Motile Compiler Internals

Michael M. Gorlick
Institute for Software Research
University of California, Irvine
mgorlick@acm.org
2011-June-12

Abstract

Abstract goes here!!!

I. Introduction

Motile is a Scheme dialect targeted toward mobile code in loosely-coupled, multi-agency, distributed systems. Its features include:

- Single-assignment within lexical scope, read-only bindings within global scope
- Persistent functional data structures throughout including lists, tuples, vectors, hash tables, (un)ordered sets, queues, priority heaps, and binding environments
- Network transfer of closures, continuations, binding environments and arbitrary data structures containing such
- Remove evaluation of closures and continuations
- Local and remote spawning of closures and continuations as green threads
- Systematic application of URLs as denotations for remote data structures and threads

We describe here the semantics of binding environments in the execution of Motile programs. Closures and continuations within Motile are fully mobile and may transit from one execution locus A to another execution locus B.[1] Given a closure or continuation λ any variable x bound within lexical scope carries its binding (value) along when λ transits from one execution environment to another. Any variable y in λ not bound within lexical scope must be rebound in the binding environment available at the destination. Thus a closure or continuation λ may begin execution within the context of one binding environment $\mathcal{E}_A$ at A and resume execution, at some later time, within the context of another binding environment $\mathcal{E}_B$ at B. The relationship between $\mathcal{E}_A$ and $\mathcal{E}_B$ may be $\mathcal{E}_A = \mathcal{E}_B$, $\mathcal{E}_A \subset \mathcal{E}_B$, $\mathcal{E}_A \supset \mathcal{E}_B$, or $\mathcal{E}_A \cap \mathcal{E}_B \neq \emptyset$.

Binding environments are a first-class construction within Motile—consequently the Motile compiler (a Scheme-to-Scheme compiler in the spirit of Mobit [**?**]) generates code in continuation-passing, environment-passing, and binding-passing styles (CPS, EPS, and BPS respectively) where both the run-time environment (generated by transitions in and out of lexical scope) and the global binding environment (defining the values of those symbols that are not closed variables) are parameters of evaluation—the equivalent of dedicated registers in an abstract virtual machine. We describe here the primitives available in Motile for manipulating binding environments and supply examples of their application.

Like closures and continuations, binding environments are mobile and may be reified and transmitted from one execution locus to another. In addition, a mobile binding environment may be "hoisted" at its destination B to act as an "extension" to the binding environment already in place at B. To our knowledge, the network transfer of a binding environment and its incorporation into a remote binding environment is a novel capability for mobile code. In this guise mobile binding environments can act as mobile modules, for example, to transfer a domain-specific library from execution locus A to another locus B for use by mobile code arriving from a third party C. As binding environments (modules) are first-class values they may be adapted and (re)generated on need. We take this one step further by reifying a binding environment as a persistent functional data structure[**?**] which simplifies the sharing and modification of binding environments among multiple threads. We also claim, but do not demonstrate or further discuss, that with a little syntactic sugar, these lightweight, first-class dynamic modules, may be applied at compile-time with many of the same features and semantics of other well-known Scheme module systems [**?**].

[1] Many execution loci L_i may reside on the same physical host but each is logically independent of the others. Each L_i is uniquely identified by an URL u_{L_i}.

II. Compiling Lambda Expressions

From the perspective of the Motile/Scheme compiler closures are:

- The fundamental units of execution of the Motile run-time
- The mechanism by which Motile generates host-independent mobile representations of user-defined Motile closures
- The mechanism by which Motile reconstructs user-defined Motile closures from their mobile representations

In other words, the closures of the host Scheme environment constitute the target execution engine of the Motile compiler and for each expression appearing in a Motile source program the Motile compiler generates at least one closure. To simplify mobility generation and mobility reconstruction each closure representing a unit of Motile execution, henceforth termed a *target closure*, is compiled in continuation-passing, environment-passing, and binding-passing style.

Specifically, each target closure $t = (\lambda\ (\mathcal{K}\ \mathcal{S}\ \mathcal{E})\ \ldots)$ accepts exactly three arguments:

- $\mathcal{K}$, a continuation
- $\mathcal{S}$, a stack representing the run-time binding environment, and
- $\mathcal{E}$, the global binding environment

$\mathcal{S}$ contains the values of the defined lexical scope variables in view at any point in time while $\mathcal{E}$ is a map of bindings for those variables that are not closed in lexical scope. The same target closure t may be called (evaluated) with many distinct global binding environments $\mathcal{E}_1, \mathcal{E}_2, \ldots$ over the course of the execution of a Motile program. The values of the three arguments are used to switch between two distinct interpretations, execution and mobility generation:

- If $\mathcal{K}$ is a procedure then $\mathcal{K}$ is a continuation, $\mathcal{S}$ is a host Scheme vector (implementing a Motile run-time binding environment), and $\mathcal{B}$ is a Motile binding environment (implemented as a persistent functional hash table). Evaluating $(t\ \mathcal{K}\ \mathcal{S}\ \mathcal{B})$ (by the host Scheme) implements the semantics of the Motile expression for which t is the target implementation.
- If $\mathcal{K}$ is false then the second two arguments are ignored and the closure returns its in-memory mobile form, a directed (possibly cyclic) graph whose nodes are Motile Assembly Language instructions (detailed in Section III)

If the subject of compilation is a Motile lambda expression $f = (\lambda\ (a_1\ \ldots\ a_m)\ \beta)$, $m \geq 0$ (for example, as a `define` or in a `letrec`) then the Motile compiler must capture the bindings of any closed variables appearing in f, in short implementing a Motile language closure. Let $t_f = (\lambda\ (\mathcal{K}\ \mathcal{S}\ \mathcal{E})\ \ldots)$ be the target closure generated by the Motile compiler for f. When evaluated t_f returns as its value a second closure

$$u_f = (\lambda\ (\mathcal{K}\ \mathcal{E}\ a_1\ \ldots\ a_m)\ \beta_f)$$

where $a_1, \ldots, a_m$ are the arguments of f and β_f is a third closure implementing the lambda body β of f. t_f is the target closure for f at its point of *definition* while u_f is the target closure for f at its point(s) of *application*. In addition, since the value $\mathcal{S}$ is fixed (at the point of definition of f) it is captured, once and for all, in the closure u_f and need not reappear as an argument.[2]

The behavior of u_f is argument-dependent. There are several distinct cases:

- If $\mathcal{K}$ is a procedure then it is taken to be a continuation, $\mathcal{E}$ is a Motile binding environment, and the arguments $a_1, \ldots, a_m$ are u_f-specific. The call implements the semantics of the Motile function f for which u_f is the implementation
- If $\mathcal{F}$

III. Motile Assembly Language

[2] $\mathcal{S}$ is implemented as a pure functional data structure and consequently its capture is, by construction, safe and immutable.