Power Domains
Supporting Recursion and Failure

Reinhold Heckmann

FB 14 – Informatik
Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

e-mail: heckmann@cs.uni-sb.de

December 9, 1991

Abstract

Following the program of Moggi, the semantics of a simple non-deterministic functional language with recursion and failure is described by a monad. We show that this monad cannot be any of the known power domain constructions, because they do not handle non-termination properly. Instead, a novel construction is proposed and investigated. It embodies both non-determinism (choice and failure) and possible non-termination caused by recursion.

1 Introduction

Following the proposals of Moggi [Mog89, Mog91b], functional languages with various notions of computations can be denotationally described by means of monads. Monads are constructions mapping domains of values into domains of computations for these values. Computations involving destructive assignments, for instance, are handled by the state transformer monad [Wad90], whereas computations by non-deterministic choice are handled by power domain constructions [Plo76, Smy78, Gun90]. All known power domain constructions and many others are covered by an algebraic theory presented in [Hec90, Hec91].

In this paper, we consider a functional language with non-deterministic choice, failure, and recursion. Surprisingly, none of the known power domain constructions — even those versions that support failure — is suitable to describe the semantics of such a language. The reason is that all constructions described by the theory mentioned above are not well-behaved with respect to non-termination caused by recursion.

Non-determinism and recursion are two independent concepts of computation. Standing alone, non-determinism is properly handled by Plotkin's power domain construction (the variant with 0), and recursion and non-termination is described by the lift monad that adjoins a bottom element. However, in case of a language where both concepts coexist, neither power domain constructions nor lifting nor any simple combination of both provides an appropriate description of the semantics.
The paper is organized as follows: In section 2, we informally discuss several aspects of non-determinism and recursion. In section 3, we briefly introduce the theoretical background: cpo's and continuity in 3.1, monads and strong monads in 3.2, and algebraic theories and free constructions in 3.3.

In section 4, the (well-known) monads are presented that describe recursion and non-determinism independently. They are implicitly characterized as free constructions w.r.t. suitable theories. This suggests to introduce the novel monad that describes both concepts of computation together as free construction w.r.t. a joint theory. In the subsequent sections, we develop several more explicit representations of the new power domains.

In section 5, the novel construction is decomposed into three steps that introduce recursion, choice, and failure in turn. In section 6, the power domains resulting from the novel construction are decomposed into terminating and non-terminating computations. In section 7, we show that the new power domain construction preserves algebraicity, and present a description of the base in the algebraic case.

2 Non-Determinism and Recursion

In this section, we want to discuss informally several aspects of non-determinism and recursion. In 2.1, a non-deterministic functional language is introduced. The monads for non-determinism and recursion are described in 2.2. In section 2.3, we present an informal semantics of our language, which indicates how a formal semantics should look like.

2.1 A Non-Deterministic Functional Language

In [Hec88], we presented a functional language with powerful patterns. They may match in different ways, thus inducing computations with different values. On the other hand, they may fail to match, which leads to a computation resulting in no value at all. An experimental implementation of this language was used to study the effects of non-determinism.

Syntactically, this language is an extension of a simply typed $\lambda$-calculus with base types including int and bool, pair, list, and function types. Non-determinism is introduced implicitly by overlapping or failing patterns, and also explicitly by the choice operator '!' and the constant fail. The actual interpreter of this language behaves similar to a Prolog interpreter: it tries to show all results of an expression to the user.

Let us consider some sample functions:

\[
\text{elem} (H : : T) = H \\
\text{elem} (H : : T) = \text{elem} T
\]

where the operator ' :: ' denotes list construction. If elem is applied to the list [1,2], then both rules in the definition of elem are applicable. Application of the first rule immediately produces the value 1, whereas the second rule induces the recursive call elem[2]. Again, both rules are applicable; the first rule producing 2, and the second rule leading to the recursive call elem[]. In this situation, no rule is applicable, whence the execution fails, i.e., terminates without producing a result. Thus, the interpreter will produce the values 1 and 2, and then terminate. More abstractly, it produces the set of values \{1, 2\}.

The same operation can be specified with explicit choice:

\[
\text{elem} (H : : T) = H \mid \text{elem} T
\]