Programming Languages Features for some Global Computer

Ulisses Ferreira
Department of Computer Science
Trinity College Dublin, Dublin 2, Ireland
E-mail: ferreirj@cs.tcd.ie or ulisses@ufba.br

***Abstract*— This paper introduces a number of programming languages concepts that are useful for mobile agents and for the Internet. In particular, we explore the ability to program with the unknown value, called *uu*, by giving examples of constructs which can be combined sequentially. Although the present characteristics are not necessarily for mobile-code languages, the underlying and unified global environment is a single one.**

I. Introduction

For writing the present paper, we observed that, in order to provide a reasonable paradigm for general Internet programming, some combined and non-traditional programming language concepts need to be introduced aiming at specific applications over the Internet.

Code mobility is a relatively new field of research that has inspired intriguing ideas on programming techniques to improve related software. New programming languages have been presented and discussed in workshops and conferences aiming at providing better standards and good examples for future language designs, commercially or otherwise.

To propose a better model for mobile agent programming for the World Wide Web, for example, the language designer should consider that the underlying connections often fail or delay. A neutral state, which would represent the lack of result, could be assigned to the variable that takes part in the request in such a way that the program carries on running safely. Mobile agents need to be robust and make their own decisions remotely.

Although the present programming language features are not necessarily specific for mobile-code languages, the underlying environment can be the same. Here we explain how this additional value can be useful in programming for a global environment where the mobile agent paradigm and technology have become increasingly important. In logics, this value has been traditionally referred to as *uu* for providing an alternative value to $true$ (*tt*) and $false$ (*ff*). Like many imperative features, the present ones also apply to functional languages.

Broadly, some pieces of work, such as [1], [2], have indicated similarities between technologies of code mobility and persistence, and some persistent languages are being explored at some universities. In the present approach, because we are looking for generality and efficiency, persistence is not provided directly by the language, but instead by programmable constructs. Because of that, this approach here is at least as applicable to mobile agents as persistent languages. As well as persistence, communication is another very active topic of research and much work has been done regarding fault tolerance and communication between mobile agents. However, relatively few satisfactory results have been achieved in terms of language facilities and abstractions.

Here we concentrate on the use of programming features in the context of a global environment and mobile agents programming.

The ability to represent and reason with partial information is well understood in the artificial intelligence and logic communities. However, very little of this work has been related to programming techniques. An exception is Extended Logic Programming, introduced by Gelfond and Lifschitz[3], [4], that can be used for the same purpose as that we are discussing here. Extended Logic Programming makes use of two forms of negation. In [5], the author suggests that an important practical problem in Extended Logic Programming is how the programmer distinguishes whether a negative condition is to be interpreted as explicit negation, or as negation due to the absence of any clause in any closed world as an assumption.

The $unknown$ value, *uu*, extends the semantics of other logics, such as classical and intuitionistic logic, according to the Łukasiewicz [6], [7] 3-valued logic. Here we extend this value for each data type in programming for a global environment or for mobile agents. In arithmetic and relational expressions, *uu* as a resulting operand implies the expression to result in *uu*. Accordingly, statements have to be adapted to make use of this value.

Most Expert System Shells made use of an $unknown$ symbol to represent lack of information in boolean variables, in a very restrictive way however. We generalize this concept to programming languages in general, although we apply it here to mobile agents programming.

Agents have to be robust and, because of this, when connections fail or delay, programs should carry on running despite the lack of information. *uu* is a constant in programming languages that can be assigned to any variable of any datatype. This new constant guarantees both safety and robustness at the same time, because variables are never committed to any value that is not in the problem domain. An introduction on types for functional programming languages can be found in [8].

In section II we review some recent programming languages for such an environment, while section III is dedicated to an illustrative programming language. Section IV introduces the

concept of *unknown* together with other related concepts, and explains how they can be used to achieve the proposed goals. As a consequence, section V complements those concepts by stressing the importance of any form of lazy evaluation in programming, as well as timeouts, no matter the adopted paradigm. Section VI briefly discusses logic programming on global computers while section VII also describes strong mobility. Section VIII contains other relevant features that are relevant enough to be mentioned. Section IX contains the conclusion.

Finally, Appendix describes a secure mobile agent system. The examples in this paper are written in a programming language. In section III, we discuss the syntax of the relevant subset of this language.

II. Some Current Mobile-Code Languages

In the past few years researchers have seen the Internet as a popular environment for systems. Some of us would like to program and compute using this structure, i.e. to view parts of an Internet-like network as *global computers*[9]. Many companies, for example, are starting to have their own internal global computers for their specific purposes.

In 1995, Sun Microsystems presented Java[10] programming language with the stress on the interesting idea of permitting code mobility on the World-Wide Web. Portability of code has become critical to software development. Some $online\ portable$ languages (i.e. taking a running program and port it to a different architecture while it is running) have recently been designed[11], whose characteristics will be discussed in this section. Type systems, scoping, name resolution and dynamic linking are some of the key concepts in this context. According to Cardelli, "languages that are not on-line portable will be abandoned because they do not provide what is increasingly perceived as basic functionality: mobility"[11].

However, one of the most interesting ideas is not only to move code, as Tcl[12] and Java[10] do, but also computation (code along with context) over the network, that is, a computation which starts at some location may continue to execute at some other location. Synchronous connections to the original site may be set while a program is running remotely in such a way that any change in some variables transparently causes the value to be stored in the original site. Alternatively, new values of variables can be sent to the original site with no need for synchronous connections. Other paradigms of mobile computation already exist and they depend on the kind of entities that are transfered over the network, with respect to what is moved (code, data, connections, etc).

When the code is moved, what happens if the names it contains are bound to resources in the source virtual machine? This issue defines two classes of strategy, *replication* and *sharing*. The first strategy may be either static or dynamic. Concerning static replication strategy, constants, system variables and libraries, for example, are regarded as *ubiquitous resources*[13] and they can adopt such strategy, where bindings are deleted and set after arrival. As for dynamic replication strategy, the code migrates to another virtual machine along with bound resources and the original bindings are deleted. The original resource in the source virtual machine may be either deleted (*replication by move*) or kept (*replication by copy*). In the *sharing strategy*, the original resource is kept and remotely accessed through network references and connections.

In both strong and weak mobility[14], security[15], [16] is a very important matter. Locations must check for authorization and capabilities in order to prevent malicious software running. However, as long as that is ensured by the system, a global network can be a very interesting and natural platform for computation. Thus, a new challenge emerges: how to provide these facilities and prevent the related problems?

Mobility should also be considered not only during the execution of programs but also during the elaboration of software. The emphasis in performance is no longer in the run-time code generated by compilers, but in the (dynamic) compilation process itself (when applicable), transmission and additional overheads to guarantee security and other requirements.

Acharya, Ranganathan and Saltz[17], [18] during the design of Sumatra, an extension of Java, consider three requirements for **individuals**: $awareness$, which is the need to monitor the level and quality of resources in their operating environment; $agility$, which is the ability to react to changes in resource availability, and $authority$, which is the ability to control the way that resources are used. Although they are important concerns, we think that these concerns should be treated at the application level, not at the language level.

Some programming languages for mobile computation are described and analyzed in [14] and other articles, and briefly described here:

- Java[10] is a strongly-typed object-oriented language. Java deals with security[19] and allows transmitting program byte-code to be interpreted by the Java Virtual Machine[20], but does not migrate computation. It supports weak mobility with dynamic linking. Security level is increased by the byte-code verifier at loading-time. Some security problems have been found[21]. It was shown that the ability to break Java type system leads to an attacker being able to run arbitrary machine code[22]. Static and dynamic type checking.
- Telescript[23] is an agent-based and object-oriented language that explicitly deals with locality, strong mobility, and finiteness of resources. There are two kinds of Execution Units: agents and places. Typically, when an agent is running on an interpreter, the instruction go causes the agent execution to be suspended, its code and current state are transmitted to a remote virtual machine and, there, the computation is resumed. However, agents do not maintain connections to remote agents. The Telescript runtime code is interpreted without security checking since security is ensured at the language level. The replication strategy is dynamic, by move. Static scoping and name resolution. Static and dynamic type checking. In spite of the historical reasons for mentioning Telescript here,

that technology was replaced by Odyssey[24], which is a Java-based version of Telescript, briefly speaking.

- Tycoon[25] provides thread migration like Telescript. It is a polymorphic, higher-order functional language with imperative features, which may support other paradigms indirectly, including object orientation. Tycoon provides strong mobility and support for persistent programming. All objects in this language have first-class status. Static and dynamic type checking, dynamic replication with strategy by copy, besides static replication strategy.
- Agent Tcl[26] provides strong mobility where the whole image of the interpreter can be transfered to a different site by executing a $jump$ instruction. Agent Tcl also provides weak mobility by executing a $submit$ instruction which allows transmission of procedures along with part of their global environment, to a remote interpreter. Typeless language therefore no type checking. Dynamic replication strategy, both by copy and by move. Agent Tcl is a PhD thesis[27].
- Safe-Tcl[28] supports active e-mail, where messages may include code to be executed when an interpreter reads the message after receiving it. However, Safe-Tcl does not support active e-mail code mobility at the language level but, instead, code mobility is achieved through a dynamic code loading mechanism. Typeless language therefore no type checking.
- Obliq[29], [30] is an object-based language that encourages distribution and mobility. While a mobile object is migrating from one place to another, new connections are automatically open between source and destination places in order to guarantee that any change in the variables will update the state in the source place. Therefore, object references are transformed into network references. Although a simple language, there is some loss of efficiency and robustness due to some possibly very large number of connections in an unreliable environment. Dynamic type checking, sharing strategy.
- Facile[31] is a functional language, a superset of ML with primitives for distribution, concurrency and communication. Mobile code programming was later added to this extension[13]. Static and dynamic type checking. Dynamic replication strategy by copy, besides static replication strategy.
- TACOMA[32], [33], the Tcl language plus primitives to allow a running Tcl script to send another script and initialization data to another host in order to execute the script remotely. Typeless language therefore no type checking. Dynamic replication by copy.
- M0[14], [34] is a stack-based interpreted language which provides weak mobility and run-time type checking. Dynamic type checking, dynamic scoping rules, dynamic replication by copy.

Aglets Workbench, developed by IBM, is a mobile agent system based on Java. Like others, such as ObjectSpace Voyager, the system security and other issues depend on the Java system[35].

As mentioned before, in the present paper, we discuss some of the features of the programming language.

III. The Present Programming Language

The present sample programming language is a language that supports mobile agents, syntactically somewhat similar to Java. It supports strong mobility, as well as some forms of knowledge and belief representation, reasoning, and uncertainty treatment. As an on-going experimental project, the language has not been scaled up and security has not been a concern. Communication between agents has not been implemented either. The virtual machine interprets byte-code and the language provides both replication strategies by programmable handlers. BNF legend: boldface letters are keywords; italic words with initial capital letter are other terminal symbols; words in lower-case letters are non-terminal symbols; meta-symbols: | indicates alternative, ε is the empty symbol of the grammar. Other terminal symbols: { (, ;) } are used in the grammar. The following BNF definition is of a very simple subset of the programming language in question, and where the first symbol denotes the starting symbol of the syntax:

aprog $\longmapsto$ classlist commandlist

classlist $\longmapsto$ ϵ | classdef classlist

type $\longmapsto$ **int** | **list**

modifier $\longmapsto$ **private** | **public** | ε

onevardef $\longmapsto$ Id | assignment

idlist $\longmapsto$ onevardef | onevardef ',' idlist

vardef $\longmapsto$ modifier type idlist ';'

handler $\longmapsto$ evaluator | reactor

evaluator $\longmapsto$ **when** Id ',' **do** command

reactor $\longmapsto$ **when** Id ':=' **do** command

classdef $\longmapsto$ **class** Id '{' defs '}'

defs $\longmapsto$ ε | vardef defs | function defs | handler defs

command $\longmapsto$ assignment | '{' commandlist '}'
| functioncall | ifcommand | **return** |
return expression

assignment $\longmapsto$ Id ':=' expression

ifcommand $\longmapsto$ **if** expression **then** command |
if expression ',' command |

if expression ',' command **ifnot** command |
if expression ',' command **else** command |
if expression ',' command **otherwise** command |
if expression ',' command
ifnot command
otherwise command

commandlist $\longmapsto$ ϵ | command ';' commandlist

where non-terminal symbols, namely function, functioncall and expression are as usual. In the programming language which is being presented, they are somewhat syntactically similar to C++ or Java. The main difference is that the symbol $ can be placed where a variable identifier is expected, as it will be explained below. There are other details that will be explained together with the examples. As stated, the Appendix formalizes the semantics that will be explained.

IV. *uu* IN GLOBAL COMPUTERS

For every data type, the language designer can add a special value, namely *uu*, to represent lack of some *domain value*, i.e. some known value in the problem domain. For integers, there is uu^i; for real numbers, there is uu^r and so on. We simply write *uu* to mean that the type is irrelevant in such context. Accordingly, we write *value* in the singular form to mean that its type is not important in the sentence. Grammatically, *uu* or $unknown$ is a constant. Variables either contain *uu* or some domain value. In advance, besides other applications of *uu* in some programming language, *uu* can support fault tolerance over the Internet, and this will be clear while the subject is introduced.

Some languages adopt a default value as initial variable contents. But since one now has *uu*, we ought to adopt this value as the initial one for every variable. The programmer should certainly want to initialize some variables with different values.

For any variable in the program, *handlers* can be attached. They can be one *evaluator* and/or one *reactor*, independently. As well as other purposes, one handler can protect a variable. The idea of evaluator is to allow the programmer to write a piece of code to produce and provide some domain value for the corresponding variable, while the idea of reactor is to inspect and protect the variable against assignments. Thus, a reactor allows the programmer to write a piece of code to react instead of letting values be stored unconditionally in the corresponding variable.

int x, y;

when x, **do** { $x := 3 * y$; }

when $x :=$ **do** { $x :=$ $; }

In the above example, two handlers are defined for the variable x. The first time that the value of x is being requested in an expression, the above evaluator is triggered, which in turn computes the triple of the value of the variable y assigning it to x. From the second time on, the computed value $3 * y$ is already available in x and, because of this, the evaluator is not triggered. This idea is not limited to *exception handling*, a mechanism supported by some other languages, and this will become clearer soon.

An evaluator can contain **return** statement (similar to C) as an alternative to assigning a value to the requested variable. In the case of the **return** statement and no prior assignment in the evaluator, the evaluator is always triggered when that variable is being used, unless some domain value has been assigned to that variable outside the evaluator.

Whenever a value is to be stored in x, the control is jumped to the corresponding reactor. Notice that the $ symbol above is used in reactors to represent the value that, in other languages, would be stored unconditionally. In the above example, the value is accepted.

Built-in predicates can be provided to check whether a variable contains *uu*, for example, **known** and **unknown**. In these cases, the value is accessed directly and the $boolean$ result from the condition is provided by the interpreter without evaluating the handler of that variable.

The use of variables in expressions can have innovative semantics:

If the variable contains some value in the problem domain, the semantics is exactly the same as in imperative languages. However, if the variable contains *uu*, the semantics is divided by two separate subcases: if there is an evaluator, it is executed. Otherwise, i.e. when there is no evaluator, *uu* is used instead. However, the semantics of the execution of evaluators is not similar to the semantics of function calls, because the latter are always executed. In the case of Remote Procedure Call or Remote Method Evaluation, this unconditional call is probably inconvenient or inefficient in programming.

In terms of design, *uu* and handlers replace exception handling in other languages. This might relatively simplify the language. Handlers are very useful during program *testing* and *debugging* phases, by inspecting what is being stored and, since mobile agents might escape from the user, *uu* together with handlers can be used in mobile agent programming. For example:

class $mycl$ {
public int x;
private list $queue$:= [ ];
when x := **do** {
x := $;
$queue$:= $queue$ +
[[$\#self$ + ".x := " + $ + " at " + $LocalTime$()]];
}
}

$mycl$ c; $c.x$:= 10; $c.x$:= 20; $c.x$:= 30;

In the above class or its subclasses, whenever x receives a value, it is also stored in the queue together with the name of the object ($\#self$), the name of the field (x) and the current

local time ($LocalTime()$). The '+' operator concatenates lists or strings, besides the arithmetic addition, as usual. The square brackets are used to construct a list of values of any type. Here the programmer chose list of lists for programming reasons.

Because we know that it is very difficult to implement the mobile agents debugging system in a satisfactory way, the above simple code can be written since we have handlers. To generalize, when a mobile agent dies, the local runtime system ought to provide a way of returning the agent to its home. By some local query, the programmer can inspect the contents of such queues, including a general queue for all classes. By writing some declaration (*trace*) in the language, a mobile object support system can internally maintain these queues.

If we think of mobile agents that can deal with resources that cannot move, the difference from other paradigms might become decisive in language design. On the one hand, a variable in an evaluating expression may cause its value to be read from a data base or requested from a remote process, provided that its current value is uu. Thus, a variable may have a *cache* because in the subsequent uses, some domain value is available locally and the handler is not triggered. On the other hand, to assign a value to a variable may cause its value to be stored on a data base or sent to a remote host.

The following piece of code exemplifies a persistent field p and a remote field r that can live together in the same class:

```
class remoteandpersistentcl {
  public int p, r;

  public void ini(int i, int j) {
    inttodb("p",i);
    p := i;
    inttourl("www.aaa.bbb.ccc/cgi/server/r.txt",j);
    r := j;
  }

  when p := do {
    p := $;
    inttodb("p",p);
  }

  when p, do {
    return intfromdb("p");
  }

  when r := do {
    r := $;
    inttourl("www.aaa.bbb.ccc/cgi/server/r.txt",r);
  }

  when r, do {
    r := intfromurl("www.aaa.bbb.ccc/cgi/server/r.txt");
  }
}
```

```
remoteandpersistentcl c;
c.p := 20; // also store the value 20 locally on data base.
sendlocally(home, c.p); // send ⟨c.p⟩ to the agent home.
c.r := 30; // also update remotely.
sendlocally(home, c.r); // send ⟨c.r⟩ to the agent home
```

Notice that, according to the evaluator definitions, while the p field is retrieved from a data base whenever its value is requested, the r field is programmed to behave as cache over the global environment. We could write a method reassigning uu to r to cause the r integer value to be retrieved from network at the next time that it is requested in some evaluating expression. The functions $inttourl$, $intfromurl$ and $sendlocally$ tell the underlying system to generate internally mobile agents to take part in the protocol. There has been a general criticism concerning mobile agents because they do not maintain connections. We agree that a programming language should hide connections from the agent, but the mobile agent support system should provide remote communication in an appropriate way. This produces positive effects and abstractions in the programming language.

Here we concentrate on features for global computing and present some new aspects of uu.

V. Lazy Evaluation and Timeouts

Lazy evaluation is one of the most interesting characteristics of programming, in particular in applications where time is regarded as important. In this paper, we are not regarding lazy evaluation as being only *call by need* of functional languages. If the language provides functions, lazy evaluation can also be very useful in the same platform, from the same point of view of the present section.

Programming for mobile agents on a global environment tends to be more personal. One of the reasons is that patience and mood vary for different people as well as for the same person at different instants, and one of the purposes of agents is to represent users.

As an example of a situation, a mobile agent ma can communicate with a stationary agent s which in its turn can send a small agent remotely to ma's home in order to return some piece of information to s which in turn can hand it to the mobile agent ma. To deal with faults and delays in communication, a timeout can be set, implicitly or explicitly, for every input operation. After that time, the result is *unknown* (uu) and the computation continues normally. Similarly, every output operation has a timeout.

In this way, the same statement can be executed at different locations[36], either sequentially or not. This situation happens often. Cache-like variables might produce a similar result as lazy evaluation. Computing with timeouts together with uu and handlers is not lazy evaluation, but it can give a somewhat similar impression of *impatience* and, because the resulting value in this case of exception is uu, variable values in programs are always sound and finally this scheme improves *agent robustness*.

Another way of dealing with faults and delays is to provide a standard semantics for basic operations such as arithmetic

and relational. In particular, if the first operand is *uu*, the expression might result in *uu* without the evaluation of the second operand. This is a form of lazy evaluation.

VI. Logic Programming

In comparison to imperative programming languages, logic programming is easy because the former requires a more difficult form of reasoning, for instance, sequences of statements. Here a program is ideally a set of facts and rules, and these are notions with which all of us are used to dealing in our daily lives. In agent-based languages and systems, the programmer typically needs to state permissions of access for users to resources and this can be done in logic programming quickly, perhaps automatically. On the other hand, logic-based languages that are boolean, such as the famous Prolog, are not very compatible with global computing because there are delays and failures in connections in the real world. Perhaps because of this, logic programming has not been interesting for code mobility or global computers. Prolog negation as failure is dependent on the closed-world assumption while, in contrast, global computers contain those mentioned characteristics. This presents an important problem if one wants to use Prolog in applications other than with non-monotonic reasoning, similar to locally answering whether there exists a flight leaving London for New York on Wednesday afternoon, as illustrated in [37].

VII. Strong Mobility

Global computers almost imply code mobility, whose most general form is *strong mobility*, which in turn can be implemented by mobile agents. To date, for all mobile agent languages and systems, an instruction that causes mobility is required, although strong mobility can be easily conceived declaratively, instead of in the form of an imperative statement. In Telescript, for instance, this statement is called **go** while in Agent Tcl the equivalent statement is called **jump**. In the programming language which is being presented for illustrative purpose, the instruction is the **flyto** statement.

The statement for strong mobility makes the agent execution freeze and the thread continues at the destination address, which is its operand. There may be some password and other details, depending on the technology.

VIII. Other Features

Because generality is desirable, choices among various strategies for binding resources should normally be programmed. Handlers may be used to implement different strategies for variables that are resources, either local or remote.

During the compilation, in order to support higher-level communication between agents, names of objects in the source program can be written in the object code, which increases the agent size but it is still a good idea. A possibility is to generate only names defined in the dynamic part of the interface. If the language supports artificial intelligence techniques, perhaps it is even interesting to consider the idea of generating all names. Communication between agents can be set from a prefix in function calls containing the name of the destination agent. For example, in $x := prov{:}func(params)$, the string variable $prov$ is a name that indicates the agent which in turn might contain the $func$ function definition. The $prov$ value is an absolute (global) or relative (to the local host) address. If such a matching name of $func(params)$ is undefined in that agent when the call is executed, x receives *uu*. In every function call (or method invocation) between two agents, a timeout can be attached. For example, in $x := prov{:}func(params)$ **timeout** 3, if the operation is not completed before 3 seconds, at that time it is interrupted and x receives *uu*.

The concepts of **home** and **Id** of agents ought to be key words in the programming language, in a similar way as exemplified above, outside the class $remoteandpersistentcl$.

Surprisingly, although *concurrent programming*[38] is an important technique that can help in certain applications, it is not a specific feature for mobile agent programming languages, as concurrency can be achieved at the operating system level.

However, *uu* permits a large number of parallel operations, not only parallel and and parallel or. For example, to evaluate $op_1\ (+)\ op_2$, the operands are evaluated first, possibly in parallel, and if both result in a known value the sum is finally performed, otherwise the result is *uu*. See semantic rules for $(+)$ in the Appendix. There are no side effects in the operation, as the language can guarantee syntactically the presence of only pure function applications and pure expression evaluations. Although we do not list all parallel operations, which lexically we would similarly surround all those sequential operators with parentheses, very similar semantic rules would apply for the other operators. In other words, in the Appendix, we only present rules for $(+)$ and not for $(-)$, $(*)$, $(/)$ etc. The systematic use of these parallel operators radically improves programming, as well as improving efficiency of agents running globally.

IX. Conclusion

Local inefficiency is an issue of the features discussed in this paper. Assuming that, in practice, mobile agent support systems entail code interpretation, the interpreter has to check the presence of *uu* whenever a variable is being requested in an evaluating expression. However, as hardware is getting faster and larger, this is not considered a significant problem. Moreover, this problem can be compensated for the fact that mobility and remote accesses are the bottleneck in applications, and that variables can behave as cache and operations can be lazy. This combination is encouraged by the language.

Acknowledgement

We would like to thank Christina for the nice and useful discussion on some parts of this paper.

APPENDIX

AN OPERATIONAL SEMANTICS

An operational semantics is defined here to make the ideas presented in this paper more precise. We define the semantics of certain language constructs, expression, assignment and conditional statements.

Let p be a program in the present object language $\mathcal{U}$ and let all of these definitions apply to the scope p. To avoid being exhaustive, we infer the uu type according to the operators, use $=$ and $\neq$ as polymorphic operators, and consider that variables have their separate scopes in each rule, although they have the same names in the set of rules. We apologize for this abuse of notation. Let A^I and A^L be isomorphic to $\mathbb{Z} \cup \{\mathit{uu}\}$ and $\{\mathit{ff}, \mathit{uu}, \mathit{tt}\}$ i.e. the set of logical values, respectively, and use these sets as carriers of the algebra that we are going to define. Because we use Łukasiewicz[6], [7] 3-valued logic in the rules, and it extends the semantics of the boolean connectives using the same symbols, we use only the 3-valued connectives to avoid a mixture of logics. Thus, $\neg \mathit{ff} \rightsquigarrow \mathit{tt}$, $\neg \mathit{tt} \rightsquigarrow \mathit{ff}$, $\neg \mathit{uu} \rightsquigarrow \mathit{uu}$, $\mathit{uu} \wedge \mathit{ff} \rightsquigarrow \mathit{ff}$, $\mathit{uu} \wedge \mathit{tt} \rightsquigarrow \mathit{uu}$, $\mathit{uu} \vee \mathit{tt} \rightsquigarrow \mathit{tt}$, $\mathit{uu} \vee \mathit{ff} \rightsquigarrow \mathit{uu}$, etc. Notice that although $\mathit{uu} \neq \mathit{uu}$ is false and $\mathit{uu} = \mathit{uu}$ is true at the rule level, both result in uu in the object language semantics. Now we can define an algebra $A \stackrel{\text{def}}{=} \langle A^I, A^L, Han, Var, Loc, Val, S, 0, \mathit{ff}, \mathit{uu}, \mathit{tt}, u, v, x, y, ev, re, \$, def, +, -, \times, /, =, \neq, <, \neg, \wedge, \vee \rangle$ for signature Σ in this analysis, where A^I is the set of integers, A^L is the set $\{\mathit{ff}, \mathit{uu}, \mathit{tt}\}$, Han is the set of handlers, Var is the set of all variables, Loc is the set of locations, $Val \stackrel{\text{def}}{=} A^I \cup A^L$, and S is the store or state. Let $u, v \in Val$, $x, y \in Var$. We only consider variables of p and not constants or operands of another nature. Then, $\Sigma \stackrel{\text{def}}{=} \langle \{A^I, A^L, Han, Var, Loc, Val, S\}, F \rangle$, where F is consisted by $+$, $-$, $\times$, $/$, $=$, $\neq$, $<$, $\neg$, $\wedge$, $\vee$ and the following functions:

$$(initialize) \quad \Omega : S$$

$$(locate) \quad \gamma : Var \to Loc$$

$$(lookup) \quad \rho : S \times Loc \to Val$$

$$(update) \quad \Delta : S \times Loc \times Val \to S$$

$$(handler\ is\ defined) \quad def : Han \to A^L$$

$$(evaluator) \quad ev : Var \to Han$$

$$(reactor) \quad re : Var \to Han$$

$$(intended\ value) \quad \$: Var \to Val$$

Intuitively, Ω initializes the whole memory; γ maps a variable to its location; ρ results in the content of a location in some particular state; and Δ updates the memory according to its parameters: location and value. As a syntactic sugar, we write $x.ev$ and $x.re$ to refer respectively to the evaluator $ev(x)$ and reactor $re(x)$ of some variable x, and use the following notation on the syntax: $def(x.ev)$ to mean that the evaluator of x exists, and $def(x.re)$ to mean that the reactor of x exists. Accordingly, we also write $x.\$$ to refer to the result from the evaluated expression that is always available during the evaluation of the reactor $x.re$ when it is defined and applied, that is, $x.\$$ is shorthand for $\$(x)$. Let $s_0, s, s', s'' \in S$, s_0 be the initial state. Then $\Omega \stackrel{\text{def}}{=} \forall x \in Var,\ \rho(s_0, \gamma x) = \mathit{uu}$.

The operational semantics rules are:

$$Introduction: \quad \frac{}{\textbf{begin}} \quad \Omega$$

$$V_1: \quad \frac{\rho(s, \gamma x) = u \quad u \neq \mathit{uu}}{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s)}$$

$$V_2: \quad \frac{\rho(s, \gamma x) = \mathit{uu} \quad def(x.ev) \quad \langle x.ev, s \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s')}{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s')}$$

$$Lazy +: \quad \frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}{\langle x + y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}$$

$$\frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{uu} \quad \langle y, s' \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v = \mathit{uu}}{\langle x + y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s'')}$$

$$\frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{uu} \quad \langle y, s' \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \mathit{uu}}{\langle x + y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u + v, s'')}$$

$$(+)_1: \quad \frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}{\langle x\ (+)\ y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}$$

$$(+)_2: \quad \frac{\langle y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}{\langle x\ (+)\ y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}$$

$$(+)_3: \quad \frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{uu} \quad \langle y, s' \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \mathit{uu}}{\langle x\ (+)\ y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u + v, s'')}$$

$$Lazy -: \quad \frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}{\langle x - y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}$$

$$\frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{uu} \quad \langle y, s' \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v = \mathit{uu}}{\langle x - y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s'')}$$

$$\frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{uu} \quad \langle y, s' \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \mathit{uu}}{\langle x - y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u - v, s'')}$$

$$Lazy *: \quad \frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}{\langle x * y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s')}$$

$$\frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (0, s')}{\langle x * y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (0, s')}$$

$$\frac{\langle x, s \rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{uu} \quad \langle y, s' \rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v = \mathit{uu}}{\langle x * y, s \rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu}, s'')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \mathit{uu}}{\langle x * y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u \times v, s'')}$$

$$Lazy\ /\colon \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}{\langle x/y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \mathit{uu}}{\langle x/y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s'')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \mathit{uu}}{\langle x/y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u/v,s'')}$$

$$Lazy\ =\colon \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}{\langle x=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \mathit{uu}}{\langle x=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s'')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \mathit{uu}}{\langle x=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u=v,s'')}$$

$$Lazy\ !=\colon \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}{\langle x\ !=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \mathit{uu}}{\langle x\ !=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s'')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \mathit{uu}}{\langle x\ !=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u\neq v,s'')}$$

$$Lazy\ <\colon \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}{\langle x<y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \mathit{uu}}{\langle x<y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s'')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \mathit{uu}}{\langle x<y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u<v,s'')}$$

$$\frac{\rho(s,\gamma x) \neq \mathit{uu}}{\langle \textbf{not}\ x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\neg\rho(s,\gamma x),s)}$$

$$\frac{\rho(s,\gamma x) = \mathit{uu} \quad \neg def(x.ev)}{\langle \textbf{not}\ x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s)}$$

$$\frac{\rho(s,\gamma x) = \mathit{uu}}{\langle x\ \textbf{known},s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff},s)}$$

$$\frac{\rho(s,\gamma x) \neq \mathit{uu}}{\langle x\ \textbf{known},s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt,s)}$$

$$\frac{\rho(s,\gamma x) = \mathit{uu}}{\langle x\ \textbf{unknown},s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt,s)}$$

$$\frac{\rho(s,\gamma x) \neq \mathit{uu}}{\langle x\ \textbf{unknown},s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff},s)}$$

$$\frac{\rho(s,\gamma x) = \mathit{uu} \quad def(x.ev) \quad \langle x.ev,s\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s')}{\langle \textbf{not}\ x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\neg v,s')}$$

$$Lazy\ \wedge\colon \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff},s')}{\langle x\ \textbf{and}\ y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff},s')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \mathit{ff} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'')}{\langle x\ \textbf{and}\ y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u\wedge v,s'')}$$

$$Lazy\ \vee\colon \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt,s')}{\langle x\ \textbf{or}\ y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt,s')}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq tt \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'')}{\langle x\ \textbf{or}\ y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u\vee v,s'')}$$

$$\frac{\neg def(x.re) \quad \langle E,s\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s')}{\langle x := E,s\rangle \overset{\text{exec}}{\rightsquigarrow} s''}\Delta(s',\gamma x,v) = s''$$

$$\frac{\begin{array}{c} def(x.re) \quad \langle E,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \\ u = \rho(s',\gamma(x.\$)) \quad \langle x.re,s'\rangle \overset{\text{exec}}{\rightsquigarrow} s'' \end{array}}{\langle x := E,s\rangle \overset{\text{exec}}{\rightsquigarrow} s''}$$

$$\frac{\langle ThVL,s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt,s') \quad \langle C1,s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle \textbf{if}\ ThVL\ \textbf{then}\ C1\ \textbf{ifnot}\ C2\ \textbf{otherwise}\ C3,s\rangle \overset{\text{exec}}{\rightsquigarrow} s''}$$

$$\frac{\langle ThVL,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff},s') \quad \langle C2,s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle \textbf{if}\ ThVL\ \textbf{then}\ C1\ \textbf{ifnot}\ C2\ \textbf{otherwise}\ C3,s\rangle \overset{\text{exec}}{\rightsquigarrow} s''}$$

$$\frac{\langle ThVL,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s') \quad \langle C3,s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle \textbf{if}\ ThVL\ \textbf{then}\ C1\ \textbf{ifnot}\ C2\ \textbf{otherwise}\ C3,s\rangle \overset{\text{exec}}{\rightsquigarrow} s''}$$

$$\frac{\langle ThVL,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff},s')}{\langle \textbf{while}\ ThVL\ \textbf{do}\ C,s\rangle \overset{\text{exec}}{\rightsquigarrow} s'}$$

$$\frac{\langle ThVL,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{uu},s')}{\langle \textbf{while}\ ThVL\ \textbf{do}\ C,s\rangle \overset{\text{exec}}{\rightsquigarrow} s'}$$

$$\frac{\begin{array}{c} \langle ThVL,s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt,s') \quad \langle C,s'\rangle \overset{\text{exec}}{\rightsquigarrow} s'' \\ \langle \textbf{while}\ ThVL\ \textbf{do}\ C,s''\rangle \overset{\text{exec}}{\rightsquigarrow} s''' \end{array}}{\langle \textbf{while}\ ThVL\ \textbf{do}\ C,s\rangle \overset{\text{exec}}{\rightsquigarrow} s'''}$$

$$\frac{\langle C1,s\rangle \overset{\text{exec}}{\rightsquigarrow} s' \quad \langle C2,s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle C1;C2,s\rangle \overset{\text{exec}}{\rightsquigarrow} s''}$$

REFERENCES

[1] M. Mira da Silva, "Mobility and persistence," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 157–176, lecture Notes in Computer Science No. 1222.

[2] M. Mira da Silva and M. Atkinson, "Combining mobile agents with persistent systems: Opportunities and challenges," in *2nd ECOOP Workshop on Mobile Object Systems*, Linz, Austria, July 1996, pp. 36–40.

[3] M. Gelfond and V. Lifschitz, "Logic programs with classical negation," in *Proceedings of 7th International Conference on Logic Programming*. Cambridge MA: The MIT Press, 1990, pp. 579–597.

[4] ——, "Classical negation in logic programs and disjunctive databases," *New Generation Computing. Ohmsha Ltd and Spring-Verlag*, pp. 365–385, 1991.

[5] A. C. Kakas, R. A. Kowalski, and F. Toni, *The Role of Abduction in Logic Programming, in Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1998, vol. 5. Logic Programming, pp. 235–324.

[6] J. Łukasiewicz, *Jan Łukasiewicz Selected Works*, ser. Series on Studies in Logic and Foundations of Mathematics. North-Holland Publishing Company and PWN - Polish Scientifi c Publishers, 1970.

[7] S. C. Kleene, *Introduction of Metamathematics*. D. Van Nostrand, Princeton, 1952.

[8] D. A. Schmidt, *The Structure of Typed Programming Languages*, ser. Foundations of Computing Series. The MIT Press, 1994.

[9] E. Freeman, "Supercomputer earth: Massively parallel internet," Yale University, Tech. Rep., December 1993, supplement to the Yale Weekly Bulletin.

[10] K. Arnold and J. Goslin, *The Java Programming Language*. Addison-Wesley Publishing Company, 1996.

[11] L. Cardelli, *Mobile Object Systems*, ser. Lecture Notes in Computer Science. Linz, Austria: Springer-Verlag, 1997, no. 1222, ch. Mobile Computation.

[12] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Adison-Wesley, 1994.

[13] F. C. Knabe, "Language support for mobile agents," Ph.D. dissertation, Carnegie Mellon University, Paittsburgh, Pa., Dec. 1995, also available as Carngie Mellon School of Computer Science Technical Report CMU-CS-95-223 and European Computer Industry Centre Technical Report ECRC–95–36. [Online]. Available: ftp://reports.adm.cs.cmu.edu/usr0/anon/1995/CMU-CS-95-223.ps.Z

[14] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna, "Analyzing mobile code languages," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 93–110, lecture Notes in Computer Science No. 1222. [Online]. Available: http://www.polito.it/~picco/papers/ecoop96.ps.gz

[15] J. Vitek, M. Serrano, and D. Thanos, "Security and communication in mobile object systems," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 177–200, lecture Notes in Computer Science No. 1222.

[16] D. Volpano, "Provably-secure programming languages for remote evaluation," *ACM Computing Surveys*, vol. 28A, Dec. 1996, participation statement for ACM Workshop on Strategic Directions in Computing Research. [Online]. Available: http://www.cs.nps.navy.mil/research/languages/papers/atsc/sdcr.ps

[17] A. Acharya, M. Ranganathan, and J. Saltz, "Dynamic linking for mobile programs," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 245–262, lecture Notes in Computer Science No. 1222. [Online]. Available: http://www.cs.umd.edu/~acha/papers/lncs97-2.html

[18] ——, "Sumatra: A language for resource-aware mobile programs," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 111–130, lecture Notes in Computer Science No. 1222. [Online]. Available: http://www.cs.umd.edu/~acha/papers/lncs97-1.html

[19] D. Dean, E. Felten, and D. Wallach, "Java security: From HotJava to Netscape and beyond," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, Cal., May 1996. [Online]. Available: http://www.cs.princeton.edu/sip/pub/secure96.html

[20] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, Massachussetts: Addison-Wesley Publishing Company, 1997.

[21] D. Dean, E. W. Felten, and D. S. Wallach, "Java security: From hotjava to netscape and beyond," in *Proceedings of the Symposium on Security and Privacity*. IEEE, 1996, pp. 190–200.

[22] D. Dean, "The security of static typing with dynamic linking," in *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[23] J. White, *Telescript Technology: the Foundation for the Electronic Marketplace*, General Magic, Inc., 1994.

[24] G. M. Corp., *Odyssey White Paper*, 1998.

[25] B. Mathiske, F. Matthes, and J. W. Schmidt, "On migrating threads," Fachbereich Informatik Universitat Hamburg, Tech. Rep., 1994.

[26] R. S. Gray, "Agent tcl: A transportable agent system," in *Proceedings of the CIKM'95 Workshop on Intelligent Information Agent*, 1995.

[27] ——, "Agent tcl: A flexible and secure mobile-agent system," Dartmouth College, Computer Science, Hanover, NH, Tech. Rep. PCS-TR98-327, Jan. 1998, ph.D. Thesis, June 1997. [Online]. Available: ftp://ftp.cs.dartmouth.edu/TR/TR98-327.ps.Z

[28] N. S. Borenstein, "Email with a mind of its own: The safe-tcl language for enabled mail," First Virtual Holdings, Inc, Tech. Rep., 1994.

[29] K. Bharat and L. Cardelli, "Migratory applications," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 131–149, lecture Notes in Computer Science No. 1222.

[30] L. Cardelli, "A language with distributed scope," *Computing Systems. The MIT Press*, vol. 8, no. 1, pp. 27–59, 1995.

[31] B. Thomsen, L. Leth, S. Prasad, T.-M. Kuo, A. Kramer, F. C. Knabe, , and A. Giacalone, "Facile antigua release programming guide," European Computer Industry Research Centre, Munich, Germany, Tech. Rep. ECRC–93–20, Dec. 1993.

[32] D. Johansen, "Mobile agent applicability," in *Mobile Agents: Second International Workshop, MA'98*, ser. Lecture Notes in Computer Science, vol. 1477. Springer, 1998, pp. 80–98.

[33] D. Johansen, R. van Renesse, and F. B. Schneider, "An introduction to the TACOMA distributed system," Department of Computer Science, University of Tromsø, Tromsø, Norway, Tech. Rep. 95-23, June 1995.

[34] C. Tschudin, "The messenger environment M0 – a condensed description," in *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, Apr. 1997, pp. 149–156, lecture Notes in Computer Science No. 1222.

[35] G. Karjoth, D. B. Lange, and M. Oshima, "A security model for agents," *IEEE Internet Computing*, vol. 1, no. 4, July/August 1997.

[36] L. Cardelli, "Global computation," *ACM Computing Surveys*, vol. 28A, no. 4, 1996.

[37] D. Gillies, *Artificial Intelligence and Scientific Method*. Oxford University Press, 1996, ch. 4 A new framework for logic, pp. 72–97.

[38] A. Burns and A. Wellings, *Concurrency in Ada*, 2nd ed. Cambridge University Press, 1998.