REQUEST-BASED MEDIATED EXECUTION

BY

SAMEER SUNDRESH

B.S., University of Illinois at Urbana-Champaign, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Professor Gul Agha, Chair
Associate Professor Sam Kamin
Associate Professor Grigore Rosu
Carolyn Talcott, SRI International

Table of Contents

Chapter 1

Introduction

It is often useful to be able to inspect and modify *how* part of a program executes, and the environment *in which* it executes. For example, variables and functions allow us to write programs that are abstract in certain values and function definitions. But we can go much further than that. For example, it can be useful to create a domain-specific language (DSL) customized to the task at hand. Rather than writing this DSL from scratch, it is expedient to reuse relevant features from the base language, creating an *embedded* DSL. But on the other hand, in some cases we may also want to remove or redefine the behavior of certain constructs inherited from the base language. One case of this is when writing a custom sandbox for 3rd party mobile code: while we may not need to introduce new constructs, it is important to regulate how the code interacts with the rest of the system. This can be done by mediating access to certain language constructs (such as direct access to memory or other resources), while directly inheriting others (such as arithmetic, conditionals, and functions).

This thesis deals with the problem of prototyping systems consisting of multiple, interacting levels of language dialects. This work focus on *dynamically* customizing the execution semantics of a programming language within a context. The key novelty of the described approach is it allows *any* language construct to be *arbitrarily* and *dynamically* customized within a *bounded scope*. By masking or virtualizing lower-level features and introducing higher-level features, a customization may introduce a new level of abstraction, up to and including a completely different programming language.

1.1 Why customize a language?

Programming languages are a means of communication amongst human software developers and computers. Languages are used to describe solutions to problems; a "good" programming language allows developers to describe solutions in a way that makes sense to themselves, other developers,

and computers. The question of what makes the most sense to people is a problem of psychology, and outside the scope of this thesis. In practice, experienced software developers tend to be skilled at coming up with abstractions that make sense to themselves and each other, but end up having to translate these descriptions to a form that is executable on a computer.

This may be because the developers make use of informal notation that is not defined in an existing programming language, or because they overload existing notation to mean something slightly different. In the latter case, code that *looks* unsafe or incorrect may actually be reasonable under the appropriate non-standard interpretation. The translation from "developer language" to "computer language" can either be done manually, or by defining a domain-specific language implementation. If the translation is purely manual, then possibly unsafe or interfering code which may at best contain bugs that are only revealed at deployment, or at worst contain intentional exploits. As such, it is useful to provide access to language customization both for developers *producing* a particular piece of code, and for developers and advanced users of *systems* that *deploy* that code. The goal of an extensible programming language is to ease the process of teaching a computer how to understand the *composition* of the developers' intended behaviors.

The following sections describe three cases in which an extensible programming language can be useful.

1.1.1 Creating domain-specific languages

The most immediate case of language extension alluded to above is domain-specific languages (DSLs). A domain-specific language introduces special constructs specifically designed for a given problem domain.

For example, TEX and PostScript are domain-specific languages for document preparation, at two different levels of abstraction. Often times, TEX code is compiled down to PostScript. Moreover, packages like PSTricks allow PostScript code to be embedded within a TEXdocument. The drawback of these domain-specific languages is that they require a complete suite of dedicated tools: parsers, translators, interpreters, and so on.

More recently, *embedded* domain-specific languages (EDSLs) have gained prominence. These are libraries within an existing programming language that provide features similar to a DSL, while reusing the host language's type system, variable binding mechanisms and control-flow constructs. In some cases, EDSLs may introduce custom syntax into the host language. The benefit of EDSLs is that since they extend an existing language, developers can reuse existing tools and knowledge of

an existing language. For example, EDSLs for modeling domains such as music and financial instruments have been developed within Haskell.

However, the EDSL also inherits the limitations of the host language. For example, an EDSL embedded in Haskell can contain nonterminating expressions and invoke operations such as `unsafePerformIO`. Unless the host language provides an ability to somehow *supervise* or *mediate* execution of an expression, an EDSL cannot provide the full flexibility of a DSL. This thesis aims to bridge that gap.

1.1.2 Reusing code in a new context

In addition to defining new domain-specific languages, our discussion in Section 1.1 also mentioned overloading the notation of an existing language to mean something slightly different. This can be a useful technique for repurposing an existing codebase in a new context. Aspect-oriented programming is an example of this phenomenon: the execution of a main program triggers additional actions and intercession by a set of *aspects.* Full-system virtualization is another example of reusing existing code in a new context. This thesis considers a technique designed to offer the features of both.

1.1.3 Full-system simulations

A concrete example of reusing code in a new context is full-system simulators. For example, one may have code designed to run on an individual networked sensing device, and wish to simulate the behavior of a network of such devices embedded in a virtual environment. Early sensor network simulators fell into two categories: (a) highly-scalable simulators that operated on *models* of sensor nodes, but could not run native sensor node code; and (b) precise simulators that ran the entire stack of operating system code for each sensor node (either at the C level, or even at the machine code level), and consequently were much less scalable.

The latter group of simulators are an example of reusing existing code (sensor network programs) in a different context (simulation, rather than deployment). Since the host language was not extensible, these simulators had to run the entire OS stack for each node in a separate process. This works because the OS mediates execution of a process. In effect, the OS is a layer of the language runtime model. This thesis considers a way of modeling such multi-level systems within a programming language.

1.1.4 Enhancing security through sandboxing

Another example of reusing code in a new context is running untrusted mobile code, such as a Java applet or a JavaScript on a web page. Obviously, it is critical that the host maintain full control over the execution of untrusted mobile code.

In Java, this has lead to a security model in which code either has access to an operation or cannot perform the operation at all. For example, if a Java applet attempts to read a local file, it receives a security exception. But what if we want to take a Java library originally written to run as trusted local code, and port it to a sandboxed environment? This requires something similar to virtualization, in which attempts to perform trusted operations are intercepted, and possibly serviced in an alternate, secure manner. For example, when a library used by a Java applet attempts to open a file, it may be given access to a virtual file stored in a filesystem archive, or a proxy for a file object stored on the server.

Sandboxing issues are even worse in the JavaScript domain. While individual web pages are isolated from each other based on the same-origin policy, scripts from different sources on the same page all run in the same environment, with access to the entire web page. When JavaScript was introduced, this was not a problem; but with the rise of 3rd-party embeddable widgets on web pages, cross-site scripting errors have become a major concern. Although it is possible to display an embedded widget in a completely separate, isolated web page originating from a different host, this option is often not taken because it prevents all communication between the widget and the host page. It would be useful if JavaScript on a web page could run the code in certain sections of the page under mediation. For example, supervisor code may control a widget's ability to access parts of the HTML document, consume processor and memory resources, and communicate with the outside world. This thesis includes a case study of such an extension to JavaScript.

1.2 In what ways might a language be customized?

Having examined the motives for customizing a programming language, now we turn to specific examples of ways in which a programming language may be customized. We will note the focus areas relevant to this thesis.

1.2.1 User-defined functions and data types

Most programming languages today allow for user-defined functions and data types. Compared to, say, assembly language, this is already quite a bit of extensibility. However, many features are still fixed by the language and not modifiable.

1.2.2 Syntax

The most obvious way in which to customize a language is to introduce special syntactic forms. New syntax simply offers a more convenient way to write down an expression. For example, an infix `_+_` operator which simply calls an `add(_, _)` function. Languages such as Maude and OMeta provide rich support for syntax customization at the parser level. This thesis focuses on dynamic semantics, and hence does not consider how to build extensible parsers.

1.2.3 Compile-time macros

Many programming languages offer static metaprogramming features as a way to customize how expressions are compiled. For example, Lisp/Scheme macros, C++ templates, and Template Haskell all employ a two-stage execution model:

1. parse the input program
2. evaluate compile-time macros on an input abstract syntax tree (AST), resulting in a transformed AST
3. compile or interpret the transformed AST

Macros customize the input language by rewriting input terms to their implementation in the base language. This staged execution model has been extended to multi-stage execution in systems such as MetaOCaml and Jumbo.

This thesis makes use of composable quoted terms, which are often encountered in multi-stage evaluation. The key piece we add is the ability to customize the definions of language constructs available when evaluating a piece of quoted code. In addition to serving as a macro system, this can be useful if different implementations of a given feature make sense at different stages of evaluation.

1.2.4 Static type system

A static type checker allows developers to reject programs that are somehow unsafe, without even executing them. Thus it can be useful to extend a language's type system in order to enforce certain constraints on what programs are considered valid. For example, adding linear types to a functional programming language allows one to perform in-place mutation on data that is not referenced elsewhere in a program's state.

This thesis focuses on the runtime semantics of programs and languages, and does not consider static type systems. The primary reason is because we would have to either restrict the language constructs which can be customized to those that can be properly typed with a given type system. Overcoming this limitation is future work; it would require a type system that is flexible enough to usefully describe the types of arbitrary language constructs, yet ensures new definitions do not compromise the safety of the type system itself.

1.2.5 Primitive operations

Every program is, fundamentally, built out of some primitive components. In a system consisting of a stack of levels of abstraction, the primitives used by some level n are defined by underlying levels $n-1, ...$: level n's primitives may well be complex defined operations at level $n-1$. For example, machine code includes primitive operations such as `add`, `jmp`, `mov` and so on. At the machine code level, extending the set of primitive operations corresponds to extending the processor instruction set. At higher levels, primitive operations include constructors, destructors, and functions on data, as well variable scoping constructs and sequential and concurrent control structures.

This thesis is essentially about defining the constructs used as primitives when executing code in a customized language environment. The simplest kinds of primitives are functions and external effects: they transform input values into output values or observable actions.

1.2.6 Control structures

In addition to primitive functions and effects, programming languages contain various fundamental control structures. These range from simple constructs like if and while, to powerful first-class functions, to more exotic features like first-class (delimited) continuations. The control structures available in a language significantly affect what the programs written in that language look like and how they work. If a base language has only simple, restrictive control structures, adding more powerful control can lead to more concise code. On the other hand, if an existing piece of code

already assumes a wide range of control structures, we must be able to faithfully capture those structures if we wish to create a language that mediates the existing code's execution.
This thesis uses the combination of delimited continuations and explicit call-by-name arguments to model arbitrary control structures within a context. This approach allows us to precisely define how a control structure makes use of each of its subterms, the surrounding "rest of the program," and even allows dynamic customization of the language itself, if so desired.

1.2.7 Variable scoping

Besides control and primitive functions and effects, probably the most important part of a language design is how it deals with variables. The most common variable scoping mechanism today is lexical scoping, but dynamic scoping can also be useful for implicit parameter passing. Even within these basic genres, there are many possible design choices. For example, variables may be read-only or writeable; using a variable within a function may require an explicit annotation for creating a new variable or for inheriting a variable from the parent environment; and these annotations may apply at the function level or the block level. More interestingly, lexical and dynamic variables may be unified in various ways. For example, one might want to treat updates to a given lexical variable as dynamic bindings only visible within a given dynamic scope, rather than imperative updates that are globally visible. This visibility aspect of variable scoping is of particular importance in the presence of first-class continuations and shared-memory concurrency. In this thesis, we address variable scoping by considering access to a variable as an explicit operation, just like primitive functions/effects and control structures. We show how variable binding can thus be defined as a case of language customization: *e.g.,* `set`$(\mathsf{x}, 7)$ has the effect of customizing the language used in the rest of the program so that `get`(x) reduces to 7 in scopes that reference the same variable `x`.

1.2.8 Concurrency semantics

Concurrent execution opens up a wide range of design choices. The two primary concerns are how concurrency is implemented, and how the effects of concurrent threads interact. The two primary implementation choices are interleaving on a single processor and parallel execution on multiple processors. At the programming language implementation level, interleaving can be considered as a layer *on top* of the base language semantics, while parallel execution requires each processor to independently implement the various language constructs. In either case, process and message

scheduling can affect a system's behavior. Different communication mechanisms are appropriate for different uses: for example, shared memory threads vs. shared-nothing processes; synchronous vs. asynchronous messaging; and locks vs. transactions.

The subject of this thesis is not specifically aimed at designing concurrent systems, but it does offer some useful applications. In a design based on interleaving concurrency, redefining a language construct works the same for single-threaded or multi-threaded programs. This is because we offer a level of control that allows interleaving schedulers to be introduced on top of a sequential language. In a parallel design, on the other hand, language customizations need to be injected into the context used by a particular processor, which we consider, as well.

1.3 Thesis outline and reader's guide

The thesis is structured as follows. Chapter 2 introduces a message-oriented execution model, which allows us to treat programs as streams of request messages. For readers with limited time, Chapter 2 and Section 5.2 are recommended; the latter relates our approach to a concrete implementation in Javascript. The rest of the thesis may then be surveyed as interest and time permits. Much of the related work in Chapter 7 is self-contained.

1.3.1 Detailed outline

Within Chapter 2, Section 2.1 covers the core of our approach. Section 2.2 then shows how this model allows us to define and redefine how certain language constructs are implemented within a given scope, while others inherit their pre-existing behavior. Chapter 3 surveys how various common programming language constructs can be defined in terms of the request mediation approach. Here we assume a simple λ-calculus-like base language. (It is important to note that our approach always requires *some* base language in terms of which definitions are written, but this base language can be completely overridden at a higher level. Only the notion of programs as streams of request messages is pervasive in our design.) Section 3.2 considers in depth a number of different ways in which *lexical variables* can be defined in our framework.

Chapter 4 then considers a case study in concurrency: we introduce several layers of language features, starting from a basic interleaving concurrency model, adding asychronous shared-nothing processes (actors), local synchronization constraints within actors, and meta-actors that supervise other actors.

Chapter 5 extends the discussion in Chapter 3 to an implementation of a request-based interpreter for Javascript. All major features of Javascript, including variables, control structures, first-class functions, and objects are covered. Our system adds a request handler mechanism to Javascript, allowing language constructs to be locally redefined. We make use of this mechanism in Chapter 6 to introduce features useful for sandboxing individual widgets within a web page. These features include execution time constraints and mediated communication with the rest of the page and the outside world.

Chapter 7 considers a range of existing work related to this thesis. Our survey includes systems technologies such as virtualization and sandboxing; reflection and metaprogramming in programming languages and actor systems; prominent semantic frameworks for prototyping languages; and the specific techniques of delimited continuations and dynamic binding. Chapter 8 further ventures into a discussion of possible related future work.

Chapter 2

A message-oriented execution model

How can one define a programming language that allows one to model systems consisting of multiple interacting levels of abstraction, where *any* language feature at a given level may be customized by lower levels? It is evident that this requires some sort of reflective programming language. But reflection as found in contemporary languages provides full power to the program at the *top* of the abstraction stack: it is able to inspect and modify the state and behavior of underlying levels. This is at odds with the goal of running sandboxed code under the control of lower levels. As such, this chapter introduces an alternative linguistic framework for mediating program execution. Succeeding chapters show how the primitives of this framework can be used to define a wide range of different language constructs.

2.1 Message-oriented computation

Our approach to defining and extending languages focuses on message-oriented computation. Object-oriented programs are of course the canonical example of message-oriented computation. Objects affect the rest of the world not via direct action, but rather only by sending messages to other objects, requesting that they perform some action. However, there is something lacking from a naive explanation of object-oriented programming: *what is an object? how is a message sent—and received?* These questions are answered by the *semantics* of a programming language. We can consider the relationship between a *program* and the *semantics* of the language in which it's written as another form of message-oriented computation. A program is of course a data structure—usually a tree—representing some algorithm. The nodes of the tree represent various language constructs—functions, if-statements, variables, and so on. Of course, the program itself doesn't define what these nodes mean; that's delegated to the language semantics. Hence to *run* the program, we must continually refer to the semantics to understand what to do next. Or, in object-oriented parlance: if we ask the *program object* to *run,* it must continually send *request*

messages to the *semantics object,* asking it what to do next in order to execute the program. Interpreters and microprocessors are concrete realizations of *semantics objects:* they define a program's dynamic behavior. But many other layers are also involved in defining a program's behavior: run-time systems, operating system kernels, virtual machine monitors, and so on. These layers form a composite *stack* of semantics objects. A particular layer, such as an OS kernel, implements some operations itself (such as a system call), while delegating other operations to a lower-level semantics object (*e.g.,* `add` instructions are implemented directly by the host microprocessor). And when we say that a particular layer implements an operation *by itself,* what we *really* mean is that layer provides a *definition* of the operation in some programming language L. The definition can only actually be *executed* by sending request messages to a lower-level semantics object that *defines* L. For example, a system call handler in an OS kernel is a sequence of machine instructions that are executed by the host microprocessor.

The value in the above way of looking at systems is it allows us to interpose new layers in the stack of semantics objects. Since such a layer can selectively intercept the request messages used to implement higher levels, it can perform *mediation.* But we still haven't defined what a program object or a semantics object *is* or how messages work. That is the subject of the rest of this section.

2.1.1 System architecture

The first thing to pick up from the above discussion is that there are some strict constraints on the system of program objects and semantics objects. We're not just saying *"write an object-oriented program."* The system architecture can be summarized as follows.

A *program object* is simply an abstract tree data structure, with a few primitive methods that cannot be overridden or extended. A program object can only send *synchronous* request messages to its corresponding semantics object. It *can't* communicate directly with other programs or several different semantics objects. On the other hand, a *semantics object* may potentially have several program objects as clients. Semantics objects are (partially) ordered; a semantics object may only send messages to semantics objects at lower levels. In fact, we can be more precise than that: a semantics object is either a *base level,* such as a host microprocessor—a black box that *computes;* or is itself (somehow) defined by a program object, in which case it can *only* send request messages to its corresponding underlying semantics object.

In other words, a system definition is structured as a *tree.* The root is a host platform. All other nodes are programs. Internal nodes are abstraction layers (semantics objects), and the leaves are

application programs. All request messages flow towards the root along the edges of the tree, and every request at an abstract level has a sequence of corresponding requests at the root. If we consider the special case of a *single-threaded* host, the tree collapses to a *list* of the active code at each level of abstraction.

In order to complete our picture of message-oriented computation, we need to define (a) what program objects look like and how they work; and (b) how program objects can be construed to act as semantics objects. The latter is how our system provides *reflection:* a program can intercept and override the behavior of *any* operation used in a higher-level program whose execution it mediates.

2.1.2 Program objects and request messages

As mentioned earlier, programs are generally represented as terms—tree data structures. For the purposes of execution, we need a way to traverse the terms in an *arbitrary* order as determined by the underlying language semantics. Along the way, we must be able to transform a (sub)term representing some (sub)program state into one representing a subsequent (sub)program state specified by the semantics. Let's begin with a simple definition of program terms:

$$\text{term} = \text{name} \times (\text{term list}) \mid \text{value}$$

A term is either a Value which cannot be further reduced, or a *compound term,* which denotes a program that needs to be simplified. Each compound term is tagged with a Name, known as its *head,* indicating *what to do* with the argument terms. For example, the term $+(1, 2)$ *may* mean "add the numbers one and two." But in another context, it could also mean "concatenate the strings '1' and '2'." We can't say exactly what the term really *means* without referring to some language semantics. Without reference to a particular semantics object, a program object is just an abstract syntax tree.

2.1.3 Term traversal

Next, we need to define a general algorithm for traversing and reducing Terms. The idea is we want to evaluate a program by traversing the nodes of its Term in the *correct order* so we can perform reductions according to the language semantics. For example, in an if-statement, we may wish to first evaluate the guard, and then choose to evaluate either the then- or the else-clause—but not both; whereas in an addition expression, we probably want to simplify both arguments before attempting to perform addition. For the time being, let's assume we have a way

```
term      =  name × (term list) | value
↑(·)      :  term → value

message   =  Req  of  name × (term list) × dcont
          |  Ret  of  value
dcont     =  term → term

eval_0    :  term → message
post      :  name
```

Figure 2.1: Terms evaluate to messages. A Req message is sent *before* visiting the subterms of a node. The special post operation can be used to force evaluation of the arguments to a term. A Ret message returns the result of evaluating a term. A dcont, or *delimited continuation* captures the state of the subterm under evaluation, with a hole where the term producing the request message was. The ↑(·) value constructor allows us to use *quoted* terms as values.

to specify and achieve the appropriate traversal order. Then what do we need to be able to do when visiting each node? There are three basic operations:

- reduce the node to a value (and then move on to the next reducible node),
- replace the node with a *different* term (and continue evaluating it), or
- if we haven't already—*visit the node's subterms.*

These three operations also allow us to perform an *arbitrary* traversal of the program tree. The basic idea is that when visiting a term, we can rewrite it to a different term to customize how its subterms are traversed before we begin to traverse those subterms.

Figure 2.1 defines an interface to the term traversal and reduction strategy described above. A program term is *evaluated* by transforming it into a sequence of *messages* corresponding to each node visited in the term, including nodes corresponding to code that is *dynamically injected* by the semantics object. The messages provide the user the ability to inspect, transform, and continue evaluating a term. There are two kinds of messages:

- a Req message is sent *before* visiting a node's subterms,
- a Ret message is sent after the *entire term* has been reduced to a return *value.*

Req messages include a *delimited continuation:*[1] a copy of the current version of the term, with the subterm we are visiting replaced by a *hole.* This allows the user to rewrite the current subterm to another term or a value. The special post operation is provided to force evaluation of some of a

[1]This is a *delimited* continuation because, as we shall see later, it does not include the state of lower levels of abstraction.

Req (if, $\langle \mathsf{true}(), 1(), 2() \rangle$, $\lambda x.x$)
  rewrite term $\mathsf{if}(-,-,-) \to \mathsf{post}(\mathsf{if}^*, -, \mathsf{quote}(-), \mathsf{quote}(-))$
Req (post, $\langle \mathsf{if}^*, \mathsf{true}(), \mathsf{quote}(1()), \mathsf{quote}(2()) \rangle$, $\lambda x.x$)
  traverse subterms (definition of post)
Req (true, $\langle\ \rangle$, $\lambda x.\mathsf{post}(\mathsf{if}^*, x, \mathsf{quote}(1()), \mathsf{quote}(2()))$)
  reduce to value $\top$ *(insert it in the delimited continuation)*
Req (quote, $\langle 1() \rangle$, $\lambda x.\mathsf{post}(\mathsf{if}^*, \top, x, \mathsf{quote}(2()))$)
  reduce to value $\uparrow 1()$
Req (quote, $\langle 2() \rangle$, $\lambda x.\mathsf{post}(\mathsf{if}^*, \top, \uparrow 1(), x)$)
  reduce to value $\uparrow 2()$ *(last argument to post reduced)*
Req (if^*, $\langle \top, \uparrow 1(), \uparrow 2() \rangle$, $\lambda x.x$)
  rewrite to unquoted then-clause term $1()$
Req (1, $\langle\ \rangle$, $\lambda x.x$)
  reduce to value 1
Ret 1

Figure 2.2: Example of how we might evaluate an if-expression via Term traversal and reduction. Control structures generally require a custom traversal order, which is accomplished here by quoting the then- and else-clauses and evaluating the unevaluated then-clause later. Note how the post and if* terms are dynamically injected by the definition of if.

term's subterms. For example, the term $\mathsf{post}(\mathsf{add}, t_1, t_2)$ will evaluate t_1 and t_2 to values v_1 and v_2, respectively (in order, from left to right), and then rewrite to $\mathsf{add}(v_1, v_2)$ (which then further reduces, depending on the definition of add in scope):

$$\mathsf{post}(\mathsf{add}, t_1, t_2) \Rightarrow ... \Rightarrow \mathsf{post}(\mathsf{add}, v_1, t_2) \Rightarrow ... \Rightarrow \mathsf{post}(\mathsf{add}, v_1, v_2) \Rightarrow \mathsf{add}(v_1, v_2) \Rightarrow ...$$

Before we go on, Figure 2.2 shows an example of how we can use the message interface to evaluate a program that requires a custom traversal order. To customize the traversal order, we initially rewrote the if term to a post term in which the then- and else-clauses are *quoted;* later, we rewrote the if* term to the (unquoted) then-clause to continue evaluating it. Notice in particular that the initial rewrite on the if treated the unevaluated subterms *opaquely.* This is important as it allows us to work with program terms that have *any structure.* We know that it's time to evaluate the then-clause by the time the if* node is visited: only after evaluating the subterms does post apply if* to their values. In general, the kind of message (Req or Ret) and the operation name are used to decide what to do when visiting a particular node.

Figure 2.3 describes the message-oriented traversal and reduction algorithm programmatically. This is an executable formalization of the strategy described above. Ignoring the post operation, terms are simply transformed into corresponding Ret or Req messages. The identity delimited

$$
\begin{array}{lcl}
\text{eval}_0\ v & = & \textsf{Ret}\ v \\
\text{eval}_0\ h(t_1, ..., t_n) & = & \textsf{Req}\ (h, \langle t_1, ..., t_n \rangle, \lambda t.t) \quad \text{where } h \neq \textsf{post} \\
\\
\text{eval}_0\ \textsf{post}(h, v_1, ..., v_n) & = & \text{eval}_0\ h(v_1, ..., v_n)
\end{array}
$$

$$
\frac{\text{eval}_0\ t_i = \textsf{Ret}\ v_i}{\text{eval}_0\ \textsf{post}(h, v_1, ..., v_{i-1}, t_i, t_{i+1}, ..., t_n) = \text{eval}_0\ \textsf{post}(h, v_1, ..., v_{i-1}, v_i, t_{i+1}, ..., t_n)} \text{ where } 1 \leq i < n
$$

$$
\frac{\text{eval}_0\ t_i = \textsf{Req}\ (h', args, c)}{\text{eval}_0\ \textsf{post}(h, v_1, ..., v_{i-1}, t_i, ..., t_n) = \textsf{Req}\ (h', args, \lambda t.\textsf{post}(h, v_1, ..., v_{i-1}, c(t), ..., t_n))} \text{ where } 1 \leq i < n
$$

Figure 2.3: eval_0 defines a traversal of terms. Each time a node is visited, eval_0 returns a corresponding message. The post operation in particular defines a left-to-right, post-order traversal: all arguments are evaluated before applying a head h to them.

continuation $\lambda t.t$ mirrors the fact that the top-level term is evaluated first, hence there is no additional context in the continuation. The post operation's behavior is defined by the eval_0 function: it traverses through the arguments $t_1, ..., t_n$ one by one from left to right, dispatching messages to evaluate each in turn. Once all argument subterms have been simplified to values $v_1, ..., v_n$, the head h is applied to the values.

2.1.4 Programs *as* request messages

In Figures 2.1 and 2.3, terms and messages were closely related via the eval_0 function. To simplify matters, we can ignore the distinction between the two, viewing a program *as* the first message in its traversal. A term t is identified with a corresponding message $[[t]]$. The necessary modifications to Figures 2.1 and 2.3 are depicted in Figure 2.4. What we have roughly done is decomposed the eval_0 function from Figure 2.3 into two parts: a mapping $[[\cdot]]$, which transforms a term into a message; and a new "eval" function that defines how request messages that reach the outermost layer of a system are interpreted. The term operation is provided to build up quoted messages corresponding to compound terms, while the eval *operation* unleashes the behavior of a quoted message.

The explicit message constructors in the message notation make it rather verbose and difficult to read. As such, in the following we will use the notation $[[t]]$ for messages which correspond to some term t. Note that not all messages have a corresponding term, since the domain of $[[\cdot]]$ only includes messages with the identity delimited continuation $\lambda m.m$. Request messages which contain a different delimited continuation will be written out explicitly.

$$
\begin{array}{lcl}
\text{term} & = & \text{name} \times (\text{term list}) \mid \text{value} \mid \text{message} \\
[[\cdot]] & : & \text{term} \to \text{message} \\
[[h(t_1, ..., t_n)]] & = & \mathsf{Req}\ (h, \langle [[t_1]], ..., [[t_n]] \rangle, \lambda m.m) \\
[[v]] & = & \mathsf{Ret}\ v \\
[[m]] & = & m \\
 & & \\
\text{message} & = & \mathsf{Req}\ \ \textit{of}\ \ \text{name} \times (\text{message list}) \times \text{dcont} \\
 & \mid & \mathsf{Ret}\ \ \textit{of}\ \ \text{value} \\
\text{dcont} & = & \text{message} \to \text{message} \\
\uparrow(\cdot) & : & \text{message} \to \text{message} \\
\text{eval} & : & \text{message} \to \text{message} \\
\mathsf{term}, \mathsf{eval}, \mathsf{post} & : & \text{name}
\end{array}
$$

$$
\begin{array}{lcl}
\text{eval}\ \mathsf{Req}\ (\mathsf{term}, \langle \mathsf{Ret}\ h, \mathsf{Ret} \uparrow m_1, ..., \mathsf{Ret} \uparrow m_n \rangle, c) & = & \text{eval}\ c(\mathsf{Ret} \uparrow (\mathsf{Req}\ (h, \langle m_1, ..., m_n \rangle, \lambda m.m))) \\
\text{eval}\ \mathsf{Req}\ (\mathsf{eval}, \langle \mathsf{Ret} \uparrow m \rangle, c) & = & \text{eval}\ c(m) \\
\text{eval}\ \mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_n \rangle, c) & = & \text{eval}\ c(\mathsf{Req}\ (h, \langle \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_n \rangle, \lambda m.m))
\end{array}
$$

$$
\frac{\text{eval}\ m_i = \mathsf{Ret}\ v_i}{\begin{array}{l}\text{eval}\ \mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_{i-1}, m_i, m_{i+1}, ..., m_n \rangle, c) \\ = \text{eval}\ c(\mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_{i-1}, \mathsf{Ret}\ v_i, \mathsf{Ret}\ m_{i+1}, ..., \mathsf{Ret}\ m_n \rangle, \lambda m.m))\end{array} 1 \leq i < n
$$

$$
\frac{\text{eval}\ m_i = \mathsf{Req}\ (h', args, c')}{\begin{array}{l}\text{eval}\ \mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_{i-1}, m_i, ..., m_n \rangle, c) \\ = c(\mathsf{Req}\ (h', args, \lambda m.\mathsf{Req}\ (\mathsf{post}, \langle h, v_1, ..., v_{i-1}, c'(m), ..., m_n \rangle, \lambda m.m)))\end{array} \text{ where } 1 \leq i < n
$$

$$
\text{eval}\ m = m \qquad \text{otherwise}
$$

Figure 2.4: In Figures 2.1 and 2.3, terms and messages were closely related via the eval function. Here we have identified the two via the injection $[[\cdot]]$. The term operation allows us to construct quoted messages that correspond to compound terms. The eval *operation* "unleashes" the behavior of a quoted message.

2.2 Message handlers and mediation

We now have a mechanism by which to traverse and reduce programs, but we still need a way to specify the language semantics in order to actually *run* a program. Figures 2.3 and 2.4 hint at how we will define language semantics; but literally extending the definition of the "eval" function for each kind of term is not scalable. As mentioned in Section 2.1.1, we would like *semantics objects* to themselves be *program objects* which mediate the execution of other, higher-level programs. Thus the key is to provide some *base-level operations* that enable mediated execution.

A *supervisor* program S can mediate the execution of another program P as follows. S must be able to manipulate a *quoted* form of program P—call it $\uparrow P$—as a *data value.* An execution of the program P corresponds to a sequence of requests. The first request message is determined by the root operation in the term representing P, while all subsequent requests are determined by a combination of the supervisor S, the levels underlying S and the first request message of P itself. S starts off by extracting the first request message, m, from P by invoking some base-level operation on $\uparrow P$. If desired, S should service request message m. On the other hand, if S cannot service m, it should be passed on to the even lower-level semantics object (call it L) with respect to which S is defined. As m passes through S, it must be adapted so that m's delimited continuation includes S itself (this ensures that L doesn't forget about S when processing the next message following m). This pass-through behavior is an essential aspect of systems consisting of multiple levels of abstraction; it allows S to override the behavior of selected operations, rather than having to define a complete language interpreter that implements all operations executed by P. After servicing m, execution continues with the next request message—again, serviced by S when possible, and passed on to L otherwise. Finally, keep in mind that all operations S itself performs are themselves implemented as request messages serviced by L or a lower-level semantics object.

In short, quoted messages and some operations on them are somehow reified as values and operations in our base language. In the following, we will show how to define these values and operations. Since this approach allows us to introduce or redefine arbitrary operations, the door is left open to other approaches that may be more appropriate in certain contexts.

2.2.1 Catching and re-throwing request messages

What's the simplest way to give our programs mediation abilities? We could allow a program S to *catch* the first message m from a quoted subprogram $\uparrow P$, perform some computation on m, and then either *continue* with P's next message, or if we don't know how to handle m, *rethrow* it to

Terms of the form $\mathsf{catch}(\uparrow x(), \uparrow \textit{req-handler-term}, \uparrow \textit{program})$:

$$\begin{array}{lcl}
\text{eval } \mathsf{Req}\ (\mathsf{catch}, \langle \mathsf{Ret}\ \uparrow[[x()]], \mathsf{Ret}\ \uparrow[[t]], \mathsf{Ret}\ \uparrow m\rangle, c) & = & \text{eval } c([[\mathsf{post}(\mathsf{catch}, \uparrow[[x()]], \uparrow[[t]], t')]]) \\
 & & \text{where } t' = t\{x() := \mathsf{Ret}\ \uparrow m\} \\
\text{eval } \mathsf{Req}\ (\mathsf{catch}, \langle \mathsf{Ret}\ \uparrow[[x()]], \mathsf{Ret}\ \uparrow[[t]], \mathsf{Ret}\ v\rangle, c) & = & \text{eval } c(\mathsf{Ret}\ v)
\end{array}$$

Terms of the form $\mathsf{rethrow}(\uparrow m)$:

$$\begin{array}{lcl}
\text{eval } \mathsf{Req}\ (\mathsf{rethrow}, \langle \mathsf{Ret}\ \uparrow \mathsf{Ret}\ v\rangle, c) & = & \text{eval } c(\mathsf{Ret}\ v) \\
\text{eval } \mathsf{Req}\ (\mathsf{rethrow}, \langle \mathsf{Ret}\ \uparrow \mathsf{Req}\ (h, \mathit{args}, k)\rangle, c) & = & \text{eval } c(\mathsf{Req}\ (h, \mathit{args}, k')) \\
 & & \text{where } k' = (\lambda m.\text{let } m' = k(m) \text{ in } \mathsf{Ret}\ \uparrow m')
\end{array}$$

$\mathsf{qret}(t)$:

$$\begin{array}{lcl}
\text{eval } \mathsf{Req}\ (\mathsf{qret}, \langle m\rangle, c) & = & \text{eval } c([[\mathsf{post}(\mathsf{post\text{-}qret}, m)]]) \\
\text{eval } \mathsf{Req}\ (\mathsf{post\text{-}qret}, \langle \mathsf{Ret}\ v\rangle, c) & = & \text{eval } c(\mathsf{Ret}\ \uparrow v)
\end{array}$$

Figure 2.5: The two basic execution mediation operations, catch and rethrow. catch intercepts the first message m from a quoted program. rethrow evaluates a quoted message $\uparrow m$ *for one step only:* it quotes and returns whatever message m reduces to. The helper operation qret allows us to evaluate a term and then quote its result.

L—the next-lower level below S. This can be done using the catch and rethrow operations defined in Figure 2.5.

Notice that catch is straightforward in our framework, because a quoted program $\uparrow P$ simply *is* a quoted instance of P's first message, $\uparrow m$. Our definition of catch amounts to a glorified while-loop: so long as m is a *request* message, run an iteration of the handler term t (with $\uparrow m$ substituted in for the "variable" $x()$), which returns the next state of the quoted program (*i.e.,* the quoted message used to evaluate the next step of the program). When m finally reduces to a value, return it.

The rethrow operation is a little bit more interesting: whereas catch intercepts the *first* message of a quoted program, rethrow intercepts the *second* message (after emitting the first). This is similar to running a microprocessor in single-step mode, where each instruction is followed by an *implicit* breakpoint. The trick to it is that the message m is modified with a custom continuation k' that quotes and returns the next message, thus allowing it to be intercepted:

$$\lambda m.\text{let } m' = k(m) \text{ in } \mathsf{Ret}\ \uparrow m'$$

2.2.2 Handling request messages by cases

While conceptually simple, the catch operation is difficult to use: we'd have to write code to explicitly check the kind of request message and what operation it pertains to. This procedural approach doesn't really allow messages to transparently *pass through* to a lower level; it *requires*

$$\text{eval } \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret}\ v\rangle, c) = \text{eval } c(\mathsf{Ret}\ v)$$

$$\frac{args = \langle a_1, ..., a_n\rangle \qquad args' = \langle \mathsf{Ret}\ \uparrow a_1, ..., \mathsf{Ret}\ \uparrow a_n\rangle \qquad k'\ \uparrow m = \uparrow(k\ m)}{\text{eval } \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Req}\ (h, args, k)\rangle, c) = \text{eval } c([[\mathsf{post}(\mathsf{rec\text{-}handler}, h, f, f\ \ args'\ \ k')]])}$$

$$\frac{h \neq h' \qquad k' = \lambda m.[[\mathsf{rec\text{-}handler}(h, f, m)]]}{\text{eval } \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Req}\ (h', args, k)\rangle, c) = \text{eval } c(\mathsf{Req}\ (h', args, k'))}$$

Figure 2.6: Definition of the selective request handler construct rec-handler. Compare to Figure 2.5.

each level to explicitly rethrow a message to the next-lower level to *simulate* the passthrough behavior. This does not accurately model real systems. For example, in an operating system or a virtual machine monitor, privileged operations are trapped by software, while unprivileged operations are directly executed by the hardware.

We will call our selective variant of the catch construct rec-handler, defined in Figure 2.6 (the rec- prefix means recursive: we continually handle messages in a loop, rather than just handling the first message). An important consideration is how a handler specifies which messages it catches. The most obvious choice is to dispatch based on the head of the message; other possible arrangements could be defined as variants of the rec-handler operation. For example, say we would like to define an if operation which first evaluates its guard subterm, and then, depending on whether the result is true or false, evaluates either its then- or else-clause. The first reduction step would be as follows:

$$\mathsf{if}(guard, then\text{-}clause, else\text{-}clause) \Rightarrow \mathsf{post}(\mathsf{post\text{-}if}, guard, \uparrow then\text{-}clause, \uparrow else\text{-}clause)$$

We can implement this first step in terms of the rec-handler operation as follows:

$$\begin{aligned}
&\mathsf{rec\text{-}handler}(\mathsf{if},\\
&\qquad \lambda args.\lambda c.\\
&\qquad\quad \text{let } \langle guard, then\text{-}clause, else\text{-}clause\rangle = args \text{ in}\\
&\qquad\qquad c(\uparrow\mathsf{post}(\mathsf{post\text{-}if}, \downarrow guard, \downarrow\mathsf{qret}(then\text{-}clause), \downarrow\mathsf{qret}(else\text{-}clause))),\\
&\qquad t)
\end{aligned}$$

The above specifies that we should evaluate term t in a context where if requests are caught and serviced by the *handler function* $\lambda args.\lambda c...$, while all other requests (and any eventual return

message) pass through to the containing context. The handler function itself is applied to the argument list[2] and delimited continuation fields of the first if request message emanting from t. The arguments are represented as *quoted* messages, and the delimited continuation as a function from quoted messages to quoted messages. The result of the handler function is itself a message m, which specifies the *next* step in the evaluation of t. As before, non-if requests pass through, while any further if requests are caught and handled, using the same handler function.
The definition in terms of rec-handler is longer than the corresponding rewrite rule, but for good reason:

- rec-handler specifies an explicit scope in which it applies: the term t.
- The explicit delimited continuation c allows *non-local* rewrites within t. For example, in Section 3.2.1 we use this feature to extend a *functional* base language with *mutable variables.*
- The explicit uses of quote and unquote allow us to perform nontrivial computations at the level of the definition of if, rather than injecting a visible sequence of actions into the term t. This is important because unlike in a standard functional language, intermediate steps in a computation are observable events in a language with handlers. In the example definition of if above, we explicitly unquote the occurrences of variables which must be evaluated in the scope of the handler. This foreshadows Section 3.2, where we will define variable lookup in terms of request messages.

2.2.3 One-shot and multi-case handlers

The discussion of catch/rethrow and rec-handler in the previous sections misses a couple of important usage patterns: one-shot handlers and multi-case handlers. A *one-shot* handler only catches the *first* message matching its pattern, rather than *all* matching messages. A *multi-case* handler expression specifies several message patterns and handler functions.

One-shot handlers. catch and rec-handler take the quoted message returned by a handler function, and further execute it within the handling context. It is sometimes useful for a handler to *pause* a computation and go do something else. For example, this can be useful for implementing interleaving concurrrency via multitasking on a single processor. In this case, it is useful to be able to distinguish between a value returned by the *mediating handler* and a value returned by the *mediated computation.*

[2]When if is applied to fewer than three arguments, we assume the let expression results in a run-time error.

Syntax:

$$\mathsf{handlers}(\mathsf{on}(h_1, f_1), ..., \mathsf{on}(h_n, f_n), \textit{on-ret-func}, t)$$

Semantics:

$$\text{eval } \mathsf{Req}\ (\mathsf{handlers}, \langle ..., \mathsf{Ret}\ g, \mathsf{Ret}\ v\rangle, c) = \text{eval } c(g(\mathsf{Ret}\ v))$$

$$\frac{\textit{cases} = ..., \mathsf{Req}\ (\mathsf{on}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f\rangle, c'), ... \qquad \textit{args}' = \langle \mathsf{Ret}\uparrow a_1, ..., \mathsf{Ret}\uparrow a_n\rangle \qquad k'\uparrow m = \uparrow(k\ m)}{\text{eval } \mathsf{Req}\ (\mathsf{handlers}, \langle \textit{cases}, \mathsf{Ret}\ g, \mathsf{Req}\ (h, \langle a_1, ..., a_n\rangle, k)\rangle, c) = \text{eval } c(f\ \ \textit{args}'\ k')}$$

$$\frac{\begin{array}{l}\textit{cases} = \mathsf{Req}\ (\mathsf{on}, \langle \mathsf{Ret}\ h_1, \mathsf{Ret}\ f_1\rangle, c_1'), ..., \mathsf{Req}\ (\mathsf{on}, \langle \mathsf{Ret}\ h_n, \mathsf{Ret}\ f_n\rangle, c_n') \quad \text{where } h' \notin \{h_1, ..., h_n\} \\ k' = \lambda m.[[\mathsf{handlers}(\textit{cases}, g, m)]]\end{array}}{\text{eval } \mathsf{Req}\ (\mathsf{handlers}, \langle \textit{cases}, \mathsf{Ret}\ g, \mathsf{Req}\ (h', \textit{args}, k)\rangle, c) = \text{eval } c(\mathsf{Req}\ (h', \textit{args}, k'))}$$

Figure 2.7: Definition of the selective, multi-case one-shot request handler construct handlers. Compare to Figures 2.5 and 2.6. In the following, we assume that an analogous multi-case variant of rec-handler called rec-handlers is available.

Multi-case handlers. It is common to need to define handlers for multiple different operations which work together: for example the definition of if assumed the existence of a compatible definition of post-if. If we need to *redefine* a pair of operations a and b, and the handler functions in our definitions of a and b both make use of the original definitions of a *and* b, we cannot linearize these definitions.

Figure 2.7 defines a new operation called handlers, which defines multiple cases of request handlers in parallel, all of which operate in a one-shot mode: the result of the first handler that matches is *returned*. Iterated execution must be implemented explicitly, for example, via recursion. In the following chapters, we assume that a similar operation called rec-handlers that generalizes rec-handler to multiple cases is also available. The handlers operation works as follows. Consider a term such as

$$\mathsf{handlers}(\mathsf{on}(h_1, f_1), ..., \mathsf{on}(h_n, f_n), g, t)$$

Here we assume $h_1, ..., h_n$ are operation names (symbols), and $f_1, ..., f_n$ and g are functions—where $f_1, ..., f_n$ are request handlers, while g is a *return value handler*. We execute term t under mediation. Any request by t other than $h_1, ..., h_n$ passes through (as with rec-handler). The first matching request causes us to evaluate the corresponding handler function and return its result as the result of evaluating the handlers expression. Unlike catch and rec-handler, the handlers operation does not further mediate execution of whatever is returned by a handler function. On the other hand, if the computation of t returns a value without invoking any of the operations bound by the handlers operation, its result is passed to the function g (rather than simply

returning the value; this is the only case in which g is used). This allows us to distinguish between values returned by t (which indicate a final result) and values and returned by handler functions (which may represent a paused computation).

Representing handlers in terms of rec-handlers. Clearly, we can represent the rec-handlers operation in terms of the handlers operation if we have another means to implement recursion. But we can also represent handlers in terms of rec-handlers. The naive solution is to simply merge the return value handler function g into the term t. However, this has two problems. First, a handler function which simply returns a quoted term as a representation of a paused computation will be misunderstood: rec-handlers will immediately continue evaluation of that quoted term! Second, g may use some of the operations $h_1, ..., h_n$; by placing it inside the mediated term, the original operations that g was coded against are shadowed by the new handlers. The following encoding fixes these problems (but still has some limitations discussed below). For clarity, we use the "match ... with ..." pseudo-notation to represent use of value decontructors (as in ML).

$$\begin{aligned}
&\mathsf{handlers}(\mathsf{on}(h_1, f_1), ..., \mathsf{on}(h_n, f_n), g, t) \Rightarrow \\
&\text{match } \mathsf{rec\text{-}handlers}(\mathsf{on}(h_1, w\ f_1), ..., \mathsf{on}(h_n, w\ f_n), \mathsf{Right}(t)) \\
&\text{with} \qquad \mathsf{Left}(v) \Rightarrow v; \qquad \mathsf{Right}(v) \Rightarrow g\ v \\
&\text{where } w\ f_i = \lambda args.\lambda k.\mathsf{Left}(f_i\ args\ \lambda q.\uparrow[[\mathsf{Right}^{-1}(\downarrow(k\ q))]]) \text{ for } i \in \{1, ..., n\}
\end{aligned}$$

The general idea is we use a union datatype to represent the two possible reasons that we returned a result from rec-handlers. For this purpose, we assume the operations Left, Right, and Right^{-1} are available. The definition of w is complicated by the fact that it needs to *remove* the outer wrapper of Right(...) from the continuation k. Since the call to Right^{-1} must be *delayed* inside k', if we used this definition of handlers in a loop, an extra layer of Right and Right^{-1} would be inserted with each iteration.

Representing multi-case handlers in terms of multiple nested single-case handlers? As mentioned earlier, we cannot linearize redefinitions of a pair of operations a and b if both of their request handler functions call the old definitions of both a and b. For rec-handlers, we can

work around this problem by introducing a new intermediate operation tmp, as in the following:

$$
\begin{aligned}
&\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{a}, f_{\mathsf{a}}), \mathsf{on}(\mathsf{b}, f_{\mathsf{b}}), g, t) \Rightarrow \\
&\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{tmp}, \lambda args.\lambda k.\text{match } \mathsf{hd}(args) \text{ with} \\
&\quad \mathsf{a} \Rightarrow dispatch\ f_{\mathsf{a}}\ args \\
&\quad \mathsf{b} \Rightarrow dispatch\ f_{\mathsf{b}}\ args), \lambda v.v, \\
&\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{a}, invoke\ \mathsf{a}), \lambda v.v, \\
&\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{b}, invoke\ \mathsf{b}), g, t))) \\
&\text{where } invoke\ h = \lambda args.\lambda k.\mathsf{tmp}(h, args, k) \\
&\text{where } dispatch\ f\ \ args\ k = f\ \ \mathsf{hd}(\mathsf{tl}(args))\ \ (k \circ \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(args))))
\end{aligned}
$$

Notice that the delimited continuations provided to the request handler functions are different in the transformed code vs. the original code. For example, the delimited continuation passed to f_{a} will include the rec-handlers terms binding a and b. If a delimited continuation does not escape a request handler function, this is probably acceptable. But the above encoding breaks down when we switch rec-handlers to handlers, because artifacts of the encoding will remain in any paused computation returned by a handlers term.

It is not clear whether there is another possible encoding of multi-case handlers in terms of single-case handlers that avoids the above problem, but it seems unlikely. Thus we suggest both handlers and rec-handlers should be available in a base language supporting request mediation.

2.3 A base language

To summarize the foregoing discussion, we now present a small *base language* consisting of the (call-by-value) λ-calculus together with our notion of request and return messages and some basic message-oriented constructs, including post, if, handler and rec-handler. The purpose of this section is to present a reasonably concise and complete semantics of such a language. In Chapter 3, we will use handlers to build constructs on top of this base language.

Figure 2.8 presents the syntax of our base language. The forms x, $e\ e$ and $\lambda x.e$ are inherited from the λ-calculus, to which we've added forms for Req and Ret messages. Informally, a request message that appears in an executable part of a program causes the program to make a request to some underlying level. Ret messages can hold values: *symbols, lambda abstractions, quoted messages,* and *lists of values.* A terminating expression always reduces to a message. We allow

m	::=	$r \mid s$	
r	::=	Req (h, ms, f)	request message
s	::=	Ret v	return message
v	::=	h	symbol (a, b, c, ...)
	|	f	function
	|	$\uparrow m$	quoted message
	|	$\langle v, ... \rangle$	list
ms	::=	$\langle m, ... \rangle$	list of messages
e	::=	m	
	|	x	
	|	$e\ e$	
	|	let $x = e$ in e	
	|	Req (h, es, f)	templated request message
	|	Ret $\uparrow e$	templated quoted message
	|	$\downarrow e$	unquotation
u	::=	$v \mid \uparrow e$	*i.e.,* Ret u is an expression e
es	::=	$\langle e, ... \rangle$	list of expressions
f	::=	$\lambda x.e$	
p	::=	$\mathcal{B}[e]$	program built on the base level

Figure 2.8: Syntax of λ-calculus + messages.

parameterized forms of request messages and quoted messages, called *message templates,* to make it easier to dynamically construct (quoted) messages. Since we support quoted messages, we also provide an *unquotation* operation, which has the effect of *evaluating* a quoted message in place.

Given the above, the most important decision remaining is how *delimited continuations* are represented. In our base language, a delimited continuation is represented as an ordinary λ-abstraction mapping quoted messages to quoted messages. The argument to a delimited continuation must be a *quoted* message to prevent it from being evaluated before being passed to the continuation. It is convenient to make the result of a delimited continuation also be a quoted message, as this means delimited continuation composition is simply function composition. This is also useful when writing user-level handler functions since, for example, the handler function of a rec-handler statement is given a delimited continuation (along with other arguments), and must produce a *quoted* message which is further evaluated subject to the rec-handler statement.

The base-level execution semantics of our extended λ-calculus is described in Figures 2.9 and 2.10. The Let Syntax and β-Reduction rules are standard; and Unquotation is straightforward. Subterm Reduction is less complicated than it looks: all it says is you can reduce a subterm as a step in reducing a larger term; *however,* these subterm reductions don't have any observable effects, since effects are only implemented by servicing request messages at the outermost level of a term.

LET SYNTAX

$$(\mathsf{let}\ x = e\ \mathsf{in}\ e') \quad = \quad (\mathsf{Ret}\ \lambda x.e')\ e$$

UNQUOTATION

$$\downarrow(\mathsf{Ret}\ \uparrow e) \quad \Rightarrow \quad e$$

β-REDUCTION

$$\frac{e[x := (\mathsf{Ret}\ v)] = e'}{(\mathsf{Ret}\ \lambda x.e)(\mathsf{Ret}\ v) \quad \Rightarrow \quad e'}$$

SUBTERM REDUCTION

$$\frac{e_1 \quad \Rightarrow \quad e_1'}{e_1\ e_2 \quad \Rightarrow \quad e_1'\ e_2} \qquad \frac{e_2 \quad \Rightarrow \quad e_2'}{e_1\ e_2 \quad \Rightarrow \quad e_1\ e_2'}$$

$$\frac{e_i \quad \Rightarrow \quad e_i'}{\mathsf{Req}\ (h, \langle ..., e_i, ... \rangle, f) \quad \Rightarrow \quad \mathsf{Req}\ (h, \langle ..., e_i', ... \rangle, f)}$$

$$\frac{e \quad \Rightarrow \quad e'}{\downarrow e \quad \Rightarrow \quad \downarrow e'}$$

REQUEST PROPAGATION

$$(\mathsf{Req}\ (h, es, c))\ e \quad \Rightarrow \quad \mathsf{Req}\ (h, es, \lambda q.\uparrow(\downarrow((\mathsf{Ret}\ c)\ q)\ e))$$

$$(\mathsf{Ret}\ v)\ (\mathsf{Req}\ (h, es, c)) \quad \Rightarrow \quad \mathsf{Req}\ (h, es, \lambda q.\uparrow((\mathsf{Ret}\ v)\ \downarrow((\mathsf{Ret}\ c)\ q)))$$

$$\downarrow(\mathsf{Req}\ (h, es, c)) \quad \Rightarrow \quad \mathsf{Req}\ (h, es, \lambda q.\uparrow(\downarrow\downarrow((\mathsf{Ret}\ c)\ q)))$$

BASE-LEVEL OPERATIONS

$$\frac{e \quad \Rightarrow \quad e'}{\mathcal{B}[e] \quad \Rightarrow \quad \mathcal{B}[e']} \qquad \frac{d\ r = e'}{\mathcal{B}[r] \quad \Rightarrow \quad \mathcal{B}[\downarrow e']}\ d \in B$$

Figure 2.9: Basic equivalences and reduction rules of λ-calculus + messages. The rules of variable substitution ($e'' = e[x := e']$) are defined in Figure 2.10. B is assumed to be a set of functions defining the base-level operations for a language; examples are provided in a subsequent figure.

VARIABLE SUBSTITUTION

$$\begin{array}{lcl} x[x := e'] & = & e' \\ y[x := e'] & = & y \text{ otherwise} \end{array}$$

EXPRESSION SUBSTITUTION

$$\frac{e_1[x := e'] = e_1' \qquad e_2[x := e'] = e_2'}{(e_1\ e_2)[x := e'] = e_1'\ e_2'}$$

$$\frac{e[x := e'] = e''}{\downarrow e[x := e'] = \downarrow e''}$$

RETURN MESSAGE (TEMPLATE) SUBSTITUTION

$$\frac{u[x := e'] = u'}{(\mathsf{Ret}\ u)[x := e'] = \mathsf{Ret}\ u'}$$

(here u is either a value (v') or a quoted message template ($\uparrow e$))

$$(\lambda x.e)[x := e'] = \lambda x.e$$

$$\frac{e[x := e'] = e''}{(\lambda y.e)[x := e'] = \lambda y.e''}\ x \neq y \wedge y \notin \mathrm{fv}(e')$$

$$\frac{e[y := z][x := e'] = e''}{(\lambda y.e)[x := e'] = \lambda z.e''}\ x \neq y \wedge y \in \mathrm{fv}(e') \wedge z \notin \mathrm{fv}(e) \cup \mathrm{fv}(e')$$

$$h[x := e'] = h \qquad \text{(names are constants)}$$

$$\frac{e[x := e'] = e''}{(\uparrow e)[x := e'] = \uparrow e''}$$

$$\frac{u_1[x := e'] = u_1' \quad \ldots \quad u_n[x := e'] = u_n'}{\langle u_1, \ldots, u_n\rangle[x := e'] = \langle u_1', \ldots, u_n'\rangle}$$

REQUEST MESSAGE (TEMPLATE) SUBSTITUTION

$$\frac{e_1[x := e'] = e_1' \quad \ldots \quad e_n[x := e'] = e1 \qquad f[x := e'] := f'}{(\mathsf{Req}\ (h, \langle e_1, \ldots, e_n\rangle, f))[x := e'] = \mathsf{Req}\ (h, \langle e_1', \ldots, e_n'\rangle, f')}$$

FREE VARIABLES

$$\begin{array}{lcl} \mathrm{fv}(\mathsf{Ret}\ v) & = & \mathrm{fv}(v) \\ \mathrm{fv}(\mathsf{Req}\ (h, \langle e_1, \ldots, e_n\rangle, f)) & = & \mathrm{fv}(e_1) \cup \ldots \cup \mathrm{fv}(e_n) \cup \mathrm{fv}(f) \\ \mathrm{fv}(\lambda x.e) & = & \mathrm{fv}(e) \setminus \{x\} \\ \mathrm{fv}(h) & = & \emptyset \\ \mathrm{fv}(e\ e') & = & \mathrm{fv}(e) \cup \mathrm{fv}(e') \\ \mathrm{fv}(x) & = & \{x\} \end{array}$$

Figure 2.10: Variable substitution, as required by β-reduction in Figure 2.9.

2.3.1 The Request Propagation rules

The only particularly notable rules in Figure 2.9 are Request Propagation and Base-Level Operations. The Request Propagation rule allows us to evaluate a subterm which depends on a lower level servicing a request message. Since requests can have effects, we require that a function must be evaluated before its argument in order to ensure deterministic evaluation.
The most careful detail in Request Propagation is how the delimited continuations are constructed. Recall that a delimited continuation c is a function from quoted messages to quoted messages, which may be evaluated by supervisor code that implements a handler. This means the evaluation of a delimited continuation should not itself emit any request messages, and should terminate. Observe that each of the Request Propagation rules has the following general form (where $\mathcal{C}[\cdot]$ is some context from which we would like to allow a request message to escape):

$$\mathcal{C}[\mathsf{Req}\ (h, es, c)] \Rightarrow \mathsf{Req}\ (h, es, \lambda q.\uparrow\mathcal{C}[\downarrow((\mathsf{Ret}\ c)\ q)])$$

Assuming c itself does not emit any request messages and terminates when applied to a quoted message, the delimited continuation $\lambda q.\uparrow\mathcal{C}[\downarrow((\mathsf{Ret}\ c)\ q)]$ constructed on the right also has this property.

2.3.2 The Base-Level Operations rules

The Base-Level Operations rule allows request messages that propagate to the base level (delimited by the "$\mathcal{B}[\cdot]$" construct) to be serviced. Base-level operation are serviced by applying some function from a set B of base-level operation definitions to a request message. Note that a request message that propagates to the base level contains the program's full continuation in its delimited continuation argument. Figure 2.10 simply defines variable substitution, as required to perform β-reduction in Figure 2.9.
Finally, Figure 2.11 introduces a set B of base-level operation definitions. The particular definitions in the figure are just one possible example. These include operations introduced earlier in this chapter, such as post, handler and rec-handler; as well as basic operations like name comparison (eq), list manipulation (car/cdr/cons), and if statements. The most important point of this figure is the definition style.
First of all, note that the definitions d_* are *not* λ-abstractions written in our base language; rather, they are simply mappings from request messages to expressions. The idea is a program making

request r is rewritten to another program which reflects whatever result and effects r causes. This corresponds with how the definitions are used in the BASE-LEVEL OPERATIONS rule of Figure 2.9. The base-level operations are part of the language definition, rather than code in the language itself; in this sense, we have defined a *language schema* parameterized on B, rather than a single concrete language.

Secondly, notice that definitions which need to evaluate subterms in a controlled fashion introduce additional rules similar to REQUEST PROPAGATION. For example, this is the case in the last rule for each of post, if, handler and rec-handler. Here post and if unconditionally propagate all request messages (much like the original REQUEST PROPAGATION rule), while handler and rec-handler propagate only those they do not catch.

For example, let's consider the definition of the handler operation in Figure 2.11. The operation as defined expects three arguments: a head/name value h, a function value f, and a quoted term t. The intent of the handler operation is to evaluate the term t, emitting messages that do not have the head h; the first message that does have the head h is passed to handler function f, whose result is returned by the handler term as a whole. If evaluting t returns a value without invoking operation h, then that value is returned by the handler term. The first case of d_{h} implements this return-a-value behavior, while the second case catches a message with head h.

The third case of d_{h} is more complicated. The basic idea is the quoted program term t contains a request message which needs to be evaluated, hence it is propagated out and inserted into the delimited continuation from which the outer request message originated. The intricate part is the delimited continuation for this new propagated request message: it takes the new quoted term resulting from servicing the first request made by t, and places it back in a handler expression. The idea is to allow one request message to escape and be serviced, yet be able to trap the next message, in case it is a return value or a request that should be serviced by the handler itself. (The definition of rec-handler, d_{rh}, is similar, but applies this idea to the second case, messages that are serviced by the handler, as well.)

2.3.3 Step-by-step reduction examples

Figures 2.12 and 2.13 give two step-by-step examples of reducing program terms to return values. Uses of base-level operations are highlighted with $\star\star\star$ in the figures; all other steps involve simplifying a term without invoking any operation requests.

$$B = \{d_{\mathsf{eq}}, d_{\mathsf{car}}, d_{\mathsf{cdr}}, d_{\mathsf{cons}}, d_{\mathsf{post}}, d_{\mathsf{if}}, d_{\mathsf{h}}, d_{\mathsf{rh}}\}$$

$$\begin{aligned}
&d_{\mathsf{eq}}\ \mathsf{Req}\ (\mathsf{eq}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ h\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ \mathsf{true})) \\
&d_{\mathsf{eq}}\ \mathsf{Req}\ (\mathsf{eq}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ h'\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ \mathsf{false})) \quad [h \neq h']
\end{aligned}$$

$$\begin{aligned}
&d_{\mathsf{car}}\ \mathsf{Req}\ (\mathsf{car}, \langle \mathsf{Ret}\ \langle v, ...\rangle\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ v)) \\
&d_{\mathsf{cdr}}\ \mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle v, ...\rangle\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ \langle ...\rangle)) \\
&d_{\mathsf{cons}}\ \mathsf{Req}\ (\mathsf{cons}, \langle \mathsf{Ret}\ v, \mathsf{Ret}\ \langle ...\rangle\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ \langle v, ...\rangle))
\end{aligned}$$

$$\begin{aligned}
&d_{\mathsf{post}}\ \mathsf{Req}\ (\mathsf{post}, \langle h, \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_n\rangle, c) \\
&\qquad = (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Req}\ (h, \langle \mathsf{Ret}\ v_1, ..., \mathsf{Ret}\ v_n\rangle, \lambda q.q))) \\
&d_{\mathsf{post}}\ \mathsf{Req}\ (\mathsf{post}, \langle h, s_1, ..., s_{i-1}, \mathsf{Req}\ (h', es, c'), r_{i+1}, ..., r_n\rangle, c) \\
&\quad = (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Req}\ (h', es, \lambda q.\mathsf{Ret} \uparrow (\mathsf{Req}\ (\mathsf{post}, \langle h, s_1, ..., s_{i-1}, \downarrow((\mathsf{Ret}\ c')\ q), r_{i+1}..., r_n\rangle, \lambda q'.q')))))
\end{aligned}$$

$$\begin{aligned}
&d_{\mathsf{if}}\ \mathsf{Req}\ (\mathsf{if}, \langle \mathsf{Ret}\ \mathsf{true}, then, else\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow then) \\
&d_{\mathsf{if}}\ \mathsf{Req}\ (\mathsf{if}, \langle \mathsf{Ret}\ \mathsf{false}, then, else\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow else) \\
&d_{\mathsf{if}}\ \mathsf{Req}\ (\mathsf{if}, \langle \mathsf{Req}\ (h, es, c'), then, else\rangle, c) \\
&\qquad = (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Req}\ (h, es, \lambda q.\mathsf{Ret} \uparrow (\mathsf{Req}\ (\mathsf{if}, \langle \downarrow((\mathsf{Ret}\ c')\ q), then, else\rangle, \lambda q'.q')))))
\end{aligned}$$

$$\begin{aligned}
&d_{\mathsf{h}}\ \mathsf{Req}\ (\mathsf{handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret} \uparrow \mathsf{Ret}\ v\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ v)) \\
&d_{\mathsf{h}}\ \mathsf{Req}\ (\mathsf{handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret} \uparrow \mathsf{Req}\ (h, a, c')\rangle, c) \\
&\qquad = (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow ((\mathsf{Ret}\ f)(\mathsf{Ret}\ h)(\mathsf{Ret}\ \hat{a})(\mathsf{Ret}\ c'))) \\
&\qquad \text{where } \hat{a} = \langle \uparrow m_1, ..., \uparrow m_n\rangle \text{ if } a = \langle m_1, ..., m_n\rangle \\
&d_{\mathsf{h}}\ \mathsf{Req}\ (\mathsf{handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret} \uparrow \mathsf{Req}\ (h', a, c')\rangle, c) \\
&\quad = (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Req}\ (h', a, \lambda q.\mathsf{Ret} \uparrow (\mathsf{Req}\ (\mathsf{handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, (\mathsf{Ret}\ c')\ q\rangle, \lambda q'.q'))))) \quad [h \neq h']
\end{aligned}$$

$$\begin{aligned}
&d_{\mathsf{rh}}\ \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret} \uparrow \mathsf{Ret}\ v\rangle, c) && = \ (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Ret}\ v)) \\
&d_{\mathsf{rh}}\ \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret} \uparrow \mathsf{Req}\ (h, a, c')\rangle, c) \\
&= (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{let}\ q = (\mathsf{Ret}\ f)(\mathsf{Ret}\ h)(\mathsf{Ret}\ \hat{a})(\mathsf{Ret}\ c')\ \mathsf{in}\ \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, q\rangle, \lambda q'.q'))) \\
&\qquad \text{where } \hat{a} = \langle \uparrow m_1, ..., \uparrow m_n\rangle \text{ if } a = \langle m_1, ..., m_n\rangle \\
&d_{\mathsf{rh}}\ \mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, \mathsf{Ret} \uparrow \mathsf{Req}\ (h', a, c')\rangle, c) \\
&= (\mathsf{Ret}\ c)\ (\mathsf{Ret} \uparrow (\mathsf{Req}\ (h', a, \lambda q.\mathsf{Ret} \uparrow (\mathsf{Req}\ (\mathsf{rec\text{-}handler}, \langle \mathsf{Ret}\ h, \mathsf{Ret}\ f, (\mathsf{Ret}\ c')\ q\rangle, \lambda q'.q'))))) \quad [h \neq h']
\end{aligned}$$

Figure 2.11: Definitions of some base language constructs. These definitions comprise the set of functions B used in Figure 2.9. Notice that most of these definitions introduce additional instances of request propagation (last case of d_{post}, d_{if}, d_{h}, and d_{rh}).

$\mathcal{B}[\ [[\mathsf{if}(\mathsf{true},\mathsf{a},\mathsf{b})]]\ ]$

$\quad [[\mathsf{if}(\mathsf{true},\mathsf{a},\mathsf{b})]]$

$\quad$ — expand term to request message —

$\quad = \ \mathsf{Req}\ (\mathsf{if}, \langle \mathsf{Ret}\ \mathsf{true}, \mathsf{Ret}\ \mathsf{a}, \mathsf{Ret}\ \mathsf{b}\rangle, \lambda q.q)$

$= \quad \mathcal{B}[\mathsf{Req}\ (\mathsf{if}, \langle \mathsf{Ret}\ \mathsf{true}, \mathsf{Ret}\ \mathsf{a}, \mathsf{Ret}\ \mathsf{b}\rangle, \lambda q.q)]$

$\star\star\star$ Base-level Operations, first case of d_{if} $\star\star\star$

$\Rightarrow \quad \mathcal{B}[\downarrow((\mathsf{Ret}\ \lambda q.q)(\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \mathsf{a})))]$

$\quad$ — Subterm Reduction —

$\quad (\mathsf{Ret}\ \lambda q.q)(\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \mathsf{a}))$

$\quad$ — β-reduction —

$\quad \Rightarrow q[q := (\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \mathsf{a}))]$

$\quad$ — Substitution —

$\quad = \ (\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \mathsf{a}))$

$\Rightarrow \quad \mathcal{B}[\downarrow(\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \mathsf{a}))]$

— Unquotation —

$\Rightarrow \quad \mathcal{B}[\mathsf{Ret}\ \mathsf{a}]$

— execution completed —

Figure 2.12: Example: reducing $[[\mathsf{if}(\mathsf{true},\mathsf{a},\mathsf{b})]]$ to $\mathsf{Ret}\ \mathsf{a}$.

$\mathcal{B}[\ [[\mathsf{handler}(\mathsf{ex}, \lambda h.\lambda\hat{a}.\lambda c.\mathsf{cdr}(\langle 1,2\rangle), \uparrow\mathsf{ex}())]]\ ]$

— (partially) expand term to request message —

$= \quad \mathcal{B}[\mathsf{Req}\ (\mathsf{handler}, \langle \mathsf{Ret}\ \mathsf{ex}, [[\lambda h.\lambda\hat{a}.\lambda c.\mathsf{cdr}(\langle 1,2\rangle)]], \mathsf{Ret}\ \uparrow\mathsf{Req}\ (\mathsf{ex}, \langle\rangle, \lambda q.q)\rangle, \lambda q.q)]$

— (further) expand term to request message —

$= \quad \mathcal{B}[\mathsf{Req}\ (\mathsf{handler}, \langle \mathsf{Ret}\ \mathsf{ex}, \mathsf{Ret}\ \lambda h.\mathsf{Ret}\ \lambda\hat{a}.\mathsf{Ret}\ \lambda c.\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q),$
$\mathsf{Ret}\ \uparrow\mathsf{Req}\ (\mathsf{ex}, \langle\rangle, \lambda q.q)\rangle, \lambda q.q)]$

$\star\star\star$ Base-level Operation d_{h}, case 2 $\star\star\star$

$\Rightarrow \quad \mathcal{B}[\downarrow((\mathsf{Ret}\ \lambda q.q)\ (\mathsf{Ret}\ \uparrow((\mathsf{Ret}\ \lambda h.\mathsf{Ret}\ \lambda\hat{a}.\mathsf{Ret}\ \lambda c.\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q))$
$(\mathsf{Ret}\ \mathsf{ex})\ (\mathsf{Ret}\ \langle\rangle)\ (\mathsf{Ret}\ \lambda q.q))))]$

— β-reduction within Subterm Reduction: $\downarrow(e_1\ e_2)$ —

$\Rightarrow \quad \mathcal{B}[\downarrow(\mathsf{Ret}\ \uparrow((\mathsf{Ret}\ \lambda h.\mathsf{Ret}\ \lambda\hat{a}.\mathsf{Ret}\ \lambda c.\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q))\ (\mathsf{Ret}\ \mathsf{ex})\ (\mathsf{Ret}\ \langle\rangle)\ (\mathsf{Ret}\ \lambda q.q)))]$

— Unquotation —

$\Rightarrow \quad \mathcal{B}[(\mathsf{Ret}\ \lambda h.\mathsf{Ret}\ \lambda\hat{a}.\mathsf{Ret}\ \lambda c.\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q))\ (\mathsf{Ret}\ \mathsf{ex})\ (\mathsf{Ret}\ \langle\rangle)\ (\mathsf{Ret}\ \lambda q.q)]$

— β-reduction —

$\Rightarrow \quad \mathcal{B}[(\mathsf{Ret}\ \lambda\hat{a}.\mathsf{Ret}\ \lambda c.\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q))\ (\mathsf{Ret}\ \langle\rangle)\ (\mathsf{Ret}\ \lambda q.q)]$

— β-reduction —

$\Rightarrow \quad \mathcal{B}[(\mathsf{Ret}\ \lambda c.\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q))\ (\mathsf{Ret}\ \lambda q.q)]$

— β-reduction —

$\Rightarrow \quad \mathcal{B}[(\mathsf{Req}\ (\mathsf{cdr}, \langle \mathsf{Ret}\ \langle 1,2\rangle\rangle, \lambda q.q))]$

$\star\star\star$ Base-level Operation d_{cdr} $\star\star\star$

$\Rightarrow \quad \mathcal{B}[\downarrow((\mathsf{Ret}\ \lambda q.q)\ (\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \langle 2\rangle)))]$

— β-reduction within Subterm Reduction: $\downarrow(e_1\ e_2)$ —

$\Rightarrow \quad \mathcal{B}[\downarrow(\mathsf{Ret}\ \uparrow(\mathsf{Ret}\ \langle 2\rangle))]$

— Unquotation —

$\Rightarrow \quad \mathcal{B}[\mathsf{Ret}\ \langle 2\rangle]$

Figure 2.13: Example: reducing $[[\mathsf{handler}(\mathsf{ex}, \lambda h.\lambda\hat{a}.\lambda c.\mathsf{cdr}(\langle 1,2\rangle), \uparrow\mathsf{ex}())]]$ to $\mathsf{Ret}\ 2$.

Chapter 3

Sequential language features

The central claim of this thesis is that the request message-based model of execution presented in Chapter 2 is useful for modeling a range of different programming language constructs. Intuitively, the idea is to build up the semantics of data and control structures for both sequential and concurrent programming in terms of request handlers. By defining the relevant constructs for a given language, the only other thing we need to do to execute programs in that language is parse the source code, rendering executable terms. Since request handlers can be dynamically redefined within a scope, the net result is that the defined language can itself be dynamically extended.
This chapter surveys how to define a variety of different sequential programming constructs in terms of request handlers. Chapter 4 extends this investigation to concurrent language features, while Chapter 5 presents a more concrete case study, defining the features of the widely-used Javascript programming language.
For the purpose of this chapter, we will assume our base language is the language of λ-calculus + messages, as described in Section 2.3. We will assume the set B of base-language operations includes all of the constructs described in Chapter 2, as well as simple variations on these operations.

3.1 Simple control structures

Sequential programming languages often include a variety of simple control structures including if statements, while loops, and variants thereof. This section shows how these can be defined in terms of request handlers.

3.1.1 if statements

Let's assume for the moment that our base-level language does not include the definition d_{if}, but instead includes an *overly eager* if-then-else operation $\mathsf{ifte}(b, t_1, t_2)$, which evaluates b, t_1 and t_2;

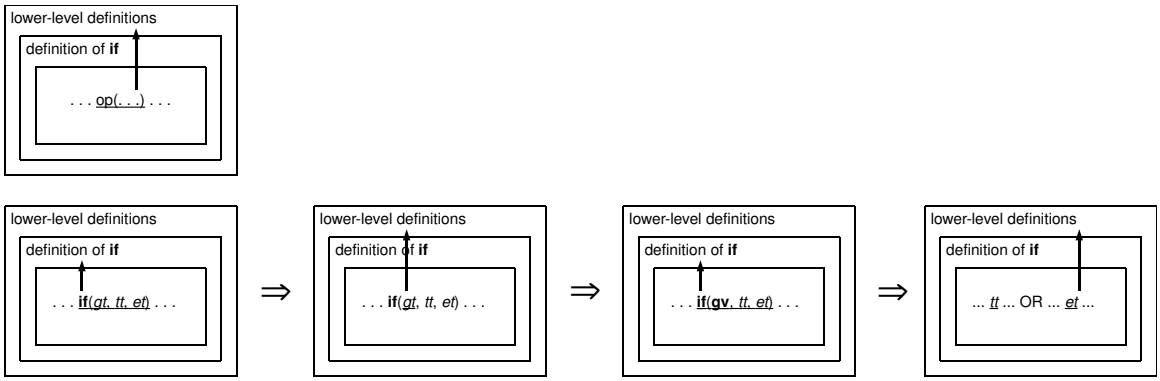


Figure 3.1: Diagram representing how the definition of the if operation works. In the first panel, all non-if operations are ignored by the layer that defines if, and handled by lower levels. The remaining panels depict the steps in evaluating an if-expression: first the if-expression itself is detected; then the guard term (gt) is evaluated, its return value is inspected, and finally, either the then-term (tt) or the else-term (et) is evaluated.

and then if b is true, returns result of t_1, otherwise returns result of t_2. Since evaluating the then- and else-clauses t_1 and t_2 could have effects such as nontermination, a *lazier* if-then-else which only evaluates *either* the then-clause *or* the else-clause is generally preferred. Although such a definition could be included directly in the base language, let's instead examine how to introduce it as a language extension. The idea is depicted in Figure 3.1. This can be defined by the following function, which takes a quoted term q and evaluates it in a context that defines the standard if operation. (Here we assume quote is an operation that takes a value, *i.e.,* a Ret v message, and returns a quoted form of that message, *i.e.,* a Ret $\uparrow$(Ret v) message.)

$$
\begin{aligned}
d_{if} \quad = \quad & \lambda q. \\
& \mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{if}, \lambda a.\lambda c.c\ \uparrow\mathsf{post}(\mathsf{if'}, \downarrow\mathsf{hd}(a), \mathsf{quote}(\downarrow\mathsf{hd}(\mathsf{tl}(a))), \mathsf{quote}(\downarrow\mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a)))))), \\
& \qquad \mathsf{on}(\mathsf{if'}, \lambda a.\lambda c.c\ \mathsf{ifte}(\mathsf{eval}(\mathsf{hd}(a)), \mathsf{hd}(\mathsf{tl}(a)), \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a))))), \\
& \qquad q)
\end{aligned}
$$

How does the above definition of if work? Say q contains a subterm $t = \mathsf{if}(\mathsf{true}, \mathsf{true}, \mathit{loop})$, where *loop* is a term that performs an infinite loop. Assuming if is not redefined within q, if at some point a request message is generated to evaluate t, it will be caught by the rec-handler definition of if. This definition constructs a term and inserts it into the continuation c; the term is *quoted* so that it will be evaluated in the calling context.

In the particular case of if, we insert a post term, which evaluates the guard $\mathsf{hd}(a)$, but *quotes* the then- and else-clauses rather than evaluating them. After evaluating these arguments, post

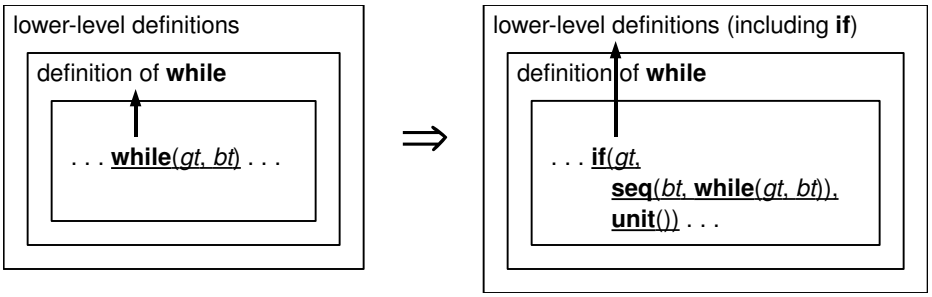


Figure 3.2: Diagram representing the definition of while. When a program attempts to execute a while loop, it is replaced by an if statement implementing one iteration of the loop.

generates an if' request with the *evaluated* arguments. The if' request simply *unquotes* the arguments using eval (since the first argument is a quoted-boolean value, and then second two arguments are quoted-quoted terms), and then uses ifte to select either the then-clause or the else-clause to insert into the delimited continuation c. The selected branch is finally evaluated when the definition of rec-handlers continues to evaluate the quoted term returned by the ifte handler function.

3.1.2 while loops

Unlike if, our base language has no construct directly corresponding to a while-loop. Since we have a recursive rec-handlers construct, it seems reasonable that we could construct a definition of while similar to the definition of if in the previous section. This is depicted in Figure 3.2, or in code, as follows.

$$d_{while} = \lambda q.$$
$$\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{while}, \lambda a.\lambda c.c \uparrow \mathsf{if}(\downarrow \mathsf{hd}(a), \mathsf{seq}(\downarrow \mathsf{hd}(\mathsf{tl}(a)), \mathsf{while}(\downarrow \mathsf{hd}(a), \downarrow \mathsf{hd}(\mathsf{tl}(a)))), \downarrow \mathsf{qret}(\mathsf{unit}()))) \; q)$$

This is the standard definition of a while loop: if the guard is true, evaluate the body once, and then (loop back and) evaluate the while loop again (in the possibly updated environment). The definition builds on the definition of if in the previous section, since we must always check the guard before evaluating the next iteration. The interesting thing is that the definitions of if and while *commute,* in the following sense:

$$\lambda q.d_{if} \uparrow (\downarrow \mathsf{qret}(d_{while}) \downarrow q) \equiv \lambda q.d_{while} \uparrow (\downarrow \mathsf{qret}(d_{if}) \downarrow q)$$

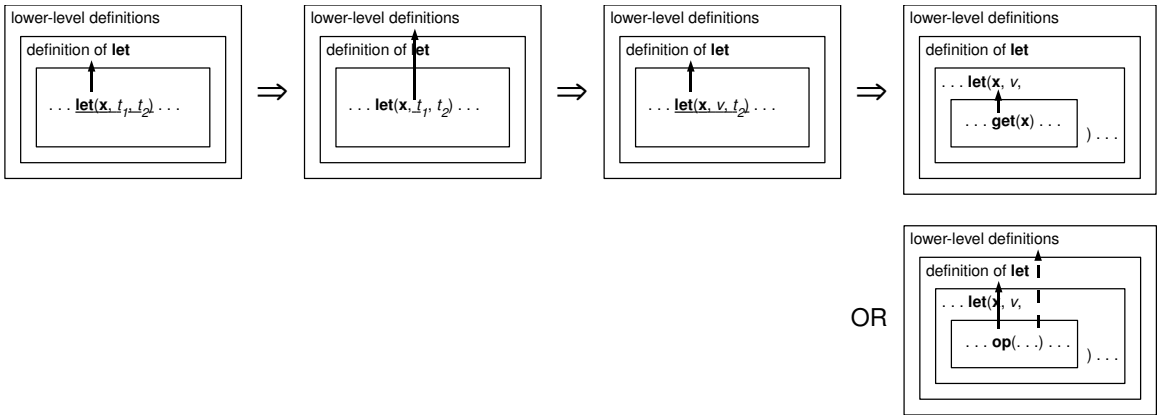


Figure 3.3: Diagram representing the definition of lexical variables. A let expression first evaluates the term t_1 (resulting in some value v), and then creates a new nested context in which lexical variable x is bound to value v, in which term t_2 is evaluated. Only get(x) operations are overriden in the custom context.

The above is written as an equivalence rather than an equation because the two terms can be distinguished if placed in a context which redefines the rec-handlers operation (or λ a function application, for that matter). But barring that difference, when the left and the right hand sides are applied to the same quoted term q, they perform the same computation. This is because the handler functions defining if and while do not directly use if or while; they only *insert* uses of if and while into the subject program. These inserted terms result in request messages which are handled by the nearest enclosing definition. If q itself redefines either if or while, that will be always be the nearest enclosing definition. Otherwise, d_{if} handles if request and d_{while} handles while requests. By construction, if the definition of d_{if} is inside the definition of d_{while}, it does not override the behavior of while requests, and vice versa.

3.2 Lexical variables

Having defined simple control structures, one might ask: can we use a similar technique to define the λ-calculus primitives themselves? This section and the next answer that question in the affirmative.

The central feature we need to define is *lexically-scoped variables.* While we could do this by translating terms using variables to corresponding λ-abstractions, doing so ties us to the λ-calculus as a base language. Instead, we will define explicit operations that create let-bindings and perform variable accesses. One approach to doing this is inserting a rec-handlers term, as illustrated in the quoted let' handler below. Figure 3.3 depicts this definition visually.

$$
\begin{aligned}
&\mathit{d\text{-}let}(q) = \\
&\quad \mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{let}, \lambda a.\lambda c.c\ \uparrow\mathsf{post}(\mathsf{let'}, \downarrow\mathsf{qret}(\mathsf{hd}(a)), \downarrow\mathsf{hd}(\mathsf{tl}(a)), \downarrow\mathsf{qret}(\mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a))))))), \\
&\qquad\qquad \mathsf{on}(\mathsf{let'}, \lambda a.\lambda c.c\ \uparrow\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{get}, \lambda a.\lambda c.\mathsf{if}(=(\mathsf{hd}(a), \downarrow\mathsf{hd}(a)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad c\ \downarrow\mathsf{hd}(\mathsf{tl}(a)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad c\ \mathsf{post}(\mathsf{get}, \mathsf{eval}(\mathsf{hd}(a))))), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \downarrow\mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a))))), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad q)
\end{aligned}
$$

The above defines let as a multi-step operation:

1. When we encounter a term $\mathsf{let}(x, t_1, t_2)$, first evaluate t_1 until we reach some value v_1; then call $\mathsf{let'}(\mathsf{quote}(x), v_1, \mathsf{quote}(t_2))$. This is implemented by the quoted post operation.

2. In response to let', insert *another* rec-handlers into the delimited continuation c, in place of the let/let' term. Evaluate the body t_2 within the scope of this new handler term.

3. When the body attempts to evaluate a term $\mathsf{get}(x)$, for the *same* symbol x bound in the original let term, insert the value v_1 in the corresponding hole in the delimited continuation, and continue.

4. On the other hand, if the get request was for a *different* variable, replay that request via the expression $\mathsf{post}(\mathsf{get}, \mathsf{eval}(\mathsf{hd}(a)))$, and return the result of that request, instead. This is essentially to patch up a mistake we made in catching the wrong request message, because the rec-handlers construct does not allow a specific enough guard.

This procedure performs a linear scan through the environment, until the appropriate binding is found. There are two shortcomings with the definition *d-let*, addressed in the following two subsections:

1. It is not clear how we would implement *mutable* variables *without* already having an imperative base language. The inserted rec-handlers term does not provide an ability to change the value that will be inserted for variable x, because the handlers remain fixed for every message it processes.

2. More generally, in the above we agreed not to translate a let term to a λ-term to avoid requiring acces to the λ-calculus operations in our subject term; and yet we freely inserted a rec-handlers term, which is certainly even *less* standard than the λ-calculus!

3.2.1 Mutable variables

In order to introduce mutable variables, we need to convert the inner rec-handlers term inserted by *d-let* to a more general handlers term. Recall that the difference is handlers only processes the *first* matching message, and then evaluates the handler function in tail position (in place of the entire handlers term). Assuming we have a fixpoint combinator that operates on λ-abstractions (call it rec), we can convert the rec-handlers term to a recursive function in *explicit state-passing form* that calls handlers at each iteration. A rec-handlers only passes one state element: the quoted subject term; the handlers remain the same in each iteration. Now we need to pass two state elements: the current value v of variable x; and the quoted subject term, q. This is illustrated below:

$$
\begin{array}{l}
\textit{d-mlet}(q) = \\
\quad \mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{mlet}, \lambda a.\lambda c.c\ \uparrow\mathsf{post}(\mathsf{mlet'}, \downarrow\mathsf{qret}(\mathsf{hd}(a)), \downarrow\mathsf{hd}(\mathsf{tl}(a)), \downarrow\mathsf{qret}(\mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a))))))), \\
\qquad \mathsf{on}(\mathsf{mlet'}, \lambda a.\lambda c.c\ \uparrow(\mathsf{rec}(\lambda r.\lambda v.\lambda q.\mathsf{handlers}(\\
\qquad\qquad\qquad\qquad \mathsf{on}(\mathsf{get}, \lambda a.\lambda c.\mathsf{if}(=(\mathsf{hd}(a), \downarrow\mathsf{hd}(a)), \\
\qquad\qquad\qquad\qquad\qquad\qquad r\ v\ (c\ v), \\
\qquad\qquad\qquad\qquad\qquad\qquad r\ v\ (c\ \mathsf{post}(\mathsf{get}, \mathsf{eval}(\mathsf{hd}(a))))))), \\
\qquad\qquad\qquad\qquad \mathsf{on}(\mathsf{set'}, \lambda a.\lambda c.\mathsf{if}(=(\mathsf{eval}(\mathsf{hd}(a)), \downarrow\mathsf{hd}(a)), \\
\qquad\qquad\qquad\qquad\qquad\qquad r\ \mathsf{eval}(\mathsf{hd}(\mathsf{tl}(a)))\ (c\ \mathsf{qret}(\mathsf{unit}())), \\
\qquad\qquad\qquad\qquad\qquad\qquad r\ v\ (c\ \mathsf{post}(\mathsf{set'}, \mathsf{eval}(\mathsf{hd}(a)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{eval}(\mathsf{hd}(\mathsf{tl}(a)))))))), \\
\qquad\qquad\qquad\qquad \lambda x.x, q))\ \downarrow\mathsf{hd}(\mathsf{tl}(a))\ \downarrow\mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a))))), \\
\qquad \mathsf{on}(\mathsf{set}, \lambda a.\lambda c.c\ \uparrow\mathsf{post}(\mathsf{set'}, \downarrow\mathsf{qret}(\mathsf{hd}(a)), \downarrow\mathsf{hd}(\mathsf{tl}(a)))), \\
\qquad q)
\end{array}
$$

The first two lines of *d-mlet* are analogous to *d-let*; only the quoted code inserted by the mlet' handler has changed. Within the mlet' handler, we have performed the explicit state-passing expansion:

$$
\mathsf{rec\text{-}handlers}(..., \ \textit{quoted-body}) \quad \Rightarrow \quad (\mathsf{rec}(\lambda r.\lambda v.\lambda q.\mathsf{handlers}(..., \lambda x.x, q))\ \textit{initial-value}\ \textit{quoted-body})
$$

Where *initial-value* is the initial value of the variable x; *quoted-body* is the body of the mlet term, in which x is bound to *initial-value;* and rec is a fixpoint combinator on λ-abstractions: $\mathsf{rec}(\lambda r.t) = (\lambda r.t)\ \mathsf{rec}(\lambda r.t)$. The complimentary changes inside the get handler function are to use v

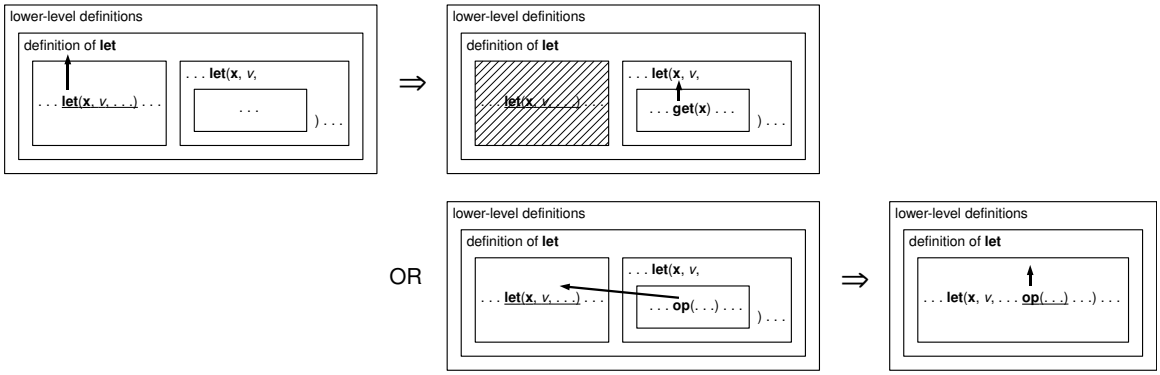


Figure 3.4: Rather than inserting a new handler inline as in Figure 3.3, the definition of nlet creates a custom handler *outside of* the main program term (this is the right subpanel in the first three panels). If the body term attempts to perform an operation not implemented by the nlet block itself, the corresponding request message is injected back into the main program term (last two panels).

in place of the hard-coded *initial-value,* and to explicitly pass the new quoted term to continue evaluating to $(r\ v)$, rather than returning it for rec-handlers to continue evaluating.

The set' handler has a form similar to the get handler: if the variable to update matches that bound, recurse with the *updated* value in place of v; otherwise, pass the set' request up to the next-closest enclosing handler, and then continue.

3.2.2 Avoiding insertion of a handlers term

The previous section generalized let to allow updates to variables, but still relied on the ability to evaluate a handlers term within the subject term. In practice, we may wish to define a language environment which is not allowed to use the handlers operation, but still can create variable bindings. How do we overcome this dilemma? The basic idea is rather than *inserting* a handlers term into the subject term, we *evaluate the body* of the let term *outside of* the subject term, and then *insert any requests that we cannot handle.* This is depicted in Figure 3.4. The key is we must insert request messages which have a *non-identity* delimited continuation—in other words, request messages that are not of the form $[[t]]$ for any term t (see Figure 2.4). This is accomplished using

the *mesg* definition below:

$$
\begin{aligned}
&\mathit{d\text{-}nlet}(q) = \\
&\quad \mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{nlet}, \lambda a.\lambda c.c\ \uparrow\mathsf{post}(\mathsf{nlet'}, \downarrow\mathsf{qret}(\mathsf{hd}(a)), \downarrow\mathsf{hd}(\mathsf{tl}(a)), \downarrow\mathsf{qret}(\mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a)))))), \\
&\qquad\qquad \mathsf{on}(\mathsf{nlet'}, \lambda a.\lambda c.c\ (\mathsf{rec}(\lambda r.\lambda v.\lambda q.\mathsf{handlers}(\mathsf{on}(\mathsf{get}, \lambda a'.\lambda c'.\mathsf{if}(=(\mathsf{qret}(\mathsf{hd}(a')), \mathsf{hd}(a)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r\ v\ (c'\ v), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c\ \mathit{mesg}(\mathsf{get}, a', \mathsf{hd}(a), v, c')), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{on}(\mathsf{set'}, \lambda a'.\lambda c'.\mathsf{if}(=(\mathsf{hd}(a'), \mathsf{hd}(a)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r\ \mathsf{hd}(\mathsf{tl}(a'))\ (c\ \mathsf{qret}(\mathsf{unit}())), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c\ \mathit{mesg}(\mathsf{set}, a', \mathsf{hd}(a), v, c')), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{other}(\lambda h'.\lambda a'.\lambda c'.c\ \mathit{mesg}(h', a', \mathsf{hd}(a), v, c')), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda x.x, q))\ \mathsf{hd}(\mathsf{tl}(a))\ \mathsf{hd}(\mathsf{tl}(\mathsf{tl}(a))))), \\
&\qquad\qquad \mathsf{on}(\mathsf{set}, \lambda a.\lambda c.c\ \uparrow\mathsf{post}(\mathsf{set'}, \downarrow\mathsf{qret}(\mathsf{hd}(a)), \downarrow\mathsf{hd}(\mathsf{tl}(a)))), \\
&\qquad\qquad q) \\
&\mathit{mesg}(h', a', x, v, c') = \mathsf{qreq}(h', a', \lambda q.\uparrow\mathsf{nlet'}(\downarrow x, \downarrow v, \downarrow(c'\ q)))
\end{aligned}
$$

What we have done here is removed the quotation on the inner handler term; inserted an *explicit* handler to catch other messages (the other term); and used the *mesg* definition to *insert request messages* into continuation c for any request that cannot be handled directly by the definition[1] of nlet'. *mesg* uses the qreq operation to create a *quoted request message* which can be composed with the delimited continuation c. The effect is that the generated request message escapes the scope of the nlet' binding, and is serviced by the nearest enclosing matching handler; but once the result of this request is inserted into the continuation $\lambda q.\uparrow\mathsf{nlet'}(\downarrow x, \downarrow v, \downarrow(c'\ q))$, *control returns to the nlet' handler.*

3.3 Lambda and apply

Up to this point, we have assumed that λ, bound variables, and function application are built into our base language. But what if we wanted to start from a different base language equipped with handlers, and yet make use of the sorts of definitions illustrated in this chapter? We have already created a definition of let-bindings and lexical variable in Section 3.2. Lambda abstractions can be considered as a more dynamic generalization of let-bindings.

[1] The "n" in nlet' means "no inserted handlers."

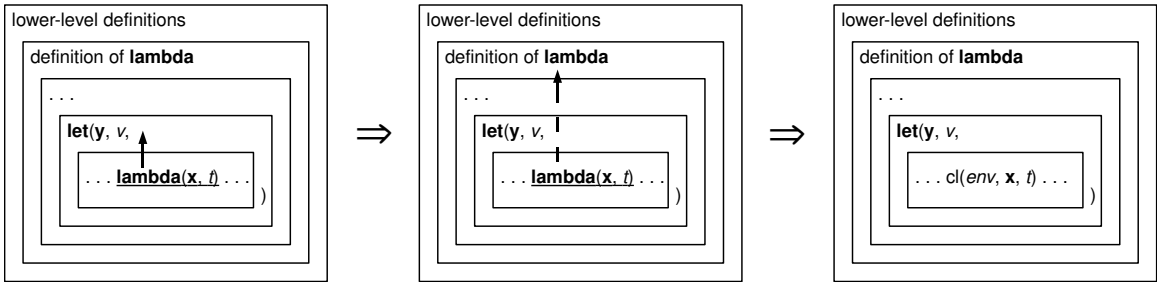


Figure 3.5: The lambda operation is defined by incrementally adding environment information to a closure at each enclosing let expression, until the full closure value has been constructed.

In this section, we define two operations, lambda and apply, which act like their λ-calculus namesakes. Note that lambda will require modifications to the definition of nlet', to gain access to the current environment and build a closure. (Had we implemented our let-binding as an eager substitution, rather than a lazy environment lookup, we *still* would need to adjust its definition to stop substituting when it encounters a lambda term binding the same variable.)

3.3.1 The lambda operation

The idea for our definition of lambda is depicted in Figure 3.5. An expression $\mathsf{lambda}(x, t)$ collects the bindings in effect from all in-scope let expressions, building an environment env, and then returns a *closure* value of the form $\langle env, x, t\rangle$. More precisely, we will define the lambda operation as a combination of two cases:

1. An additional handler within the handlers term of the nlet' handler (see *d-nlet*). This case is used to incrementally build up the environment in a closure.

2. A base case, which packages up the closure when there are no more bindings to add to the environment.

Additional nlet' handler. The additional handler that needs to be inserted into *d-nlet* is as follows.

$$\begin{aligned}&\mathsf{on}(\mathsf{lambda}, \lambda a'.\lambda c'.\mathsf{if}(=(\mathsf{qret}(\mathsf{hd}(a')), \mathsf{hd}(a)),\\ &\qquad\qquad c\ \mathit{mesg}(\mathsf{lambda}, a', \mathsf{hd}(a), v, c'),\\ &\qquad\qquad c\ \mathit{mesg}(\mathsf{lambda}, \mathsf{list}(\mathsf{hd}(a'), \uparrow\mathsf{nlet'}(\downarrow\mathsf{hd}(a), \downarrow v, \downarrow\mathsf{hd}(\mathsf{tl}(a')))), \mathsf{hd}(a), v, c'))\end{aligned}$$

First of all, the reason the above handler must be inserted into *d-nlet*, rather than somehow composing with it is that it needs access to the variables a and v, which are internal to the loop

rec(...) loop within *d-nlet*. In words, what this handler does is it checks whether the variable bound by lambda matches that bound by nlet'. If *yes,* the arguments to the lambda abstraction are unchanged, and the message is propagated outwards, since the bound variable within the lambda abstraction would shadow any binding due to the nlet' term. However, if the variables are *different,* it *extends the body* of the lambda abstraction to include the additional nlet' binding. An important thing to consider any time we have mutable variables is: *how* are they captured by functions? In the above construction, mutable variables are captured by *value:* the body of the lambda abstraction is extended with a *new* variable with the same name hd(a) and its value v at the time of capture. If the v in the original nlet' term were later changed via a set operation, this would *not* affect the captured variable inside a lambda term that has already been evaluated to produce a closure. The alternative is to introduce a common *mutable store* that wraps all terms, and capture an immutable address in the store, rather than the current value of a mutable variable. This is addressed in Chapter 5, where we reuse the store of a host Javascript interpreter as the store for a defined language.

Base case handler. Once a lambda request escapes all enclosing nlet' terms, it eventually has to be transformed into some sort of value—a closure representing the first-class function. For now, we will do this by simply creating a pair consisting of the bound variable name and the body of the lambda abstraction, and inserting that back into the delimited continuation. Since this means lambda abstractions can be forged, in Section 3.4, we will consider how to create abstract data structures that cannot be forged within a given scope. The definition of the base case is as follows. We separate it into two steps because in the next section we need to make closures elsewhere.

$$
\begin{aligned}
&\textit{d-lambda}(q) = \\
&\quad \text{rec-handlers}(\text{on}(\text{lambda}, \lambda a.\lambda c.c\ \text{qreq}(\text{make-closure}, a, \lambda q.q)), \\
&\qquad\qquad \text{on}(\text{make-closure}, \lambda a.\lambda c.c\ \text{qret}(a)), \\
&\qquad\qquad q)
\end{aligned}
$$

(Where a already consists of exactly two terms—the bound variable name and the body.) Although λ is used in the definitions of the handler functions above, note that λ-abstractions are not inserted as values into the quoted term q.

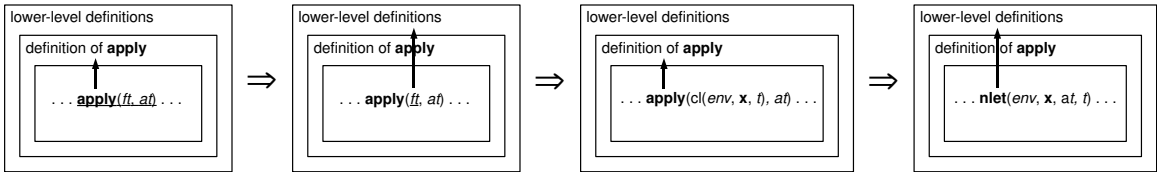


Figure 3.6: The apply operation simply evaluates the function term *ft* to a closure value, and then builds a corresponding let-expression, binding the environment from the closure and the argument while evaluating the function body.

3.3.2 The apply operation

Given a closure represented as a pair of a bound variable name and the body of a lambda abstraction as above, it's a fairly simple matter to apply that closure to a value. All we need to do is create a corresponding let or nlet term:

$$\begin{aligned} &d\text{-}apply_1(q) = \\ &\quad \mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{apply}, \lambda a.\lambda c.c\ \uparrow\mathsf{nlet}(\downarrow\mathsf{hd}(\mathsf{eval}(\mathsf{hd}(a))), \downarrow\mathsf{hd}(\mathsf{tl}(a)), \downarrow\mathsf{hd}(\mathsf{tl}(\mathsf{eval}(\mathsf{hd}(a)))))), \\ &\qquad\qquad q) \end{aligned}$$

To better understand the structure of this term, note that a well-formed argument list to apply can be generated by the expression $\mathsf{list}(\mathsf{qret}(\mathsf{list}(\mathsf{qret}(x), \mathsf{qreq}(body))), \mathsf{qreq}(arg))$. Hence $\mathsf{eval}(\mathsf{hd}(a))$ is the closure, which is a list containing the bound variable name and body (both quoted), and so on. As discussed earlier in the context of if and while, the definition $d\text{-}apply_1$ commutes with the definitions *d-lambda* and *d-nlet*, because the apply handler does not directly use or override the lambda or nlet/nlet' handlers.

There are two basic problems with above solution ($d\text{-}apply_1$):

1. It calls eval on a term passed in by user code within the context of a handler. Normally, this term would be the closure, which is a quoted list, *i.e.,* a value. But a malformed request could result in arbitrary code execution. This is remedied in Section 3.4.

2. The body of a lambda abstraction may attempt to access lexically free variables via get or set' requests. If the caller has a lexical bound variable by the same name, the free variable will be captured. This is really a problem for the caller, since it means a function it calls may peer into its lexical environment. This is addressed below.

Preventing variable capture. How can we address variable capture in the request-based framework? In the λ-calculus semantics based on eager substitution, α-renaming is used to prevent

variable capture. However, when it is possible to dynamically generate variable get and set' requests, we may not know all of the free variables that an argument term needs to access at the time of α-renaming.

Since we are using an explicit environment, the alternative is to create a "terminator" term which catches all unhandled variable requests within an apply term, as follows:

$$
\begin{aligned}
&\textit{d-apply}_2(q) = \\
&\quad \text{rec-handlers}(\text{on}(\text{apply}, \lambda a.\lambda c.c\ \uparrow\text{terminator}(\text{nlet}(\downarrow\text{hd}(\text{eval}(\text{hd}(a))), \downarrow\text{hd}(\text{tl}(a)), \downarrow\text{hd}(\text{tl}(\text{eval}(\text{hd}(a))))))), \\
&\qquad\qquad \text{on}(\text{terminator}, \lambda a.\lambda c.\textit{terminator}(a, c)), \\
&\qquad\qquad q)
\end{aligned}
$$

There are several choices for how the terminator operation could catch and handle get and set' requests to free variables:

1. Trigger some sort of error condition.
2. Make all get requests respond with an undefined value, and make set' requests have no effect (*i.e.,* undefined, read-only free variables).
3. Dynamically introduce new variable bindings local to the function body.
4. Dynamically introduce new global variables.

We will address cases 1-4 in the variants of *terminator* defined below. First of all, we can trigger an error condition by simply rewriting the first occurrence of a get or set' request that reaches the terminator into an error request. Some other handler would then be responsible for handling error requests. Note that any get or set' request that reaches the terminator is by necessity an access to a free variable, since accesses to bound variables would be caught by intervening let or nlet layers.

$$
\begin{aligned}
&\textit{terminator}_1(a, c) = \\
&\quad \text{rec}(\lambda r.\lambda q.\text{handlers}(\text{on}(\text{get}, \lambda a'.\lambda c'.c\ \uparrow\text{error}(\text{get}, \downarrow\text{qret}(a'), \downarrow\text{qret}(c'))), \\
&\qquad\qquad\qquad\qquad \text{on}(\text{set'}, \lambda a'.\lambda c'.c\ \uparrow\text{error}(\text{set'}, \downarrow\text{qret}(a'), \downarrow\text{qret}(c'))), \\
&\qquad\qquad\qquad\qquad \text{on}(\text{lambda}, \lambda a'.\lambda c'.r\ (c'\ \text{qreq}(\text{make-closure}, a', \lambda q.q))), \\
&\qquad\qquad\qquad\qquad \text{other}(\lambda h'.\lambda a'.\lambda c'.c\ \text{qreq}(h', a', \lambda q.\uparrow\text{terminator}(\downarrow(c'\ q)))), \\
&\qquad\qquad\qquad\qquad \lambda x.x, q))\ \text{hd}(a)
\end{aligned}
$$

As in *d-nlet*, we use the other handler case to unwind requests not serviced by the definition of

$terminator_1$ in a controlled manner. We also need to introduce the base case handler for lambda here, to avoid capturing free variables when we create a new closure. We will omit the on(lambda, ...) and other(...) handlers in the other versions of *terminator* below, since they do not change.

The second option is to rewrite get and set' requests as if they referred to an immutable variable with an undefined value. This is illustrated below. It is again written in a style similar to *d-nlet*.

$$
\begin{aligned}
&terminator_2(a, c) = \\
&\quad \mathsf{rec}(\lambda r.\lambda q.\mathsf{handlers}(\mathsf{on}(\mathsf{get}, \lambda a'.\lambda c'.r\ (c'\ \mathsf{qret}(\mathsf{undefined}()))), \\
&\qquad\qquad \mathsf{on}(\mathsf{set'}, \lambda a'.\lambda c'.r\ (c'\ \mathsf{qret}(\mathsf{unit}()))), \\
&\qquad\qquad ..., \\
&\qquad\qquad \lambda x.x, q))\ \mathsf{hd}(a)
\end{aligned}
$$

Finally, we can extend the set' handler to actually insert a new nlet layer binding any free variable that is written to. The only change is in the set' handler, which wraps c' with a let-binding:

$$
\begin{aligned}
&terminator_3(a, c) = \\
&\quad \mathsf{rec}(\lambda r.\lambda q.\mathsf{handlers}(\mathsf{on}(\mathsf{get}, \lambda a'.\lambda c'.r\ (c'\ \mathsf{qret}(\mathsf{undefined}()))), \\
&\qquad\qquad \mathsf{on}(\mathsf{set'}, \lambda a'.\lambda c'.r\ \uparrow\mathsf{nlet'}(\downarrow\mathsf{hd}(a'), \downarrow\mathsf{hd}(\mathsf{tl}(a')), \downarrow(c'\ \mathsf{qret}(\mathsf{unit}())))), \\
&\qquad\qquad ..., \\
&\qquad\qquad \lambda x.x, q))\ \mathsf{hd}(a)
\end{aligned}
$$

Notice that since each terminator term corresponds to a particular instance of an apply term, the definition $terminator_3$ above is inherently restricted to converting free variables to *local* variables. The alternative is to introduce a new *global* variable. We will introduce on possible such solution below, and revisit this problem later in Chapter 5.

The basic challenge in dynamically introducing a new global variable is we need to skip over all of the local variables in the intervening levels of callers. In implementation terms, this means we need to stop crawling the stack and refer directly to the heap. We will do this via a combination of the strategies in $terminator_1$ and $terminator_3$: rather than transforming get and set' requests into *errors,* $terminator_4$ will transform them into *global variable requests,* which are only caught at the

base level.

$$
\begin{aligned}
&\mathit{terminator}_4(a, c) = \\
&\quad \mathsf{rec}(\lambda r.\lambda q.\mathsf{handlers}(\mathsf{on}(\mathsf{get}, \lambda a'.\lambda c'.c\ \mathsf{qreq}(\mathsf{gget}, a', c')), \\
&\qquad\qquad \mathsf{on}(\mathsf{set'}, \lambda a'.\lambda c'.c\ \mathsf{qreq}(\mathsf{gset'}, a', c')), \\
&\qquad\qquad \ldots \\
&\qquad\qquad \lambda x.x, q))\ \mathsf{hd}(a)
\end{aligned}
$$

At the base level, a global variable request is transformed back into a normal variable request—*i.e.,* gget becomes get and gset' becomes set' again. As in $\mathit{terminator}_3$, a set' request on an undefined variable results in the insertion of a new nlet' layer. It is crucial that this new variable binding surround the code that translates global variable requests back to normal variable requests. For this reason, we have separated the handlers in the *globals* definition into two separate rec-handlers layers:

$$
\begin{aligned}
&\mathit{globals}(q) = \\
&\quad \mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{get}, \lambda a.\lambda c.c\ \mathsf{qret}(\mathsf{undefined}())), \\
&\qquad\qquad \mathsf{on}(\mathsf{set'}, \lambda a.\lambda c.\uparrow\mathsf{nlet'}(\downarrow\mathsf{hd}(a), \downarrow\mathsf{hd}(\mathsf{tl}(a)), \downarrow(c\ \mathsf{qret}(\mathsf{unit}())))), \\
&\quad \uparrow\mathsf{rec\text{-}handlers}(\mathsf{on}(\mathsf{gget}, \lambda a.\lambda c.c\ \mathsf{eval}(\mathsf{qreq}(\mathsf{get}, a, \lambda q.\mathsf{qret}(q)))), \\
&\qquad\qquad \mathsf{on}(\mathsf{gset'}, \lambda a.\lambda c.c\ \mathsf{eval}(\mathsf{qreq}(\mathsf{set'}, a, \lambda q.\mathsf{qret}(q)))), \\
&\qquad\qquad \downarrow\mathsf{qret}(q)))
\end{aligned}
$$

3.4 Abstract data structures

Abstract data structures are useful in any system, as a means of separating interfaces from implementations. Normally, data structures are considered as concrete within the scope of an implementation module, and abstract everywhere else that they are visible in the program. Thus an abstract data structure has global extent, although its internals are only visible within a specific scope.

But when a program involves multiple levels of abstraction, it is no longer reasonable to assume high-level definitions have global extent. For example, a CPU or an OS kernel has no understanding of the classes defined in a Java program running in user mode; yet the lower levels must be able to execute on behalf of the Java program. This is because the concrete bit-level representations of the Java classes are exposed to the lower levels of the system: the JVM itself, the OS kernel, and the CPU.

We can describe data structures in terms of their relationship to three different levels of the system:

1. At levels *below* a data structure's definition, instances appear as their concrete meta-representation.

2. At the level at which a data structure is defined, and above, instances are represented concretely.

3. At some level, the data structure is *abstracted;* at all levels above that, instances are treated opaquely, and only accessible through defined constructor and accessor operations in scope.

Intuitively, we can think of a data structure as a *box* in which we store values. At higher levels (3), where the data structure is abstract, the box is locked; whereas at lower levels (1 & 2), it is unlocked. The box contains some value, together with a label describing what sort of box this is. At levels above the definition (2), that label is meaningful, while at underlying levels (1), it is readable, but meaningless.

We can implement the above idea with the following operations:

make-box(s, v)	create a box with label the symbol s and contents v
is-box(v)	is v an unlocked box?
box-label(v)	return the label of an unlocked box
box-contents(v)	return the contents of an unlocked box
lock-box(s, t)	lock all boxes with label s and evaluate term t

Notice that boxes are *themselves* an abstract data structure in our system, which are made accessible via the above operations. Thus we must have some definitions corresponding to make-box, is-box, box-label and box-contents in our base language. If we adopt the notation $\#s\{v\}$ for a box with label the symbol s and contents the value v, and assume boxes are abstract except in the base-level eval function, we can use the following operation definitions in our base language.

$$B = \{..., d_{\mathsf{mb}}, d_{\mathsf{ib}}, d_{\mathsf{bl}}, d_{\mathsf{bc}}\}$$

$$\begin{array}{lll}
d_{\mathsf{mb}}\mathsf{Req}\ (\mathsf{make\text{-}box}, & \langle \mathsf{Ret}\ s, \mathsf{Ret}\ v\rangle, c) & = (\mathsf{Ret}\ c)(\mathsf{Ret}\ \#s\{v\}) \\
d_{\mathsf{ib}}\mathsf{Req}\ (\mathsf{is\text{-}box}, & \langle \mathsf{Ret}\ \#s\{v\}\rangle, c) & = (\mathsf{Ret}\ c)([[\mathsf{true}()]]) \\
d_{\mathsf{ib}}\mathsf{Req}\ (\mathsf{is\text{-}box}, & \langle ...\rangle, c) & = (\mathsf{Ret}\ c)([[\mathsf{false}()]]) \\
d_{\mathsf{bl}}\mathsf{Req}\ (\mathsf{box\text{-}label}, & \langle \mathsf{Ret}\ \#s\{v\}\rangle, c) & = (\mathsf{Ret}\ c)(\mathsf{Ret}\ s) \\
d_{\mathsf{bl}}\mathsf{Req}\ (\mathsf{box\text{-}label}, & \langle ...\rangle, c) & = (\mathsf{Ret}\ c)([[\mathsf{undefined}()]]) \\
d_{\mathsf{bc}}\mathsf{Req}\ (\mathsf{box\text{-}contents}, & \langle \mathsf{Ret}\ \#s\{v\}\rangle, c) & = (\mathsf{Ret}\ c)(\mathsf{Ret}\ v) \\
d_{\mathsf{bc}}\mathsf{Req}\ (\mathsf{box\text{-}contents}, & \langle ...\rangle, c) & = (\mathsf{Ret}\ c)([[\mathsf{undefined}()]])
\end{array}$$

On the other hand, the abstraction operation lock-box is actually definable using rec-handlers: it simply needs to redefine the other four operations so that they fail on boxes with label s within the scope of t. As a consequence, abstract values that are exported from their definition context automatically become concrete.

Chapter 4

Concurrent language features

4.1 Actors

The *actor model* [5] is an approach to concurrent programming based on asynchronous message communication between independent objects, known as *actors.* Each actor has its own thread of control which processes incoming messages. Importantly, actors share no state: all interaction is via explicit message communication. In this section, we develop a definition of actors in terms of request-based reflection.

4.1.1 Informal semantics

Operationally, an actor consists of a triple (a, q_m, b):

- a is the actor's *address*—each actor is assumed to have a unique, unforgeable address (much like a URL on the internet).
- q_m is the actor's *mailbox*—a set of messages waiting to be dispatched to the actor. In this paper, we assume a message can be any value.
- b is the actor's *behavior*—a function that accepts a message and performs some computation. When the behavior is applied to a message and the actor is engaged in computation, it is said to be *running.* An actor which is not running is said to be *ready* [to accept a message].

In addition to sequential code, an actor's behavior may use these operations:

- send(a, v)—asynchronously transmit value v as a message to the actor with address a. If the recipient is busy, the message is queued in its mailbox.
- ready(b)—causes the calling actor to become ready, with behavior b.
- newactor(b)—creates a new actor with behavior b, and returns its address.

If an actor's behavior returns without calling ready, we assume it has terminated.

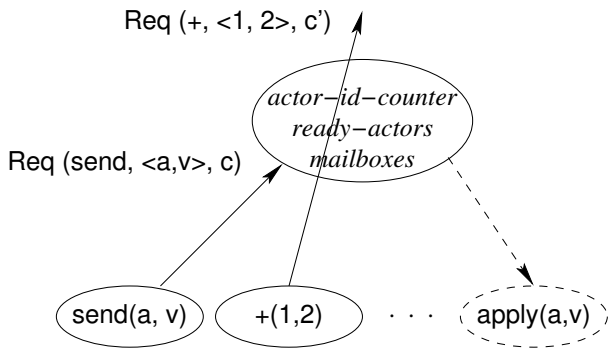


Figure 4.1: Illustration of actor computation in our system. Each of the lower terms represents an actor, while the upper term represents the implementation of actor semantics. Messages that propagate out of the upper term are implemented by the base language.

4.1.2 Defining actors

Fig. 4.1 graphically depicts an example of our construction of an actor system in terms of request messages. We have defined the actor model as described above in terms of request handlers in Fig. 4.2. For readability, we use pseudocode for the sections that are not related to reflection. The figure consists of five definitions: *actors*, *wrap*, *send*, *ready* and *newactor*. We describe the salient details of our system below.

***actors*.**

The *actors* definition serves as the main interpreter loop for an actor system. It accepts two arguments:

- a_0—the address to be assigned to the first actor created.
- t—a *quoted term* which evaluates to a sequence of newactor and send requests to set up an initial actor configuration.

The three let-ref operations define the mutable data structures for an actor configuration, *except for* the running actors. The core of the *actors* definition is the rec-handlers term. Subsequent section describe the actual handler functions.

***wrap*.**

Before we can properly handle ready requests from actors, we need to overcome one technical problem. If the term t in the *actors* expression contains a parallel composition of several actors,

```
actors(a_0, t) =
  let-ref(actor-id-counter, a_0,
  let-ref(ready-actors, empty-map(),
  let-ref(mailboxes, empty-map(),
  rec-handlers(
    on(send, send),
    on(ready, ready),
    on(newactor, newactor),
      t))))

wrap(a, b, v) =
  return ↑handler(ready,
      λ(⟨b′⟩, c).ready(↓a, b′),
      apply(↓b, ↓v))
```

```
send = λ(⟨a, v⟩, c).
  if ∄ actor with address a, return c(∅)
  if actor a is ready:
    remove (a, b) from ready-actors
    return ↑par(↓wrap(a, b, v),   ↓c(∅))
  else, add v to a's mailbox

ready = λ(⟨a, b⟩, c).
  if v = remove message from a's mailbox:
    return c(wrap(a, b, v))
  else:
    insert (a, b) into ready-actors
    return c(∅)

newactor = λ(⟨b⟩, c).
  a = fresh actor address (from actor-id-counter)
  insert (a, b) into ready-actors
  return c(a)
```

Figure 4.2: A definition of actors in terms of request handlers.

then a request from one actor, of the form Req (ready, $\langle b \rangle$, c), is ambiguous. This is because the request does not explicitly specify *which* actor is to take on the behavior b.

The *wrap* operation solves this problem. It takes an actor address a, the corresponding behavior b, and a message v to dispatch to that actor. It then constructs and returns a quoted term describing the computation to be performed by the actor, which includes a *wrapper* that transforms implicit ready requests of the form Req (ready, $\langle b' \rangle$, c) to explicit requests of the form Req (ready, $\langle a, b' \rangle$, c).

***send*.**

When a running actor calls send(a, v), it intends that the actor system send the value v as a message to the actor with address a. Within the request handler framework, the send term is transformed to a request of the form Req (send, $\langle a, v \rangle$, c), which is then serviced by the *send* definition.

An important detail is the contents of the delimited continuation c. The extent of c is the entire t argument in the rec-handlers term of the *actors* definition. Thus it includes the current state of *all* running actors, with a hole where the send term under service occurred. For example, we might have:

$$c_{\text{example}} = \lambda x.\uparrow\mathsf{par}(t_1, t_2, \mathsf{handler}(\mathsf{ready}, ..., \mathsf{seq}(\downarrow x, \mathsf{ready}(...))), t_4)$$

where t_1, t_2 and t_4 are the paused computations of other actors, and the handler(ready, ..., ...)

```
buffer-behavior =
  letrec empty = λm. cases of m:
    ⟨put, v⟩ ⇒ ready(full(v))
    ⟨get, a⟩ ⇒ ???
  and full v = λm. cases of m:
    ⟨put, v⟩ ⇒ ???
    ⟨get, a⟩ ⇒ send(a, v)
                ready(empty)
  in empty
```

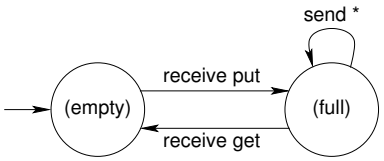


Figure 4.3: An actor behavior defining a single-element message buffer.

expression is that introduced by *wrap*.

Since the actor send operation has no return value, *send* injects $\emptyset$ into the caller and returns a quoted term that rec-handlers continues evaluating. The handlers for the ready and newactor operations are analogous to that for send; they do not introduce any new concepts.

4.2 Local synchronization constraints

This section explores an extension to the definition of actors that supports local synchronization constraints. These general constraints present a convenient alternative to hand-coded guards on messages received in actor behaviors. We start with a motivating example, and then show how code injection can be used to implement local synchronization constraints without changing the definition of the *actors* operation.

4.2.1 Example: a single-element buffer

Now that we have established a definition of the actor semantics, we can consider how a simple actor program would run under our definition. We consider an actor implementing a single-element buffer: it is either empty or holds a value. The *buffer-behavior* is defined in Fig. 4.3. For example, we can define a computation that creates two buffer actors and inserts the value 7 into one of them as follows.

```
actors(0, ↑par(newactor(buffer-behavior),
               send(newactor(buffer-behavior), ⟨put, 7⟩)))
```

However, if we tried to write variants on this program which attempt to get a value from an empty buffer, or put a value into an already-full buffer, we would run into a problem: the handlers for those cases are not defined in Fig. 4.3. We have encoded these constraints as an automaton. This

```
local-constraint = λ(⟨dfa, b0⟩, c).
  c(↑(let dfa = ↓dfa in
    let ref qm = empty-set() in
    let ref s = initial state of dfa
    letrec dispatcher = λb.
      if ∃(lbl, v) ∈ qm
       s.t. (s, receive lbl, s′) ∈ dfa:
        remove (lbl, v) from qm
        s := s′
        call-behavior(b, v)
      else, wait for a message:
        ready(λm.
          if m = ⟨lbl, v⟩ where
          ∃(s′, receive lbl, s″) ∈ dfa:
            insert (lbl, v) into qm
          else (unrecognized message):
            insert (∗, m) into qm
          dispatcher(b))
    in
      dispatcher(↓b0)))

actors-with-local-constraints(n, t) =
  rec-handler(local-constraint, local-constraint,
    actors(n, t))

call-behavior(b, v) =
  letrec f = λt.
    handlers(
      on(ready, λ(⟨b⟩, c).dispatcher(b)),
      on(send, λ(⟨a, m⟩, c).
          send(a, m)
          if m = ⟨lbl, v⟩
          and (s, send lbl, s′) ∈ dfa:
            s := s′
          f(c(∅))),
        t)
  in f(↑apply(↓b, ↓v))
```

Figure 4.4: A definition of local synchronization constraints.

local synchronization constraint [31] automaton can be used to govern message queueing.

4.2.2 Defining local synchronization constraints

As illustrated above, it is often the case that an actor can only handle certain kinds of messages in certain states. The simplest way to describe such a local synchronization constraint is as a finite-state automaton, with edge labels corresponding to messages. Since message passing in actor systems is asynchronous, we can implement a local synchronization constraint by deferring incoming messages which cannot be processed in the current state. When an actor enters a state that accepts a waiting message, that message may be dispatched. In some sense, this is a refinement of the notion ready vs. running actors.

Figs. 4.3 and 4.5 contain examples of local synchronization constraint automata. Both message sends and receives can change the state of the automaton; however, only received messages are deferred if there is no corresponding edge from the current state. Message sends not mentioned do not change the state. The definition of local synchronization constraints in Fig. 4.4 is described below.

***actors-with-local-constraints*.**

We extend the base *actors* operation with a new operation called local-constraint. Interestingly, this doesn't require any changes to the definition of *actors*. Because our base definition of actors does not recognize an operation called local-constraint, corresponding requests escape the rec-handlers term of the *actors* operation, and can be caught by *actors-with-local-constraints*. This is the same mechanism by which sequential code within an actor executes: the computation requests are not caught by the definition of actors, so they default to whatever definition is available in the underlying base language.

***local-constraint*.**

Because the local-constraint handler catches requests emitted by *actors*(n, t), delimited continuation c encompasses the entire actor system. Hence *local-constraint* is global across all actors; it does not execute within the context of one actor. But we would like *local-constraint* to execute within the context of an actor, wrapping a supplied behavior with some additional code to enforce local synchronization constraints.

The definition of *local-constraint* solves this problem by injecting code into the calling actor. This is accomplished by passing a quoted term to the delimited continuation c, which yields a quoted term containing the full actor configuration, with the quoted code inserted into the calling actor. When we resume execution of the actor system, the injected code is executed. Our use of generative programming here is analogous to aspects that activate on a join point in aspect-oriented programming [48].

The code injected by *local-constraint* creates a new behavior from the user-specified initial behavior b_0 and a local synchronization constraint automaton, *dfa*. Two mutable variables are created that are local to the generated behavior's closure: q_m is the deferred message set, and s is the actor's current state. The behavior itself consists of a *dispatcher* loop, which does the following:

1. *dispatcher* checks for deferred messages that can be handled in the current state. If any are available, an arbitrary message is dispatched to the user-specified behavior (see the description of *call-behavior* below).

2. Otherwise, *dispatcher* waits for an incoming message; when it receives one, it adds it to q_m and loops back, to try to dispatch it. We use label $*$ to denote messages that do not have a label explicitly mentioned in the automaton.

***call-behavior*.**

The *call-behavior* definition is reminiscent of *wrap* in Fig. 4.2: it dispatches a message to a behavior in a controlled manner:

- ready is intercepted to ensure that the code injected by *local-constraint* maintains control of the actor when user code invokes ready.
- send is intercepted to allow state transitions triggered by outgoing messages. The handler recurses, since we have used a handlers rather than rec-handlers.

All other operations—for example, newactor or local-constraint are not intercepted by *call-behavior*; they pass through and are handled by the existing lower-level definitions.

4.3 Meta-actors

Thus far, we have restricted our extensions to operate within the context of the actor being extended, or as a shared platform beneath all actors. Actor systems have traditionally proposed another arrangement, in which a meta-actor supervises the external activity of an object-level actor. Meta-actors are a general mechanism that has been used to implement a variety of coordination constructs. In the following two sections, we show how to define meta-actors using the request-based reflection mechanism.

As part of this exercise, we consider two different, but related ways in which actors can be assigned meta-actors. In Sec. 4.3.1, we statically assign a meta-actor to an actor at creation time, using a newactor-meta operation in place of newactor. In Sec. 4.3.2, we generalize to allow dynamic assignment and reassignment of meta-actors to extant actors. Our designs allow a number of other points of flexibility: a meta-actor may service either a single actor, or a group, or all actors in a system; and meta-actors may be stacked, allowing an object-level actor's meta-actor to be supervised by a meta-meta actor. In Sec. 4.3.3, we consider how these meta-actor architectures may be used as a platform for coordination services.

4.3.1 Statically-assigned meta-actors

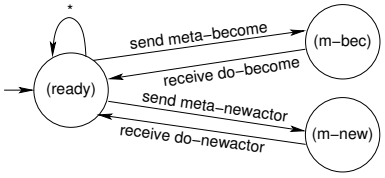
The left column of Fig. 4.5 defines a newactor-meta operation and associated helpers, which allow an actor to call newactor-meta(a, i, b) to create a new actor with behavior b that is supervised by meta-actor a (the new actor is identified by index i, in case meta-actor a is already supervising

```
actors-with-newactor-meta(n, t) =
  rec-handler(newactor-meta, newactor-meta,
    actors-with-local-constraints(n, t))

newactor-meta = λ(⟨ma, i, b0⟩, c).
  c(↑newactor(local-constraint(dfa_nm,
    let (ma, i) = ↓(ma, i) in
    let ref b = ↓b0 in
    letrec f = λm. cases of m:
      ⟨do-dispatch, ma, m′⟩
        ⇒ f′(↑apply(↓b, ↓m′))
      m ⇒ send(⟨meta-message, i, m⟩)
    and f′ = λt.
      rec-handlers(
        on(send, onsend),
        on(ready, onready),
        on(newactor, onnewactor),
          t))
    in f)))

onsend = λ(args, c).
  send(ma, ⟨meta-send, i, args⟩)
  return c(∅)
onready = λ(args, c).
  send(ma, ⟨meta-ready, i, args⟩)
  ready(λm = ⟨do-ready, ma, b′⟩.
          b := b′
          ready(f))
onnewactor = λ(args, c).
  send(ma, ⟨meta-newactor, i, args⟩)
  ready(λm = ⟨do-newactor, ma, a′⟩.
          f′(c(a′)))

dfa_nm =
```




```
actors-with-newactor-meta-dyn(n, t) =
rec-handler(newactor-meta, newactor-meta-dyn,
      actors-with-local-constraints(n,
        rec-handler(newactor,
          λ(⟨b⟩, c).c(newactor-meta(∅, ∅, b)), t)))

newactor-meta-dyn = λ(⟨ma0, i0, b0⟩, c).
  c(↑newactor(local-constraint(dfa_nmd,
    let ref (ma, i, b) = ↓(ma0, i0, b0) in
    letrec f = λm. cases of m:
      ⟨message, m′⟩ ⇒
        if ma = ∅, f′(↑apply(↓b, ↓m′))
        else, send(ma, ⟨meta-message, i, m′⟩)
      ⟨do-setmeta, ma, ma′⟩ ⇒ ma := ma′; ready(f)
      ⟨do-ready, ma, b′⟩ ⇒ b := b′; ready(f)
      ⟨do-dispatch, ma, m′⟩ ⇒ f′(↑apply(↓b, ↓m′))
    and f′ = λt.
      rec-handlers(
        on(setmeta, onsetmeta),
        on(send, onsend’),
        on(ready, onready’),
        on(newactor, onnewactor’),
          t))
    in f)))

onsetmeta = λ(⟨ma′, i′⟩, c).
  if ma = ∅:
    (ma, i) := (ma′, i′)
    f′(c(∅))
  else:
    send(ma, ⟨meta-setmeta, i, ⟨ma′, i⟩⟩)
    ready(λm = ⟨do-setmeta, ma, ma″⟩.
            ma := ma″
            f′(c(∅)))

onsend’ = λ(⟨a, m⟩, c).
  if ma = ∅, return c(send(a, ⟨message, m⟩))
  else, return onsend(⟨a, m⟩, c)
onready’ = λ(⟨b′⟩, c).
  if ma = ∅, b := b′; ready(f)
  else, onready(⟨b′⟩, c)
onnewactor’ = λ(⟨b′⟩, c).
  if ma = ∅, return c(newactor-meta(∅, ∅, b′))
  else, return onnewactor(⟨b′⟩, c)

dfa_nmd =
```

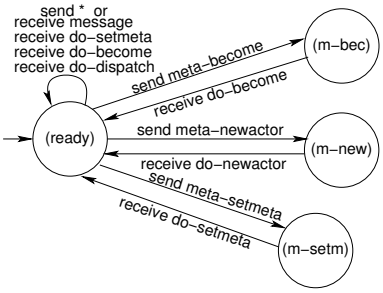


Figure 4.5: Two different definitions of meta-actors. The definition in the left column fixes an actor's meta-actor at creation time. The right column eases this constraint, allowing dynamic redefinition of an actor's meta-actor via the new setmeta operation.

some other actors). When we say a meta-actor a_m *supervises* another actor a_o, we mean that whenever a_o attempts to perform an external action (send, ready, or newactor) this action is not directly performed. Instead, a message is sent from a_o to a_m, and the meta-actor is given the responsibility of actually performing the action—or doing something else instead. The supervised actor a_o accepts certain command messages that allow its meta-actor a_m to manipulate the internals of a_o's state.

***actors-with-newactor-meta*.**

Similar to *actors-with-local-constraints* (Sec. 4.2.2): handles the newactor-meta operation, which is not caught within *actors-with-local-constraints* or *actors*.

***newactor-meta*.**

Like *local-constraint* (Sec. 4.2.2), it injects code into the calling actor. The injected code adapts the user-specified behavior b_0, wraps the adapted behavior in a local synchronization constraint specified by dfa_{nm}, and creates a new actor with that behavior. We'll call that new actor a *supervised actor.*

The adapted behavior contains a fixed meta-actor address ma and actor index i, and a mutable behavior b, initialized to the user-specified behavior b_0. Normally, a supervised actor only accepts a $\langle$do-dispatch, $ma, m'\rangle$ message from its meta-actor ma, which cause the supervised actor to dispatch message m' to its current behavior b. This dispatch is within a rec-handlers expression, which allows us to mediate all send, ready and newactor requests made by b.

***onsend*, *onready*, and *onnewactor*.**

These definitions handle actor operations by a supervised actor, relaying them to its meta-actor. Since the ready and newactor operations return results, they are implemented *synchronously,* by waiting for reply messages. We use local synchronization constraints to prevent unrelated messages from interrupting these synchronous primitives. Note that the synchronous behavior within *onready* and *onnewactor* is only safe because the code for these handlers has been injected into an actor. Had we left the handlers at the level of the *newactor-meta* definition, it would be impossible for the meta-actor to actually execute while *onready* or *onnewactor* is waiting.

4.3.2 Dynamically-reassignable meta-actors

A clear drawback of the above formulation of meta-actors is we cannot assign a meta-actor to an extant unsupervised actor, or dynamically change the meta-actor assigned to a supervised actor. One solution is to create a proxy meta-actor along with each actor, which relays messages to the actor's *current real meta-actor*. Another is to make the *ma* and *i* variables in our previous definition mutable; this is presented in the right column of Fig. 4.5 and described below.

***newactor-meta-dyn*.**

The definition of *newactor-meta-dyn* is very similar to *newactor-meta*. This is both satisfying and vexing: we have made a small evolutionary change to introduce a useful feature, but that change is not composable as a separate piece of code that can be applied to *newactor-meta* (compare to the clean composition of *actors*, *actors-with-local-constraints*, and *actors-with-newactor-meta*). This is because we are redefining local behavior that is not directly exposed by *newactor-meta*.[1] The major changes besides making the meta-actor address *ma* and supervised actor index *i* into mutable variables are:

1. The new setmeta operation allows an actor to assign itself a meta-actor.
2. The meta-actor can now command the supervised actor with do-setmeta and do-ready messages in the ready state.

***on...* handlers.**

Unlike the handlers in Sec. 4.3.1, the handlers here have two different cases: direct manipulation, when the current actor has no meta-actor assigned; and a message to the meta-actor if one is present. The *onsetmeta* handler is synchronous because the new meta-actor must be set before the supervised actor performs any further external operations. The *onnewactor'*, handler is interesting because the unmediated case is *not* a call to the base-level newactor operation. Instead, it creates an unsupervised actor that can later adopt a meta-actor.

***actors-with-newactor-meta-dyn*.**

This is perhaps the trickiest definition in this section. The outer rec-handler term introduces the definition of *newactor-meta-dyn*. As discussed above, *onnewactor'* ensures that when an

[1] Technically, even very simple operations like lambda and variable accesses are exposed to lower levels of abstraction. But catching and redefining these simple, general operations to effect very specific changes is a non-solution.

unsupervised actor created by *newactor-meta-dyn* calls newactor, the newly-created actor is actually created using *newactor-meta-dyn*, so that it has the option of calling setmeta later to adopt a meta-actor. This definition extends the same property to the code in the term t that sets up the initial actor configuration.

One may wonder whether there is an infinite loop in this definition, since the inner rec-handler defines newactor as a call to newactor-meta, while the outer rec-handler defines newactor-meta to inject code that calls newactor. This is not a bug because two *different* definitions of newactor are in scope within t vs. within the handler $c(\mathsf{newactor\text{-}meta}(\emptyset, \emptyset, b))$. The handler evaluates as follows:

$$c(\mathsf{newactor\text{-}meta}(\emptyset, \emptyset, b)) \rightarrow c(\mathsf{newactor}(\mathsf{local\text{-}constraint}(...)))$$

Because the call to newactor is evaluated *before* substituting its result into the delimited continuation c, it is not caught by the inner rec-handler expression; instead, newactor is serviced by the base definition in *actors*.

4.3.3 Using meta-actors for coordination

To use our actor meta-architecture for coordination, we need to design meta-actors to supervise hos object-level actors interact. A common choice is to assign a unique meta-actor to each actor. In this case, we factor out a coordination protocol into the meta-actor behaviors: object-level actors define application logic, while meta-actors define coordination policy.

Example: a replica protocol.

We can use this pattern to define a master-backup replication protocol, as in [4]. Assume we are trying to build a replicated service, which is made available to several clients. The clients are written to communicate with a single actor address, while the service is implemented by several back-end replicas, which are instances of the same service. We need to be able to do three things:

- *Intercept all messages sent from a client to a replicated service, and forward them to the replicas.* The client's meta-actor serves as a forwarder.
- *Intercept all reply messages sent from the service replicas back to the client, discarding duplicates.* This could be done by the client's meta-actor, or amongst the replicas' meta-actors. In the latter case, if one replica already has a result, we may choose to terminate the other replicas' computations and update their state. Without a reflective base

language, one must change the meta-architecture to allow a meta-actor to interrupt a computation. In our framework, a meta-actor can add this ability by injecting supervisor code that wraps the behavior of an object-level actor.

- *Coordinate access to shared resources by the replicas.* This is important when a replicated service needs to use another service that has persistent state. This is actually the dual of a service replication protocol: now our replicas are multiple *clients* acting as one. In a master-backup arrangement, only the master's meta-actor would actually send its requests, with the results broadcast to all replicas.

In this scenario, we need to modify all clients to transform service requests into requests to multiple replicas. Creating a separate meta-actor for each client would lead to a large number of meta-actors to coordinate. Another approach is *group reflection,* in which a meta-actor uniformly modifies a collection of actors. For example, all clients on a single host could use the same replica-access meta-actor. We could even override the newactor operation, as in Sec. 4.3.2, to assign all actors the replica-access meta-actor by default.

One potential downside with group reflection is that some clients may actually need custom meta-actors. For example, this is the case when a replicated service acts as a client to another service. This calls for an ability to *stack* meta-actors. The simplest approach is to allow meta-actors to be managed by other meta-actors—which our definitions do allow. But there are other approaches. For example, we might consider meta-actors as something like our handler expressions, where certain messages are caught and sent to certain meta-actors. This can be done by wrapping the code injected by *newactor-meta* with a rec-handlers expression that catches and reroutes meta-actor messages.

On the other hand, one may also want to inject a stack of handlers into an object-level actor, rather than using a meta-actor at all. In practice, this would be useful for packaging existing imperative code as actors. We could use reflection to transform direct I/O requests into communication with services. This is similar to what full-system virtualization is used for today.

Example: generating meta-actors from synchronization constraints.

Meta-actors simplify direct implementation of coordination constructs by allowing us to factor out synchronization policy from application code. But implementing the policy itself can still require significant work. Research on synchronization languages has focused on generating meta-actor behaviors from higher-level specifications. A reflective programming language is a natural fit for

the generative approach. As we have seen in the pseudocode figures, quasiquoted program terms can be used as code templates, with other code snippets inserted into unquoted positions. For example, $\uparrow\mathsf{apply}(\downarrow b, \downarrow v)$ generates a quoted term that, when evaluated, computes $b(v)$.
Often, synchronization constraints may include code snippets which act as predicates on messages and distributed state. For example, we may wish to dispatch a grant-lock message to actor C only if actors A and B are in a state where they will not need the lock to proceed. How can we compute the state predicate on A and B?

1. Use reflection to capture each actor's state as a delimited continuation.

2. Evaluate the predicate within some meta-actor. Requests for properties of A and B must be forwarded to their corresponding meta-actors. We can catch these requests using a handlers expression.

3. Evaluate the predicate fragments related to a given actor within a copy of that actor's continuation. We can use request handlers to catch and ignore operations that have side effects, like send.

4. If the predicate evaluated to true, dispatch the message. In any case, we discard the copies of the actors' continuations used to compute the predicate, since computation of the predicate itself is not allowed to change the system state, and the copies become out of sync with the actual state of the actors as computation proceeds.

4.4 Evaluation

In the previous sections, we have defined actors as an extension to our base language with a notion of reflection and simple parallel composition; we then extended this actor language with local synchronization constraints, and meta-actors. How effective has our notion of reflection been for creating these extensions? We revisit criteria from the introduction:

How invasive is the extension?

The definitions of actors, local synchronization constraints, and meta-actors are all non-invasive in the sense that they required no modifications to the language constructs upon which they build. This is primarily thanks to two language features:

- Request propagation. Requests for operations propagate out until they get to a corresponding definition. We used this to sandwich new features around old features. For example, actors requesting the local-constraint operation are placed *inside* a context managed by the *actors* definition, while the definition of local-constraint is wrapped around the *outside* of the *actors* term. This same pattern was used with the definitions of meta-actors.

- Code injection. As part of the sandwiching pattern, the definition of, say, local-constraint outside of the actor configuration injected code into the requesting actor within the *actors* term.

On the other hand, the definition *newactor-meta-dyn* is invasive in the sense that it is a modified version of *newactor-meta*, rather than a compositional extension. This is because we needed to change the behavior of operations whose requests did not escape *newactor-meta*—for example, reading the current address of the meta-actor, *ma*.

Overall, different extensions are composable, while variations on the same theme may require some code duplication. This tradeoff seems reasonable, as it is consistent with how we actually develop code.

How faithfully does it express the desired construct?

In all cases, we were able to implement the desired constructs. Most cases were straightforward, because we either defined new operations, or we created handlers that locally redefined lower-level operations. (*e.g.,* trapping ready requests within an actor).

The most difficult case was redefining newactor in Sec. 4.3.2. We needed to introduce a global definition (applicable to all actors), but the *actors* construct already catches and services newactor requests. Hence we could not introduce the operation outside of an *actors* term. Our solution was to wrap the seed term *t* with our redefinition of newactor, and to "infect" new actors with this definition as they are created. In other words, we introduced a form of early or lexical binding, to compliment the message propagation model's dynamic scoping.

How well do the different extensions work together?

The different extensions compose, and build upon each other. All of the constructs build on actors; and the definition of meta-actors builds upon local synchronization constraints. In practice, it is necessary to avoid naming conflicts when independently developing and composing extensions.

The redefinition of the newactor operation in Sec. 4.3.2 indicates that different definitions of the same operation can coexist. However, this phenomenon can be confusing.

An interesting case to consider is whether actors with local synchronization constraints can be supervised by meta-actors. After all, the definition of a supervised actor itself makes use of local synchronization constraints. Fortunately, this is not a hindrance. Since the local-constraint operation is a behavior transformer, we can write code like newactor-meta($ma, i,$ local-constraint(dfa, b)). Since the meta-actor layer symmetrically labels and unlabels messages when interacting with the user-supplied behavior, we have a protocol stack, with "on the wire" messages like $\langle$message, $\langle$put, $v\rangle\rangle$ and $\langle$do-dispatch, $\langle$get, $a\rangle\rangle$.

Chapter 5

Implementation in Javascript

The request message-based execution model studed in this thesis has been implemented in the form of a reflective Javascript interpreter, itself written in Javascript. Introduced in 1995 in Netscape Navigator, today Javascript is the universal client-side scripting language available in all major web browsers. Together with its twin ActionScript (essentially a variant of Javascript used in Adobe Flash), Javascript is by far the most popular language for mobile code. However, the current means for supervising and mediating the execution of Javascript code are rather primitive. Today, you generally have to choose *either* secure code isolation *or* powerful cross-domain scripting; reflective code mediation would be useful to allow a hybrid of the two. This is examined in Chapter 6.
This chapter describes the implementation of the message-based execution model in Javascript. The prototype interpreter allows us to execute Javascript code, and to dynamically redefine the behavior of Javascript constructs within a scope. This case studay serves an examples of how a variety of different language features (such as those described in Chapter 3) can be implemented in terms of request messages in practice. We also consider implementation strategies for a low-level interpreter (in C or assembly language), and the factors that inherently limit the performance of implementations of request-based execution.

5.1 The Javascript programming language

Javascript is a multi-paradagim programming language, drawing inspiration from languages like the C family (syntax), Self (prototype-based object inheritance), and Scheme (first-class functions, closures and tail recursion). It is a dynamically-checked language that offers a combination of imperative, functional, and object oriented constructs. Despite the word “script” in common, Javascript is not related to Unix shell scripting languages. Later sections in this chapter will assume familiarity with Javascript, both from a programming perspective and a language implementation perspective.

5.1.1 Variables

The statement `var x;` declares a new variable `x`. Variables are scoped to the nearest enclosing *function,* not the enclosing *block.* For example, this is valid Javascript code:

```
function (x) {
    if (x > 0) {
        var y = 3;
    } else {
        var y = 7;
    }
    return y;
}
```

5.1.2 Functions

In Javascript, functions are objects that encapsulate executable code. This section describes the basics of functions; the section on objects includes additional details.

Named functions. *Named* functions look similar to C functions or Java methods, but without type annotations *(proposed upcoming Javascript versions do include optional type annotations),* and with the addition of the `function` keyword:

```
function f(...) {
    ...
}

function addPair(x, y) {
    return x + y;
}
```

Function bodies consist of zero or more statements, separated by semicolons or newlines *(Javascript parsers use a heuristic to insert missing semicolons at newlines—this is usually more trouble than it's worth).* Return statements are optional.

```
function doStuff(x, y) {
    var sum = x + y;
    alert(sum); // Displays the value of sum
    // retun x + y; // This line is commented out.
}
```

Nested functions. Function definitions may appear inside of other functions. In that case, they are (lexically) locally scoped.

```
function outer(x) {
```

```
    return inner(x, x);

    function inner (y, z) {
        return y * z;
    }
}
```

Notice that the function definition may appear before or *after* its use.

Anonymous functions. Same syntax, but without a name.

```
function (...) {
    ...
}

function (x, y) {
    return x + y;
}
```

An anonymous function is an *expression* which can be used as a *value.* The name of a named function can also be used as a value. Assigning an anonymous function to a variable isn't *quite* the same as a function declaration—this wouldn't work correctly:

```
function outer(x) {
    return inner(x, x);

    var inner = function (y, z) {
        return y * z;
    };
}
```

The solution would be to assign to `inner` before its use.

Anonymous functions can be assigned *local* names to allow recursion, without introducing the function's name into the surrounding namespace. To do so, simply use a named function as an expression rather than a statement.

```
var f = function myfun(x, y) {
    if (x > 0) {
        return myfun(x-1, y+1);
    } else {
        return y;
    }
};

//    typeof f     == "function"
// && typeof myfun == "undefined"
```

Closures. Nested functions capture variables in their lexical scope, creating a *closure*. The variables referenced by the closure are *mutable.*

```
function makeFn() {
    var counter = 0;
    return function () {
        counter++}
        return counter;
    }
}

var f = makeFn();
var g = makeFn();

f(); // => 1
f(); // => 2
g(); // => 1
f(); // => 3
g(); // => 2
```

The `arguments` variable. Sometimes it's useful to write functions that accept a variable number of arguments. In C, this is done via the `stdarg` stack-inspection mechanism. In Scheme, this is done by binding a variable to the `cdr` position of the argument list. In Javascript, the list of arguments to a function is bound to the `arguments` variable.

```
function concat() {
    var str = '';
    for (i = 0; i < arguments.length; i++) {
        if (str != '') {
            str += ' ';
        }
        str += arguments[i];
    }
    return str;
}
var s = concat('welcome', 'to', 'javascript'); // s = 'welcome to javascript'
```

This is particularly useful when writing higher order functions that need to wrap the behavior of other functions.

5.1.3 Conditionals

The conditional constructs in Javascript are much like those in C and Java, both in syntax and semantics. Javascript inherits the following C-family conditional constructs:

```
if (guard) { then-clause }        if (guard) { then-clause } else { else-clause }
guard ? then-expr : else-expr
switch (expr) { case string: ... break; ... default: ... }
for (init; guard; incr) { body }
while (guard) { body }            do { body } while (guard);
```

Note that the *guard* expressions are considered to *fail* if their result is equivalent (via the `==` operation) to either `false` or `null`. This includes values such as `0`, `''` (the empty string), and `undefind` (value of an uninitialized variable). Any other value for a guard expression is considered as a *pass,* regardless of whether it is equivalent (`==`) to `true`. See Section 5.1.4 for more details. Also note that `switch-case` statements match on *strings* rather than *integers*; integer values are automatically coerced to strings as necessary.

In addition to the above constructs, Javascript introduces a *for each* loop, which iterates over the names of the fields in an object (including fields inherited from its prototype); and the `with` construct, which extends the local environment with fields of a given object. The syntax is as follows:

```
for (var field-name in object-expr) { body }
with (object-expr) scope
```

For example, the program on the left and the porgram on the right are equivalent.

```
var x = 1;                  | var x = 1;
var obj = { z: 2 };         | var obj = { z: 2 };
with (obj) {                |
    var y = x + z;          | var y = x + obj.z;
}                           |
```

5.1.4 `undefined`, `null` and comparison

Javascript offers *two* different equality comparison operators: `==` (double-equals: *equivalent*) and `===` (triple-equals: *identical*). A variable declared with `var` initially has value `undefined`, which is *different* from the value `null`. However, `==` treats the two as *equivalent:*

```
undefined == null    // true
```

```
undefined === null    // false
```

Some other values are also considered as equivalent by `==`, for example:

```
'' ==  0              // true
'' === 0              // false
```

Object instances are *always* considered as distinct, even if they have the same fields:

```
({a:1}) == ({b:2})    // false
({ })   == ({ })      // false
```

5.1.5 Other operators

Other operators in Javascript are mostly the same as those of C and Java. For example, arithmetic, bitwise, and assignment operations. There are a few important differences, though.

Strings and concatenation. Strings can be surrounded by single or double quotes; characters are not distinguished from strings. The `+` operation is used for string concatenation, as well as arithmetic. Numbers are automatically coerced to strings as necessary.

```
'Hello ' + "World"; // => 'Hello World'
0 + 'abc';          // => '0abc'
1 + 2;              // => 3
```

Array literals. Array literals are defined using brackets (not braces as in C or Java). Arrays may include holes, which default to `undefined`. Arrays are simply objects with fields named `0`, `1`, `2`, etc., along with a `length` field. Example:

```
var a = [11,'z',,3,,,'abc',9];
a[0] // => 11
a[1] // => 'z'
a[2] // => undefined
// a.length == 8;
for (var i = 0; i < a.length; i++) {
    // do something with a[i]
}
```

The `typeof` operator. The `typeof` operator allows you to dynamically determine the type of a value. As in Java, values in Javascript are *either* objects or primitives. For example, `typeof 'hello' == 'string'` and `typeof 3 == 'number'`, while `typeof { } == 'object'` and `typeof null == 'object'` . Other type names include `'function'`, `'undefined'` and `'boolean'`.

5.1.6 Objects

Object literals. A Javascript object is fundamentally a key-value assocation. Javascript is dynamically typed, and does not use classes. Object literals are denoted as follows.

```
var obj = { field1: expr1, field2: expr2, ... };
```

Object literals cannot be used directly in contexts where they may be confused with code blocks. In those cases, surround an object literal in parentheses.

Accessing fields. An object field may be accessed via either dot-notation (similar to C structures) or bracket-notation (similar to C arrays). Strings are used as indices into the fields of an object. Numbers may be used as indices as well, but they will be coerced to decimal strings.

```
obj.field1
obj['field2']
obj[3] // same as obj['3']
```

Note that when using dot-notation, you can only use field names that are proper identifiers; but when using bracket-notation, field names may be arbitrary strings.

Methods. A method in Javascript is simply a function stored in an object field. Methods are called using dot-notation. An object's methods can also be assigned, removed, or updated dynamically, just like any other object field.

```
obj = { };
obj.y = 1;
obj.m = function (x) { return 2*x + this.y; }
var seven = obj.m(3);
```

The `this` variable. Within a method, the associated object is dynamically bound to the `this` variable. Because methods are simply functions stored as object fields, the value of `this` depends on the context in which a function/method is called. The three different cases are:

1. plain functions
2. functions used as methods
3. functions used as constructors

Function not used as a method: `this` is bound to the global scope object. In a browser, the global scope is stored in the `window`

object.

```
function f() {
    alert(this == window);
}
f(); // displays "true"
```

When a function is stored as a field of an object, it is treated as a method. When the method is invoked, the `this` variable will point to the object.

```
var obj = {
    m: function () { alert(this.x); },
    x: "hello"
};
obj.m(); // displays "hello"
```

However, if the method is considered as a field, and its value is stored to a separate variable, it is treated as a standard function again: `this` points to the global scope (the `window` object).

```
var m = obj.m;
window.x = "goodbye";
m(); // displays "goodbye"
```

The `call` and `apply` methods. Now say you have an object, and a function you want to use as a method of that object. One way to do so is to store the function in a field of the object. However, if you don't want to modify the object (or if you want to hide the method from other code that has access to the object), what do you do?
In Javascript, functions are themselves objects. Function objects have methods `call()` and `apply()` that allow you to explicitly specify the value of `this` to use.

```
function g(y, z) {
    return this.x + y + z;
}
var obj = { x: 3 };
```

In order to invoke `obj.g()`, we'd need to update `obj`.

```
// obj.g = g;
// var a = obj.g(4, 5); // a = 12
```

But if we don't want to modify `obj`, we can use
`g.call()`,

```
var b = g.call(obj, 4, 5); // b = 12
```

Or we can use `g.apply()`, especially if the number of arguments can vary.

```
var args = [4, 5];
var c = g.apply(obj, args); // b = 12
```

Constructors. In Javascript, constructors are simply functions. To use a function as a constructor, it is called using the "`new`" operator. Doing so binds a fresh object to "`this`". The `new` expression automatically returns the newly-allocated object, after allowing the constructor to modify it as desired. By convention, constructors are named to describe the kind of object they initialize.

```
function MyObject(name) {
    this.name    = name;
    this.counter = 0;
}
var mo = new MyObject("bob");
// result:
   mo.name    == "bob"
&& mo.counter == 0;
```

In particular, object literal expressions implicitly call the `Object` constructor—*i.e.,* the following two code sequences are equivalent.

```
// object literal
var obj = { x: 7 };

// or equivalently, call the Object constructor
var obj = new Object();
obj.x = 7;
```

Prototypes. Javascript doesn't provide classes for structuring object-oriented programs *(classes are proposed for a future version of the language, but ar not necessary to implement object inheritance).* Instead, each object has an associated *prototype object* which specifies the default values of fields not stored directly in the object. The prototype is set when an object is constructed.

```
function Point() { }
Point.prototype.x = 0;
Point.prototype.y = 0;

var obj = new Point();
// result: obj inherits fields from p.
//    obj.x == 0
// && obj.y == 0;
```

If you set a field in an object, that will *override* the value stored in the prototype; however, it will not *modify* the prototype object itself. This is similar to how a local variable can shadow a global

variable in most languages; in fact it makes Javascript objects very convenient for implementing chained environments in an interpreter.

```
obj.x = 3;
// result: p.x unchanged
//    obj.x == 3
// &&   p.x == 0;
```

On the other hand, changing fields of the prototype object directly *will* affect all objects that reference that prototype, *unless* they have already overridden those fields.

```
p.x = 5;
p.y = 6;
p.z = 7;
// result:
//    obj.x == 3 // Because obj.x was already set to a custom value
// && obj.y == 6
// && obj.z == 7;
```

A common application of prototypes in Javascript is to encapsulate a collection of methods shared by a class of objects. The prototype mechanism is provided as an alternative to classes in other object-oriented programming languages. For example, we may define two different constructors which share one common prototype, as follows:

```
function Point() { }
function PointAt(x, y) {
    this.x = x;
    this.y = y;
}
PointAt.prototype =
Point.prototype = {
    constructor: Point,
    x: 0,
    y: 0,
    add: function (pt) {
        this.x += pt.x;
        this.y += pt.y;
    }
};

var pt = new PointAt(2, 3);
pt.add(pt);
// result:
//    pt.x == 4
// && pt.y == 6;
```

The `instanceof` operation and `.constructor` field. The `instanceof` operation allows you to check whether a given object's prototype is the same as a given *constructor*'s `.prototype` field

object, or is an *ancestor* of that object. For example:

```
   obj   instanceof Point   == true
&& obj   instanceof PointAt == true
&& obj   instanceof Object  == true
&& ({ }) instanceof Point   == false;
```

Every prototype object has a `.constructor` field which is supposed to point to the constructor function used to define objects which have that prototype. A function's initial `.prototype` object includes a `.constructor` field pointing to the function itself. If one changes the function's `.prototype` field (as in the above example of `Point` and `PointAt`), one must also set the `.constructor` field in the new prototype object. For example:

```
function f() { }
var a = new f();
// a.constructor == f;

f.prototype = { };
var b = new f();
//    a.constructor == f
// && b.constructor != f;

f.prototype.constructor = f;
// b.constructor == f;
```

5.1.7 Exceptions

Any object can be thrown as an exception. As in Java (and unlike the Common Lisp condition system, or our request messages), Javascript exceptions are non-restartable. The syntax is similar to Java The keywords are `throw`, `try`, `catch` and `finally`. A `try` block must be followed by either a `catch` clause or a `finally` clause, or both. For example:

```
try {
    if (b)
        throw { name: "something" };
} catch (e) {
    alert(e.name);          // displays "something" if b == true
    throw e;                // raises the exception again
} finally {
    alert("finished try");  // always displays "finished try"
}
alert("no exception");      // only displays "no exception" if b == false
```

5.2 General implementation strategy

Our interpreter is divided into two phases: first, we parse an input string, producing a request message describing the entire input program. Then we evaluate the program by servicing its individual request messages one-by-one. The traversal order is determined by the definitions of language constructs; for example, evaluation of arguments to functions is sequenced. Request and return messages are direct transliterations of the execution model introduced in Chapter 2; their constructors are as follows:

```
function RequestMessage(head, args, cont) {
    this.head = head;
    this.args = args;
    this.cont = cont;
    return this;
}
function req(head, args, cont) { return new RequestMessage(head, args, cont);}

function ReturnMessage(val) {
    this.val = val;
    return this;
}
function ret(val) { return new ReturnMessage(val); }
```

Given a request message that has reached the base level of the system, it is evaluated by looking up the operation name (`mesg.head`) in a table of message handlers, and dispatching one of them. The base level message handlers are grouped together like one big rec-handlers expression. Each handler returns a new request message describing the rest of the behavior of the *entire* system—a zero-argument full continuation (or perhaps one argument, if you consider the handler environment in which the message is serviced as an argument). The main interpreter loop is implemented by the `baseHandler` function:

```
function baseHandler(mesg) {
    if (mesg instanceof RequestMessage) {
        var handler = baseEnv[mesg.head];
        if (handler) {
            return baseHandler(handler(mesg.args, mesg.cont));
        }
    }
    return mesg;
}
```

5.2.1 Interactive evaluation

Notice that if the program returns a value (a `ReturnMessage`) or it requests an operation for which there is no defined handler, the message is simply returned by `baseHandler`. Such messages could be presented to the user (for example, to manually simulate an unimplemented operation), and handled by an even lower-level interpreter. The next message following servicing of an unhandled message can then be fed back to the `baseHandler` function to continue evaluation. In order to support this interactive evaluation feature—as well as multiple concurrent interpreters within the same Javascript environment—it is important that handler functions do not maintain any state in global variables (or persistent variables local to the interpreter loop—hence there are none). As such, the request messages handled (or returned) by `baseHandler` encapsulate the entire state of a program.

5.2.2 The post operation

Earlier, in Chapter 3, we discussed how various language features can be defined in terms of request messages. Now we turn to how request handlers are implemented at the base level. One of the request handler functions included in the `baseEnv` table implements the post operation, which provides the feature of call-by-value evaluation. To review, the term $\mathsf{post}(\mathsf{h}, e_1, ..., e_n)$ means we should evaluate the subterms $e_1, ..., e_n$ in order, producing the values $v_1, ..., v_n$; and then evaluate the term $\mathsf{h}(v_1, ..., v_n)$, in place of the whole post term. This is implemented in Javascript as follows:

```
baseEnv['post'] = function (args, cont) {
    for (var i = 1; i < args.length; i++) {
        if (args[i] instanceof RequestMessage) {
            var post_cont = function (m) {
                var newargs = args.slice(); // copy the args array
                newargs[i] = m;
                return req('post', newargs, id);
            }
            post_cont.handles = { };
            return applyCont(cont,
                             req(args[i].head,
                                 args[i].args,
                                 composeCont(post_cont, args[i].cont)));
        }
    }
    return applyCont(cont, req(args[0].val, args.slice(1), id));
};
function id(m) { return m; }
id.handles = { };
function doPost(head, args, cont) {
```

```
        return applyCont(cont, req('post', [ret(head)].concat(args), id));
    }
```

Where `composeCont` and `applyCont` act as function composition and application operations; they are defined in more detail in Section 5.4.

The for-loop above iterates over the argument subterms and evaluates them, while the return statement after the loop rewrites the post term to $\mathsf{h}(v_1, ..., v_n)$. Within the loop, the if-statement finds the first subterm which requires further evaluation. The call to `applyCont` substitutes a new request message into the program's continuation, effectively rewriting the post term to a request to evaluate a subterm. That request includes a delimited continuation which stores the other subterms and ensures control will return to the post operation. This is accomplished by wrapping whatever delimited continuation came with the subterm in the `post_cont` delimited continuation, which generates a new post request containing the updated subterm.

The `doPost()` function is simply a shortcut used in the following sections, much like `req()` and `ret()`. The line "`post_cont.handles = { };`" in the post request handler is an optimization discussed in Section 5.4, and can safely be ignored for now.

For example, the `RequestMessage` objects corresponding to the first three steps of evaluating the abstract term $\mathsf{post}(\mathsf{add}, 1(), 2())$ are as follows (assuming the term $1()$ evaluates to the numeric value `1`):

$\mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ \mathsf{add}, \mathsf{Req}\ (1, \langle\rangle, \lambda x.x), \mathsf{Req}\ (2, \langle\rangle, \lambda x.x)\rangle, \lambda x.x)$ is represented by:

```
{ head: 'post',
  args: [{ val: 'add' },
         { head: '1', args: [ ], cont: id },
         { head: '2', args: [ ], cont: id }],
  cont: id }
```

$\mathsf{Req}\ (1, \langle\rangle, \lambda m.\mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ \mathsf{add}, m, \mathsf{Req}\ (2, \langle\rangle, \lambda x.x)\rangle))$ is represented by:

```
{ head: '1',
  args: [ ],
  cont: function (m) {
            return { head: 'post',
                     args: [{ val: 'add' },
                            m,
                            { head: '2', args: [ ], cont: id }],
                     cont: id }; } }
```

$\mathsf{Req}\ (\mathsf{post}, \langle \mathsf{Ret}\ \mathsf{add}, \mathsf{Ret}\ 1, \mathsf{Req}\ (2, \langle\rangle, \lambda x.x)\rangle, \lambda x.x)$ is represented by:

```
{ head: 'post',
  args: [{ val: 'add' },
         { val: 1 },
         { head: '2', args: [ ], cont: id }],
```

```
cont: id }
```

(Note that the body of the continuation in the second request message has been pre-evaluated for readability.)

5.3 Implementations of key Javascript constructs

This section describes how various Javascript constructs are implemented in our interpreter. This serves as both a more concrete analogue to and a validation of the methods of Chapter 3. Each language construct is defined by a collection of operations, and where appropriate, a representation for related runtime values (such as environments, objects, and function closures). Since our extended Javascript language includes support for user-defined request handlers, *any* of the operations can be arbitrarily dynamically overridden within a scope.

5.3.1 Primitive values

Since Javascript is both our host and target language, Javascript primitive values like booleans, numbers, strings, `null` and `undefined` can simply be identity-mapped. In order to differentiate between `null` and `undefined` values in the interpreter and in the interpreted code, all Javascript primitive values are wrapped in `JSValue` objects:

```
function JSValue(jsValue) {
    this.jsValue = jsValue;
}
var jsUndefined = new JSValue(undefined);
var jsNull = new JSValue(null);
```

Quoted code (technically, quoted request and return messages) and delimited continuations reified as values within the target language are similarly wrapped in `JSQuoted` and `JSCont` objects.

5.3.2 Conditional statements: `while` loops

We will discuss the definition of the while operation and related accessories; other conditional statements are defined analogously. The interpreter parses a Javascript `while` statement such as

`while (`*guard-expr*`)` *body*

into the request message

Req (while, $\langle$*guard-expr*, *body*$\rangle$, $\lambda x.x$)

which is represented by an object of the form

{ head: 'while', args: [*guard-expr*, *body*], cont: id }

The while operation is then implemented starting with the following reduction.

$$\mathsf{while}(g, b) \quad \Rightarrow \quad \mathsf{post}(\mathsf{post\_while}, g, \uparrow g, \uparrow b)$$

which evaluates the guard g in the current context, but uses quotation to save unevaluated copies of the guard g and body b of the while loop for use in subsequent iterations. Assuming the guard evaluates to a success value v (`true`, a non-zero number, a non-empty string, a non-`null` object, etc.), the term is then reduced through the following two steps.

$$\mathsf{post\_while}(v, \uparrow g, \uparrow b) \quad \Rightarrow \quad \mathsf{seq}(b, \mathsf{while}(g, b))$$

where the seq operation simply evaluates its arguments from left to right, returning the value `undefined` as its result (represented in the interpreter by `jsUndefined`). Incidentally, seq is also used to implement Javascript's "{ ... }" blocks Similarly, when the while loop terminates, it returns `undefined`.

5.3.3 Environments

As mentioned in Section 5.2.1, the entire state of a program must be contained in a request message that arrives at the base level. Thus variable bindings must be stored in explicit environment-binding terms. We define the operation with_env, used in the form with_env(*env*, *body*), which binds an environment *env* within the code of *body*. The with_env construct is then defined to evaluate *body*, intercepting and servicing requests that need to access the environment, while passing other requests (or a final return value) through to lower levels.

Before we discuss how environment-related requests are handled, let's consider how other requests are passed through. This is implemented by the following code snippet (compare this to `baseEnv['post']`).

```
baseEnv['with_env'] = function (args, cont) {
    var env = args[0].val;
    var body = args[1];

    if (body instanceof ReturnMessage) {
        return applyCont(cont, body);
    }

    var we_cont = function (m) {
        return req('with_env', [ret(env), m], id);
    };
    we_cont.handles = makeSet('declare', 'post_set', 'post_get', 'function');

    // (Handle declare, post_set, get, and function here.)
```

```
        return applyCont(cont,
            req(body.head, body.args, composeCont(we_cont, body.cont)));
    };
```

In the above, a return message issued by the body *replaces* the with_env term, while any other request message *passes through* the with_env term: the `return` statement on the last line propagates the request message from the body, saving the environment in the message's delimited continuation field (`with_env_cont`). (As with post, the `with_env_cont.handles` line is an optimization, to be discussed in Section 5.4.)

As written above, `baseEnv['with_env']` simply evaluates *body*. Now we introduce the requests that with_env intercepts and services. The basic environment-related functions in Javascript are: declaring a new variable (via the `var` keyword), getting and setting the value of a variable, and creating a function closure which captures the current environment. These are represented by the following four forms:

with_env(*env*, ...declare(*var-name*)...)

with_env(*env*, ...set(*var-name*, *expr*)...)

with_env(*env*, ...get(*var-name*)...)

with_env(*env*, ...function(*bound-var-name*, *body*)...)

Notice that the *expr* argument of set(*var-name*, *expr*) must be evaluated, while both of the arguments of function(*bound-var-name*, *body*) must *not* be evaluated. Since argument subterms are not automatically evaluated in our framework, function is correct as-is, while set must go through an extra evaluation step. The handlers for the different messages intercepted by the with_env operation are defined below. This code snippet replaces the comment in `baseEnv['with_env']` above.

```
    // Handle declare, post_set, get, and function:
    var cc = composeCont(cont, composeCont(we_cont, body.cont));
    switch (body.head) {
    case 'declare':
        env.declare(body.args[0].val.jsValue);
        return applyCont(cc, ret(jsUndefined));
    case 'get':
        return applyCont(cc, env.get(body.args[0].val.jsValue));
    case 'set':
        return doPost('post_set', args, cc);
    case 'post_set':
        env.set(body.args[0].val.jsValue, body.args[1].val);
        return applyCont(cc, body.args[1]);
    case 'function':
        return applyCont(cc, ret(new JSFunction(env, body.args[0].val,
```

```
                                                    body.args[1])));
}
```

Environment objects. In accordance with Javascript semantics, the environment object `env` is structured as a linked list of activation records, with the head of the list containing variables defined in the scope of the current function, the next entry describing the lexically containing scope, and so on. Environment objects are created via the `JSEnvironment` constructor:

```
function JSEnvironment(parentEnv) {
    this.parentEnv = parentEnv;
    this.bindings = {};
}
```

The `JSEnvironment.prototype` defines three methods for manipulating variables:

`declare(`*var-name*`)` adds a new variable to the local environment, with initial value `jsUndefined`.

`get(`*var-name*`)` searches through the list of activation records and returns the value of the *first* entry named *var-name*. But what happens if the variable is not defined? According to the definition of Javascript, a *ReferenceError* exception should be thrown. We implement this by *returning a request message* Req (throw, ⟨Ret `exnReferenceError`⟩, $\lambda x.x$). This is why, in the "`case 'get':`" clause above, the *message* returned by `env.get(...)` is *inserted into* the composed continuation `cc`.

`set(`*var-name*`,` *value*`)` simply updates the value of a variable. If the variable does not already exist, it is added to the *global* scope.

5.3.4 Objects (including arrays)

Javascript includes several operations related to objects: object literal expressions, field access, and object construction using `new`. We will first define a representation for objects, and then describe how each of the operations on objects is serviced.

Representation of objects. All objects in the interpreted Javascript world are implemented as `JSObject` objects in the interpreter. The structure of a `JSObject` is quite similar to a `JSEnvironment`; for example, the constructor is as follows:

```
function JSObject(prototype) {
    this.prototype = prototype;
```

```
    this.fields = {};
}
```

However, the methods defined in `JSObject.prototype` have slightly different semantics from the environment accessor methods (this is purely a consequence of the definition of the Javascript language). If a particular field is not found in a `JSObject`, `get(`*field-name*`)` returns `jsUndefined` (rather than throwing an exception); and `set(`*field-name*`,` *value*`)` always writes to fields of the object itself, not to its prototype.

Object literal expressions. Object and array literal expressions in Javascript are parsed as follows:

Javascript code	**parsed term**
{ f_1 : e_1 , ..., f_n : e_n }	object(field(f_1, e_1), ..., field(f_n, e_n))
[e_1 , ..., e_n]	array(e_1, ..., e_n)

An object request is evaluated by simply (a) creating a new empty object (`obj = new JSObject()`), and then (b) iterating through the field requests, evaluating the expressions e_i to values v_i and inserting new bindings into the representation of the object (`obj.set(`f_i`,` v_i`)`). Since the e_i must be evaluated in the context where the object expression appears, we use a convention similar to that used by post and with_env to pass requests made by the e_i terms through. array is implemented similarly.

Getting and setting fields. There are several forms for getting and setting object fields in Javascript. In general, we have the following four cases:

Javascript code	**parsed term**
object-expr`.`*field-name*	get_field(*object-expr*, *field-name*)
object-expr`[`*field-name-expr*`]`	get_field(*object-expr*, *field-name-expr*)
object-expr`.`*field-name* `=` *value-expr*`;`	set_field(*object-expr*, *field-name*, *value-expr*)
object-expr`[`*field-expr*`]` `=` *value-expr*`;`	set_field(*object-expr*, *field-name-expr*, *value-expr*)

The get_field and set_field operations simply call the `get` or `set` method of a `JSObject`, respectively. Both operations use post to implement call-by-value semantics.

Constructing objects with `new`. Objects constructed with `new` are similar to object literals. The key differences are we need to (a) set the new object's prototype to that specified by the constructor's `prototype` field, and (b) invoke the constructor on the newly-created object. If we assume the new object resulting from step (a) is called *obj*, this means we reduce a new request to

an apply_method request:

$$\text{new}(constructor, args...) \quad \Rightarrow \quad \text{apply\_method}(constructor, obj, args...)$$

In code, this is as follows:

```
baseEnv['new'] = function (args, cont) {
    return doPost('post_new', args, cont);
};
baseEnv['post_new'] = function (args, cont) {
    var prototype = args[0].val.get('prototype');
    if (prototype instanceof JSValue) {
        prototype = prototype.jsValue;
    }
    var newObj = new JSObject(prototype);
    return applyCont(cont,
        req('apply_method', [args[0], ret(newObj)].concat(args.slice(1)), id));
};
```

Here the `if` statement is necessary to transform the objects `jsNull` and `jsUndefined` to the corresponding native values `null` and `undefined`. Also notice that since we are able to inject an apply_method request into the caller, the implementation of `new` (along with all the other operations on objects) is completely independent of the implementation of functions.

5.3.5 Functions and methods

We've already seen two operations related to functions and methods: function and apply_method. Functions in Javascript source code are parsed to terms as follows:

Javascript code	**parsed term**
`function (`*vars...*`)` { *body* }	function($\langle vars... \rangle$, $\hat{body}$)

Where $\hat{body}$ is the parsed form of *body*. The function operation, which creates a closure, is implemented by with_env, since the closure needs to capture the current environment. It is reproduced below:

```
case 'function':
    return applyCont(cc, ret(new JSFunction(env, body.args[0].val,
                                            body.args[1])));
```

Since a function request has the form Req (function, $\langle$Ret $\langle vars... \rangle, body\rangle, \lambda x.x$), the above code creates a new `JSFunction` from the current environment, the bound variable names, and the body of the function. Note that since functions in Javascript are defined to be objects, `JSFunction` inherits from `JSObject`.

Function and method application. Our interpreter provides two application operations: apply (for ordinary functions) and apply_method. In Javascript, a method is a function stored as a field of an object. The only special thing about a method call is the associated object is automatically assigned to the local `this` variable. Thus apply is simply implemented as a reduction to apply_method:

apply(f, *args*...) ⇒ apply_method(f, `jsUndefined`, *args*...)

A term apply_method(*method-expr*, *obj-expr*, *args*...) is evaluated in five steps, as described in the code below:

```
baseEnv['apply_method'] = function (args, cont) {
    // 1. Evaluate method and object expressions and all arguments
    return doPost('post_apply_method', args, cont);
};
baseEnv['post_apply_method'] = function (args, cont) {
    var f = args[0].val;
    var thisObj = args[1].val;
    var funArgs = args.slice(2);
    // 2. Create the local environment
    var env = new JSEnvironment(f.parentEnv);
    // 3. Bind the argument values to argument variables
    for (var i = 0; i < f.boundVars.length && i < funArgs.length; i++) {
        env.declare(f.boundVars[i].jsValue);
        env.set(f.boundVars[i].jsValue, funArgs[i].val);
    }
    // 4. Bind the object to the this variable
    env.declare('this');
    env.set('this', thisObj);
    // 5. Evaluate the function body in the local environment
    return applyCont(cont,
        req('with_return', [req('with_env', [ret(env), f.code], id)], id));
};
```

The last step is the most interesting: first, it creates a with_env request which will take care of evaluating the body of the function within the newly-created local environment. Second, it nests this within a with_return request, which, as we shall see, implements the `return` construct. Finally, this request is injected into the calling context, so that when the whole expression reduces to a return value, that value is automatically returned to the caller. Another interesting characteristic of this structure is if certain operations were overridden in the caller via user-defined request handlers (Section 5.3.7, those handlers also apply to the called function—*i.e.,* it ensures handlers have *dynamic scope*.

The with_return operation. The with_return operation implements Javascript's `return` construct. A term of the form with_return(*body*) is evaluated by passing through all non-return request messages from *body*. If *body* issues a return request, the argument is evaluated in the calling context within *body*, and the whole with_return term then evaluates to the result value. On the other hand, if *body* simplifies to a value without issuing a return request, then the with_return term simplifies to that value. So we have the two cases:

with_return(...return(*expr*)...) ⇒ with_return(...post(post_return, *expr*)...) ⇒

... ⇒ with_return(...post_return(*value*)...) ⇒ *value*

and

with_return(*value*) ⇒ *value*

The implementation of with_return is similar to that of with_env, but simpler: only post_return requests are intercepted.

5.3.6 Exceptions

Throwing an exception in Javascript is quite similar to returning from a function. We know that "`return` *expr*`;`" delivers the result of evaluating *expr* to the nearest dynamically enclosing function call site, discarding the intervening delimited continuation. Similarly, "`throw` *expr*`;`" delivers the result of evaluating *expr* to the `catch` or `finally` clause of the nearest dynamically enclosing `try` block, again, discarding the intervening delimited continuation. As such, the definitions of the operations try_catch_finally and throw are analogous to the operations with_return and return described in the previous section.

5.3.7 User-defined request handlers

In addition to the standard features of Javascript, we have extended the language with support for user-defined request handlers. These are provided by the handlers and rec-handlers operations, as introduced in Chapter 2. We will present the implementation of handlers in detail; rec-handlers is analogous. Our implementation generalizes the implementation structure of post and with_env described above.

Recall that the usage pattern for handlers is as follows:

$$\mathsf{handlers}(\mathsf{on}(h_1, f_1), ..., \mathsf{on}(h_n, f_n), g, t)$$

where the pairs (h_i, f_i) specify operation names and corresponding request message handler

functions used to mediate the execution of term t, while the function g is used to handle a value returned by term t. As before, request messages that do not match one of $h_1, ..., h_n$ are propagated out of the handlers term, while the result of the first matching request handler is returned by the handlers term (*i.e.,* handlers operates in a single-shot mode, while rec-handlers mediates a stream of requests).

Our Javascript implementation of handlers follows. We begin with the case that the body term t simply returns a value:

```
baseEnv['handlers'] = function (args, cont) {
    var body = args[args.length-1];
    if (body instanceof ReturnMessage) {
        return applyCont(cont, req('apply', [args[args.length-2], body], id));
    }
```

Since the return handler g is the second to last argument to handlers, the above applies g to the value returned by the body. On the other hand, if the body emits a request message, we need to decide whether that message is one handled by this handlers term, and either dispatch the corresponding handler function or propagate the message out of the handlers term, as follows.

```
    handler_funcs = { };
    for (var i = 0; i < args.length-2; i++) {
        handler_funcs[args[i].args[0].val] = args[i].args[1];
    }
    var f = handler_funcs[body.head];
    if (f) {
        return applyCont(cont, req('apply',
            [f, encodeArray(body.args), encodeCont(body.cont)], id));
    } else {
        var handlers_cont = function (m) {
            var newargs = args.slice();
            newargs[newargs.length-1] = m;
            return req('handlers', newargs, id);
        };
        handlers_cont.handles = handler_funcs;
        return applyCont(cont,
            req(body.head, body.args, composeCont(handlers_cont, body.cont)));
    }
};
```

The then-clause of the `if` statement above, dispatches a request handler function. The `encodeArray` and `encodeCont` helper functions encode the array `body.args` and the function `body.cont` as a `JSObject` and a `JSFunction`, respectively. The else-clause propagates the request message outward, much like in the definitions of post and with_env.

5.3.8 Reified messages and continuations

Handlers written in the Javascript object language require access to reified representations of request and return messages and continuations. These are provided using two additional datatypes, `JSQuoted` and `JSCont`, together with a collection of constructor and deconstructor operations. The operations are analogous to the functions with corresponding names already introduced. We need to add the operations deconstructMessage and cont because unlike the interpreter itself, the object language does not have direct access to the concrete representations of messages and continuations. Operations on messages:

> req$(h, args, k)$ — constructs a quoted `RequestMessage`, *i.e.,* $\uparrow$Req $(h, args, k)$
>
> ret(v) — constructs a quoted `ReturnMessage` which returns the value v, *i.e.,* $\uparrow$Ret v
>
> deconstructMessage$(qm, on\text{-}req, on\text{-}ret)$ — checks whether qm is a quoted request message or a quoted return message, and calls either the function *on-req* or *on-ret,* respectively, in tail position (or neither, if qm is not a message). *on-req* is called with three arguments, while *on-ret* is called with one argument, corresponding to the arguments to the constructors req and ret.

Operations on delimited continuations:

> id() — returns the identity delimited continuation
>
> cont$(h, \langle args_1 ... \rangle, \langle args_2 ... \rangle, k)$ — constructs a single-level delimited continuation with a hole between $args_1$ and $args_2$, *i.e.,* returns the delimited continuation $(\lambda m.\uparrow$Req $(h, \langle args_1 ..., \downarrow m, args_2 ... \rangle, k))$. Note that if the hole is nested several levels deep, the delimited continuation must be constructed using a combination of cont and composeCont.
>
> applyCont(k, qm) — applies the delimited continuation k to the quoted message qm, *i.e.,* $k(qm)$
>
> composeCont(k_1, k_2) — composes two delimited continuations, *i.e.,* $k_1 \circ k_2$

All of the above operations return `null` on invalid arguments. (Throwing an appropriate exception would be another reasonable implementation choice.) Syntactically, uses of the above built-in operations appear like function calls, except the operation name is prefixed by an "`@`" character. For example, `@id()` or `@ret(7)` or `@req('some_op', [@ret(7)], @id())`.

5.4 Performance considerations

Given that our language provides the ability to override the definition of any construct within a scope at any time, it is natural to expect that execution performance may suffer. In fact, the initial experimental implementations of the interpreter described in this chapter (and its precursors) suffered an asymptotic slowdown on even simple programs!

For example, consider a program that adds 2^n copies of the number `1`, using tree of binary addition operations (`1+1`, or `(1+1)+(1+1)`, or `((1+1)+(1+1))+((1+1)+(1+1))`, or...). Such a program should operate in a number of steps linear in the number of ones. Surprisingly, these programs were found to operate in a number of steps *quadratic* in the number of ones!

Why is this? The key problem was that after each evaluation step of some inner term, each outer term (its parent term, its grandparent term, and so on, up to the root term) were given an opportunity to take a step. This is an artifact of a naive approach to applying a continuation to a request message. If we assume a continuation is simply a term with a hole in it, inserting a request message into the continuation fills the hole; but the next message extracted from the continuation will be the request message to evaluate the *root* term. For example, in the following, in the middle of multiplying the numbers 2 and 3, the addition operation is given an opportunity to regain control:

$$
\begin{aligned}
&[[+(*(2(),3()),*(4(),5()))]] \Rightarrow \ldots \\
&\Rightarrow (\lambda m.[[\mathsf{post}(\mathsf{post}+, m, *(4(),5()))]])\ (\mathsf{Req}\ (\mathsf{post}, \langle \mathsf{post}*, 2, 3(), \lambda x.x\rangle)) \\
&\Rightarrow [[\mathsf{post}(\mathsf{post}+, \mathsf{Req}\ (\mathsf{post}, \langle \mathsf{post}*, 2, 3()\rangle, \lambda x.x), *(4(),5()))]] \\
&= \mathsf{Req}\ (\mathsf{post}, \langle \mathsf{post}+, \mathsf{Req}\ (\mathsf{post}, \langle \mathsf{post}*, 2, 3()\rangle, \lambda x.x), *(4(),5()))\rangle, \lambda x.x)
\end{aligned}
$$

In many other cases similar to the above, a continuation only needs to be able to intercept certain kinds of messages. For example, the post operation only handles return messages; with_env only handles declare, get, set, post_set and function requests; and handlers only handles those operations in its list of on-clauses. Thus when we try to apply a continuation c to a request message m that the continuation does not need to intercept, what we should *actually* do is return an updated version of m in which the continuation c is composed with m's delimited continuation.

This optimization requires us to keep track of which requests a (delimited) continuation handles; that is exactly the purpose of the `handles` field of the delimited continuation functions in the Javascript code throughout the previous sections. We now define the composeCont and applyCont functions, which implement the optimization in the interpreter.

```
function composeCont(f, g) {
```

```
    var h = function (x) {
        return applyCont(f, applyCont(g, x));
    }
    h.handles = union(f.handles, g.handles);
    return h;
}

function applyCont(c, t) {
    if (t instanceof RequestMessage
    &&  c.handles && !c.handles[t.head]) {
        return req(t.head, t.args, composeCont(c, t.cont));
    } else {
        return c(t);
    }
}
```

Note that the above code optimizes in terms of the number of interpreter steps. Of course, if the overhead of checking and maintaining the `handles` sets of delimited continuations is greater than the cost of executing extra steps, then actual execution time will suffer. Thus, this is best considered as a *model* of a performance optimization—or an optimization in terms of conciseness of program execution traces.

Possible further performance improvements are discussed in Section 8.2.

Chapter 6

Applications

This chapter considers how we can use the Javascript dialect extended with request handlers to prototype solutions to various problems. Given the choice of Javascript, our problems are relevant to client-side web programming. We take as motivation the need to sandbox widgets on web pages (Section 6.1). We then create request handlers that implement prototypes of time and memory resource limits (Section 6.2) and mediated communication with the rest of the web page and other servers (Section 6.3).

6.1 Sandboxed widgets on web pages

The *World Wide Web* has quickly grown from a collection of hyperlinked documents to a diverse application platform. In the first generation of web apps, all application logic resided on the server side, with the client web browser serving as a display mechanism. The introduction and standardization of the *Javascript* programming language and the *Document Object Model* gradually allowed developers to migrate application code to the client side.

6.1.1 Relationship to the rest of this thesis

The security models provided by operating systems and other programming languages serve as good inspiration for security within a web page. One example is the request-based execution model studied in this thesis, which attempts to allow programmers to limit or override the behavior of programs within a given scope. Since there are many ongoing attempts at improving client-side web security, this serves as a worthwhile problem for assessing the usefulness of the request-based execution model. Because we have already implemented an extensible Javascript interpreter *in* Javascript, we can use this as a basis for a prototype implementation of sandboxed Javascript code that runs in a web page.

6.1.2 Widget architecture

A *widget* is a chunk of user interface functionality. For example, scroll bars, weather maps, text editor fields, videos, search boxes, and so on. As discussed above, in this section, we are interested in creating manageable widgets for embedding in web pages. Security is a particular issue when a widget originates from a different administrative domain than the host page. But even within a single administrative domain, well-defined widgets make it easier to define and revise user interfaces.

More specifically, we can characterize the requirements for widgets as follows. It must be easy instantiate a widget anywhere on a web page. It must be possible to create multiple independent instances of the same kind of widget without any special care. Widgets should be nestable and composable. The source-level definition of a collection of widgets should be similar to HTML: easy to write, easy to edit. Operationally, widgets on different parts of the page must not be tightly coupled. It should be easy for a program to dynamically change the contents of a page: add, remove and reconfigure widgets.

A good way to design widgets is as objects whose interaction with the rest of the world is mediated by their parent context. This allows a widget's parent to reconfigure it, choose what events trigger activity in the widget, and customize the effects of events signaled by the widget. This corresponds closely to general idea of mediated request messages studied in this thesis. To solve the problem, we need to identify how to perform the following specific mediation patterns.

- Execution time limits: allow the code within a widget to run for a bounded amount of time and then be preempted.
- Memory resource limits: limit the number of objects a widget can allocate.
- Limit the ability to access the HTML document's DOM tree: we need to create proxies for DOM objects representing the part of the page controlled by the widget, while blocking access to parts of the page outside the widget's scope.
- Limit the ability to communicate with a server: this includes both sending data to and receiving data from the originating server and other servers. Communication can occur indirectly, by creating DOM objects representing things like images to be loaded into a web page, as well as directly through XMLHttpRequest objects.

The following sections describe how to prototype each of these features using request mediation.

6.2 Resource limits

Execution time and memory usage limits are useful for ensuring that an errant widget does not cause the rest of a web page to become unresponsive. This is particularly important on the web, since third party widgets included in a page may be changed by their developers at any time. This section considers the use of mediated execution to enforce time and memory constraints.

6.2.1 Execution time limits

Execution time limits are conceptually the simplest extension to our Javascript dialect. This is because request messages already provide us with a convenient preemption mechanism. Although the base Javascript language does support asynchronous timers, it does *not* support timer interrupts: timer events may only fire when the Javascript interpreter is idle. As a simple alternative, we can count the number of request messages that a subterm makes, and preempt it after a certain limit. Our syntax is as follows:

time_limit(*num-steps*, *on-preempt*, *on-return*, *body*)

where *num-steps* is (an expression that evaluates to) some integer number of execution steps, and *body* is a term which we will run for *num-steps*-many steps (or fewer, if it returns a value first). Either the function *on-preempt* or *on-return* is called in tail position when we preempt *body* or *body* returns a value, respectively.

Definition as an extension to the Javascript interpreter. First, we will examine how we would implement timed preemption as an extension to the request-based Javascript interpreter. This will follow a similar form as the definitions of post and with_env in Chapter 5. Later, we will consider how to implement it using the `handler` construct instead.
As with earlier definitions (such as the if statement), the definition of time_limit is separated into (a) a phase where we pre-evaluate the arguments *num-steps, on-preempt* and *on-return;* and (b) a phase called post_time_limit, where we mediate the execution of *body.* As shown below, step (a) is accomplished using the post operation, while quoting the *body* argument.

```
baseEnv['time_limit'] = function (args, cont) {
    return applyCont(cont,
        req('post', [ret('post_time_limit'), args[0], args[1], args[2],
                     ret(new JSQuoted(args[3]))], id));
};
```

The post_time_limit operation implements most of the behavior of the time-limited execution policy.

```
baseEnv['post_time_limit'] = function (args, cont) {
    var num_steps  = args[0].val.jsValue;
    var on_preempt = args[1];
    var on_return  = args[2];
    var body       = args[3].val.codeTerm;
    if (body instanceof ReturnMessage) {
        return applyCont(cont,
            req('apply', [on_return, body], id));
    } else if (num_steps > 0) {
        function cont2(m) {
            return req('post_time_limit',
              [ret(num_steps-1), on_preempt, on_return, ret(new JSQuoted(m))],
              id);
        }
        return applyCont(cont,
            req(body.head, body.args, composeCont(cont2, body.cont)));
    } else {
        return applyCont(cont,
            req('apply', [on_preempt, args[3]], id));
    }
};
```

Recall that `args` is an array of request or return messages; since post_time_limit is called *after* evaluating the arguments, we assume they are all return messages. Hence `args[0].val.jsValue` retrieves the number of steps requested (assuming it is indeed a numeric value), and `args[3].val.codeTerm` unquotes the *body* term. The three clauses of the `if-else` correspond to the three cases: (i) *body* has returned a value; (ii) *body* can be executed for at least one more step; (iii) *body* has more steps remaining, but we're out of time and must preempt it. Case (ii) is the most intricate: we must propagate *one* message from *body,* but catch its next message; this is analogous to the rethrow operation introduced in Section 2.2.1. It is also similar the pattern used in the definitions of post and with_env, except now we potentially need to catch *any* message, rather than only messages with certain heads.

Definition in terms of handlers. As discussed earlier, it is preferable that we do not have to extend the core Javascript interpreter. The handlers operation and its variants should instead be used to implement language extensions. Here we show how to define time-limited execution in terms of handlers.

The first phase—preevaluating arguments—is similar to the previous definition. The names of functions in the interpreter have simply been changed to names of corresponding built-in operations provided by the interpreter. We use the syntax `@`*op-name* in our Javascript extension,

as introduced in Section 5.3.8.

```
function on_time_limit(args, cont) {
    return @applyCont(cont, @req('post',
        [@ret('post_time_limit'), args[0], args[1], args[2], @ret(args[3])],
        @id()));
};
```

The post_time_limit phase is a bit different from the earlier definition, mainly because the object-language Javascript code has an opaque interface to the structure of quoted terms. Here we use the operation deconstructMessage as a dual to the quoted message constructors, req and ret; and also the cont operation, used to build continuations. Given these changes, the handler for post_time_limit can be written as follows.

```
function on_post_time_limit(args, cont) {
    var num_steps = 0;
    @deconstructMessage(args[0],
        function () { },
        function (value) {
            num_steps = value;
        });
    var on_preempt = args[1];
    var on_return  = args[2];
    var body = null;
    @deconstructMessage(args[3],
        function () { },
        function (value) {
            body = value;
        });
    @deconstructMessage(body,
        function (body_head, body_args, body_cont) {
            if (num_steps > 0) {
                var cont2 = @cont('post_time_limit',
                    [@ret(num_steps-1), on_preempt, on_return], [], @id());
                return @applyCont(cont, @req(body.head, body.args,
                    @composeCont(cont2, body.cont)));
            } else {
                return @applyCont(cont,
                    @req('apply', [on_preempt, @ret(body)], @id()));
                    // Note that "@ret(body)" is equivalent to "args[3]"
            }
        },
        function (value) {
            return @applyCont(cont,
                @req('apply', [on_return, @ret(value)], @id()));
                // Note that "@ret(value)" is equivalent to "body"
        });
};
```

Notice that use of the explicit message deconstructor increases the code size, but for good reason:

it forces us to consider the case that we do not have a quoted return message where one is expected. The time_limit and post_time_limit operations defined above can be enabled within a scope using the rec-handlers form built into our extended Javascript dialect, as follows:

```
@rec_handlers({time_limit:      on_time_limit,
               post_time_limit: on_post_time_limit},
              ...);
```

where "`...`" is the context in which we would like to be able to use the new form `time_limit(...)`

6.2.2 Discussion

The execution time constraints provided by time_limit as implemented in Section 6.2.1 are very much like an instruction count implemented using single-step debugging mode on a processor. This provides the flexibility to preempt executions after a precise number of requests, or at a specific point. However, executing explicit checks in response to each request invariably results in poor performance. In practice, if time_limit turned out to be a useful operation, the solution would be to implement it as a built-in operation in the base interpreter, perhaps using native timer interrupts. This highlights an interesting feature of the request-based execution model: since users cannot distinguish requests implemented by handlers within the language from requests implemented by underlying native code, it is possible to transition from either one to the other.

6.2.3 Memory resource limits

In addition to time, memory is the other major internal resource limitation on computations. There are two main differences between time and memory constraints. First of all, a program's memory usage is not monotonic: some operations increase total memory usage, while others do not affect memory usage, or even decrease it. Secondly, if we know which operations potentially affect memory usage and which do not, we can choose to intercept only those that affect memory usage, allowing all other operations to execute directly.

It is relatively easy to put a bound on the number of allocations that a Javascript program has made, using a technique similar to the time_limit construct, but limited to counting memory-allocation operations such as variable declaration, object creation, and function calls. Unfortunately, in a system using implicit memory management—for example, garbage collection or reference counting—it is difficult to determine whether a given operation that *may* decrease memory usage *actually does*. Thus without access to the underlying memory manager, we cannot

determine the program's current total memory usage. As with time constraints, we could implement a crude prototype of memory usage constraints using request mediation, but production-quality memory constraints would need to be integrated into the language implementation.

In the specific case of a web page containing multiple widgets, we have the added complexity that an object may be allocated by some widget A, and then transferred to another widget B. In this case, it can be unclear against which widget's budget a particular allocated object should be counted. Even if A has discarded all references to the object, it is possible that the object may contain more data than B expects or requires. Solving this problem requires a sufficiently detailed protocol specification.

6.3 Mediated communication

The two key ways we can modify the behavior of a program at the langauge level are (a) how its internal operations evolve the state of the program, and (b) how the program is allowed to interact with other entities. The first case was introduced in Chapter 3, and the second in Chapter 4. In the case of Javascript embedded on a web page, the three mains of interaction between scripts are:

- global variables
- the HTML Document Object Model
- communication with web servers

Manipulation of the environment and mediated access to global variables has already been discussed in Chapters 3 and 5. We discuss the other concerns below.

6.3.1 Limiting DOM tree access

The DOM is a tree data structure representing the full contents of the web page [81]. All nodes in the DOM tree are represented as Javascript objects, which contain browser-defined fields (*e.g.,* a description of the display style used to render a given element) and methods (*e.g.,* appendChild() to insert a new sub-element within an existing element on the web page). As with other Javascript objects, scripts are free to add new fields into a DOM object or modify existing values to restructure the page.

The ability to modify the DOM clearly must be limited in order to prevent multiple independent scripts embedded in the same page from conflicting. Moreover, even reading from certain parts of the page should be restricted—for example, an advertisement needn't know a person's user name when logged into a website that happens to display interactive ads.

Since the DOM consists of a collection of objects that model the HTML page, this problem amounts to creating proxy objects that model just part of the page. Implementing the proxies strictly as library code in Javascript is not possible without changing the DOM API. This is for two reasons: (a) since Javascript object fields do not have access control, any object fields can be read or modified, including pointers to the underlying DOM nodes; and (b) the DOM API has certain fields that act as *properties:* simply modifying the field value executes code that changes some aspect of the underlying HTML page.

Our solution is to introduce an extension to Javascript that supports *proxy objects,* which implement two methods, `getField` and `setField`. These methods are called instead of directly accessing the fields of an underlying object. The extension operates as a handler, which executes client code in a custom context. The following Javascript operations (as introduced in Chapter 5) are locally redefined:

post_new(...) — This operation needs to create a non-proxy object. Since non-proxy objects could coincidentally include fields called `getField` and `setField`, we need to use a special encoding. A simple choice is to create what we call a *default proxy object,* which includes a single field, named `object`, that points to the actual object constructed with the base-level implementation of post_new.

post_get_field(*obj*, *field-name*) — If *obj* is a proxy object, this operation needs to call its `getField` method, while if it is a default proxy object, it simply needs to look up the value in *obj*`.object`:

```
function on_get_field(obj, field_name) {
    if (obj.getField) {
        return obj.getField(field_name);
    } else {
        return obj.object[field_name];
    }
}
```

post_set_field(*obj*, *field-name*, *value*) — If *obj* is a proxy object, this operation needs to call its

`setField` method, while if it is a default proxy object, it simply needs to set the value in *obj*`.object`:

```
function on_get_field(obj, field_name, value) {
    if (obj.setField) {
        obj.setField(field_name, value);
    } else {
        obj.object[field_name] = value;
    }
    return value;
}
```

Given the ability to run code with these three overridden handlers in place, limiting DOM tree access simply amounts to creating appropriate proxy objects that provide a view into the full HTML DOM. In practice, however, there is one additional twist: each DOM element may have a unique id. If a script only has access to a subset of the DOM tree, it may create an element with an id that conflicts with another element elsewhere. This can be alleviated by associating a prefix string with all element ids created by a given script. Since we cannot trust a script to consistently use its assigned prefix, we associate some additional private information with each restricted script context, much like how the fields of a proxy object are not directly visible to client code.

6.3.2 Limiting communication with servers

At first glance, web standards already require a "same origin" policy for scripts contacting servers. However, as it turns out, there are many ways for a script to indirectly send information to an arbitrary server, or inject scripts from a foreign server into the web page. Moreover, since scripts can be fetched from a server other than the origin of the host HTML page, we may not want a particular script to even be able to access data on the HTML origin host.
The three basic ways in which script running on a standard web browser may cause communication with a server are as follows.

1. Create an `XMLHttpRequest` object and call its `send` method, which sends an HTTP request to an arbitrary URL on the HTML origin server. The response is provided via an asynchronous event.

2. Create a new HTML DOM `img`, `iframe`, `object` element. These cause external data to be loaded from an arbitrary URL, which may be used to transmit arbitrary data to an arbitrary host.

3. Create a new HTML DOM `script` element. A script element will execute code fetched from an arbitrary URL on an arbitrary remote host, using the same environment as code embedded in the page.

4. Modify the `location` field of an existing `iframe` element on the web page, or even the whole page.

5. Modify the CSS style of an element on the web page such that it uses a background image loaded from an arbitrary URL.

The basic feature we would like to implement is a way to keep track of and enforce the patterns of URLs from which a given script context may (a) make HTTP requests (which may both fetch data and cause side effects on the server), and more strictly, (b) fetch and execute code. Method 1 above can be accomplished by introducing an alternate definition of `XMLHttpRequest` into the Javascript environment which makes appropriate checks. Since methods 2-5 above all rely on the HTML DOM, all DOM operations which communicate with remote hosts must check request URLs against the URL patterns currently in effect.

Now how do we actually restrict what URLs can be access by a particular section of code? A straightforward approach is to define operations makeURL and makeCodeURL which take a string and return an opaque, unforgeable URL value (much like proxy objects in the previous section cannot be constructed within client code). By using handlers to override these operations at several levels of abstraction, we can incrementally filter out URLs that are not allowed within a particular scope.

Chapter 7

Related work

The message-based mediated execution model studied in this thesis can be compared to related work on two major fronts: the problem domain we are addressing (Sections 7.1-7.3), and the method by which we provide mediation (Sections 7.4-7.6).

The core problem we are addressing is a variation on the theme of virtualization and sandboxing (Section 7.1), in which the virtual machine environment presented within a particular scope is customizable at runtime. Historically, language customizations have been provided by means such as reflection, metaprogramming, and aspect-oriented programming; we compare these to our work in Section 7.2. Alternatively, one may view language customization as the process of composing new specification modules with an explicit model of the language semantics; we survey related semantic frameworks in Section 7.3.

The three key features identified in our approach to mediated execution were explicit call-by-name execution, dynamic scoping, and delimited continuations. The explicit call-by-name feature requires standard operation for manipulating quoted code, similar to those provided in the metaprogramming systems discussed in Section 7.2.2. Some of the finer points on dynamic scoping are surveyed in Section 7.5, while the background on delimited continuations is presented in Section 7.4. From a system perspective, our approach to mediation can be viewed as an extension to the notions of actors and meta-actors, discussed in Section 7.6.

7.1 Virtualization and sandboxing

Virtual machines have been used both at the system level and in programming language implementations, in order to provide software with the illusion of a particular underlying platform. System-level virtualization focuses on a thin layer of abstraction, allowing system resources to be multiplexed; while language virtual machines focus on providing the key primitives for implementing a given programming language. Mediated request-based execution aims to provide a

customizable virtual machine abstraction, and hence builds on both of these genres.

7.1.1 System-level virtualization

Mainframes. Though it has only recently become popular on commodity platforms, *system-level virtualization* has a long history on mainframe platforms. The earliest practical virtualization platform was IBM's VM/370 operating system [22], which began as a research project in the mid 1960's, and achieved commercial deployment in the 1970's. VM/370 and its successors are typically used to host per-user operating system instances, allowing multiple concurrent users to run legacy batch-mode applications and online development environments. The early success of VM/370 is due to several factors unique to mainframes: vertically-integrated custom hardware designs; the high cost of mainframe equipment and service; emphasis on data processing (as opposed to outright computational speed); and abstracted, relatively high-level IO channels.

VMware and Xen on PCs. More recently, in the late 1990's and early 2000's, virtual machine monitors such as VMware [85] and Xen [14] have made virtualization a practical capability of commodity x86 microcomputers. The key challenge in designing these systems has been how to implement sufficiently complete virtualization on top of an architecture not designed for it. First of all, the x86 processor architecture has a relatively complicated set of protected-mode constructs, including several unprivileged instructions which behave slightly differently in user vs. supervisor mode. Secondly, the surrounding PC architecture includes a diverse set of IO devices, which can be challenging to efficiently virtualize. While traditional virtualization systems like VMware, VirtualPC, VirtualBox, QEMU, Parallels, Mac-on-Linux (and so on) run unmodified guest operating systems, Xen requires restricted modifications to guest OSes. This *paravirtualization* approach simplifies virtualization by eliminating the need to emulate certain tricky constructs. In the mid-to-late first decade of the 2000's, AMD and subsequently Intel introduced hardware virtualization instructions for the x86 platform, significantly simplifying the job of virtual machine monitors. VMware, Xen, Linux's Kernel Virtual Machine, VirtualBox, and other systems today can employ virtualization instructions to mediate the execution of unmodified native operating system images.

Operating system multiplexing. An alternative, though more restrictive approach is operating system multiplexing, as in Solaris Zones [89], FreeBSD Jails [45], Linux-VServer, and OpenVZ [69]. In these systems, a single operating system kernel provides several independent user

level environments, each with their own restricted `root` account. Operating system multiplexing is generally faster than full system virtualization, since only one OS kernel is present, and privileged operations such as IO run natively. This also has the downside that all system images are all tied to the same kernel—*e.g.,* this approach alone would not be suitable for running a Windows environment atop a Linux host.

The common goal of the above work is to provide complete virtualization of a standard system platform. The proposed thesis work serves a complimentary goal: making it feasible to construct custom virtual environments with an application.

7.1.2 Language virtual machines

Several programming languages use custom virtual machine environments as an intermediate target for code generation. Virtual machine code (often called *bytecode*) typically consists of simple, low-level operations along with somewhat higher level operations which are used as primitives by higher-level languages. While the Java Virtual Machine [61] and its younger cousin, the .Net Common Language Runtime [27], are the most widely used language virtual machines, they are far from the first. Some of the earliest language virtual machines were O-code, for BCPL [73]; and p-code for Pascal [40]. Smalltalk [33]—an influential reflective object-oriented language—was also implemented atop a virtual machine. In the case of Smalltalk, live VM program images can be saved to disk, ported to other machines, and restarted.

Language virtual machines come in both general-purpose and language-specific varieties. In addition to Smalltalk, contemporary languages such as OCaml and Haskell maintain custom virtual machines. The Java VM was originally designed for the Java programming language alone; but as Java grew in popularity, languages such as Scala [68] chose to use the JVM as their own target platform. Indeed, upcoming versions of the JVM plan to add support for bytecode instructions useful for dynamically-checked languages other than Java. Observing this pattern, the .Net CLR and the LLVM compiler infrastructure [56] were both designed as targets for multiple different languages. The key difference between the two is is that .Net aims to encompass the "typical" features of object oriented programming languages, to enable multi-language interoperability; while LLVM models an idealized typed processor architecture, to enable language-neutral analyses and optimizations.

The principle property of language virtual machines is that they provide an abstracted execution environment for higher-level languages. However, most language virtual machines do not provide

virtualization in the sense of system-level virtualization. While it may be slightly easier to construct a nested virtual machine as compared to, say virtualizing x86; there are typically no constructs to automatically create a full nested virtual machine environment within a language VM.

7.1.3 Javascript sandboxing

Over the past 15 years, Javascript [26] embedded on web pages [95] has become the de facto standard for mobile code in user-facing applications. Javascript interpreters are embedded in the web browser installed on every modern PC. Since mobile code cannot be trusted with a user's full permissions, web browsers execute Javascript code in a restricted sandbox. Scripts cannot directly access the local hard drive or other devices: all interaction is via their host web page; and scripts cannot retrieve data from servers other than that from which their host web page originates. But with the rise of web platforms, many web pages run scripts originating from several different hosts—ranging from advertising services, which tend to be trusted by the creator of a website, to arbitrary user-created widgets. The current solution to Javascript security is fairly limited, and in practice, if not properly understood, can easily lead to cross-site scripting vulnerabilities. The monolithic Javascript sandbox does not help because exploits need not break out of the sandbox to attack other code running within the same Javascript sandbox. The discussion in Chapter 6 considered ways that we might use request mediation to construct a hierarchy of nested sandboxes.

7.2 Reflection, metaprogramming and aspect-oriented programming

A variety of techniques have been developed to allow language extension. These range from complete dynamic access to the language implementation through purely static transformations. Additionally, some techniques offer very open access, with few guidelines; while others are more constrained. The work in this thesis falls on the more dynamic end of the spectrum, though with some intentional constraints, since code can only mediate the execution of nested levels.

7.2.1 Reflection and metacircular evaluators

A metacircular evaluator is simply an interpreter written in the same language which it interprets. Consequently, a metacircular evaluator can evaluate a copy of itself (naturally, at some cost in

performance). The first machine-executable metacircular evaluator was written in the original Lisp programming language; a λ-calculus together with representations of lists and symbols is sufficient for this purpose. Somewhat more recently, the Smalltalk-80 virtual machine, for example, is itself written in Smalltalk-80. The fact that the interpreter and interpreted program are written in the same language opens up an interesting possibility: the ability to expose the interpreter state to an interpreted program and allow modification of the interpreter state, commonly known as *reification* and *reflection,* respectively (or collectively, *"reflection"*).

Reflection is commonly used as a flexible means to extend languages and systems. Pattie Maes proposed a definition of computational reflection in [62], and studied reflection in procedural languages, logic programming, and object-oriented languages. For example, in Smalltalk, programs have access to classes as objects; and messages not explicitly handled by another method are available to an object's `doesNotUnderstand:` method. Moreover, since all components of the Smalltalk-80 VM are themselves represented as Smalltalk objects, the machine state is available for modification—for example, addition or removal of classes and methods. As an exploratory dynamic environment, two key factors which are not addressed in Smalltalk are security and static analysis. While its reflection capabilities are very useful for extending programs, they are not appropriate for strongly *limiting* a program's abilities.

The 3-Lisp system, by Brian Smith [74], is one of the earliest examples of a fully-reflective programming language. 3-Lisp defined a tower of interpreters, allowing one to modify the execution of any level by executing code at the next level up. This model is similar to our levels of handlers, except growing in the opposite direction: we allow any level to act as a meta-level for a new object level that it defines and controls, whereas 3-Lisp allows any level to access its pre-existing meta-level. New meta-levels are created on an as-needed basis to simulate an infinite stack of levels.

The Common Lisp programming language offers a huge library of features, including a reflective implementation of object-oriented programming, known as the Meta-Object Protocol [49] (MOP). As in our work, a program's execution may be mediated by some *meta-object;* but in the MOP, program behavior is modified by updating a program component's corresponding meta-object (as opposed to running that program in an custom context). The MOP focuses on customization of OO features, such as method dispatch, inheritance policies, and creating instance of objects from classes. Thus a collection of handlers which override only the object-oriented features in our dialect of JavaScript would be similar to a use of the Meta-Object Protocol.

Java is another object-oriented language providing reflection, but in a more limited sense. Java's reflection provides runtime read access to information about types and classes: for example, given an object, one can access a representation of its class, and look up a method with a given name and type signature. However, one cannot, for example, add a method an object's class at runtime, or introduce multiple inheritance into the language. While sometimes inconvenient, these restrictions also serve a purpose: program semantics cannot be so drastically changed that trusted code no longer functions properly. An alternative approach to this problem is to recognize multiple domains of execution, where a reflective modification only affects its local domain. For example, we take this approach with handlers that only service requests made within their scope.
The Maude system [18] provides yet another example of reflection. As a language designed for the specification of executable semantics, it is useful to be able to write tools which can manipulate modules as data or run them as programs. Hence Maude provides a `META-LEVEL` module, which allows one to represent modules as data, and reduce terms in meta-represented modules using the underlying Maude engine. This is similar to our concept of mediated execution, but term reduction in Maude does not have side effects, whereas our system supports the ability to mediate some operations, while others pass through. Maude's flexible syntax allows meta-represented modules to appear very similar to Maude source code. In some ways, our request mediation mechanism takes this concept further, by allowing builtin and defined operations to be indistinguishable and interchangeable.

7.2.2 Metaprogramming

Reflective modification of a metacircular evaluator is one approach to metaprogramming; customized code generation is another. The latter may be called *compile-time metaprogramming.* In practice, most compile-time metaprogramming systems also involve some run-time support. Nonetheless, these systems are generally more restrictive in terms of the scope of metaprogramming effects available at runtime.

Lisp and Scheme macros. *The* classic example of compile-time metaprogramming is Lisp macros [34, 71, 16]. The simplest formulation of a macro is a function which runs at compile time, translating lists representing new syntactic forms into existing syntactic forms. Since macros are responsible for generating other code, this means that macros must run in a different environment from the normal application code. In practice, Lisp systems take measures to bridge the

compile-time and run-time environments, including allowing shared helper functions to be used in both contexts; maintaining access to macros for code generated at runtime via `eval`; and in some cases, using the resultant environment following macroexpansion as the initial runtime environment. The key problem with unrestricted macros is that their behavior is difficult to predict. On the one hand, code using macros can be incredibly succinct; but on the other, macro-laden code can be difficult to understand and refactor. Complex macros may also themselves contain subtle bugs which spread pervasively throughout a program. One such problem is variable capture: a macro which introduce variables may in some cases result in conflicts if the same variable name is already in use. Hygienic macros as in Scheme [55, 20, 37, 46] provide a solution to the variable capture problem, by treating all variables within a macro definition as *meta-variables.* In addition to lower-level facilities, Scheme provides simplified higher-level definition constructs for macros which are direct mappings from new forms to code templates.

C++ template metaprogramming. Perhaps the most popular modern compile-time metaprogramming environment is C++ templates [83, 79]. While originally designed for straightforward generic programming (such as parameterized container data structures), due to the interaction between template partial specialization and ad hoc polymorphism, C++ templates are Turing-complete. In stark contrast to Lisp macros, the template language is totally different from the C++ runtime language; template code may *only* execute at compile time. Nonetheless, several libraries have been created that make extensive use of templates to generate application-specific code. For example, Blitz++ uses templates to generate high-performance implementations of matrix math [92]. Similarly, the Boost library [72] provides a wide range of features based on C++ templates, ranging from parsers to graph algorithms [41] and more. Perhaps the best lesson to be learned from C++ is the practical value of powerful extensions to popular programming languages.

Other multi-stage programming systems. Though Lisp macros and C++ templates have historically enjoyed the most popularity, experimental multi-stage programming [88] extensions have been created for several other programming languages. Projects include Jumbo (for Java) [44]; MetaOCaml [58] (a followup to [88]); and Template Haskell [78]. These systems generally consist of quote and unquote mechanisms (similar to quasiquotation in Lisp) which allow quoted program expressions to be composed and evaluated at runtime. This feature also appears in language-level virtualization; the key addition being the ability to manipulate the definition of the language context in which an expression is evaluated.

7.2.3 Aspect-oriented programming

Aspect-Oriented Programming [48] has a long history, originating in research on more open-ended reflection mechanisms, particularly the Meta-Object Protocol. The key differentiator of AOP is the ability to usefully apply aspects to pre-existing source code, for example, as in AspectJ [47]. This is accomplished by specifying *aspects*, which include patterns that match at particular *join points* in a program (typically function call sites), and *advice,* which is code that should be run before, after, or in place of the application code at those points. While the aim of language-level virtualization is the ability to define or restrict the behavior of all operations used within a subprogram, AOP focuses on enriching functions which match certain patterns.

Several studies have been conducted on the implementation and analysis of aspect-oriented programming, including the following. Bockisch, et al. [15] investigated the integration of aspect-oriented features into language virtual machines, in order to provide good performance while enabling join points to be chosen at run time. In contrast, [77] developed a static analysis which enables more efficient compilation of dynamic aspects. A type system for aspect-oriented computation is presented in [39], for the polyadic μABC language. μABC is similar to the π-calculus, modeling nonterminating computations of interacting peers. In contrast, language-level virtualization focuses on mediating a subprogram's behavior by way of a meta-level. Another theoretical study of aspects is presented in [76], which defines a semantics of aspects by translation to an ML-like functional language. This contrasts with our approach of building up a language by using virtualization constructs.

On the experimental side, [7] explored the use of user-defined analyses to identify pointcuts. Notably, their analysis accounts for the effects of aspects, and can remove dynamic tests which may be resolved statically. Similar techniques could be useful in a static, translational implementation of language-level virtualization. An interesting complement to our work is *aspect mining,* in which aspects are extracted from existing code bases; see, for example [21]. Both aspect mining and language level virtualization are useful tools for refactoring and reusing existing code bases in controlled environments.

7.3 Semantic frameworks

Programming languages can be considered both in terms of implementation strategies and mathematical definitions of the underlying semantics. Many approaches exist to definining

executable semantics, wherein a semantic specification can be used to interpret real programs. From this perspective, our request-based model of execution can be viewed as an approach to dynamically creating new, extended language semantics for executing certain program fragments. We compare our approach to various systems of operational semantics surveyed in [75].

Small-step structural operational semantics (SOS) defines a syntax-directed reduction function on the states of a program [70]. A computation is expressed as a derivation tree, where each node corresponds to a single-step rule in the semantics. As pointed out in [75], SOS does not provide facilities for the modular definition of effectful operations: (1) the full state of a program be captured in the syntax of each semantic rule; and (2) conceptually simple operations like halt require changes to many seemingly unrelated rules. Our approach does not suffer from these limitations because our basic execution model allows request messages to automatically propagate to the appropriate level. For example, the definition of halt is simple and modular:

$$\mathsf{handler}(\mathsf{halt}, \lambda args.\lambda k.\mathsf{unit}(), ...)$$

Similarly, program state related to certain operations can be abstracted away, and ignored by unrelated operations. Examples include our definitions of lexical variables in Chapters 3 and 5.

Big-step operational semantics abstracts SOS's reduction function on program states to an evaluation function from programs to results [43]. The return messages (Ret v) in our execution model borrow this idea. While big-step semantics can make language definitions simpler and higher-level as compared to small-step semantics, it does not provide a way to peer into the individual evaluation steps. In addition to debugging concerns, a program which does not terminate has no result *value*. Our execution model gets around this problem by exposing intermediate steps as request messages, which may or may not be captured by various parts of a program or language definition, depending on whether they are of interest.

Modular SOS (MSOS) extends SOS with the capacity to describe state information that is not part of a program term itself, and which may be used or ignored by rules [65, 66]. Rules which ignore components of the state are assumed not to change those components. One could, theoretically, define a dynamic language extension mechanism which merged new "language modules" with a copy of the MSOS semantics of the base language in effect, and instantiating a new interpreter for the new, extended semantics. While this is feasible when all modules of the language semantics are equally trusted, it does not account for access permissions to state

components, untrusted language extensions, and the coexistence of different definitions of the same construct within different parts of the *same* program. Our approach intrinsically accounts for these additional concerns by explicitly scoping language extensions as handler operations within the language. Hence a "language module" in our approach can only access those state components and operations which a normal program defined at the same point can access.

Context reduction semantics differs from the previously-described systems in that it splits a program, at each step of execution, into an *evaluation context* and a *redex* [30, 97]. This corresponds closely to our request messages: the redex corresponds to the pair of an operation name and its arguments, and the context corresponds to the delimited continuation. But there are several key differences. Evaluation contexts represent the *entire* context surrounding a redex, hence they correspond to *full continuations,* in contrast to our *delimited continuations.* Context reduction semantics involves simple, unconditional rules based on pattern matching on the context and the redex, whereas in this thesis, handlers may make use of arbitrary operations that are available in their scope, and we may rethrow requests that a handler cannot itself service. This is possible because new handlers can be defined anywhere within our programs, whereas context reduction semantics, as with the other systems above, separates the semantic rules from the program. In context reduction, valid contexts are specified via a grammar, and inferred via a parsing mechanism; whereas our execution model involves an explicit traversal directed by the programmer (or language designer). We made this choice because it is easier to understand how to arbitrarily redefine operations if the operation definition has full control over (and responsibility for) the traversal. It would be interesting to understand how to integrate the grammar/parsing aspect of contxt reduction into the request-based execution model.

All of the systems discussed above make use of an interleaving model of concurrency, as do we in Chapters 4 and 6. For example, a two-level reflective semantics has been developed for distributed systems in terms of the actor model [94], which has been applied to problems such as distributed garbage collection. In principle, we could also offer a true concurrency model, where the par operation accepts and evaluates a number of concurrent terms, internally servicing certain operations on its own, only emitting requests for operations it does not know how to implement. In hardware terms, this corresponds to executing programs on multiple slave processors, emitting messages on a shared bus to perform communication. However, this possibility is not explored in detail in this thesis.

7.4 Delimited continuations

Continuations formalize the intuitive notion of "the rest of the program" at any point of execution [82]. A continuation is a function that, when called, executes the rest of the program—and since programs either terminate or diverge, a call to a continuation does not return. This behavior makes continuations useful for describing control flow constructs. For example, a function may be passed normal return continuation and an exception continuation, and then internally choose which continuation to call, depending on whether an exceptional condition occurs during its execution. In practice, continuations have been used as a compilation mechanism—the CPS transform—as well as presented to the application programmer as a general-purpose construct, such as `call-with-current-continuation` in Scheme [19].

Delimited continuations [29] represent the rest of the execution of a program *up to a certain point.* In practice, delimited continuations often represent the rest of the computation of a particular subterm of a program, where the root of that subterm serves as the delimiter. One may alternatively think of a delimited continuation as a *continuation prefix,* which may be composed with another (delimited) continuation to form a larger (delimited) continuation.

Historically, delimited continuations have been defined in terms of a pair of operators, most notably either the dynamic delimited continuation operators prompt/control, or the static delimited continuation operators shift/reset. For example, the expression prompt $\mathcal{C}$[control $k.e$] captures the delimited continuation $\mathcal{C}$, evaluating to prompt $e[k := \mathcal{C}]$, as presented in [51]. The corresponding case for shift/reset is similar, but slightly different: in the case of prompt/control, if the captured delimited continuation $\mathcal{C}$ is returned by the prompt expression and then executed, any control operation it invokes may capture a much larger delimited continuation than was apparent from the original program code. The static operators, shift/reset, fix this problem by wrapping the delimited continuation returned by reset in a shift block. Thus shift/reset has a straightforward transformation to prompt/control; but the question of a transformation in the opposite direction was open for quite some time. As it turns out, these and various other approaches to delimited continuations, as well as full continuations, are mutually expressible in terms of each other [17].

The delimited continuations used in this thesis are of the dynamic variety, since it is convenient to be able to take code out of one dynamic context and put it into another. Another key difference is that we use different delimiters for different operators (corresponding to their handler scope). Other implementations of delimited continuations make similar generalizations [52].

The continuations within our request messages are in fact *delimited continuations* [24]. While a

standard continuation is a non-returning procedure which represents the rest of a program, a delimited continuation represents the rest of the computation of a *subterm,* and hence returns (unless the subterm diverges). Kiselyov et al. [53] showed how dynamic binding can be expressed in a delimited control framework. Their notion of *delimited dynamic bindings* coincides with the behavior of request messages (producers of delimited continuations) and handlers (the scope of dynamic bindings and consumers of delimited continuations) in language-level virtualization.

7.5 Dynamic binding

One of the key features enabling request mediation is dynamic binding. The idea is that when a program term is placed within the scope of a handler, that handler's binding takes effect within the program term. If the program term is later placed in the scope of an alternate handler for an operation, the new handler should take effect. This section briefly reviews the literature on dynamic binding. This work can be classified into three general areas: theoretical results, implementations, and applications. Additionally, for an overview of binding constructs, see [12].

7.5.1 Theoretical results

Several different approaches to dynamic binding have been studied in a theoretical context. The system λN provides a model of dynamic binding based on name-indexed variables [23]. Rather than storing a single value in a variable, λN allows multiple values to be passed along different named channels associated with the same lexical variable. This model is a generalization of both lexical scoping and (lexically-global) dynamic scoping, since each lexical variable can be treated as a dynamic environment or record. Hence records are essentially built-in to λN, which allows for the development of a type system with different value types at each name index of a variable. However, names are not first-class values, which means that private names must be managed by convention, and new names cannot be created dynamically.

Dynamic binding a la Lisp was formalized in [64]. The system Λ_d and its derivatives retain Lisp's distinction between lexical and dynamic variables. The article studied applications of dynamic variables to defining exceptions, which is similar to our use of request messages in this thesis, since request messages can be viewed as restartable exceptions. The work also covered a number of practically-motivated evaluation strategies, including a dynamic-environment passing transformation from dynamic to lexical scoping; and both deep and shallow binding. Deep binding

corresponds to searching through a list for a variable's binding, while shallow binding is implemented by saving and restoring the contents of a global variable. These topics were explored from an implementer's perspective in [13].

One of the more recent advances has been the combination of dynamic binding and and delimited control within a single, consistent formalism [53]. The system, called DB+DC, enriches the language from [64] with a type system and delimited control. The authors conclude that delimited dynamic binding must allow for delimited continuations which close over part of the dynamic environment, rather than all or none. Indeed, this is exactly what we do in this thesis with handlers and rec-handlers expressions, since these dynamic binding sites also serve as delimiters for continuations in request messages. The paper also shows that DB+DC can in fact be macro-expressed in terms of delimited control.

An approach to implicit parameter passing which is slightly different from dynamic variables was presented in [60]. This system allows for Hindley-Milner type inference of implicitly propagated function arguments. However, a comparison of the implementation of implicit parameters versus other dynamic binding systems has revealed a discrepancy [50]—whereas reading a dynamic variable returns the value of the latest binding, implicit parameters may be irrevocably substituted sooner. However, this may arguably be the desired behavior in the situations implicit parameters are designed to solve.

7.5.2 Implementations

Lisp implementations have a long history of dynamic variables. Earlier Lisps relied heavily or exclusively on dynamic scoping [13]. Common Lisp [80] brought lexical scoping as the default, while still allowing a separate namespace of (lexically-global) dynamically-scoped *special variables*. Some other languages, such as Perl, also follow this pattern (Perl uses `local` for dynamic scoping and `my` for lexical scoping). TEX [54] and Emacs Lisp [59] also retain dynamic scoping. In many of these cases, however, the choice of dynamic scoping is not an absolutely necessary choice so much as an engineering compromise based on use cases and implementation technology.

The Scheme programming language, inspired by the untyped λ-calculus, is the most popular lexically-scoped Lisp dialect. Owing to the history of the Lisp community, several Scheme implementations include additional features for dynamic scoping—particularly, lexically-scoped dynamic variables. The SRFI 39 standard [28] defines a system of *parameter objects* which can be dynamically created, passed by reference, and dynamically bound. While parameters are distinct

from variables, the fact that they are first-class objects which can be stored in lexical variables means they are essentially lexically-scoped dynamic variables—much like how operations are not first-class functions, but operation names are represented as symbols in this thesis. One existing weakness of parameter objects is the fact that different Scheme implementations handle parameters differently in multi-threaded programs. The preferable approach is for parameter bindings in separate threads to be independent, just as sequential parameter bindings would be. The paper [36] proposed a minimal practical implementation of dynamic scoping for imperative languages. This project focused on the structure of dynamic environment frames, lookup methods based on hashing, and a pragmatic approaches to introducing dynamic variables using C++ classes and C preprocessor macros.

7.6 Actor systems

The actor model [5] presents a convenient way to describe and implement concurrent object-oriented systems. In relation to this thesis, the concept of meta-actors—supervisors which mediate the visible actions of other actors—represent a restricted form of our request mediation. In Chapter 4, we examined definitions of several actor constructs in terms of request mediation.

7.6.1 Background on actors

Actors have been implemented as both libraries and native features in many production languages. Erlang [9], developed for safety-critical distributed systems such as telephone switches, is essentially an industrial-strength implementation of the actor model. Historically, object-oriented languages like Smalltalk share a very similar model—however, Smalltalk's messages are synchronous. Even Scheme was originally inspired by a desire to create an actor system [87], although Scheme ended up taking a rather different evolutionary path.

More recently, actors have been implemented as library abstractions in many languages, including Java [10], Scala [35], Python [57], and C++ [11, 63]. The primary downside of actor libraries is they cannot always enforce actor semantics. For example, the behaviors of two actors in Java could hold references to a shared variable, because Java's threads allow shared memory. This can happen inadvertently if an actor sends a message which contains an object that is only a shallow copy of an object it retains. Similarly, if actors are conflated with base-language objects, a programmer may call a synchronous object method rather than sending an asynchronous message; this may be

done inadvertently or inconsistently. It is possible to define front-end preprocessors which rule out these violations, but this essentially entails creating a new, derived language [91].

An alternative to creating a new language or living with a leaky system is to catch violations dynamically. This is the approach we have taken with our request handlers for actor operations. While it does not give us strong static guarantees, it does allow us to detect violations when testing a prototype. Another concern which requires run-time support is distributed garbage collection actors [42, 93, 25, 90].

7.6.2 Actor coordination and reflective customization

As mentioned in the introduction, there are many approaches to coordination in actor systems. In general, program control structures can be viewed as patterns of message passing amongst actors [38]. One approach we examined in this paper is finite automaton local synchronization constraints. This idea can be extended to protocol automata describing sessions involving multiple actors—for example, a client interacting with a server. A related concept is message receive patterns: for example, in Erlang, an actor's behavior may be defined piecewise, with different handlers for different message patterns. This idea has been generalized to patterns involving a conjunction of multiple messages [86].

An alternative to applying constraints locally within each actor is to mediate and filter messages as they pass between actors. Meta-actors are hence a useful implementation strategy for message filters [6]. In addition to the two cases examined in this paper (fixed and reassignable meta-actors), we can consider other tradeoffs: one meta-actor per actor vs. a stack of multiple meta-actors per actor vs. multiple actors supervised by a single meta-actor, known as group reflection [96]. Each of these cases can be described using the definitions of meta-actors we presented.

Finally, there are several approaches to describing complex coordination patterns amongst multiple actors. One form is protocol description languages, such as DIL, which can be used to customize program behavior, such as failure semantics [11, 84]. Another approach, called synchronizers, allows us to describe synchronization constraints in terms of message patterns and methods of various actors that they enable and disable—a distributed generalization of local synchronization constraints [1, 32]. Synchronizers have been extended to describe realtime behavior [67]. An alternative to enabling and disabling methods is creating and trapping the corresponding messages in flight. For example, Reo circuits provide a way to specify rules that govern the creation, propagation, and consumption of messages [8]. Protocol customization mechanisms have been used

to introduce fault-tolerance and dependability features into existing systems [2, 3].

Chapter 8

Conclusions

This thesis has studied an approach to programming languages that allows arbitrary language features to be redefined and extended from within the language itself. The key to this approach has been the view that a program executes by dispatching a sequence of request messages to an underlying underlying *semantics object,* which is itself defined as a lower-level program. In our model, program terms are trivially equivalent to a subset of these messages. Hence uses of all language features—even those of our base language—can be treated as request messages.

Our model conveniently captures the standard notion of a *stack* of multiple levels of abstraction, such as a script interpreted by a user process hosted by an operating system running on a processor. Having multiple levels of semantics objects, we generalized our notion of request message dispatch, to allow messages to either be handled by a particular level, or *automatically* propagated to the underlying level. As a consequence, we are able to create contexts which trap and service arbitrary request messages—hence define and redefine arbitrary operations; while we transparently pass all other requests to the underlying semantics—hence inherting all the other definitions in scope.

To better understand the request-based execution model, we used it to define a number of language features. Notably: all major features of JavaScript, as well as a simple actor system with meta-actors. The fact that even base language features are treated as messages had pluses and minuses when applied to a real language. On the negative side, it *required* a base-level language interpreter defined in the request-based style. To get the full reflective power of our model, we had to define a custom interpreter. But on the plus side, given such a base-level interpreter, we are able to dynamically create derived language interpreters within custom contexts by redefining only the relevant operations.

An important feature of the request-based model is that it enables supervisory control over the effects a program may have, while allowing the program itself to radically redefine the language in which it is written. Moreover, any program may act as a supervisor for subprograms. This is

because custom language extensions, by construction, (a) are only as powerful as "normal" code in the same context, and (b) have the ability to fully mediate code within their scope. Unlike models of reflection in which a program "reaches down" and modifies its own interpreter, this aspect of the request-based model makes it useful in sandboxed environments, such as web pages.
Finally, it is worth noting that the request-based execution model is enabled by the combination of three important preexisting language features:

- **dynamic scoping**—request messages are serviced by their nearest *dynamic* enclosing handler.
- **explicit call-by-name evaluation**—allowing the definition of an operation to customize how arguments are evaluated
- **delimited continuations**—*continuations* allow us to replicate, pause, abort, or customize the evaluation of a program following invocation of an operation; while their *delimited* nature allows us to restrict the scope of these control effects

Each of these features is important in our model. For example, dynamic scoping *combined with* delimited continuations is what allows us to define imperative variables on top of a functional base language. The correspondence between terms and messages is what unifies these features.

8.1 Contributions

The key contributions of this thesis can be summarized as follows:

1. Introduced a request-based execution model that allows arbitrary language features to be dynamically redefined, which is consistent with (and in fact, useful for) program sandboxing.
2. Showed how to define a real programming language (JavaScript) in terms of the request-based execution model, allowing dynamic extensions to JavaScript in a browser-based prototype implementation.
3. Related the request-based execution model to prior work, including virtual machines, reflection, and meta-actor systems.
4. Serves as a *(we believe, compelling)* argument for the inclusion of dynamic scoping, explicit call-by-name/nonstrict evaluation, and delimited continuations in real programming languages. While the request-based execution model as described in this thesis is still

immature, and will invariably be superseded by better formulations, these three language features are well-studied, and yet not as commonly available as they ought to be.

8.2 Future work

Possible future work related to this thesis can be divided into three categories:

1. evolution of the request-based execution model
2. implementation improvements
3. further applications

8.2.1 Evolution of the request-based execution model

It would be useful to make it easier to correctly and concisely describe implementations of language features via request handlers. One avenue of improvement would be to create a type system which allows us to define the type signature of an operation, and then check that their definitions and redefinitions (via request handlers) and their uses are all consistent with the type signature. Since our goal would be to type *any* language construct (rather than just, say, functions and variables), this would need to be a fairly rich type system. We would need the ability to introduce new types and new *typing rules,* show that those typing rules are consistent, and annotate terms with *effect types* as well as result types. One possible approach is a dependent type system, in which the type environment and typing rules are reified as type-level functions. This work will require a careful examination of the state-of-the-art in type-based reasoning systems, such as Coq, NuPRL, and Twelf.

Another, related, avenue for improvement is to examine and refactor our approach to variable binding. This thesis considers variables as names that are accessed via explicit `get` and `set` operations. This is consistent with the low-level realization of variables in memory, but it has two potential issues going forward. First of all, is it easier to type (or infer the types of) variables if they are expressed differently, for example, as operations themselves, rather than as names? Secondly, it would be useful to provide variable binding mechanisms as a component out of which operation definitions can be constructed. For example, it may be more convenient to adopt a higher-order abstract syntax (HOAS) style, perhaps providing a few different binding options (*e.g.,* lexical and dynamic variables).

Variable binders aren't the only potentially reusable language feature components (or *subfeatures*). It would be useful to define a library of subfeatures. To make this useful, we would need to investigate design patterns for intentionally making language features and subfeatures easily composable. For example, as it stands right now, if a new operation is defined via a handler and used within the scope of that handler, and we cannot introduce another mediating handler in between the two, then we have no good way to customize that feature. In solving this problem, we should be careful not to violate the sandboxing model: nested programs should not be able to arbitrarily examine and modifying their containing scope unless they are explicitly trusted to do so.

8.2.2 Implementation improvements

A key limiter on the performance of request-based interpreters is the cost of capturing delimited continuations. In most cases, this capture could well be performed lazily, as is the case in production implementations of delimited continuations. However, preliminary experiments with delimited continuations in OCaml [52] has turned up a slight inconsistency between the type signatures control/prompt and friends and the semantics of our handler operation. Traditionally, delimited continuations do not distinguish between "normal" return via the delimited term reducing to a value, and "forced" return by capturing a delimited continuation and inserting a value in its place. Our handler operation does distinguish between these cases; and implementing it in terms of control/prompt together with a discriminated sum type annoyingly requires us to insert an additional layer of injections/projections to the delimited continuation at each step of evaluation. It should be possible to optimize this away in a lower-level implementation.

A related concern is an efficient implementation of dynamic scoping, for the use cases typically encountered in the request-based execution model. One possibility is to eagerly inline dynamic handlers (or their addresses) into a program. To make this work, we must keep track of what has been inlined into a term, so that we can undo the inlining if a delimited continuation containing that term is later evaluated in a modified dynamic environment. In this case, it is particularly important that we know whether a given delimited continuation is used in only one place, or in multiple parts of the program simultaenously. In the latter case, we need to either not perform the inlining, or make a copy of the code.

8.2.3 Further applications

Javascript on web pages is an interesting domain to consider because of the clear need to better mediate foreign code running on a web page. If implementations challenges such as those outlined above can be overcome, it could be useful to create a native Javascript interpreter for a browser that supports request mediation. This would reclassify the techniques we have explored from early prototypes to more plausible solutions.

Aside from web pages, other computing environments also benefit from the ability to manipulate and safely mediate arbitrary code. It would be interesting to build features like our request mediation constructs into an operating system. For example, we might build upon existing debugging and tracing facilities. What would make this really useful is if we could use a simple, high-level scripting language to easily, safely and completely mediate the execution of any software component or fragment thereof. In essence, this would allow us to perform virtualization at an arbitrary granularity.

References

[1] Gul Agha, Svend Frlund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. In *in David Skillicorn and Domenico Talia (editors), Programming Languages for Parallel Processing, pp 146-157, IEEE Computer Society, May*, 1995.

[2] Gul Agha, Svend Frlund, Rajendra Panwar, and Daniel Sturman. A linguistic framework for dynamic composition of fault-tolerance protocols. In *Conference on Dependable Computing for Critical Applications (DCCA-3), pp 197-207, International Federation of Information Processing Societies, Palermo (Sicily), Italy, September*, 1992.

[3] Gul Agha, Svend Frlund, Rajendra Panwar, and Daniel Sturman. A linguistic framework for dynamic composition of dependability protocols. In *in C. E. Landwehr, B. Randell, and L. Simoncini (editors), Dependable Computing and Fault-Tolerant Systems VIII, pp 345-363, IFIP Transactions, Springer-Verlag*, 1993.

[4] Gul Agha and Daniel C. Sturman. A methodology for adapting to patterns of faults. In *G. Koob (ed.), Foundation of Ultradependability*, pages 23–60, 1994.

[5] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[6] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *Object-Based Distributed Processing, Lecture Notes in Computer Science 791*, pages 152–184. Springer Verlag, 1993.

[7] Tomoyuki Aotani and Hidehiko Masuhara. Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, 2007.

[8] Farhad Arbab. Reo: A channel-based coordination model for component composition. In *Mathematical Structures in Computer Science*, pages 329–366, 2004.

[9] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[10] Mark Astley, Thomas Clausen, James Waldby, Rajesh Kumar, and Amin Shali. Actor foundry. http://osl.cs.uiuc.edu/af/.

[11] Mark Astley, Daniel C. Sturman, and Gul A. Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44:2001, 2001.

[12] M.P. Atkinson and R. Morrison. Types, bindings and parameters in a persistent environment. pages 3–20, 1988.

[13] Henry G. Baker. Shallow binding in lisp 1.5. *Commun. ACM*, 21(7):565–569, 1978.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Symposium on Operating System Principles*, 2003.

[15] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004.

[16] Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. *SIGPLAN Lisp Pointers*, V(1):278–287, 1992.

[17] Chung chieh Shan. Shift to control. In *Proceedings of the 5th workshop on scheme and functional programming, ed. Olin Shivers and Oscar Waddell, 99107*, 2004.

[18] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[19] Will Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 124–131, New York, NY, USA, 1988. ACM.

[20] William Clinger. Macros in scheme. *SIGPLAN Lisp Pointers*, IV(4):17–23, 1991.

[21] Grigoreta Sofia Cojocar and Gabriela Şerban. On some criteria for comparing aspect mining techniques. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, 2007.

[22] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.

[23] Laurent Dami. A lambda-calculus for dynamic binding. *Theor. Comput. Sci.*, 192(2):201–231, 1998.

[24] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160, New York, NY, USA, 1990. ACM.

[25] Peter Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and euler cycles. In *in Proceedings Workshop on Distributed Algorithms 96, O. Babaoglu, K. Marzullo, Eds., Lecture Notes in Computer Science 1151*, pages 141–158. Springer-Verlag, 1996.

[26] ECMA-262. *ECMAScript Language Specification.* ECMA, 3rd edition, December 1999.

[27] ECMA-335. *Common Language Infrastructure.* ECMA, 4th edition, June 2006.

[28] Marc Feeley. Srfi 39: Parameter objects. 2003.

[29] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Marrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, 1987.

[30] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, page 103(2):235271, 1992.

[31] Svend Frolund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *European Conference on Object-Oriented Programming, in O. L. Madsen (Editor), Lecture Notes in Computer Science 615*, pages 185–196. Springer Verlag, 1992.

[32] Svend Frlund and Gul Agha. A language framework for multi-object coordination. In *Proceedings of the European Conference on Object-Oriented Programming*.

[33] Adele Goldberg, David Robson, and Michael A. Harrison. *Smalltalk-80: The Language and its Implementation*. Longman Higher Education, May 1983.

[34] Jr. Guy L. Steele. *Common LISP: The Language*. Digital Press, 2nd edition, 1990.

[35] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

[36] David R. Hanson and Todd A. Proebsting. Dynamic variables. *SIGPLAN Not.*, 36(5):264–273, 2001.

[37] David Herman and Mitchell Wand. A theory of hygienic macros. In *ESOP '08: Proceedings of the Seventeenth European Symposium On Programming*, March 2008.

[38] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[39] Peter Hui and James Riely. Typing for a minimal aspect language: preliminary report. In *the 6th workshop on Foundations of aspect-oriented languages*, 2007.

[40] Institute for Information Systems, University of California, San Diego. *UCSD PASCAL System II.0 User's Manual*, March 1979.

[41] Jaakko Järvi, Matthew A. Marcus, and Jacob N. Smith. Library composition and adaptation using c++ concepts. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA, 2007. ACM.

[42] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. *SIGPLAN Not.*, 25(10):126–134, 1990.

[43] Gilles Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.

[44] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: run-time code generation for java and its applications. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 48–56, Washington, DC, USA, 2003. IEEE Computer Society.

[45] Poul-Henning Kamp. Jails: Confining the omnipotent root. In *2nd System Administration and Network Engineering Conference*, 2000.

[46] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11:7–105, August 1998.

[47] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *15th European Conference on Object-Oriented Programming*, 2001.

[48] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, 1997.

[49] Gregor Kiczales and Jim D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[50] Oleg Kiselyov. Haskell's 'implicit parameters' are not dynamically scoped. 2005.

[51] Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, 2005.

[52] Oleg Kiselyov. Native delimited continuations in (byte-code) OCaml, 2005. http://okmij.org/ftp/Computation/Continuations.html#caml-shift.

[53] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. *SIGPLAN Notices*, 41(9):26–37, 2006.

[54] Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[55] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM.

[56] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[57] Jacob Lee. Parley: Python actor runtime library. http://osl.cs.uiuc.edu/parley/.

[58] C. Lengauer and W. Taha. *Special Issue on the First MetaOCaml Workshop 2004*, volume 62 of *Science of Computer Programming*. September 2006.

[59] B. Lewis, D. LaLiberte, and R. Stallman. *GNU Emacs Lisp Reference Manual*. http://www.gnu.org/software/emacs/manual/elisp.html.

[60] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 108–118, New York, NY, USA, 2000. ACM.

[61] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.

[62] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987.

[63] Rick Malloy, Niklas Gustafsson, Mike Chu, and Stephen Toub. Parallel computing platform: Asynchronous agents for native code. Microsoft, 2008a.

[64] Luc Moreau. A syntactic theory of dynamic binding. *Higher Order Symbol. Comput.*, 11(3):233–279, 1998.

[65] Peter D. Mosses. Foundations of modular SOS. In *MFCS*, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80. Springer, 1999.

[66] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, pages 60–61:195–228, 2004.

[67] Brian Nielsen, Shangping Ren, and Gul A. Agha. Specification of real-time interaction constraints. In *Proc. of First Int. Symposium on Object-Oriented Real-Time Computing, IEEE Computer Society*, pages 206–214, 1998.

[68] Martin Odersky. The scala langue specification, version 2.7. Technical report, Programming Methods Laboratory, EPFL, 2009.

[69] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59, HP Laboratories, April 2007.

[70] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[71] Christian Queinnec. Macroexpansion reflective tower. In *Proceedings of the Reflection96 Conference*, pages 93–104, 1996.

[72] Robert Ramey. Making a boost library. In *OOPSLA '05*, October 2005.

[73] Martin Richards. *The BCPL Cintcode and Cintpos Users Guide*. University of Cambridge, September 2005.

[74] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 331–347, 1984.

[75] Grigore Rosu. K: A rewriting-based framework for computations —preliminary version—. Technical Report UIUCDCS-R-2007-2926, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[76] S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, 2007.

[77] Damien Sereni and Oege de Moor. Static analysis of aspects. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003.

[78] Tim Sheard and Simon Peyton Jones. Template metaprogramming for haskell. In *Haskell Workshop 2002*, pages 1–16, 2002.

[79] Jeremy Siek and Walid Taha. A semantic analysis of C++ templates.

[80] Guy L. Steele. *Common Lisp: The Language*. Digital Press, 1990.

[81] Johnny Stenback, Philippe Le Hegaret, and Arnaud Le Hors. *Document Object Model (DOM) Level 2 HTML Specification*. W3C, January 2003.

[82] Christopher Strachey and Christopher P. Wadsworth. Continuations: a mathematical semantics for handling full jumps. Technical Report PRG-11, Oxford University Computing Laboratory, January 1974.

[83] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[84] Daniel C. Sturman and Gul A. Agha. A protocol description language for customizing failure semantics. In *In The 13th Symposium on Reliable Distributed Systems, Dana Point*, pages 148–157. Society Press, 1994.

[85] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, 2001.

[86] Martin Sulzmann, Edmund S.L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In D. Lea and G. Zavattaro, editors, *COORDINATION '08: Proc. 10th Intl. Conf. Coordination Models and Languages*, pages 315–330, 2008.

[87] Gerald Jay Sussman. Scheme: An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab*, 1975.

[88] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Notices*, 32(12):203–217, 1997.

[89] Andrew Tucker and David Comay. Solaris Zones: Operating system support for server consolidation. In *USENIX Virtual Machine Research and Technology Symposium*, 2004.

[90] Abhay Vardhan and Gul Agha. Using passive object garbage collection algorithms for garbage collection of active objects. *SIGPLAN Not.*, 38(2 supplement):106–113, 2003.

[91] Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. In *16th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[92] Todd L. Veldhuizen. *Active Libraries and Universal Languages.* PhD thesis, Indiana University Computer Science, May 2004.

[93] Nalini Venkatasubramanian, Gul Agha, and Carolyn L. Talcott. Scalable distributed garbage collection for systems of active objects. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 134–147, London, UK, 1992. Springer-Verlag.

[94] Nalini Venkatasubramanian and Carolyn L. Talcott. A semantic framework for modeling and reasoning about reflective middleware. *IEEE Distributed Systems Online*, 2, 2001.

[95] W3C. *HTML 4.01 Specification*, December 1999.

[96] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 405–425, 1991.

[97] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, page 115(1):3894, 1994.

Author's biography

Sameer Sundresh received a Bachelor of Science degree in Computer Science from the University of Illinois at Urbana-Champaign in 2001; he has been enrolled in the graduate program in Computer Science since then. From 2001-2005, Sameer worked on sensor networks, including collaborative fieldwork on localization based on radio and acoustic signals and other known constraints, and a sensor network simulator of questionable value other than accumulating citations. Being tired of overcalibrating dodgy sensors on bombing ranges and army bases in the middle of nowhere, during 2005-2006, Sameer made a failed attempt to develop a language-parametric module system (with a few unsuccessful revivals as he learned more about programming languages and type systems). Since December of 2006, he has intermittently been working on the request-based execution model reported on in this thesis; for about the first year, this included an attempt at a type-and-effect system which never quite worked right and obscured the central ideas. From the August 2007 through August 2008, he consulted part-time for Pattern Insight, Inc., which inspired the application to Javascript. Since the Spring 2007 semester, Sameer has worked as a teaching assistant for CS 241 (systems programming), CS 421 (programming languages), and CS 423 (operating systems).

Since 2004, Sameer has also been actively involved in the University of Illinois ACM student chapter. His participation includes work on a mobile phone-based payment system with SIGEmbedded, founding of a student SIGPLAN chapter, service as the 2005 Reflections Projections Conference chair, guiding the formation of a corporate relations committee, and stints as secretary and vice chair.