneously provides an algebraic reasoning system and an elegant reduction semantics of higher-order imperative languages. We show that the evaluation of applications can still take advantage of parallelism and that the major cost of these evaluations stems from the necessary communication for substitutions. Since this is also true for functional languages, we conjecture that if a successful parallel evaluation scheme for functional languages is possible, then the same strategy will also solve the problem of parallel evaluations for imperative programming languages.

1. Pro Imperative Higher-Order Languages

A programming language is a medium for expressing thoughts about problems and their solutions. This statement is folk wisdom, yet, it has been ignored since programming became an activity performed on real machines. In the beginning, programming languages were considered as command languages for computers. This view grew out of the popular imperative programming languages for the early computing machines and the necessity for maximal utilization of scarce resources.

Another phase in programming language research was determined by the advent of non-von-Neumann computer architectures. The realization was that the traditional way of processing programs had a bottleneck and that this bottleneck should be eliminated in favor of as much parallel processing as possible. But, instead of implementing redesigned traditional languages on these modern machines, new languages were invented. The prevailing opinion was [4, 17] that "increasing performance by exploiting parallelism must go hand in hand with making the programming of these machines easier."¹ This argument, together with a trend for more mathematical languages, ignited interest in applicative languages [1].

Applicative languages are easy to implement on non-traditional reduction architectures. They have a simple operational model based on reduction semantics. However, these languages lack abstractions for expressing evaluation control and state change because these facilities invalidate ordinary reduction semantics and thus complicate parallelization of program evaluations. Programmers must simulate these imperative effects in applicative languages by using and optimizing tricks of denotational semantics, e.g., accumulators, auxiliary functions, or the clumsy passing around of state variables. Again, the burden is borne by the programmer.

Our basic premise is that the language user should not feel any restrictions caused by the underlying implementation machine. We agree that a language must have a clean, mathematical reasoning system, but, we also insist that a language must include control and assignment facilities as fundamental abstractions for expressing evaluation control and state change. They are necessary ingredients for secure, modular, and compositional programming.

The starting point of our development is the λ-calculus [2], more precisely, the λ-value-calculus [20], which is simultaneously a language and a reasoning system. Its advantage is that all objects, including functions, are first-class, i.e., there is no restriction on their use. The programming language contains two additional facilities which preserve this property. One gives full access and control over

¹ [17, p.350], emphasis ours.
the current continuation of an expression; the other abstracts the right to reassign a value to a variable. Programming paradigms such as logic, functional, or object-oriented programming are easily emulated in this language by implementing the respective constructs as syntactic abstractions [3, 13].

Recently [6, 7] we presented two extensions of the λ-calculus which independently model control and assignment facilities. Here we demonstrate that the two extensions can be unified. The new calculus automatically yields a reduction semantics for imperative higher-order languages. The standard reduction strategy of the calculus reveals considerable potential for parallelism in the evaluation of programs. A minor result is the introduction of a new control construct. While simplifying the reductions of the control calculus [6], it syntactically subsumes all traditional control constructs and permits the design of an entirely new class of programs [5].

The emphasis here is on the development of the reduction semantics since it is crucial for the parallel implementation of programs. The reader who is more interested in the proof system is referred to the earlier reports, but the relevant material is repeated here for completeness. In the next section we formalize the syntax and semantics of the programming language and demonstrate with a few examples how some commonly found facilities of other languages are simple syntactic abstractions. The third section contains the transformation of our abstract machine semantics into a program rewriting system. The derivation method is new; some of the intermediate stages are remotely related to Plotkin-style operational semantics [19]. In Section 4 we reformulate the rewriting system as a set of freely applicable notions of reduction. Section 5 addresses implementation issues. An alternative definition of standard evaluation reveals that the reduction system for the programming language offers ample opportunity for evaluating programs in parallel.

2. The Programming Language

The programming language is an idealized version of such imperative higher-order programming languages as ISWIM [14], GEDANKEN [22], and Scheme [23]. Its term set \( \mathcal{A} \) is an extension of the term set \( \Lambda \) of the \( \lambda \)-calculus. The two new kinds of expressions are \( \mathcal{F} \)-applications and \( \sigma \)-abstractions. An \( \mathcal{F} \)-application is of the form \( \mathcal{F} M \) where \( M \) is an arbitrary expression; when evaluated, it applies \( M \) to a functional abstraction of its current continuation, i.e., a functional representation of the rest of the computation. These abstractions have the same status and behavior as functions created by \( \lambda \)-abstractions. The syntax of a \( \sigma \)-abstraction is \( (\sigma x. M) \) for a variable \( x \) and a term \( M \). The abstraction does not bind the variable, but abstracts the right to reassign a value to the variable. When invoked on a value, a \( \sigma \)-abstraction performs the reassignment and then continues to evaluate its body, which yields a result for the entire application. The meaning of the remaining constructs should be adapted accordingly: variables are assignable placeholders for values, abstractions correspond to call-by-value procedures, and applications invoke the result of the function part on the value of the argument part. The syntax is summarized in Definition 2.1.

The sets of free and bound variables of a term \( M \), \( \text{FV}(M) \) and \( \text{BV}(M) \), are defined as usual; the only binding construct in the language is the \( \lambda \)-abstraction. The set of assignable variables in \( M \), \( \text{AV}(M) \), contains all variables that occur in the variable position of a \( \sigma \)-abstraction. Terms with no free variables are called closed terms or programs. To avoid syntactic issues, we adopt Barendregt's \( \alpha \)-congruence convention of identifying \( (\equiv_\alpha \) or just \( \equiv \) terms that are equal modulo some renaming of bound variables and his hygiene convention which says that in a discussion, free variables are assumed to be distinct from bound ones. Substitution is extended in the natural way and we use the notation \( M[x := N] \) to

\[ \text{FV}(M) \]

\[ \text{BV}(M) \]

\[ \text{AV}(M) \]

\[ (\sigma x. M) \]

\[ (\equiv_\alpha \) or just \( \equiv \) terms that are equal modulo some renaming of bound variables and his hygiene convention which says that in a discussion, free variables are assumed to be distinct from bound ones. Substitution is extended in the natural way and we use the notation \( M[x := N] \) to

2 C. Talcott [24] and I. Mason [18] also have designed reasoning systems for control and [first-order] assignment abstractions, respectively. However, their systems are not extensions of \( \lambda \)-calculi, but are equational theories based on rewriting machines similar to the ones we present in Section 3. Neither addresses the issue of reduction systems; work on a unification of the two systems is in progress [personal communication].