Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 276 (2011) 191–213

www.elsevier.com/locate/entcs

Capsules and Closures

Jean-Baptiste Jeannin[1]

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA

Abstract

Capsules are a clean representation of the state of a computation in higher-order programming languages with effects. Their intent is to simplify and replace the notion of closure. They naturally provide support for functional and imperative features, including recursion and mutable bindings, and ensure lexical scoping without the use of closures, heaps, stacks or combinators. We present a comparison of the use of closures and capsules in the semantics of higher-order programming languages with effects. In proving soundness of one to the other, we give a precise account of how capsule environments and closure environments relate to each other.

Keywords: Capsule, Closure, Functional Programming, Imperative Programming, State of Computation, Higher-Order Functions, Mutable Variables, Scoping, Programming Language Semantics.

1 Introduction

This paper compares *Capsules* and *Closures*. *Capsules* are a representation of the state of a computation for higher-order functional and imperative languages with effects, and were introduced in [1]. Many authors have studied the state of a computation, for example [2–14]. However, capsules are intended to be as simple as possible, and they correctly capture lexical scoping and handle variable assignment and recursion without any combinators, stacks or heaps, and while keeping everything typable with simple types.

Closures were first introduced by Peter J. Landin along with the SECD machine [13], and first implemented in the programming language Scheme [15]. The early versions of Lisp implemented *dynamic scoping*, which did not follow the semantics of the λ-calculus based on β-reduction. By keeping with each λ-abstraction the environment in which it was declared, thus forming a closure, closures were successful at implementing *static scoping* efficiently.

[1] Email: jeannin@cs.cornell.edu

doi:10.1016/j.entcs.2011.09.022

In [1], capsules are shown to be essentially finite coalgebraic representations of regular closed λ-coterms. Because of recursion and therefore of possible cycles in the environment, the state of computation should be able to represent all finite λ-terms and a subset of the infinite λ-terms, also called λ-coterms. Capsules represent all the regular λ-coterms, and that is enough to model every computation in the language. λ-coterms allow to represent recursive functions directly, without the need for the Y-combinator or recursive types.

The language we introduce is both functional and imperative: it has higher-order functions, but every variable is mutable. This leads to interesting interactions and allows to go further than just enforcing lexical scoping. In particular, what do we expect the result of an expression like ($\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0$) to be? Scheme (using $\mathsf{set!}$ for $:=$) and OCaml (using references) answer 2. Capsules give a rigorous mathematical definition that agrees and conservatively extends the scoping rules of the λ-calculus. Our semantics of closures also agrees with this definition, but this requires introducing a level of indirection, with both an environment and a store, à la ML. Finally, recursive definitions are often implemented using some sort of backpatching; this construction is known as "Landin's knot". We build this directly into the definition of the language by defining $\mathsf{let\ rec}\ x = d\ \mathsf{in}\ e$ as a syntactic sugar for $\mathsf{let}\ x = a\ \mathsf{in}\ x := d; e$, where a is any expression of the appropriate type.

There is much previous work on reasoning about references and local state; see [16–19]. State is typically modeled by some form of heap from which storage locations can be allocated and deallocated [9–12]. Others have used game semantics to reason about local state [20–22]. Mason and Talcott [2–4] and Felleisen and Hieb [5] present a semantics based on a heap and storage locations. A key difference is that Felleisen and Hieb's semantics is based on continuations. Finally, Moggi [8] proposed monads, which can be used to model state and are implemented in Haskell.

This paper is organized as follows. In section 2, we formally introduce a programming language based on the λ-calculus containing both functional and imperative features. In section 3, we describe two semantics for this language, one based on capsules and the other on closures. In section 4, we show a very strong correspondence (Theorem 4.5) between the two semantics, showing that every computation in the semantics of capsules is bisimilar to a computation in the semantics of closures, and vice-versa. In section 5, we show (Propositions 5.1–5.4) that closure semantics retains some unnecessary information that capsule semantics omits, attesting of the simplicity of capsules. We finish with a discussion in section 6.

2 Syntax

2.1 Expressions

Expressions $\mathsf{Exp} = \{d, e, a, b, \ldots\}$ contain both functional and imperative features. There is an unlimited supply of *variables* $x, y, z, \ldots$ of all (simple) types, as well as