

MAST: A Dynamic Language for Programmable Networks

Technical Report

Dimitris Vyzovitis and Andrew Lippman
MIT Media Laboratory
{vyzo,lip}@media.mit.edu

May 31, 2002

Abstract

In this paper we present MAST, a dynamic Scheme variant for network, distributed, and peer-to-peer programming. MAST extends Scheme with additional constructs and semantics for network and distributed programming. We introduce distributed first class environments, support for distributed continuation with a loosely coupled model, and a sound security model. The result is a high level mostly functional language suitable for programming a wide range of network and distributed protocols and applications. Low level point-to-point and multipoint protocols, high level distributed protocols, and interacting protocols can all be designed and implemented with ease and reasonable performance.

1 Introduction

Modern distributed applications require more than the traditional byte-stream abstraction of the network. Applications require multiple protocols for actively dealing with various parts of complex interactions: peer-to-peer lookup, service location, dynamic configuration and control, and transport in application dependent data units. At the same time, these protocols are characterized by common algorithms, object interactions and data representation within the application domain, calling for a language that can capture and efficiently express programs in a common framework.

In this paper we present MAST, a dynamic programming language for network, distributed, and peer-to-peer programming using distributed environment and continuation abstractions. MAST is based on Scheme – in fact the reference implementation is intended to be a fully *R⁵RS* [15] compliant Scheme implementation. MAST extends Scheme with additional constructs and semantics for network and distributed programming. We introduce first class environments and dynamic procedures, a built-in lightweight threading model integrated with continuations, remote evaluation and support for distributed continuation with a loosely coupled model, code mobility and a sound security model. The result is a high level mostly functional language suitable for programming a wide range of network and distributed protocols and applications. Low level point-to-point and multipoint protocols, high level distributed protocols, and interacting protocols can all be designed and implemented with ease and reasonable performance.

MAST differs from other distributed variants of Scheme [14, 17] and other distributed programming toolkits and languages, such as Java/RMI [20], network objects [2] and Obliq [4] to name a few, as it does not attempt to mask the network or provide strongly coupled, coherent, type-safe or statically checked distributed programming capabilities. Instead, it operates at a lower level than these systems, yet providing powerful abstractions which capture the computation flow and environment of network programming. The distributed object model is loosely coupled with no back pointers, thus simplifying garbage collection, eliminating network overhead, and allowing transparent interaction with objects accessible through multicast and unicast channels alike. First class environments, remote evaluation, and distributed continuation offer a high level language for doing low level network programming. Distributed continuations in particular move the language semantics beyond remote evaluation [19], allowing us to handle unicast and multicast protocols in the same

framework. And finally, the built-in language support for mobility [12, 16], makes it easy to implement high-level protocols based on the mobile agent paradigm.

The idea of implementing network protocols as mobile code is the driving force behind the Active Networks [8] initiative. Protocols based on MAST, however, depart from the general active network paradigm, as there is no modification to the IP service model required. MAST code is not required to execute on routers or require active IP [22] extensions; rather, it is encapsulated on normal IP packets and is executed at end hosts. The language design adheres to the end-to-end design principle of the Internet [18], and offers a natural tool for building protocols with application-level framing [6]. Finally, as the use of mobile code has serious security implications [5], the language is designed with security in mind, and features a fine-grained, multi-level security model. The result is a language with distributed and network computation semantics which match the requirements of a loosely coupled system model.

In the remaining of the paper we present the language in a fairly informal manner. We begin by describing a fundamental view of network computation using an end-to-end data transfer example in Section 2. We then present a model of network computation with distributed continuations, as employed by MAST in Section 3 and proceed to analyse the language as a differential to Scheme¹ in Section 4. We continue with the security model in Section 5, and conclude by presenting a host of examples in Section 6, covering both high and low level protocol and service implementation. Finally, we briefly discuss related work in Section 7 and conclude the paper in Section 8

2 A Fundamental View of Network Computation

Network computation can be viewed interactions between objects living in different address spaces, logically connected through network channels. Network channels have service access points as their end-points: a service access point consists of a network address (IP address), a port number, and a low level protocol specification. The network address may designate a specific host in the Internet, as is the case for point-to-point connections, or a logical address which delivers to an open set of objects in the case of multicast. The low level protocol specification describes the service model of communication: TCP offers stream communication semantics, while UDP offers unreliable, datagram oriented communication semantics. We will use the term address interchangeably with service access point throughout the discussion.

Each object has an internal state, captured by an *environment*, and each interaction is captured by the *continuation* of the computation. The environment consists of the symbols accessible in a computation within the object, and the values associated with them (the denoted values). The continuation of the computation captures the exact state in the evaluation of an expression. Interactions between objects are driven by the exchange of messages. The *protocol* of interaction between objects dictates the acceptable contents of the messages – as to what expressions they may contain and what symbols they may reference during evaluation – and the effect of message exchange in the continuation of the computation.

Consider the following example interaction between two objects A and B, where A requests a byte from B, by specifying a sequence number (Figure 1). This simple interaction appears at some level in virtually all network interactions that require the transfer of information between two objects. The protocol is simple in this case: there are two types of messages, `get-message` and `put-message`. `get-message` carries a requested sequence number, and `put-message` carries a sequence number and the byte which is associated with this sequence number, as specified by the following grammar:

```
message ::= <get-message> | <put-message>
get-message ::= ("get" <integer>)
put-message ::= ("put" <integer> <byte>)
```

The semantics of the protocol, as illustrated in Figure 1, are stop and wait: A transmits a `get-message`, and waits for a response (`put-message`) from B.

We can describe the computation of A in Scheme as

(begin

¹General working knowledge of Scheme may be useful for following some subtle details; The reader is referred to [15, 1] for further details.

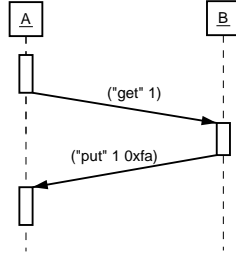


Figure 1: Example point-to-point object interaction

```

(sendto peer (message "get" next))
(let ((msg (receive)))
  (if (and (eqv? (message:type msg) "put") (eqv? (message:next msg) next))
    (begin
      (put-data next (message:data msg))
      (set! next (+ next 1))))))
  
```

Here we assume that exist procedures `sendto` and `receive` that send and receive the contents of the message over the network respectively. Similarly, `message` and related procedures construct a message formatted according to the protocol specification, and similarly extract fields from it.

In this code snippet, the environment of object A consists of all the symbols referenced and their associated values. The symbols `begin`, `sendto`, `let`, `if`, `and`, `eqv?`, `set!`, and `+` are bound to primitive scheme procedures. The symbols `sendto`, `receive`, `message`, `message:type`, `message:next`, and `message:data` bound to procedures for sending, receiving, and manipulating messages from the network. The symbols `peer` and `next` represent the local state of the object in the course of interaction, the symbol `peer` bound to the access point of B and `next` representing the sequence number of the next byte expected. Similarly, the symbol `put-data` is bound to an external procedure which modifies the local interaction state by inserting data into the buffer that holds the information transferred from object B. Finally, `msg` is a symbol whose lifetime is limited in the scope of the `let` expression.

The final part to note about the computation in object A is the continuation. The term continuation is used to denote the future or remaining part of the computation [9]. When evaluating a sequence of expressions $A_1 A_2 \dots A_n$, during the evaluation of expression A_k , the continuation is $A_{k+1} \dots A_n$. In our specific example, when `(sendto ...)` is evaluated, the continuation is `(let ...)`. Within the `(sendto ..)` expression, the sub-expression `(message "get" next)` appears in operand position. Hence it must be evaluated before the procedure `sendto` can be applied. During the evaluation of this expression, the continuation is the application of `sendto` plus the remaining expressions.

There some fine points to be made in this particular example, which is typical of network computation. Evaluation of expressions that involve the network may take arbitrary time to complete and may fail for reasons which are beyond the control of the local object. The application of `sendto` may fail, while the application of `receive` suspends the computation until there is a message to be received in the network stack (and may also fail). Most importantly, these two primitives implicitly modify the continuation of a computation in two different ways: `sendto` *activates* a continuation in a remote object, while `receive` *suspends* a continuation in a local object until a message is received.

In order to fully appreciate the effects of this activation and suspension in the computation, we need to consider the code that implements the functionality of object B:

```

(let loop ()
  (let ((msg (receive)))
    (if (eqv? (message:type msg) "get")
      (sendto peer (get-data (message:next msg))))))
(loop))
  
```

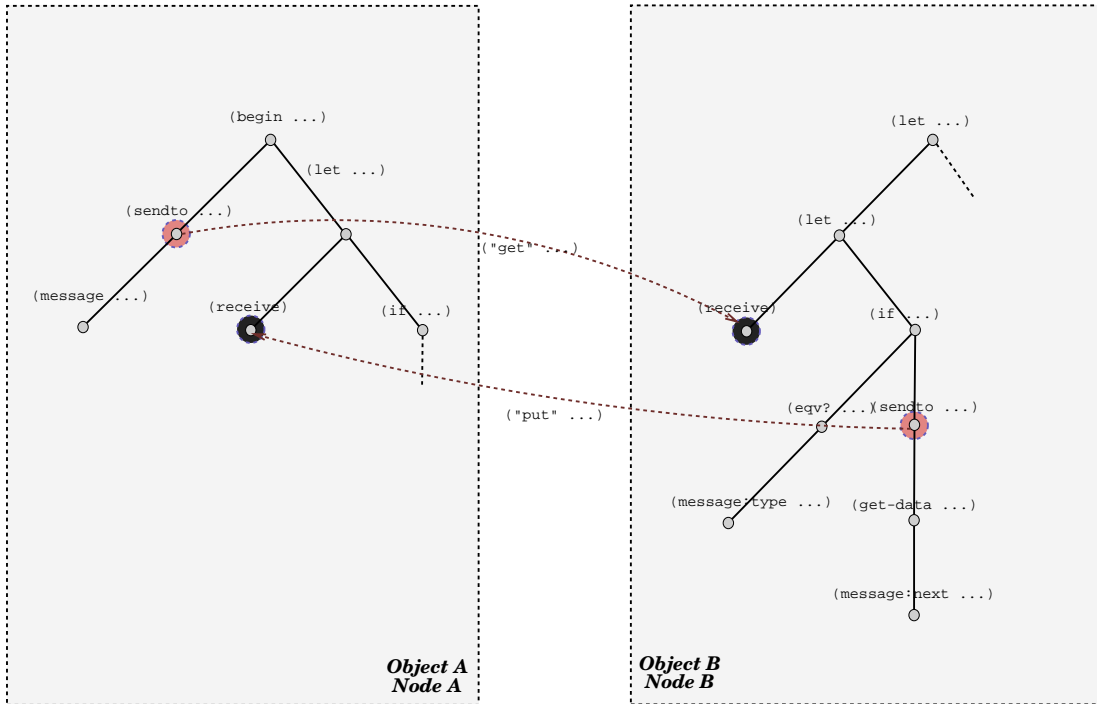


Figure 2: Transfer of control in the interaction of objects A and B

With this code, object B is in a loop. The computation is suspended in the beginning of the loop by the application of `receive` until a message is available, and in tandem re-activates the remote computation with `sendto`. Obviously this code omits details related to boundary conditions, but it is sufficient to demonstrate the the transfer of control between the two interacting objects.

The transfer of control is illustrated in Figure 2, where we show parts of the computation trees for the two objects. The trees are visited in left-to-right, depth-first order as each procedure is evaluated. Sub-trees represent expressions in operand positions that are evaluated before the parent node procedure. The points of interest are the connections between `sendto` and `receive` nodes, marked with arrows in the diagram. A computation that enters a `receive` subtree will suspend until an activation is received by a concurrent computation that enters the respective `sendto` node.

It is this transfer of control between processes executing in environments residing in different address spaces that characterises network and distributed computation. Even as interactions grow increasingly more complex than point-to-point communication, we can still express communication as transfer of control between remote processes. However, in order to fully capture the most complex of interactions, like multicast communication, we need to move beyond single activation of a suspended computation.

3 Network Computation with Environments and Distributed Continuations

With this view, environments and distributed continuations are the fundamental concepts behind network computation. MAST builds on this view by explicitly manipulating environments and distributed continuations as first-class objects; and in the process we move from a passive protocol specification to an active protocol implementation.

We distinguish between two flavours of environments: local and remote. A *local environment* is the local state of an object, as exported for access by other objects. The environment contains symbols which are accessible by code executing within its scope. A *remote environment* provides a local hook or proxy for an

environment exported by some object in an address, offering a logical connection to the actual environment where code may execute, as illustrated in Figure 3. To disambiguate between environments accessible through the same address, each environment carries a unique identifier. The identifier of an exported environment is supplied by the code which exports the environment to an address. Similarly, in order to evaluate code in a remote environment, its identifier must be known. In the same spirit, we introduce a node identifier, which logically identifies nodes in the network.

First class environments allow us to transform protocol specifications to procedure evaluations. We can show the transformation with the preceding example of basic get-put interaction as:

```
get-message → (get next) ⇒ put-message:data
```

That is, `get-message` is transformed to the `get` procedure. The `put-message` is now extraneous; the data is provided by the return from `get`.

With this transformation, the protocol implementation is completed by defining `get` at B's environment, and by transforming explicit message passing to procedure evaluations:

```
A → B: (get next)
;; ...
B: (define (get next)
    ...
    ⇒ <data>)
```

where `A → B` implies that the procedure is evaluated from the local environment of `A` to the exported environment of `B`, and `⇒` denotes a return of the procedure to the calling environment.

Figure 4 illustrates this interaction, together with the transfer of control during the computation. When a remote evaluation is attempted, the code for performing the remote evaluation is serialized, transmitted over the network, deserialized in the target environment, and gets evaluated. Similarly, we can transfer all forms of code over the network, including procedures, handles to environments, and so on. Notice how the explicit control of messaging, which required use of `sendto` and `receive` and handling of message formatting, disappears into the procedure evaluation, while preserving the semantics of the interaction.

Although `get` is defined as a procedure in its local environment, describing both the protocol interaction and data format, evaluation across environments is not as simple as a remote procedure call [19]. Rather, it is part of a distributed continuation. An evaluation in a remote environment captures the control of the current continuation and transfers it to a remote environment for further evaluation. Control returns to the point of execution after the code has been evaluated, but it *may return more than once*. Remote procedure calls on the other hand return exactly once or fail.

A single return cannot capture interactions that occur in *multipoint* communication scenarios. For instance, when the remote environment is bound to a multicast address, it is natural for the evaluation to return multiple times concurrently, as the procedure *is* in fact evaluated in multiple environments. This is the scenario illustrated in Figure 5. The figure illustrates the `A → B: (get next)` evaluation, when `B` is bound to a multicast environment, and `C` and `D` are environments exported to the address of `B`.

The flow of the computation becomes more important in multi-object interactions. For example, consider and interaction between three objects `A`, `B`, and `C`:

```
A → B: (get next)
;; ...
B: (define (get next)
    → C: (get next))
;; ...
C: (define (get next)
    ...
    ⇒ <data>)
```

In this example, `A` evaluates `(get ...)` in `B`'s environment. `B` however forwards the request to `C`, which performs the actual evaluation. The computation flow for this example is illustrated in Figure 6. Notice that

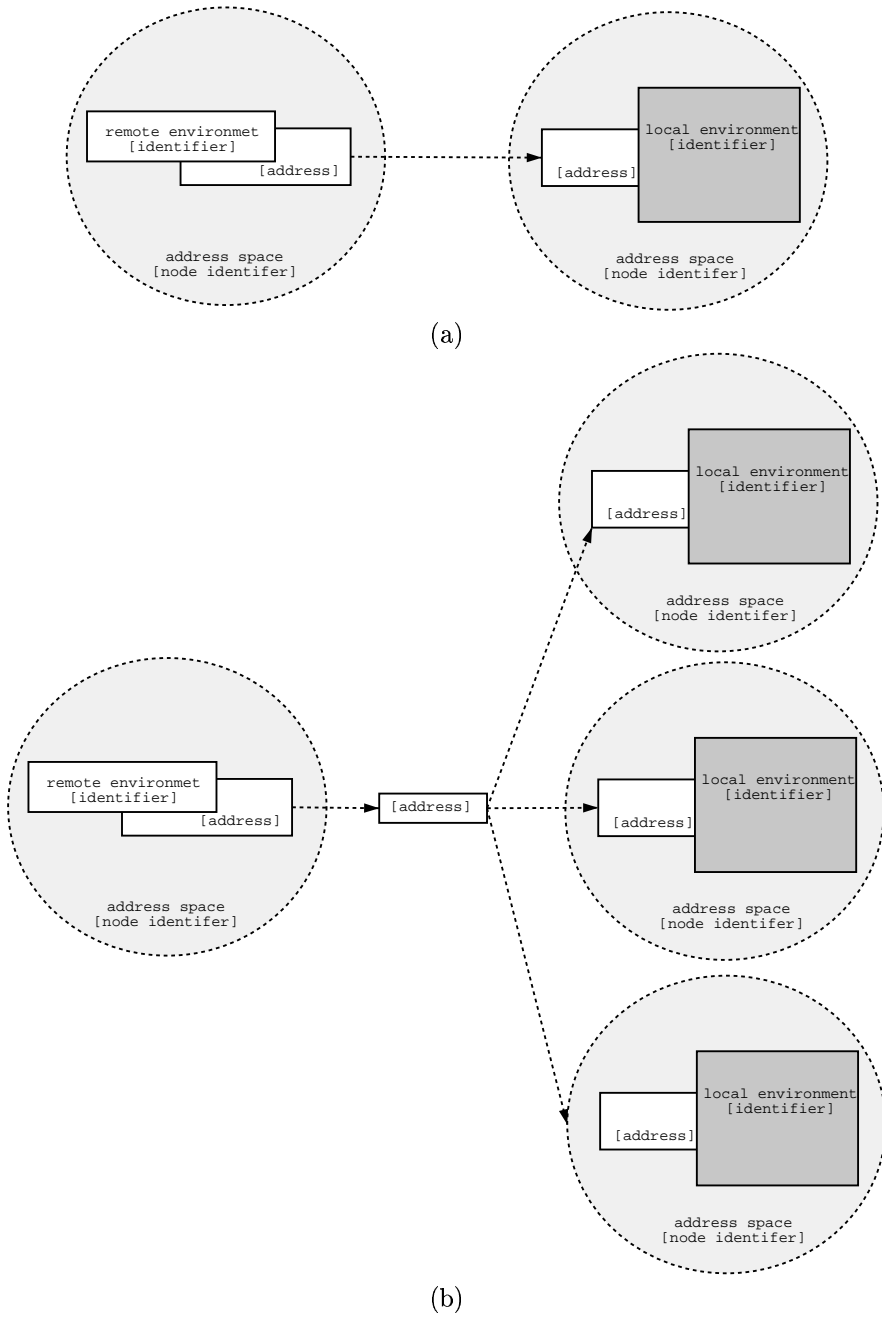


Figure 3: Local and Remote environments. (a) Remote environment in a unicast address (b) Remote environment in a multicast address

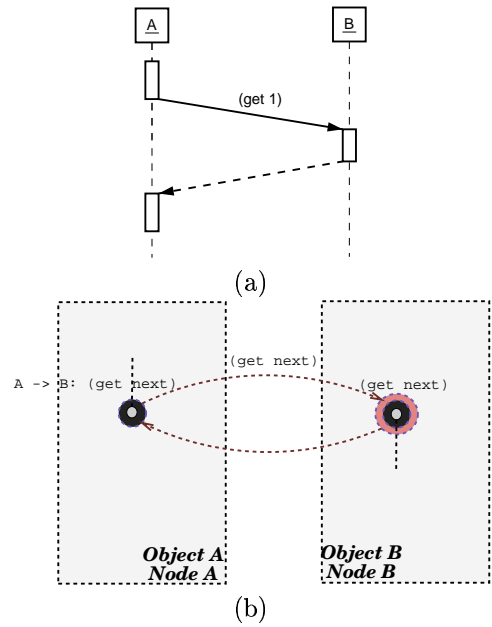


Figure 4: Protocol interaction with procedural abstraction and distributed continuation. (a) High level interaction. (b) Computation tree

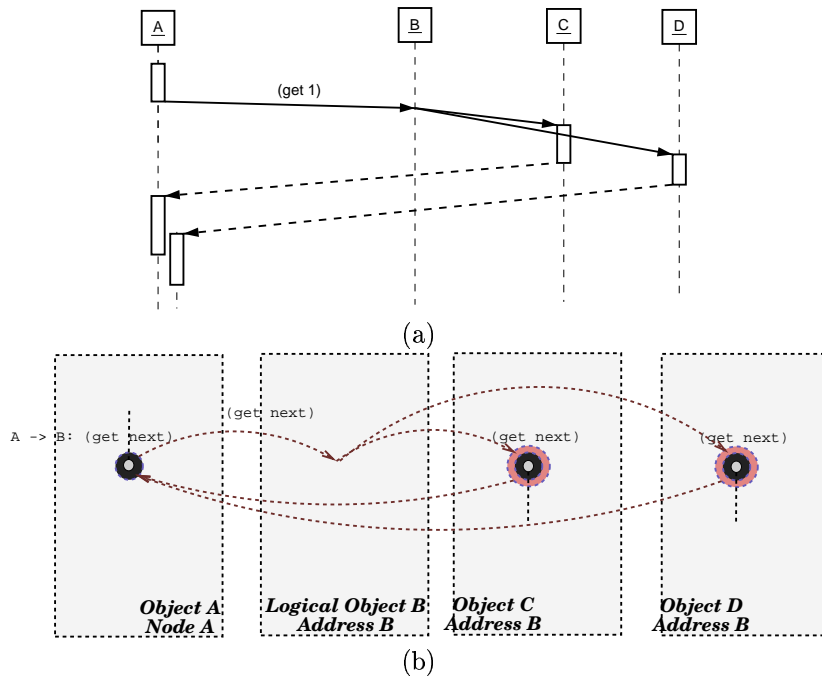


Figure 5: Multiple returns in a distributed continuation, through a multicast channel: (a) High level interaction (b) Computation trees

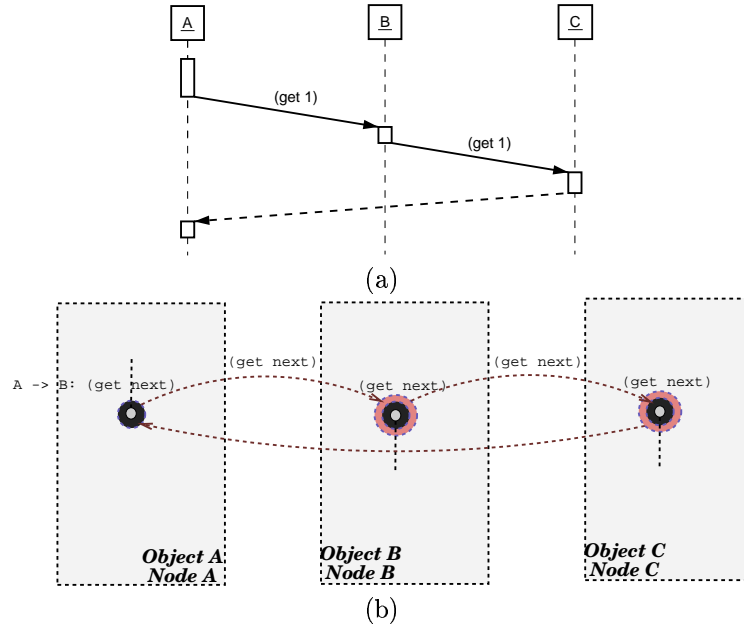


Figure 6: Forwarding as distributed tail-recursion. (a) High level interaction (b) Computation trees

the continuation of the evaluation ins C is in A; B only forwards the result to A. In a network environment, we want to avoid returning the result of the evaluation through B, as this only adds an additional hop in the computation, together with delay and the need to maintain context for the computation in B. The distributed continuation can easily capture the semantics of the computation flow: the evaluation from B to C of `(get ...)` is *tail-recursive*. The introduction of distributed continuations allows us to describe distributed tail recursion².

Finally, it should be noted that in Scheme continuations are first class objects. A continuation can be captured by using the primitive `call/cc` and thrown back later for an arbitrary number of times, introducing another way for performing multiple returns.

4 The Language

MAST builds on Scheme by adding language extensions for asynchronous and secure network computation. Language extensions are orthogonal to the original semantics, so that valid scheme code can be executed in a MAST interpreter without modifications.

The main language extensions can be summarised in the following:

- **Dynamic procedures and scoping.** In Scheme, procedures are statically (lexically) scoped and all referenced symbols are resolved at the definition environment. MAST provides dynamic procedures (α -forms), which have unresolved symbols, resolved at the evaluation context.
- **First class environments.** Together with scoping rules, first class environment allow us to capture local and distributed computation in a concise manner.
- **Remote evaluation, distributed continuation and tail recursion across address space boundaries.** We extend the continuation model of Scheme to include evaluations that transcend address space boundaries. Tail recursion is also extended for distributed calls.

²Scheme implementations are *required* to be tail-recursive. By adding distributed continuation we naturally capture distributed tail-recursion as well.

- Threading and parallelism primitives. We introduce asynchronous and parallel continuations, which are used as primitive constructs for concurrent evaluation, in a manner which is integrated with the continuation model of the language.
- Security model. Finally, we complement the language with a security model for distributed computation. Any code transmitted over the network can be verified and encrypted with user code. We also integrate fine-grained sandbox security, by allowing user code to explicitly define the symbols that can be accessed by code that is received over the network.

4.1 Scoping Operators

In order to facilitate scope resolution and enhance program readability we introduce two special scoping operators: `@` and `&`. `@` is used in conjunction with environments and sets a dynamic context for the evaluation of expression. `&` essentially performs the reverse action: it up-levels the context of evaluation for an expression.

Hence, in the expression

```
(@foo (bar sth))
```

`foo` should evaluate to an environment, which is then used as the dynamic context for the evaluation of `(bar sth)`. Both symbols are resolved in the context of the denoted value of `foo`. Similarly, in the expression

```
(@foo (bar &sth))
```

`sth` is uplevelled, and is resolved in the enclosing context of the entire expression. The operators expand to special forms `set-context!` and `uplevel!` which alter the current environment and evaluate an expression within their context.

```
(@A B) => ((set-context! A) (lambda () B))
```

```
&A => ((uplevel!) (lambda () A))
```

The use of the operators shall become apparent in the following sections.

4.2 Exception Handling

Scheme support for handling errors and exceptions comes from `error` and explicit uses of `call/cc`. However `error` is inadequate for most forms of sophisticated exception handling, as it is intended for interactive use. `call/cc` is an extremely powerful primitive that can implement any form of exception handling, but requires the explicit introduction of an error reporting protocol (i.e illegal values) for any procedure that may fail. To facilitate error checking we add exception handling primitives:

```
(raise <object>)
```

```
(catch <handler> body ...)
```

```
<handler> → (lambda (exc) ...)
```

`raise` raises its argument as an exception, which is handled by the closest enclosing exception handler. Exceptions may cross evaluation boundaries, so unhandled exceptions occurring at a remote environment will propagate backwards. `catch` evaluates the body. If an exception `exc` is raised within the body, the expression evaluates to `(handler exc)`. Side-effects of expressions evaluated before the exception occurs are not undone. The convention used for exception objects is `(<symbol> <string> ...)`, where the first element of the pair denotes the error type, the second element contains a human readable error message, and remaining elements contain additional information specific to the exception type.

4.3 Procedures and Symbol Binding

MAST supports two forms of procedures: λ -forms, which are usual statically bound scheme procedures, and α -forms, which are dynamically bound.

Syntactically, a λ -form is created by

```
(lambda (args ...) body ...) ⇒ <procedure>
```

For example,

```
(lambda (x y) (if (eqv? x y) x y))
```

creates a procedure which takes two arguments, and if the two arguments are equivalent it evaluates to the first; otherwise it evaluates to the second. Thus, the following expression evaluates³ to 2:

```
((lambda (x y) (if (eqv? x y) x y)) 1 2) ⇒ 2
```

The syntax for creating an α -form is similar:

```
(alpha (args ...) body ...) ⇒ <dynamic-procedure>
```

However, this expression creates a procedure with all free symbols unbound. Therefore, in the procedure created by

```
(alpha (x y) (if (eqv? x y) x y))
```

the symbols `if` and `eqv?` are unbound. The result of the evaluation of the following expression cannot be determined at the defining environment:

```
((alpha (x y) (if (eqv? x y) x y)) 1 2) ⇒ ?
```

The rationale behind dynamic procedures is support for evaluation in external environments. When a dynamic procedure is transferred over the network for evaluation, unbound symbols will not be transferred; they are bound at the remote evaluation context. Furthermore, since unresolved symbols can be locally determined, the communication pattern for transferring and remotely evaluating a procedure is deterministic. This is in contrast to systems like Kali Scheme [14], where denoted values are transferred on demand. That approach is unsuitable for the environment where MAST operates, as remote environments may be connected to multicast addresses or in potentially untrusted hosts. In the case of multicast, the incremental approach is guaranteed to cause implosion, while in the untrusted host case the programmer has no control over which denoted values will actually be remotely provided.

In the above example, when the procedure is transferred over the network for evaluation at a remote environment, the definitions of `if` and `eqv?` need not be transferred. Instead, they will be bound to the local definitions upon evaluation. Note that the same effect can *not* be achieved with quotation or quasiquotation. The reason is that `eval`, operates on the root environment. This not only hinders controlled resolution at the remote end, it also presents a significant security threat by giving access to the root environment to untrusted code.

In tandem, we introduce some more special forms for dealing with dynamic procedures:

```
(dynamic-procedure? <obj>) ⇒ <boolean>  
(free-symbols <obj>) ⇒ list-of <symbol> | exception: type-error  
(static-bind <obj> . symbols) ⇒ <dynamic-procedure> | <procedure> | exception: type-error  
(dynamic-bind <environment> <obj> . symbols) ⇒ <dynamic-procedure> | <procedure>  
| exception: type-error | exception: unresolved-symbol
```

³assuming that `if` and `eqv?` have not been rebound at the definition environment.

`dynamic-procedure?` evaluates to `#t` if the argument is a dynamic procedure and `#f` otherwise. `free-symbols` evaluates a list of all free symbols in the procedure (the empty list if the procedure has no free symbols) or raises a `type-error` if the argument is not a procedure. Finally, the binding operators evaluate to a procedure with the symbols specified in the optional variable bound to the current context (or all the free symbols if used with one argument). `static-bind` binds the symbol the current lexical environment, while `dynamic-bind` explicitly binds symbols from a specific environment. Both procedures raise a `type-error` if the first argument is not a procedure. `dynamic-bind` also raises an `unresolved-symbol` if some of the symbols could not be found in the environment argument. Note that a dynamic procedure with no free symbols is equivalent to a procedure. In addition, evaluation of a dynamic procedure will first statically bind the procedure and then evaluate.

For example, since `(lambda (x y) (if (eqv? x y) x y))` is a normal procedure:

```
(dynamic-procedure? (lambda (x y) (if (eqv? x y) x y))) => #f
(free-symbols (lambda (x y) (if (eqv? x y) x y))) => '()
```

Similarly, for `(alpha (x y) (if (eqv? x y) x y))`:

```
(dynamic-procedure? (alpha (x y) (if (eqv? x y) x y))) => #t
(free-symbols (alpha (x y) (if (eqv? x y) x y))) => (if eqv?)
(free-symbols (bind (alpha (x y) (if (eqv? x y) x y)) 'if)) => (eqv?)
(free-symbols (bind (alpha (x y) (if (eqv? x y) x y)))) => '()
```

The `&` operator can also appear in the body of a dynamic procedure to force a static binding at the defining environment. Hence, the following two expressions are equivalent:

```
(alpha () (display &foo))
(static-bind (alpha () (display foo)) 'foo)
```

with `foo` statically bound in the resulting procedure.

4.4 Environments and Scoping

Environments are first class objects which contain bindings from symbols to MAST objects. Environments can exist in isolation, adding simple object-oriented capabilities to Scheme, can be exported to local network service access points becoming execution contexts for incoming code, and can be logically connected to remote network access points providing a way to evaluate code to remote contexts. Conceptually, they map to the environments of the computation model presented in Section 3. In conjunction with dynamic procedures and the scoping operators of Section 4.1 we can write code which completely captures the semantics of distributed interactions and issues of symbol resolution [21].

Environments have a multiple inheritance model with left-to-right depth first resolution. A new environment can be created by extending one more existing environments, using procedure `extend-env`. The root environment is always an empty environment and can be obtained with the procedure `root-env`. Finally, the predicate `environment?` checks if an object is an environment:

```
(root-env) => <environment>
(extend-env <environment> . rest) => <environment>
(environment? <obj>) => <boolean>
```

Symbol resolution and evaluation in the context of an environment uses the `@` operator. For example, consider the following sequence of expressions:

```
(define apple (extend-env (root-env)))
```

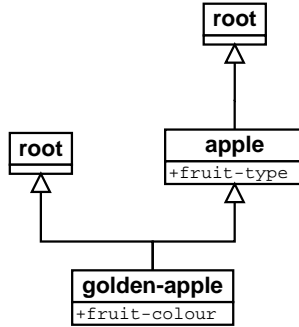


Figure 7: Environment inheritance and symbol resolution for the fruit example.

```

(@apple (define fruit-type "apple"))
(define golden-apple (extend-env (root-env) apple))
(@golden-apple (define fruit-color "golden"))

```

The environment tree created by this sequence of expressions is illustrated in Figure 7. In the expression `(@apple (define fruit-type "apple"))`, we bind the symbol `fruit-type` to the value `"apple"` to the `apple` environment. The expression is fully evaluated in the `apple` dynamic context. However, as the symbol `define` is unbound within the dynamic context, it is resolved to the lexical context of the expression, and similarly for the definition of `fruit-color` at the `golden-apple`. Now consider the expression

```
(display (@golden-apple fruit-type))
```

This expressions displays the string `"apple"`, with resolution of the symbol visiting first the left parent of `golden-apple` (an empty environment) and then the right parent (`apple`).

4.5 Remote Environments and Distributed Continuation

An environment can be made accessible over the network by exporting it to a service access point. This is achieved by the `export-env` procedure:

```

(export-env <environment> <environment-identifier> <network-address> . options)
⇒ <exported-environment>

```

An exported environment is connected to an underlying environment as by inheritance. Incoming code will have this environment as dynamic context, and symbol resolution will occur using the left-to-right inheritance model. Notice however that incoming code has no lexical scope, hence symbols that have not been explicitly bound to the environment are not externally accessible, providing environment sandboxing (Section 5.3).

Environment identifiers in MAST are 16-element byte vectors⁴, allowing for a 128-bit environment space. Network addresses encapsulate service access points in the network:

```

(make-inet-address <host> <port> <type>) ⇒ <network-address>
(make-inet6-address <host> <port> <type>) ⇒ <network-address>
(make-unix-address <path> <type>) ⇒ <network-address>
<type> → 'stream | 'dgram

```

Similarly there are predicates for checking the address type of an object, and procedures for extracting various fields from an address.

⁴Since byte vectors are frequently used for representing and transferring raw data, MAST provides a built-in `byte-vector` datatype.

The options in `export-env` specify ancillary parameters for the exported environment, like timeouts, background exception handlers, and end-to-end security policy as explained in Section 5.2. Options are (`<symbol>` `<value>`) pairs. For instance,

```
(export-env an-env an-identifier an-address '(timeout 120000))
```

exports `an-env` with an activity timeout of 2 minutes.

Conversely, in order to do remote evaluation, it is necessary to connect to a remote environment:

```
(connect-env <environment-identifier> <address> . options )  
⇒ <remote-environment>
```

with similar option semantics. Having obtained a `remote-environment`, we can remotely evaluate an expression with the usual scoping rules. For example the expression

```
(display (@a-remote-fruit fruit-type))
```

where `a-remote-env` is bound to a remote environment, will serialise the expression `fruit-type`, fully resolve it, and evaluate at the referenced environment. If the actual environment(s) to which `a-remote-fruit` contain a binding to `fruit-type`, this will be displayed to the standard output stream. Environments where the symbol cannot be resolved will throw an `unresolved-symbol` exception.

Environments can be referenced with environment handles. An environment handle is a pair consisting of the environment identifier and the address to which it is bound. A handle can be obtained with `environment-handle`:

```
(environment-handle <exported-environment> | <remote-environment>) ⇒ <environment-handle>  
<environment-handle> → (<environment-identifier> <address>)
```

It should be noted that remote evaluations are *blocking*. When the result of an evaluation is retained, the continuation will block until the evaluation is complete or an exception results in case of errors. Furthermore, there is no binding between a `remote-environment` and the actual environment to which it logically connects. Therefore, if there is no exported environment existing at the remote end of the logical connection, a blocked expression will eventually time out; there are no exceptions thrown back for non-existence of an environment at the remote end.

An additional consequence of the loose coupling between environments is the outcome of multiple returns to a continuation. In the example above, if the remote environment is connected to a multicast address, the continuation of the remote evaluation (in this case `(display .)`) will return multiple times. However, the result will be displayed as long as the local part of the continuation is accessible. If the current continuation is not locally accessible when the evaluation returns control, a `invalid-continuation` exception will be delivered to the background exception handler and also be propagated back to the caller.

Since remote evaluations may return more than once, we introduce an additional primitive that evaluates to the peer environment of the current continuation:

```
(peer-env) ⇒ <remote-environment>
```

With this primitive, we can obtain information about the current peer during multiple returns as is the case of remote evaluation in multicast address.

Finally, remote evaluations are only possible within the context of an exported environment. The reason is twofold: any remote evaluation must have a return path and it must carry identification information for security purposes (Section 5).

4.6 Continuation Control, Threads, and Synchronisation Primitives

Continuations constitute an important part of the language, as we have presented it so far. Continuations in MAST go beyond Scheme continuations, as they can capture distributed control transfer (Section 3). Naturally, as in Scheme, continuations captured with `call/cc` have unlimited extent but with one important caveat: The extent of a distributed continuation cannot transcend the extent of its local counterpart. To make this more concrete, consider the following code snippet:

```
(define orange (extend-env (root-env)))
(@orange (define eater '()))
(@orange (define (take-a-bite)
  (call/cc (lambda (exit)
    (set! eater exit)
    (exit #t))))))
(export-env orange ...)
```

In this case, `orange` is exported to some address. When a remote object evaluates `take-a-bite` in the `orange` context, the continuation is *captured* by `(set! eater exit)`. Subsequently, local code could jump back into the remote evaluation context by evaluating `(@orange (eater sth))`. This call will only be valid while the remote context is still accessible in its local environment. In other words, garbage collection is oblivious to remote references: distributed continuations are *weakly* referenced. If a jump to a distributed continuation that has been reclaimed is attempted an `invalid-continuation` exception will be raised.

However, not all remote evaluations need evaluate to a meaningful value. This also comes in place with multicast environments: in order to control implosion, some of the remote ends will need to suppress the result of the computation. In order to enable this type of control, we provide an `abort/cc!` (`abort-current-continuation!`) primitive. Invocation of `abort/cc!` results in aborting the current continuation. If the continuation is local, an `aborted-continuation` exception will be thrown. If the continuation is in a remote address space, the exception will not propagate, thus controlling implosion. In addition, tail-recursion across address space boundaries is automatically handled by the interpreter as part of the distributed continuation. Therefore, when a nested tail calls to remote environments will forward the results directly to the originating environment.

Without a form of concurrency, the blocking semantics of remote evaluation limit the type of interactions we can effectively program to serial ones. While continuations can be used for implementing user level threading [7], the use of threads is an essential part of network programming. Therefore, MAST provides built-in threading capabilities, with two primitives: `call/ac` (`call-with-asynchronous-continuation`) and `call/pc` (`call-with-parallel-continuation`).

Continuations created with `call/ac` are concurrently executing *subcontinuations* [13], which end at an asynchronous completion token:

```
(call/ac <proc>) ⇒ <asynchronous-completion-token>
proc → (lambda (<current-continuation> ...) ⇒ <object>
```

An `asynchronous-completion-token` is a special object which will hold values or exceptions resulting from the evaluation of the asynchronous procedure. It is a synchronisation primitive and concurrently evaluating continuations can wait on it for the evaluation result of the asynchronous procedure with `wait-for-completion`:

```
(wait-for-completion <asynchronous-completion-token> . time) ⇒ <boolean>
(wait-for-any-completion <tokens> . time) ⇒ <number>
tokens → list-of <asynchronous-completion-token>
```

The procedure can have an optional upper bound on the waiting time, and evaluates to `#t` if the asynchronous continuation has returned in the meantime. If 0 is used, the procedure returns immediately. The second form, `wait-for-any-completion`, can be used for multiplexed wait on multiple completion tokens. It returns the number of completion tokens with pending conditions or 0. The completion condition and value can be retrieved with the following procedures:

```
(value-pending? <asynchronous-completion-token>) ⇒ <boolean>
(exception-pending? <asynchronous-completion-token>) ⇒ <boolean>
(accept-completion <asynchronous-completion-token>) ⇒ <object>
```

For example, the following code snippet⁵ will greedily take bites at a remote orange until it has succeeded 10 times, and dispatch 'done-eating to the token:

```
(@an-orange-eater
  (let*
    ((bites 0)
     (token (call/ac
             (lambda (exit)
               (let loop ()
                 (if (@remote-orange (take-a-bite))
                     (begin
                       (set! bites (+ bites 1))
                       (if (> bites 10) (exit 'done-eating))))
                 (loop))))))
     (wait-for-completion token)))
```

The formal argument of the asynchronous procedure is the continuation. As with any other continuation, it can be captured and return multiple times. Results are queued in the asynchronous completion token until they are consumed. For example, consider in the following snippet of code:

```
(define signal '())
(define pend (call/ac (lambda (exit) (set! signal exit) #t)))
```

every time we evaluate the expression (signal <object>), the argument will be added to the pend completion queue.

To avoid potential memory leaks, asynchronous continuations are also weakly referenced. An asynchronous procedure which is not referenced by user code will evaluate only once, as a one-shot continuation [3]. The decision for whether the continuation is one-shot is made when the asynchronous procedure completes. If the asynchronous completion token or the continuation (the formal argument of the procedure) is not accessible by other continuations, it is marked for reclaim by the garbage collector and can no longer execute if remote evaluations jump into it. In addition, user-code can forcibly terminate an asynchronous continuation by using the `force-complete!` procedure, which delivers a `continuation-terminated` exception to the completion token and marks the continuation as non executable.

The second primitive, `call/pc`, spawns a list of procedures which continue in the current continuation:

```
(call/pc <procs>) ⇒* <proc-result>
procs → list-of <proc>
proc → (lambda (<current-continuation>) ...) ⇒ <proc-result>
```

To make this more concrete, consider the following simple example:

```
(display (call/pc (list (lambda (exit) "apple") (lambda (exit) "orange"))))
```

This expression will spawn the two procedures and *concurrently* continue in the `display` expression. Hence, "apple" and "orange" will be concurrently displayed to the standard output stream. Parallel continuations are particularly useful in constructing tail-recursive expressions for application-level multicast. For example,

```
(@eater (define (eat all-fruit)
  (call/pc (map (lambda (fruit)
                 (lambda (exit) (@fruit (take-a-bite))))
               all-fruit))))
```

⁵assuming `remote-orange` is connected to an orange remote environment as described above

the `eater` environment will take bites to all the fruit environments contained in the `all-fruit` list in parallel. If `eat` is part of the distributed continuation, the results of the bites will return directly to the remote-end of the continuation.

Finally, MAST includes usual synchronisation primitives for avoiding race conditions: mutex and read/write locks, condition variables, and semaphores.

4.7 Serialisation

Serialisation of code to external bytecode representation and vice-versa is prevalent throughout the language⁶. Whenever a remote evaluation is requested, the code to be evaluated is pickled to its bytecode representation, transferred over the network, unpickled at the destination, and evaluated. Serialisation primitives are also available at the user-level:

```
(pickle <object>) => <byte-vector>
(unpickle <byte-vector>) => <object>
```

When an object is serialised, symbols it refers to will also be serialised, creating a full environment that captures the relevant state of the interpreter. The difference between procedures and dynamic procedures becomes relevant in serialisation. In a dynamic procedure free symbols are encoded as missing references, and bound to symbols provided by the context when used. When the first expression is serialised, the result is a dynamic procedure with a reference to an external symbol `apple`. In the second case, the symbol `apple` and its definition and dependencies will be included in the serialised form

4.8 Revisiting the Example

Let us now revisit the opening example about the get-put interaction of two communicating objects. Let `A` and `B`'s environment carry the identifiers I_A and I_B respectively, and P_A and P_B be their access points. Note that we hid P_A and P_B in the previous section, by implicitly assuming they are bound to the `peer` symbol, and avoided defining the way connections are established and maintained. With the following code, `A` will retrieve `bytes` bytes:

```
(define A (export-env (extend-env (root-env))) I_A P_A)
(define B (connect-env I_B P_B))
(@A (let loop ((next 0))
      (put-data (@B (get &next)))
      (if (< next bytes) (loop (+ next 1)))))
```

The code for `B` is even simpler:

```
(define B (extend-env (root-env)))
(@B (define (get next) (get-data next)))
(export-env B I_B P_B)
```

Notice how the structure of the code changes to more closely reflect the semantics of the interaction. The actual protocol is simple: the consumer requests a byte and the producer returns it, in a stop-and-wait fashion. Previously we required external procedures `message`, `message:type` to explicitly construct and access the fields of the messages. But these parts were actually irrelevant to what we were trying to achieve with the interaction; the only things that really matter are the procedures `put-data` and `get-data`, which provided access to the consumers and producers of the bytes transferred through the network.

⁶Natively defined procedures and primitives like `if` cannot be serialised.

5 Security Model

5.1 Basic Model

The security model of the language is designed to preserve end-to-end semantics. The objective is to provide a sound security model for protecting the interpreter and the user-space, and support for user code to define further security and encryption primitives according to its requirements. This is a language-based protection model, where user-code completely controls the level of security required.

The measures necessary to protect a mobile code system can be broadly classified in three orthogonal categories: interpreter protection, computational environment or user-space protection, and end-to-end protection. Interpreter protection ensures that malicious code cannot harm critical sections of the runtime system and cause it to behave in an invalid way. User-space protection requires clear separation between environments that are part of different processes running in the interpreter, and restricting malicious code from crossing the boundaries between them. Thus, code being evaluated in some environment should not have unauthorised access to symbols or procedures defined in other environments that may be present in the interpreter. Finally, end-to-end protection involves user-dependent encryption and/or verification of code. User-code should be able to verify incoming segments of code and grant them suitable credentials for the extent of their evaluation within the interpreter. Similarly, user-code may require encryption of code before transmission in order to protect its function or embedded data.

In order to achieve interpreter and user-space protection and enable end-to-end protection, we take a multi-level security approach:

- An access controller and a verifier/decoder can be attached to any exported environment.
- An encoder can be attached to any remote environment proxy.
- Code credentials, assigned by the verifiers, are accessible throughout the computation.
- Environments are isolated, and only explicitly bound symbols are accessible to code evaluated in the context.

5.2 Primitives

Every exported environment handle has an associated user-defined access controller and a message decoder. These are specified by the `'access-control` and `'decoder` options of `export-env`:

```
access-control → (lambda (<address> <node-identifier> <environment-identifier>) ...) ⇒ <boolean>
decoder →
  (lambda (<node-identifier> <environment-identifier> <byte-vector>) ...)
    ⇒ (<object> <byte-vector>) | #f
```

The access controller and decoder can be attached to the environment post-creation with the `set-access-control!` and `set-decoder!` primitive procedures.

The access controller grants access according to source address filtering of incoming messages. Every incoming message carries the source node identifier, source and target environment identifier, and a code segment which contains the raw byte-level description of the code to be evaluated. The decoder operates on the code segment: it extracts the actual code from the segment according to user-defined criteria and grants a user-defined credential which escorts the code during the continuation of its computation. If verification or decoding fails, the verifier evaluates to `#f`, and the message is discarded. A `verification-failure` exception is also delivered to the background exception handler if present.

The procedure may verify a digital signature on the message or decryption of the message, and is described as a user procedure. The source node identifier is a unique key which can be used for referencing encryption keys or digital signatures for these purposes. The process of receiving a message and extracting the code for evaluation is summarised in Figure 8.

The code credentials can be any MAST object, and can later be accessed and checked by user code at any point during the subsequent computation with the procedure `credentials`:

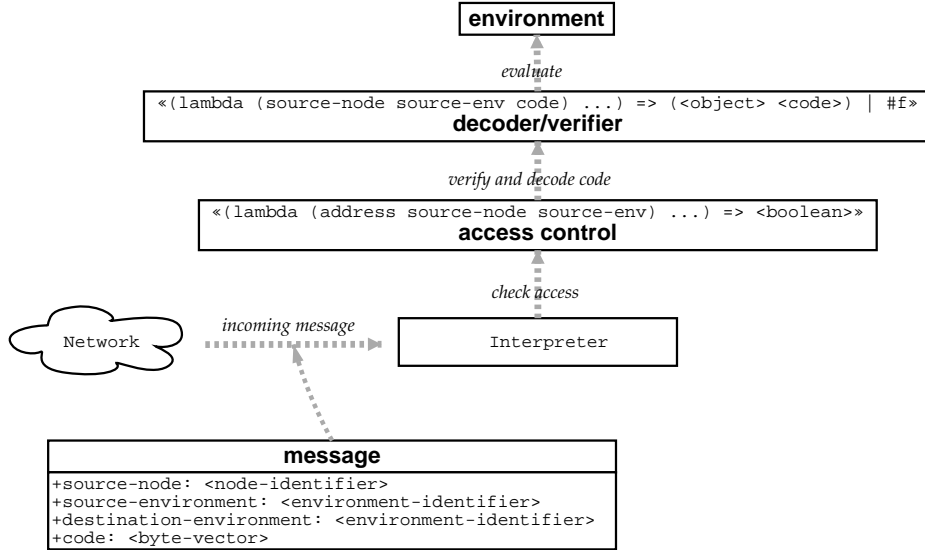


Figure 8: Access control, verification and decoding of incoming messages

`(credentials) => <object> | #f`

In addition, the procedure `local-continuation?` is provided, which will evaluate to `#t` if and only if the current continuation is not part of a distributed continuation:

`(local-continuation?) => <boolean>`

User code can use this procedure for restricting access to sensitive procedures to the local address space only.

A similar process is followed for encoding messages for evaluation to remote environments: Remote environments may have an encoder, passed as a `'encode` option to `remote-env`:

`encode -> (lambda (<environment-identifier> <object> <byte-vector>) ...) => <byte-vector>`

The encoder accepts three arguments: the target environment identifier, the credentials object in the continuation, and the raw byte level representation of the transported code. Similar to the decoder, the encoder can be set post-creation using the primitive procedure `set-encoder!`.

The encoder transforms the raw byte representation of a message according to user-defined criteria. For instance, if encryption or authentication is required, the encoder encrypts or signs the data representation of a message before being transmitted over the network. The verifier/decoder of the local environment is responsible for transforming messages that are received during a control transfer as the distributed computation unwinds to a form that can be evaluated by the interpreter.

Credentials do not accrue over successive remote evaluations. For example, consider the continuation $A_1 R_1 A_2 R_2 A_3 \dots$, where A_k are local and R_k remote evaluations. On evaluation of $A_2 R_2 \dots$ the credentials are the credentials of the result of R_1 , and in $A_3 \dots$ the credentials are the credentials of the result of R_2 . The evolution of the credential as the computation unwinds is shown in Figure 9.

5.3 Environment Sandbox

The sandbox security model was introduced with Java, as a means to execute untrusted code in a restricted environment. The original model was coarse-grained and was later refined [11] to offer more fine-grained control on defining a security policy. Exported environments in MAST offer equivalent fine-grained sandbox

Continuation	(credentials)	(local-continuation?)
$A_1 R_1 A_2 R_2 A_3 \dots$	#f	#t
$R_1 A_2 R_2 A_3 \dots$	#f	#t
$A_2 R_2 A_3 \dots$	c_1	#f
$R_2 A_3 \dots$	c_1	#f
$A_3 \dots$	c_2	#f

Figure 9: Credential accumulation in the evaluation of $A_1 R_1 A_2 R_2 A_3 \dots$. c_k denote the credentials that accomodate a return from R_k

protection. Incoming code can only access symbols explicitly bound in an environment, and cannot cross environment boundaries. By default, environments are empty until populated with symbols by user code.

We can illustrate environment sandboxing with the following example, implementing a simple mutex for synchronising distributed processes⁷:

```
(define a-lock (extend-env (root-env)))
(@a-lock (define owner '()))
(@a-lock (define lock (make-mutex-lock)))
(@a-lock (define (acquire)
  (with-lock lock
    (if (null? owner)
      (begin
        (set! owner (peer-env))
        #t)
      #f))))
(@a-lock (define (release)
  (with-lock lock
    (if (equal? owner (peer-env))
      (begin
        (set! owner '())
        #t)
      #f))))
(export-env a-lock ...)
```

Distributed processes can synchronise on this lock by evaluating `acquire` and `release` in the context of an environment connected to `a-lock`. External processes can also access the `owner` and `lock` symbols, but they cannot modify them. For example, evaluating `(@a-lock-proxy (set! owner "sth"))` will fail with an `unresolved-symbol` exception, as `set!` is not accessible to incoming code. The symbol is accessible to the defined procedures `acquire` and `release` though, as it is part of their defining environment. The effect of sandboxing is illustrated in Figure 10.

6 Examples

We close the presentation of MAST with some non-trivial examples that illustrate how the language can be used for building active services and protocols. Our examples are a high-level example of brokered data transfer service, similar to Napster, an active messaging example that implements a Gnutella-like system, and a data-transfer service that uses a downloadable transport protocol.

6.1 Brokered Data Transfer

Brokering is a common pattern of interaction in distributed and multi-agent systems. In this pattern, an intermediary entity – the broker – receives advertisements from producers of information. Consumers conduct

⁷The example uses the `make-mutex-lock` and `with-lock` MAST procedures. The first constructs a mutex lock, and the second evaluates the a sequence of expressions while holding a mutex

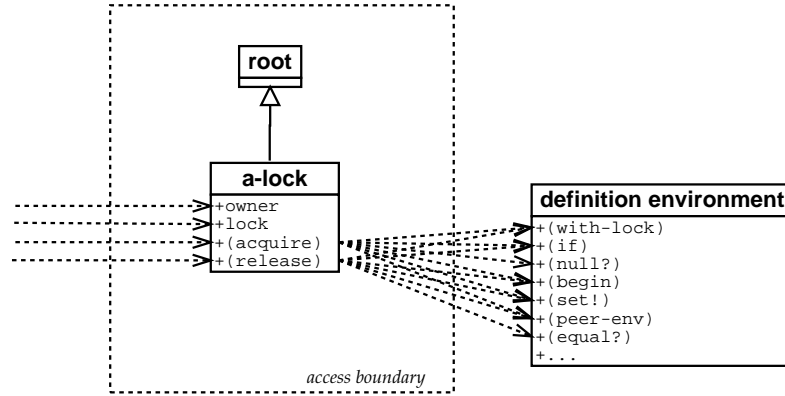


Figure 10: Environment sandbox for the distributed mutex example

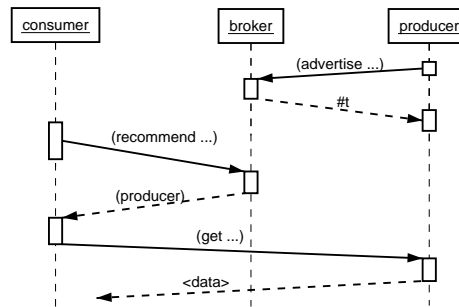


Figure 11: Recommendation and data transfer in the brokering example.

the broker with a query that is matched against the advertisements from producers. Then the broker has a handful of choices: it can forward the query to a matching producer with the reply forwarded directly to the consumer (recruiting); it can forward the query to a producer, receive the reply and forward it to the consumer (facilitation); and it can return handles to the matching producer to consumer (recommendation). The broker in our example works as a recommender, as Figure 11 illustrates.

6.1.1 The Broker

The interface of the broker consists of two procedures:

```
(advertise <string> <object>) ⇒ #t
(recommend <string>) ⇒ list-of (<environment-handle> <string> <object>)
```

Producers advertise strings and supply arbitrary objects to be used as keys with the procedure `advertise`. Consumers ask for a recommendation using `recommend`, with a regular expression as an argument. The procedure evaluates to a list of matches; a match consists of an environment handle, the description string, and the key supplied by the producer. The complete code of the broker is shown in Figure 12. The procedure evaluates to a broker environment which has been exported to the supplied address with the supplied identifier.

6.1.2 The Producer

For the producer, we take a similar approach, illustrated in Figure 13. The main procedure of the interface is `advertise`, which establishes an advertisement with a broker and keeps a reference to a data source

```

;; make a broker environment.
;; id is the environment identifier to use
;; addr is the address where the environment should be exported
(define (make-broker id addr)
  ;; advertisement list
  (define entries '())
  ;; read-write lock for accessing entries
  (define lock (make-read-write-lock))
  ;; broker environment
  (let (broker (extend-env (root-env)))
    ;; advertise: add an advertisement to the advertisement list
    (@broker (define (advertise ad key)
      (with-write-lock lock
        (if (null? entries)
            (set! entries (list (list (environment-handle (peer-env)) ad key)))
            (append! entries (list (environment-handle (peer-env)) ad key))))))
    ;; recommend: return a list of matches to a regular expression
    (@broker (define (recommend query)
      (with-read-lock lock
        (filter (lambda (entry) (regexp:match? query (list-ref entry 1)))
          entries))))
    ;; export the environment and return the result
    (export-env broker id addr)))

```

Figure 12: Broker implementation in the broker example

constructor procedure. The constructor procedure, when invoked with a key, should evaluate to a procedure that sequentially produces all the data in the stream. Data transmission takes place when a consumer invokes `get`. The producer spawns a new asynchronous continuation which pushes the data in the background. Finally, it should be noted that no assumptions are made for the formatting of the data. The source can produce any type of object deemed convenient, in an application-level framing fashion. The only assumption in the interaction is that transmission is reliable; hence, this producer code should be used over TCP, by a specifying a `stream`-based address.

6.1.3 The Consumer

The code that implements the consumer environment is illustrated in Figure 14. To use this environment, client code should call `recommend` for obtaining a match list from the broker. Then, after selecting a source from the match list, it should use the `get` procedure of the environment. Similarly to the producer, client code supplies a procedure that will receive the actual data from the producer, supplied as an argument to `get`.

6.2 Gnutella with Active Queries

Gnutella is well-known peer-to-peer system. It uses a diffusion algorithm for propagating queries around an overlay network, controlling the depth of the search with a `ttn` parameter. In Figure 15, we show how we can implement a gnutella node which uses *active queries* for searching reachable nodes. With active queries, instead of having a restricted language like the wildcard match of the standard gnutella protocol, we can have arbitrary procedures to control the search. This is similar to the approach taken by mobile agent systems like MESSENGERS [10], with the distinction that the message is not self-routed.

The interface to the node is simple: `send-query` will send a query to the network, and `accept` will accept a network query, and propagate it further if `ttn` has not been exceeded and there are no matches in the local node. There are only two subtle details in the code: local node security and concurrent searching.

```

;; make a producer environment
;; id is the environment identifier to use
;; addr is the address where the environment should be exported
(define (make-producer id addr)
  ;; internal procedure to asynchronously put data to the peer.
  ;; local is the exported local environment
  ;; peer is the remote target environment
  ;; source is a data producer procedure, which evaluates to #f as eof
  ;; exit is the asynchronous continuation
  (define (put local peer key source exit)
    (let loop ()
      (let ((next (source)))
        (@local (@peer (put key &next)))
        (if (not next) (exit 'done))
        (loop))))
  ;; producer environment
  (let ((producer (extend-env (root-env))))
    ;; list of active asynchronous continuations delivering data
    (@producer (define active '()))
    ;; list of sources for providing data
    (@producer (define sources '()))
    ;; mutex lock for accessing the lists
    (@producer (define lock (make-mutex-lock)))
    ;; get: send the data for the file exported with the key
    ;; file transfer occurs in an asynchronous continuation
    (@producer (define (get key)
      (let ((source-entry (with-lock lock (assoc key source))))
        (if source-entry
            (let ((token (call/ac
                          (lambda (exit)
                            (put (current-env) (peer-env) key ((cadr source-entry) key) exit))))
              (with-lock lock
                (if (null? active)
                    (set! active (list token))
                    (append! active token)))
                #t)
            #f))))
    #f)))
  ;; make an advertisement
  ;; broker is the broker remote environment to use
  ;; ad is the advertisement string
  ;; key is the key for the advertisement
  ;; source is a procedure that can provide a new data stream for the key
  (@producer (define (advertise broker ad key source)
    (@broker (advertise &ad &key))
    (with-lock lock
      (if (null? sources)
          (set! sources (list (list key source)))
          (append! sources (list key source))))))
  ;; export the env and return the result
  (export-env producer id addr))

```

Figure 13: Producer implementation in the broker example

```

;; make a consumer environment
;; id is the environment identifier to use
;; addr is the address where the environment should be exported
(define (make-consumer id addr)
  (let ((consumer (extend-env (root-env))))
    ;; data receptor list
    (@consumer (define receptors '()))
    ;; receptor list mutex
    (@consumer (define lock (make-mutex-lock)))
    ;; put: invoked by the producer to deliver a frame of data
    ;; the frame is forwarded to the appropriate receptor
    (@consumer (define (put key data)
      ((cadr (assoc key receptors)) data)
      (if (not data) ;; no more frames
          (with-lock lock
              (set! receptors
                    (filter (lambda (entry) (not (equal? (car entry) key)))
                          receptors))))))
    ;; recommend: ask a broker for a recommendation
    (@consumer (define (recommend broker query)
      (@broker (recommend &query))))
    ;; get: retrieve data from a producer
    ;; source is the producer environment
    ;; key is the producer specific key
    ;; rcv is a procedure that will receive data frames
    (@consumer (define (get source key rcv)
      (with-lock lock
        (if (null? receptors)
            (set! receptors (list (list key rcv)))
            (append! receptors (list key rcv))))
      (if (@source (get &key))
          #t
          (begin
            (with-lock lock
              (set! receptors
                    (filter (lambda (entry) (not (equal? (car entry) key)))
                          receptors)))
            #f))))))
    ;; export the environment and return the result
    (export-env consumer id addr)))

```

Figure 14: Consumer implementation in the broker example

```

(define (make-node id address neighbour-list local-info)
  (let ((node (extend-env (root-env))))
    ;; neighbours at the overlay network
    (@node (define neighbours neighbour-list))
    ;; locally available stuff
    (@node (define local local-info))
    ;; accept: accept an active message as a visitor.
    ;; If the message finds any matching data in the local node, send the
    ;; results back to the original node. Otherwise, and if the ttl is positive
    ;; propagate to the neighbours in the overlay network
    (@node (define (accept visitor ttl)
      (let ((matches (filter visitor local)))
        (cond
          ((not (null? matches)) matches)
          ((and (> ttl 0) (> (length neighbours) 0))
           (call/pc (map (lambda (hop)
                         (lambda (exit) (@hop (accept &visitor &(- ttl 1))))))
                     neighbours)))
          (else (abort/cc!)))))) ;; suppress flooding from non-matching leaves
    ;; allowed symbols for visitors (security)
    (@node (define if &if))
    (@node (define cond &cond))
    (@node (define and &and))
    (@node (define or &or))
    (@node (define regexp:match? &regexp:match?))
    ;; and so on...
    ;; ...
    ;; send a query to the overlay.
    ;; It may return multiple times or never, so do it asynchronously
    ;; Access is restricted to local continuations only.
    (@node (define (send-query query ttl)
      (if (local-continuation?)
          (map (lambda (hop)
                (call/ac (lambda (exit)
                          (let ((match (@hop (accept &query &ttl))))
                            (list (peer-env) match))))))
              neighbours)
          (raise '(permission-denied "Restricted procedure!")))))
    ;; export the environment
    (export-env node id addr)))

```

Figure 15: Node implementation for the gnutella example.

For security purposes, we cannot allow the visitor to execute arbitrary code to the local node. Hence, we restrict the symbols available to the query by explicitly binding them to the context. In the example we only allow boolean tests and regular expression matching:

```
(@node (define if &if))
(@node (define cond &cond))
(@node (define and &and))
(@node (define or &or))
(@node (define regexp:match? &regexp:match?))
```

The query language can be easily extended by binding more symbols to the environment.

Concurrent search is handled by using `call/ac`:

```
(map (lambda (hop)
      (call/ac (lambda (exit)
                (let ((match (@hop (accept &query &t1))))
                  (list (peer-env) match))))
      neighbours)
```

The code constructs a list containing the completion token for each query thread. User code can then asynchronously check for results to the query with `wait-for-any-completion`. The result of each query includes both the match list and the environment where the match occurred.

Futher queries are spawned by non-matching nodes:

```
(cond
  ((not (null? matches)) matches)
  ((and (> ttl 0) (> (length neighbours) 0))
   (call/pc (map (lambda (hop) (lambda (exit) (@hop (accept &visitor &(- ttl 1))))
                 neighbours)))
   (else (abort/cc!)))) ;; suppress flooding from non-matching leaves
```

Notice the use of `call/pc` for directly forwarding the result to the original sender of the query, and the suppression of fail messages at non matching leaves with `abort/cc!`. The way the parralel search occurs is illustrated in Figure 16. The figure shows how the computation unwinds in the overlay network, and how we may have multiple returns in some paths of the continuation. This is not a problem however, as results are queued in the completion tokens.

6.3 Downloadable Transport Protocols

In this final example we show how we can implement a service with a downloadable transport protocol. Here, a sender wishes to transmit data to a receiver, but does not have an implemenation of the protocol for transmitting the data. To solve the problem, it conducts the receiver and retrieves an implementation of the transport protocol as a dynamic procedure. It then uses this dynamic procedure for fetching the data from the source. The receiver-side of the implementation is shown in Figure 17, and the sender-side in Figure 18. Although the implemented protocol is a simple and grossly inefficient stop-and-wait, any type of protocol can be transported in the same way. The moral of the example is how complete protocol implementations can be embedded in mobile code, in a manner similar to the scheme employed by active routers.

7 Related Work

Many of the ideas employed in MAST have analogs in other systems and languages. In particular, we discuss remote evaluation and distributed continuation, the use of environments for abstracting distributed computation, and sandbox security models in the light of previous work.

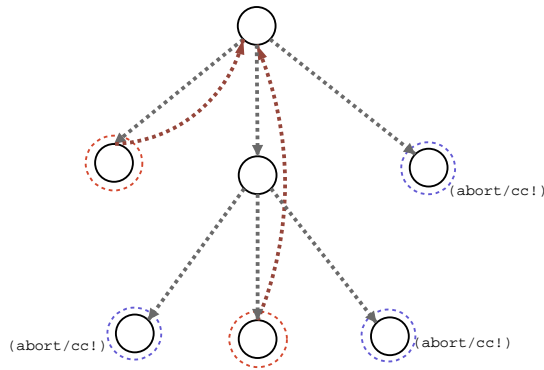


Figure 16: Computation trace of parallel search in the gnutella example.

```

;; make a dynamic receiver
;; id is the environment identifier to use
;; addr is the address where the environment should be exported
;; consume is the data consumer procedure
;; initial-timeo is the initial stop+wait timeout
(define (make-dynamic-receiver id addr consume initial-timeo)
  (let ((receiver (extend-env (root-env))))
    ;; get-transport: send a transport protocol sender
    (@receiver (define (get-transport)
      ;; return a dynamic procedure implementing the sender side of transport
      ;; tgt is the target environment
      ;; produce is a local procedure that produces data frames
      (alpha (tgt produce)
        (let loop ((next (produce))
                   (cur-timeo &initial-timeo)) ;; initial timeout
          (let send ((token (call/ac (lambda (exit) (@tgt (put &next))))
                    (timeo cur-timeo))
                    (if (wait-for-completion token timeo)
                        (if (value-pending? token)
                            (if next
                                (loop (produce) timeo) ;; send next frame
                                #t)) ;; eof
                            (raise (accept-completion token)) ;; failure. raise the exception
                        (send ;; timeout - retransmit
                            (call/ac (lambda (exit) (@tgt (put &next))))
                            (* 2 timeo))))))))) ;; double the timeout
      ;; receiver side of transport
      (@receiver (define (put next) (begin (consume next) #t)))
      ;; export eht environment to the supply address
      (export-env receiver id addr)))

```

Figure 17: Receiver and transport provider of the downloadable transport protocol example.

```

(define (make-dynamic-sender id addr)
  (let ((sender (extend-env (root-env))))
    ;; add symbols necessary for the transport protocol to work
    (@sender (define let &let))
    (@sender (define lambda &lambda))
    (@sender (define call/ac &call/ac))
    (@sender (define if &if))
    (@sender (define wait-for-completion &wait-for-completion))
    (@sender (define value-pending? &value-pending?))
    (@sender (define raise &raise))
    (@sender (define accept-completion &accept-completion))
    (@sender (define * &*))
    ;; send: retrieve a transport protocol from the receiver and send the data
    ;; produce is a procedure which produces data frames.
    (@sender (define (send receiver produce)
      ((@receiver (get-transport)) receiver produce)))
    (export-env sender id addr)))

```

Figure 18: Sender of the downloadable transport protocol example.

The idea of remote evaluation, as an exchange of control between distributed entities, was introduced in [19]. Distributed continuations move a step further, allowing programs to explicitly capture the transfer of control across address space boundaries. Kali Scheme [14] supports distributed continuations with a strongly-coupled distributed model, completely abstracting the notion of the network. Internally, Kali uses an on-demand consistency scheme for symbol linking and continuation migration, and includes mechanisms for distributed garbage collections. Unlike Kali Scheme, MAST is designed to operate in loosely coupled environments and potentially untrusted environments. Hence, we take a lower level approach and give complete control to the programmer concerning symbol linking and garbage collection. Furthermore, MAST is intended to be used for low level protocol implementation, including multicast protocols, where the actual details of the interaction should be in the control of the programmer. Finally, the ability to explicitly link symbols at local environments, allows MAST to be used as a partial distributed evaluation mechanism.

The use of environments as a mechanism for controlling remote evaluation was explored with the Network Objects toolkit [2]. RMI [20] uses a similar mechanism for representing remotely accessible objects through stubs. MAST takes a similar environment-based approach to distributed continuation. Unlike other systems, however, the loosely coupled model employed by MAST allows us to express distributed computation in loosely coupled and multicast environments.

Finally, sandboxing and security policies were popularised with Java [11]. Typically, in policy based security, a security manager is installed and permeates all interactions between local and potentially untrusted code. The approach take by MAST does not require a security manager. Rather, the symbol resolution mechanism provides a secure policy, where the programmer has complete control over the symbols and resources accessed by untrusted code. Additional security primitives are enabled in a purely end-to-end fashion, as the language allows user code to provide custom code encoding, verification, and authentication.

8 Conclusion

In this paper we present MAST, a dynamic programming language for network, distributed, and peer-to-peer programming using distributed environment and continuation abstractions. MAST extends Scheme with additional constructs and semantics for network and distributed programming. We introduce first class environments and dynamic procedures, a built-in lightweight threading model integrated with continuations, remote evaluation and support for distributed continuation with a loosely coupled model, code mobility and a sound security model. The result is a high level mostly functional language suitable for programming a wide range of network and distributed protocols and applications. Low level point-to-point and multipoint

protocols, high level distributed protocols, and interacting protocols can all be designed and implemented with ease and reasonable performance.

References

- [1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.
- [2] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Technical report, Digital Systems Research Center, 1995.
- [3] C. Bruggeman, O. Waddell, and R.K. Dybvig. Representing control in the presence of one-shot continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.
- [4] L. Cardelli. A language with distributed scope. In *Proc. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1995.
- [5] D. Chess. Security issues in mobile code systems. In *Mobile Agents and Security*, number 1419 in Lecture Notes in Computer Science, 1998.
- [6] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM'90*, 1990.
- [7] R.K. Dybvig and R. Hieb. Continuations and concurrency. *ACM Transactions on Programming Languages and Systems*, 1990.
- [8] D. Tennenhouse et. al. A survey of active networks research. *IEEE Communications*, 1997.
- [9] D. Friedman, C. Haynes, and E. Kohlbecker. Programming with continuations. *Program Transformation and Programming Environments*, 1984.
- [10] M. Fukuda, L. Bic, M. Dillencourt, and F. Merchant. Messages versus messengers in distributed programming. In *Proc. International Conference on Distributed Computing Systems*, 1997.
- [11] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [12] D. Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, 1997.
- [13] R. Hieb, R.K. Dybvig, and C. Anderson. Subcontinuations. *Lisp and Symbolic Computation*, 1994.
- [14] S. Jagannathan, H. Cejtin, and R. Kelsey. Higher order distributed objects. *ACM Transactions on Programming Languages and Systems*, September 1995.
- [15] R. Kesley, W. Clinger, and J. Rees. The Revised⁵ report on the algorithmic language scheme. *Higher Order and Symbolic Computation*, September 1998.
- [16] F. Knabe. *Language Support for Mobility*. PhD thesis, Carnegie Mellon University, 1995.
- [17] L. Moreau, D. DeRoure, and I. Foster. NeXeme: a distributed Scheme based on Nexus. In *Proc. International Europar Conference (EURO-PAR'97)*, 1997.
- [18] J. Saltzer, D. Reed, and D. Clark. The end-to-end argument in system design. *ACM Transactions on Computer Systems*, 1984.
- [19] J. Stamos and D. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 1990.
- [20] Sun Microsystems. *Java Remote Method Invocation Technical Specification*, 1996.
- [21] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, March 1997.
- [22] D. Wetherall and D. Tennenhouse. The active IP option. In *Proc. ACM SIGOPS European Workshop*, 1996.