Language Support for Mobile Agents

Frederick Colville Knabe

December 1995

CMU-CS-95-223

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee
Jeannette Wing, Chair
Peter Lee
Doug Tygar
Alessandro Giacalone, ECRC

Also published as Technical Report ECRC-95-36, European Computer-Industry Research Centre, Arabellastrasse 17, 81925 Munich, Germany.

This research was supported by the European Computer-Industry Research Centre, Munich, Germany, through a Ph.D. student contract.

Keywords: mobile agents, functional languages, distributed programming, remote resource access, distributed typing, software mobility, lazy compilation, heterogeneous code, Facile, ML, Internet applications.

Abstract

Mobile agents are code-containing objects that may be transmitted between communicating participants in a distributed system. As opposed to systems that only allow the exchange of nonexecutable data, systems incorporating mobile agents can achieve significant gains in performance and functionality.

A programming language for mobile agents must be able to express their construction, transmission, receipt, and subsequent execution. Its implementation must handle architectural heterogeneity between communicating machines and provide sufficient performance for applications based on agents. In addition to these *essential properties,* an agent language may support *desirable properties* such as high-level abstractions for code manipulation and the ability to access resources on remote execution sites.

We designed and implemented an agent programming language that satisfies the essential properties and a number of desirable ones. A key feature of our language is the use of strong static typing for remote resource access. Agents may be linked dynamically to resources on remote sites, and this linking is always guaranteed to be type safe. We provide this guarantee without requiring that all components of an agent-based system be compiled together.

Our language also includes several features to improve the performance of mobile agents. Before an agent is transmitted, it is trimmed of values that are expected to be available on the recipient, thus shrinking transmissions. Agents may be interpreted or compiled depending on the application and the relative performance trade-offs. When compilation is used, it is done *lazily:* Each component of an agent is only compiled as it is needed. Furthermore, machine-specific representations for an agent can be transmitted with machine-independent ones, opening the possibility for recipients to skip compilation or interpretation altogether.

To evaluate our language and to explore the potential of mobile agents, we developed a programming framework for agents. Several applications were implemented by other programmers within this framework using our language. Their work served to validate our design and our choice of agent language properties. We also analyzed the performance of our language on these applications and several synthetic benchmarks. The analysis shows that the features we incorporated into the implementation significantly improve performance.

To my parents, Richard and Linda Colville, and the team of Kevin Garvey, Brett Bates, Karen Thorson, and Jon Ruth, who together steered me back on course before I went too far astray.

Contents

List of Tables viii

List of Figures ix

Acknowledgments x

1 Mobile Agents 1
- 1.1 Applications of Mobile Agents . . . 1
  - 1.1.1 Heterogeneous Communication . . . 3
  - 1.1.2 Reduced Communication . . . 3
  - 1.1.3 Specialized Protocols . . . 4
  - 1.1.4 Reduced Server Loads . . . 5
  - 1.1.5 Enriched Interfaces . . . 6
  - 1.1.6 Temporary Applications . . . 6
  - 1.1.7 Intelligent Data . . . 7
- 1.2 Programming Languages for Mobile Agents . . . 7
- 1.3 Overview of the Dissertation . . . 8

2 Towards an Agent Language 10
- 2.1 Properties of Agent Languages . . . 10
  - 2.1.1 Code Manipulation . . . 11
  - 2.1.2 Heterogeneity . . . 12
  - 2.1.3 Performance . . . 12
  - 2.1.4 Remote Resource Access . . . 13
  - 2.1.5 Strong Typing . . . 13
  - 2.1.6 Automatic Memory Management . . . 14
  - 2.1.7 Stand-Alone Execution . . . 14

2.1.8 Independent Compilation . . . 15
2.1.9 Security . . . 15
2.2 The Development Platform . . . 15
2.2.1 The Facile Language and Compiler . . . 16
2.2.2 Agent Language Properties Supported by Facile . . . 17
2.2.3 Properties Not Supported by Facile . . . 20
2.2.4 Constraints from the Facile Project . . . 21
2.3 Summary . . . 21

3 The Language and Compiler Design 22
3.1 Transmissible Code Representations . . . 23
3.1.1 Marshalling Strategies . . . 24
3.1.2 Reducing Space Costs with Annotations . . . 26
3.1.3 Other Space Reduction Methods . . . 31
3.2 Typed Remote Resource Access . . . 32
3.2.1 Proxy Structures . . . 32
3.2.2 Types . . . 35
3.2.3 Linking . . . 37
3.2.4 Access Context . . . 40
3.3 Ubiquitous Values . . . 40
3.3.1 Ubiquitous Structures . . . 41
3.3.2 Closure Trimming . . . 42
3.4 Lazy Run-Time Compilation . . . 43
3.5 Related Work . . . 44
3.5.1 Transmissible Representations . . . 44
3.5.2 Typed Remote Resource Access . . . 46
3.5.3 Lazy Compilation . . . 48
3.6 Summary . . . 49

4 The Agent Language Implementation 51
4.1 Naming via Signatures . . . 52
4.2 Transmissible Code Representations . . . 53
4.2.1 Stand-Alone Functions . . . 54
4.2.2 Representations . . . 57
4.2.3 Storage . . . 61
4.3 Closures . . . 63
4.4 Run-Time Compilation . . . 65

4.4.1 Laziness . . . 66
4.4.2 Compilation and Linking . . . 71
4.5 Summary . . . 72

5 Agent Applications **74**
5.1 An Overview of Mobile Service Agents . . . 75
5.2 Design of the MSA Framework . . . 75
5.3 MSA Applications . . . 78
5.3.1 No Communication . . . 79
5.3.2 Required Communication . . . 81
5.3.3 Preferred Communication . . . 82
5.3.4 A Sample Implementation . . . 89
5.4 Evaluation . . . 92

6 Performance **95**
6.1 Facility Configuration . . . 96
6.1.1 Performance of Benchmarks . . . 97
6.1.2 Performance of MSA Applications . . . 108
6.2 Closure Trimming . . . 113
6.3 Function Sizes . . . 115
6.4 Evaluation . . . 115

7 Conclusions and Future Work **118**
7.1 Contributions . . . 118
7.2 Future Work . . . 120
7.3 Security . . . 121
7.4 Lessons Learned . . . 123

Bibliography **126**

List of Tables

6.1 Performance data for the sorting function benchmarks 103
6.2 Sizes of the transmissible representations for the large and small sorting functions 104
6.3 Combined marshalling and transmission times 107
6.4 Calculated performance for the sorting function benchmarks using multiple representations 108
6.5 Performance data for the MSA applications 110
6.6 Sizes of the transmissible representations for the MSA service agents 111
6.7 Calculated performance for the MSA service agents using multiple representations 111
6.8 Times between a series of user interactions with the MSA agents . 112
6.9 Effect of closure trimming on the sizes of marshalled agents . . . 114
6.10 Effect of closure trimming on transmission times 114
6.11 Effect of closure trimming on compilation times 114

List of Figures

3.1 Signatures and structures in Facile 33
3.2 A signature for polymorphic stacks 35
3.3 A functor parameterized by one argument 38

4.1 The factorial function in lambda language 58
4.2 The factorial function in final CPS 59
4.3 The factorial function in middle CPS 61
4.4 Layout of a compiled code string 62
4.5 A code string prepended with transmissible representations 63

5.1 Bootstrapping the connection to a directory service 77
5.2 The user interface of a directory agent 78
5.3 An interactive map agent 80
5.4 Establishing the file system browser service 82
5.5 A conference schedule agent 84
5.6 Establishing the conference schedule service 85
5.7 The agent tracking system in use 87
5.8 Entry point to the file system browser agent 90
5.9 The callback function for the file system browser agent 91

Acknowledgments

After seven years of graduate work, I am grateful to many people for the help they have given me in bringing it to a successful conclusion.

First, I thank Alessandro Giacalone and Jeannette Wing, my two advisors. Alessandro's confidence in me and his encouragement of my ideas were instrumental in carrying me through the definition and implementation of the work. He also taught me about the research "business" and gave me the chance to participate in many decisions surrounding the Facile project. Most importantly, I learned that even Italians in Germany can roast a Thanksgiving turkey you will never forget.

It was a distinct privilege to be able to work with Jeannette in the writing of my dissertation. Jeannette taught me how to write a research paper, and any coherence or clarity of arguments in these pages is due to her detailed comments and rigorous standards. Jeannette was willing to work with me long distance, and I was continually astounded by how quickly she read chapters and replied to emails.

I thank Doug Tygar and Peter Lee, my other committee members, for spending the time and energy to review my dissertation and to offer many helpful comments. They challenged my arguments and sharpened my thinking.

I relied greatly on members of the Facile group as the "technical support team" for my thesis. Bent Thomsen and Lone Leth were always willing to lend an ear when I burst in with my latest question. Tsung-Min Kuo patiently explained distributed typing and other concepts to me. André Kramer helped me track down bugs and work out implementation approaches, and always kept an eye out for references related to my area. Pierre Cregut guided me through some of the dark corners of the SML/NJ compiler. I would have been lost during the performance analysis without the advice of Ioanis Nikolaidis. Pierre-Yves Chevalier ("Pitch") actually sat down and used my system, a true act of courage. I thank them all.

Expatriate life in Munich was highlighted by the good company of Konrad Slind and Alexandra Karl, Ann Elbling, and Deirdre Tincker. From nights at

the Oktoberfest to impassioned discussions of American politics, they kept me in touch with the world outside computer science. And home would never have been so *gemütlich* without the Westpreußenstraße club.

Most of all, I thank my wife, Kim, for standing by me through all the stresses and rough spots. Her support gave me the strength to keep going and made the whole journey worthwhile.

Chapter 1

Mobile Agents

Mobile agents are code-containing objects that may be transmitted between communicating participants in a distributed system. As opposed to systems that only allow the exchange of nonexecutable data, systems incorporating mobile agents can achieve significant gains in performance and functionality.

A programming language for mobile agents must be able to express their construction, transmission, receipt, and subsequent execution. Its implementation must address such practical problems as handling architectural heterogeneity between communicating machines and providing sufficient performance for applications based on agents. In this dissertation we present the design and implementation of an agent programming language. We develop several new language features and implementation techniques, and we analyze their utility and performance in real applications.

In Section 1.1 we start by examining the capabilities and uses of mobile agents. We then argue the need for languages that specifically support agent programming in Section 1.2. We give an overview of the dissertation in Section 1.3.

1.1 Applications of Mobile Agents

The recent and dramatic growth of the Internet is a clear sign that computing has entered a new era. Networking has changed computers from isolated data processors into powerful communication devices. Users now have access to a vast, expanding array of information and services. The computer in its new role is steadily permeating and transforming society, much as the telephone did 100 years

ago.

The existence of widespread distributed application infrastructures, in particular the World Wide Web, has enabled the quick spread of new services. These infrastructures permit computers to communicate and interact via standardized interfaces. Each interface determines such characteristics as how resources at network nodes can be accessed, what information can be exchanged, how control flow in an application can be distributed over the network, and what connectivity must be maintained between interacting partners.

As interfaces incorporate richer data formats and communication protocols, more possibilities for innovative applications emerge. In the World Wide Web, the drive for increased functionality has led many service providers to make their own piecemeal extensions to the interface. However, this approach breaks down the common standard and leads to a lack of interoperability.

What users of the Internet need is an infrastructure that allows clients and servers to extend the interface to support specialized tasks. For example, a service provider may want to use a custom protocol to broadcast information to subscribers. A client may need to run a special terminal emulator for interacting with a central legacy database. A scientist may want to distribute a computation-intensive task to colleagues who volunteer CPU cycles. In each case, a common interface is needed initially to distribute the components of the application, but that interface must then be extended to suit a particular purpose.

The means to provide such extensibility is for the interface to support *the transmission and execution of mobile agents.* By sending mobile agents through the network to execute at remote nodes, servers and clients can install special-purpose interfaces with whatever properties they require. The agents can embody anything from communication protocols to data filters to complete applications.

Mobile agents offer solutions to a range of problems frequently encountered in distributed applications. At the same time, they make possible new kinds of applications with novel functionalities. The uses of agents include

- Linking together existing heterogeneous systems
- Reducing communication costs
- Encoding specialized communication protocols
- Off-loading work from servers
- Enriching client–server interfaces
- Simplifying temporary installation of applications
- Permitting self-processing data

We discuss each of these uses in the following sections.

1.1.1 Heterogeneous Communication

Although networking offers many potential benefits, it can raise just as many challenges when existing centralized systems must be retrofitted for communication and distribution. Tying together two systems quickly reveals incompatibilities in command languages and data formats, and as more systems are added the incompatibilities multiply.

Mobile agents can solve these kinds of problems by serving as a lingua franca that many systems can share. Because they include code, agents can be used to map between complex interfaces on heterogeneous systems. The one requirement is for the agent language primitives to be executable by each type of system.

This idea is not new, and it dates back to the very founding of the Internet. In 1968, the original Arpanet designers were already considering the problem of emulating the terminals for one timesharing system on the terminals of another. Their solution was the Decode–Encode Language, or DEL [Rulifson 1969; Reynolds and Postel 1987]. On connecting to a remote system, the appropriate terminal emulator would be sent back to the client as DEL code. The client would execute the emulator using its local implementation of DEL. Unfortunately this innovative work was set aside before it could be fully developed.

Falcone [1987] also faced the problem of providing common views of several centralized services across a heterogeneous distributed environment. These services and their clients were located on a variety of machines running different operating systems. Service interfaces generally consisted of a set of system calls, but each operating system, and thus the clients running on it, had its own view of what those calls should be. The different views were resolved by having each server support a generic interface consisting of many simple routines. A system call from a client on any of the operating systems could then be broken down into an agent program. The agent, written in a variant of Lisp, would be transmitted to the appropriate server for execution, where it would use the simpler calls to recreate the behavior of the original one.

1.1.2 Reduced Communication

A key advantage of agents is that they can decrease communication costs. In a client–server system structured around agents, part of the client or the server can

move to the other side of the communications link. Once the client or server has moved, interactions between the two can bypass the network.

Falcone's system also demonstrated this use of agents. In principle, system calls made by clients could have been translated into multiple RPCs made to generic interface routines on the appropriate servers. However, the resulting overhead from communication latencies would have been unacceptable. By sending agents to make the calls, these latencies were avoided.

The NeWS window system [Gosling et al. 1989] provides another example. In NeWS, clients communicate with the display server by sending PostScript programs. Instead of drawing a grid by sending several thousand messages for individual points, it is possible to send one brief program that will compute and draw the entire grid. The code sent by the client can also be used to extend the server, so that complex actions can be carried out in the future using a single message.

By reducing communication needs, agents can make interactivity possible even in cases where the network connection is poor. When Ever Systems developed a home banking system for Unibanco in Brazil, they reduced the need for data transmissions over slow, noisy phone lines by downloading application agents onto customers' personal computers [McCartney and Friedland 1995]. In the Wit project [Watson 1994, 1995], portable palmtop computers can access applications on Unix servers via an infrared link. To overcome the weaknesses of the link (low bandwidth and frequent interruptions during movement), the user interface components of an application are moved to the palmtop when the application is first accessed.

Communication can also be reduced by moving the client to the server. In a wireless networked environment, even if agents are used for service interfaces, communication costs may still be too high for a user to navigate the network and to find the resources and services of interest to her. Instead of only receiving agents from services, it might be possible for a user to upload her own agent to the wired network where it can search for information and services and carry out other tasks on her behalf. The only communications that the user's computer must make are the initial upload followed by the eventual receipt of the agent's results.

1.1.3 Specialized Protocols

Mobile agents allow servers to use custom communication protocols with clients. To receive an agent initially, the client and server must share some standard

protocol. Once the agent is running, though, it can use a specialized protocol for communication back to its home server. Furthermore, an executing agent can communicate repeatedly with the server without intervention from the user, allowing the construction of dynamic services. As an example of both these concepts, a weather server or stock ticker might transmit regular updates to agents on scattered clients by using a special multicast protocol.

1.1.4 Reduced Server Loads

One of the challenges in providing services on the Internet today is that the computational components of a service must reside on the server machine. Because commonly used access protocols such as the World Wide Web, Gopher, and Telnet only provide for the exchange of nonexecutable data, servers must take complete responsibility for performing any service-related computation for clients.

For example, a number of services now provide historical financial data. Providing this data in a simple textual format is straightforward and requires little computation by the server. However, many users prefer to see the data formatted as a graph. The service provider can add value by plotting the data and then transmitting an image of the resulting graph for display by the client. Note that the service provider cannot assume that the client has any plotting software itself.

The addition of graph plotting has a number of consequences. First, the computational demands on the server are increased. Popular servers are overloaded, so more and more resources must be dedicated by the service provider to ensure adequate performance. Second, plotting the data on the server means the server must send an image of a graph rather than a numeric data set. The image will almost certainly be many times larger than the original data set, so more packets have to be transmitted to the user. Larger data transmissions drive up bandwidth requirements on the network and mean the user must wait longer for the service to deliver its results. Finally, the user only receives an image, and therefore cannot locally apply useful plotting operations (such as adding a grid). Any modifications to the graph involve requests to the server, adding communication latency to the other problems. On a global network, such latency may be noticeable to a large number of potential users.

Structuring a service with agents can solve or reduce all of these problems. The most important characteristic of agents is that they allow computation to move. A server can off-load work onto a client by sending an agent to the client. The client presumably is willing to dedicate resources such as CPU time to the service

interaction, and these resources can be used directly by the agent. In our example, the transmitted agent could do the plotting of received numeric data itself, while also providing a user interface to the plotting parameters. The user gains better performance while the load on the server and network is lessened.

The same effect can be achieved without mobile agents. If the client already has specialized software installed for interactions with a particular service, there is no need for agents to be transmitted over the network. However, there are thousands of publicly available services on the Internet today. Their success largely depends on *ease of access* from a wide range of potential clients. It will often be too much to require a first-time client to install manually special software for accessing a service. Mobile agents provide a sufficiently general and transparent means of moving computation from servers to clients.

1.1.5 Enriched Interfaces

Agents permit servers to present more sophisticated interfaces to the user. For example, a virtual reality interface would normally require far too much computation and network communication for a server to provide it remotely. Even if the CPU time and bandwidth were available, communication latency would make interactivity difficult. But by sending an agent containing the interface to the client, the server can sidestep these problems.

1.1.6 Temporary Applications

An agent need not be part of a larger distributed application; it may be self-contained and have no communication needs at all. For example, an agent could consist of the data and manipulation code for a simple model of the globe. Downloading such an agent from a server is no different from downloading an application from an FTP site. However, the agent has the advantage that it can leverage off the client's existing agent infrastructure. The infrastructure (i.e., the program on the client that receives and executes agents) allows the user to download and run the agent without the steps of creating source directories, configuring, compiling, and installing binaries. This simplicity makes it much easier to download applications temporarily; the user can retrieve an application with little effort and discard it later.

Agents that can function in a stand-alone mode are suitable for portable computers that have no wireless connections. Before disconnecting from the network

at her home location, a user might download an agent with train schedules and a route planner. Such an agent may have no need for communication and can be thrown away after the trip. If the user periodically reconnects to the network, the agent may be able to take advantage of these moments to retrieve updates silently.

As a user moves into a new environment such as a city, company, or conference site, she may want to access special services that are particular to the environment. In addition to services that are only usable while some connectivity can be maintained, an environment may provide stand-alone agents with domain-specific information. A simple example is an agent that contains a local guide book along with software to search and scan it in various ways.

The HotJava browser [Gosling and McGilton 1995] for the World Wide Web encourages this style of agent programming. With HotJava, Web pages may contain *applets* written in the Java language. These applets are self-contained programs that start execution once they have been downloaded. HotJava applets range from games and simulations to interactive proofs, and many more are being developed.

1.1.7 Intelligent Data

Associating agents with data provides a way for the data to "know" how to process itself. An example is provided by the MPEG 4 compression standard for video [ISO 1995], where the decompression algorithm is bundled with the data. This approach makes the standard highly flexible and adaptable to different needs.

In the SOFTNET packet radio system [Zander and Forchheimer 1984; Olofsson 1985], each network packet is an agent written in FORTH. The code for how to process a packet is included in the packet itself; routers simply execute the packets they receive. Once again, flexibility is one of the primary advantages of the system.

1.2 Programming Languages for Mobile Agents

Mobile agents offer compelling advantages for constructing flexible and adaptable distributed systems. However, the transmission of code places new demands on programming languages and run-time systems. At a minimum, the following are required:

Code manipulation. The language must provide a means for manipulating and transmitting the code-containing objects[1] that represent mobile agents. A means must also exist for receiving the objects, converting them into an executable form, and executing them.

Heterogeneity. Heterogeneous machine architectures are a common characteristic of distributed systems. Mobile agents constructed on one architecture must be executable on other architectures.

Performance. The use of mobile agents in an application can eliminate communication latencies or enable enriched functionality. These benefits can only be realized if the implementation of mobile agent transmission and execution delivers sufficient performance.

Conventional languages and compilers do not satisfy these requirements. All the code in a program is assumed to stay in a single address space. To maximize performance, the compiler strives to make the code as machine-specific as possible. Ad hoc approaches to supporting mobile agents, such as transmitting source files and compiling them on receipt, suffer from poor performance and lack of integration with the language.

We therefore see a need for the development of programming languages that specifically address mobile agents. In the design of such languages we want features that make mobile agents easy to program correctly. We also want to implement the low-level mechanisms necessary for heterogeneity and performance.

1.3 Overview of the Dissertation

In this dissertation we present the design and implementation of an agent language that meets the preceding requirements as well as several others in the areas of strong typing, stand-alone execution, remote resource access, and independent compilation. Novel aspects of our work include

- Multiple agent representations and execution methods
- Typed access to remote resources through extended lexical scoping
- Automatic unlinking of local resources on transmission
- Lazy compilation of received agents

[1] We use the term *object* informally here; we are not referring to object-oriented programming.

- A programming framework for mobile agents

We also assess the performance of our implementation and applications.

In Chapter 2 we examine the requirements for agent programming languages and present a higher-order language that provides a good starting point for an agent language. In Chapter 3 we develop a design that addresses the necessary language and compiler features and then relate this design to other work. We elaborate the design with descriptions of the significant implementation methods and decisions in Chapter 4. We then turn in Chapter 5 to practical use of the agent programming language with descriptions of the agent applications that we have built. In Chapter 6 we give performance results for the agent language and the agents built with it, allowing us to evaluate the merits of the implementation features. We discuss our conclusions and areas for future research in Chapter 7.

Chapter 2

Towards an Agent Programming Language

In the previous chapter we saw the value of mobile agents in distributed systems and identified the need for languages that meet the requirements for agent programming. Our task in this chapter is to expand those requirements and to consider how we may satisfy them.

We begin in Section 2.1 by examining our minimal requirements for an agent language. We also introduce several other properties that such a language should (but need not) satisfy; support for these properties aids the development of distributed systems based on mobile agents.

With these properties established, in Section 2.2 we choose an existing language, Facile, as the basis for implementing an agent language. We present the features of Facile and argue why they provide a good starting point for satisfying the properties. We also note constraints from the Facile project on this work.

We summarize the chapter in Section 2.3.

2.1 Properties of Agent Languages

Agent languages are distinguished from other programming languages by a set of *essential* properties. These properties form a bare minimum for building mobile agent applications in real distributed environments. They are as follows:

- Support for manipulating, transmitting, receiving, and executing code-containing objects

- Support for heterogeneous computer systems
- Performance sufficient to meet the needs of applications

We can also add several *desirable* language properties that improve agent programming:

- Remote resource access
- Strong typing
- Automatic memory management
- Stand-alone execution
- Independent compilation
- Security

We explain both the essential and desirable properties in the following sections.

2.1.1 Code Manipulation

The construction and use of mobile agents requires that the language support several kinds of code manipulation:

Identification. The language must provide a means to identify the code of an agent, allowing it to be distinguished from other agents and program code. For example, if an agent consists of a number of functions, one designated function (its main entry point) may serve to identify it.

Transmission. There must be suitable language primitives or library functions with which the programmer can express that an agent should be transmitted.

Receipt and execution. The language run-time environment must be able to execute received agents *within the existing address space of the recipient.* This condition is an important assumption underlying many of the possible applications for agents and the potential performance gains.[1]

Languages may support these requirements in different ways. We add as a desirable property that the language provide constructs and abstractions that allow code to be manipulated completely within the language. For example, a language

[1] Note that without this distinction, Unix `rsh` would be sufficient to meet the code manipulation requirements.

might provide a construct that allows blocks of code to be identified as agents. Special operators might then be used to transmit, receive, and execute these blocks.

A language without this property might require that agents be developed as separate programs that are placed in files managed by the operating system; the language might then provide a means to transmit such a file, to load its contents dynamically into its address space, and to start executing the new code. This approach is more awkward to program than when all parts of code manipulation are integrated into the language.

2.1.2 Heterogeneity

Distributed systems are characterized by heterogeneity in machine architectures and operating systems. In open networks such as the Internet, users may access services from hundreds of different machine environments. For agent technology to be widely useful, it must be possible for agents built on one machine to execute on other machines. Though homogeneous implementations of agents are certainly easier to implement, they are not as general and are unrealistic for real distributed applications.

The heterogeneity problem is not restricted to agent programming; in conventional distributed programming, nonexecutable data must be transmissible between different machine architectures.

2.1.3 Performance

An agent language implementation should minimize the performance overhead of using agents. Agents incur space and time costs through their size, transmission time, time to convert to an executable form, speed of execution, etc. These costs must be kept low enough to meet the needs of applications.

Because this requirement is by nature application dependent, it can be difficult to determine if an implementation meets it. Comparing applications based on agents with those based on standard techniques may be misleading. As a simple example, the time to transmit a million consecutive integers is almost surely longer than the time to transmit and execute an agent that generates those integers, but this difference tells us very little about whether agents have been efficiently implemented.

In the absence of a standard suite of agent benchmarks, implementors should report the performance of their languages on real applications.

2.1.4 Remote Resource Access

Agents should be able to interact with locally available services and resources in their eventual execution environments. An agent may need to access library routines, system data, and other values to accomplish its tasks.

The agent language should provide a means for specifying and controlling these accesses. The access mechanism should be specific enough that the programmer can indicate particular resources for an agent to use.

2.1.5 Strong Typing

The agent language should use strong static typing. The benefits of strong typing for normal programming are well known. They become even more important when programming with agents.

Debugging distributed programs is extremely difficult. Conventional debugging tools only provide control over a single address space. In contrast, distributed programs comprise multiple address spaces as well as a communication network. Conventional tools are of limited help in tracking down a type error whose source could lie in any of these components or in their interactions.

If our distributed programming is restricted to building a new client for a pre-existing and presumably debugged service, we sidestep some of these difficulties. If a client fails due to type errors, we can use conventional tools to monitor its execution and track them down. The service with which the client interacts can be treated as a black box.

When the client can send agents to the server, though, we reintroduce the problems. Now the client program can fail in another address space. The client programmer may not be able to access the address space of the server, ruling out the use of even conventional debugging tools. The client's failure might even bring down the server. (If the agent language provides a strongly typed *run-time* environment, the server may be able to detect and recover from client type errors. However, the client programmer is still left with the problem of finding the source of the error.)

Strong static typing can help prevent programming errors, and an agent language should support it. It is particularly valuable for checking agent code that accesses remote resources. An agent should not be able to compile unless its execution is guaranteed to be type safe.

2.1.6 Automatic Memory Management

For many of the same reasons that strong typing is desirable, automatic memory management is as well. Dangling pointers are eliminated as a potential problem, and memory leaks are reduced. The latter are particularly insidious, because agents may run without errors on a server but leave unreclaimed memory that gradually accumulates. Detecting the source of such leaks, difficult enough in conventional programming, is confounded by the use of agents. Automatic memory management makes these problems less common.

2.1.7 Stand-Alone Execution

Agents should not require a connection to their originating site to execute unless the requirement arises from the application. In Section 1.1.6 we pointed out the usefulness of agents in cases where connections back to a service may not be available, such as mobile computing. The agent programming environment should make agents stand-alone unless the programmer explicitly programs them otherwise.

Supporting stand-alone execution rules out the use of *implicit callbacks* in the language. A callback is a communication back to the originating site to retrieve a value or to call a function. Implicit callbacks are those that are not directly obvious from the program code; they are inserted by the compiler or initiated by the run-time system.

Implicit callbacks may be used to hide the distributed nature of applications, the argument being that hiding distribution simplifies programming. The problem is that dependencies and the costs of access are hidden as well, leaving programs fragile and prone to serious flaws.[2]

The counterargument is that implicit callbacks may improve performance. Rather than transmitting all the components making up an agent, the system only transmits some subset. The remaining components are only retrieved when they are needed by the executing agent. However, each retrieval incurs the latency associated with the original agent transmission; these costs may outweigh any benefits from an initially smaller transmission. Moreover, if in the meantime the

[2] Waldo et al. [1994] argue that trying to hide the distinction between local and remote resources and the communication in a distributed system is doomed to failure. They discredit the "unified view" that hides the distinction because of its inability to handle the problems of latency, memory access, partial failures, and concurrency.

connection to the originating site has been lost, then the callbacks fail and the agent does as well. Callbacks should only be programmed explicitly, leaving the programmer to evaluate their appropriateness.

2.1.8 Independent Compilation

In providing any other properties (particularly static typing), the agent language should not require that all components of a distributed application using agents be compiled together. In open distributed systems, servers and clients are frequently developed and maintained separately. It is unrealistic to require that entire systems pass through a single compilation at a single site.

2.1.9 Security

Agent languages make it easy for systems to receive and execute code off the network. Though this feature offers great flexibility to agent-based applications, it also makes them an ideal route for attacks on a system. A user of an application may have no idea that it is based on agents and that it may introduce foreign code into her system. Using agents to create worms and viruses is trivial.

An agent language for open distributed systems should either provide its own run-time security model or be coupled with some external model that provides system security. As an example of the former approach, the run-time system might only execute agents written in a restricted language that does not include system calls and does not permit the examination or modification of arbitrary memory locations. In the latter approach, agent-based applications might be required to execute behind internal firewalls provided by the operating system.

2.2 The Development Platform

We support all of the preceding properties except security in our agent language. Deferring the problems of security allowed us to concentrate on satisfying the other properties.

Rather than designing and implementing an agent language from scratch, we chose to start with an existing language (Facile) that partially fulfills the essential and desired properties. We could then reuse large parts of a design and implementation and focus on the features specific to agent programming.

In this section we first briefly discuss the original Facile language. We then compare Facile against the agent language properties and see which ones it supports and which ones remain for us to address. Finally, we discuss some constraints from the Facile project on modifying the language.

2.2.1 The Facile Language and Compiler

Facile is a higher-order, mostly functional programming language that integrates support for concurrency and distribution. The original formal foundations for the language, based on the Calculus of Communicating Systems [Milner 1980], were developed at the State University of New York at Stony Brook by Giacalone et al. [1989]. Since 1991 a group at the European Computer-Industry Research Centre has refined and implemented the language. In 1994 ECRC made the Facile Antigua Release [Thomsen et al. 1993] freely available.

The functional language core of Facile is based on the Standard ML language [Harper et al. 1986; Milner et al. 1989]. Facile itself is implemented as an extension of the Standard ML of New Jersey compiler [Appel and MacQueen 1991]. Because it is implemented as an extension of an existing language, Facile's features fall into two rough categories. The first consists of those parts of Facile largely inherited from ML and oriented towards general-purpose programming. The second category builds on the first to support concurrent and distributed programming.

In the first category, Facile provides high-level constructs for defining new complex data types. In the process of defining a data type the programmer simultaneously defines the constructors that can be used to build instances of that data type. In the reverse direction, traversing and breaking down instances of the type can be programmed using a general pattern-matching facility. Combined with automatic memory management, these features simplify data manipulation. Facile also supports generic polymorphism, allowing the creation of functions that can operate over more than one type.

Facile is strongly typed. Any program that passes the type checker cannot fail with a memory fault or illegal instruction at run time. In addition, the compiler can infer the types in a program without their being specified by the programmer (the inferred types may be checked against a programmer-supplied specification if desired).

Facile is higher order. Functions can be defined on the fly, in any scope. They may be passed to other functions or returned as results. Combined with

the polymorphic type system, higher-order functions allow code to be reused and specialized in different contexts. Facile also includes high-level exception handling features and a sophisticated module system for name management.

The second category of Facile's features includes support for concurrency and communication. In Facile, *processes* communicate via *channels.* (Facile processes are lightweight threads and should not be confused with heavyweight operating system processes.) Individual processes are created by spawning process *scripts.* A script can be spawned in the same address space, where it executes concurrently, or on a homogeneous remote machine. The process model is tightly integrated with the functional language model, allowing functions to be implemented using processes and vice versa. The programmer can use whichever model is most natural.

Communication in Facile is synchronous. The language provides a `send` function that takes a channel and a value to transmit, and a `receive` function that takes a channel and returns a value. These functions handle all the low-level details involved in communication. Facile also supports an *alternative* operator that allows a process to attempt multiple simultaneous sends and receives, with only one send or receive succeeding [see Knabe 1993]. Communication channels in Facile are typed, and *any type of value* can be transmitted over channels. In particular, process scripts, functions, and channels are all values in the Facile language model and can be transmitted, though the first two are restricted to homogeneous transmissions.

Facile has been used to build several distributed applications. These include Calumet [Talpin 1994], a multi-user presentation system that distributes visual and audio information across a network, and Einrichten [Ahlers et al. 1995], an application that combines distribution, real-time graphics, and live video to permit collaborative interior design work for widely separated participants.

2.2.2 Agent Language Properties Supported by Facile

Facile's principal advantage is that it already supports all of the code manipulation requirements for agent programming, including integration of code manipulation into the language. As we noted previously, Facile has first-class functions that we can create at run time, apply to arguments, pass to and from functions, and transmit with the `send` and `receive` primitives. The natural representation for an agent in Facile is simply a function.

Functions in Facile consist of the function's code and the bindings from its

lexical environment. An agent can therefore be structured as many functions with one of them designated as the main entry point; all of the other functions will be part of the main function's environment.

An example will show how in principle an agent could be programmed in the original version of Facile (though the code shown contains some flaws, as we shall see in the next section). We will consider an agent that transmits the local time in seconds back to its sender. First we write a function for getting the time:[3]

```
fun getTimeSeconds () =
    let val TIME{sec=seconds,...} = gettimeofday()
    in seconds
    end
```

We next create a channel called `answerChannel` for sending the time back. A main agent function named `reportTimeAgent` calls `getTimeSeconds` and sends the result over `answerChannel`. Because they are free inside `reportTimeAgent`, both `answerChannel` and `getTimeSeconds` are part of its environment:

```
val answerChannel : int channel = channel()

fun reportTimeAgent () =                    (* Main agent function *)
    send(answerChannel, getTimeSeconds())
```

Because main agent functions look the same as other functions, the programmer may want distinguish them using a naming convention (such as appending `Agent` to their names).

On the sender side we transmit `reportTimeAgent` through a preexisting channel named `clientChannel` and await the result. The value transmitted for `reportTimeAgent` consists of a *closure* record that contains both `reportTimeAgent`'s code and the components of its environment.

```
val remoteTime =
    (send(clientChannel, reportTimeAgent);
     receive(answerChannel))
```

The remote host receives the agent from `clientChannel` and executes it by applying it to a null argument:

[3] We will use Facile syntax in our examples. An overview of the syntax can be found in [Thomsen et al. 1993].

```
let val agent = receive clientChannel
in agent()
end
```

The point of this example is that using lexically scoped functions makes agent programming straightforward. Any function can be an agent; we simply create one and transmit it.

The implementation of functions as closures is not a feature of Facile oriented specifically towards agents; rather, closures are generally necessary to implement higher-order functions. A closure encapsulates a function and allows it to be executed in a different environment while still preserving static scoping. The different environment can be in another part of the program or it can be on a different machine.

When the `send` primitive is passed an argument, it recursively descends through the components of the argument and prepares it for transmission. Closures in Facile are simply records consisting of a code string and the values free in that code, and they are treated like any other values by `send`. When a function closure is passed to the `send` primitive, all the components of the closure record are prepared for transmission, including components that are themselves closures. The `send` primitive can therefore easily transmit the entire set of functions making up an agent when passed just the main function of the set.

During its recursive descent, the `send` primitive makes a complete copy of its argument's components, which is then transmitted to the remote site. The important implication for agents is that a transmitted closure contains all the values it needs from the sending site. Reference cells pointed to by the transmitted argument are also copied. They do not provide an implicit means to share state with the sending site. (For cases when state must be shared, Facile provides a separate library for distributed references that is implemented using channels and explicit communication.) Thus Facile supports stand-alone execution; there are no implicit callbacks.

Facile already supports strong typing and automatic memory management. Strong typing is preserved in the presence of distribution through a module exchange facility that allows types and typed values to be shared between different systems. However, one restriction of this facility is that it can only be used after the values have been created on one system (i.e., after the system is running) but before the other systems that will share them have been fully compiled.

In sum, Facile is an attractive starting point for constructing an agent language.

It supports code manipulation, strong typing, automatic memory management, and stand-alone execution. Nevertheless, significant work is required to satisfy the other properties, which we consider in the next section.

2.2.3 Properties Not Supported by Facile

Our example of the time-retrieval agent might seem to show that the original version of Facile supports everything necessary for agent programming. However, several important components are missing which we must design and implement.

Facile does not support heterogeneity. The `send` and `receive` primitives can be used to transmit nonfunction values between heterogeneous systems. However, functions are sent as compiled code, restricting their transmission to homogeneous architectures. An exception is raised at run time when a cross-platform transmission is attempted.

Facile does not provide a means for functions to access remote resources other than by having the recipient explicitly pass in local values as arguments. Functions are strictly lexically scoped. We can therefore see a flaw in our time-retrieval agent in the call to the library function `gettimeofday`. We want the agent to call a local version of this function wherever it executes. However, there is no way to specify such access to a remote resource. Instead, when the agent is sent, the version of `gettimeofday` available on the sender will be sent with it. Because the `gettimeofday` function may have an architecture-specific implementation, we generally do not want to transmit it. Moreover, the inability to refer to remote resources means that if `gettimeofday` were not available on the sender to begin with, we would not be able to write the agent at all. (One solution would be for the recipient to supply `gettimeofday` as an argument to the agent, but then the recipient's code would not be as general; it would be specialized to a particular agent.)

In adding support for access to resources in remote environments, we will want to preserve strong typing. However, Facile's module exchange facility is not suitable for sharing types because it requires that all the components of a distributed system must be compiled in a particular order, which is not compatible with our independent compilation property.

Finally, in adding support for these properties, we will need to address performance.

2.2.4 Constraints from the Facile Project

In changing Facile into a language suitable for agent programming we must take into account certain restrictions introduced by working within a larger project with its own goals. The Facile implementation at ECRC is the basis for several different research efforts, of which support for agents is only one.

One requirement is to preserve the properties of the Facile language. Because it is the properties of Facile noted in Sections 2.2.1 and 2.2.2 that make it attractive for agent programming in the first place, this requirement is an important one. Extending strong typing to remote resource access is an example of maintaining an underlying property of Facile programming.

A second requirement is to minimize changes to the Facile compiler and run-time system. Other researchers are also changing the Facile system (for example, the type system is being overhauled to incorporate subtyping). Maintaining a coherent system requires that work be partitioned carefully. Another reason for minimizing changes results from Facile's implementation as an extension of Standard ML of New Jersey. Fewer changes to the system decrease the time necessary to upgrade Facile to a new version of SML/NJ.

2.3 Summary

Facile provides a good starting point for developing an agent language by providing first-class functions, full copies of closures on transmission, strong typing, and automatic memory management. It satisfies the essential property of code manipulation, and moreover allows code to be manipulated entirely within the language.

Facile is not a complete agent language, however. We still need to address heterogeneity, remote resource access, independent compilation, and performance. The challenge is to provide these capabilities while continuing to satisfy the properties that Facile already does and obeying the constraints introduced by the Facile project.

With these goals outlined, we can now turn to the design and implementation of our agent language.

Chapter 3

The Language and Compiler Design

Facile must be modified to satisfy the properties of heterogeneity, remote resource access, independent compilation, and performance. At the same time we want to preserve the agent language properties it supports already.

To accomplish this goal, we introduce the following language and implementation changes:

- We modify the compiler to generate code in a representation that can be executed on heterogeneous architectures. We concentrate on keeping space costs low.
- We introduce a means for referring to remote values in a type-safe way. We allow the remote value that is used by an agent to depend on the agent's execution site via dynamic linking.
- We modify the transmission of closures to handle *ubiquitous values,* which are never transmitted and thus improve performance.
- We change the run-time system to support lazy compilation of received agents.

We present our design and rationale for each of these changes in the next four sections. In Section 3.5 we then discuss related work, and we summarize the chapter in Section 3.6.

3.1 Transmissible Code Representations

A basic difficulty in implementing transmission of values across heterogeneous processors is differences in value representations. For example, suppose we transmit the bits making up an unsigned integer from a little-endian machine to a big-endian one. Because each machine interprets the bits differently, we will receive an integer different from the one we transmitted (unless, of course, the integer happens to be a binary palindrome).

A common method for handling the differences in how machines represent various data types is to define standard transmissible representations. Before transmission, each value is converted into a standard representation determined by its type in an operation called *marshalling*. On receipt, the complementary *unmarshalling* operation converts the value from its standard representation into an appropriate local one. A number of widely used standard representations exist, each defining a set of transmissible data types [Xerox 1981; ANSA 1986; Dean et al. 1987; Zahn et al. 1990; Sun 1992].

Marshalling is a recursive process. Complex types such as structures and arrays are built on top of primitive ones such as integers and reals. Marshalling progressively decomposes a value until it terminates with conversion of its primitive components. The representation of these components is often highly processor dependent as well as language dependent.

In Facile, the underlying primitive component of closures is *machine code*. Unlike other processor primitive types, machine code cannot be converted to an architecture-independent representation in a straightforward way. Translating machine code from one native format to another is impractical except in special applications, such as porting legacy code [Andrews and Sand 1992].[1]

In Facile and most other higher-order languages, code is not created at run time; functions produced during execution are just new closures containing old code. All the code in a program is known at compile time. We therefore have the possibility to perform code marshalling at compile time. Note that we cannot normally marshal data at compile time, because the data values are unknown until the program runs.

In the next section, we consider several different approaches to marshalling code at compile time, developing a hybrid approach for Facile. Because generating

[1] Vegdahl [1986] implemented dedicated translation routines to convert between two machine-dependent byte-code representations for Smalltalk (thus allowing exchange of *compiled methods*), but noted that his approach was not a general solution.

transmissible representations for every function in a program imposes high space costs, we then introduce function annotations into Facile that allow the programmer to identify which functions should be marshalled. We also discuss two methods for reducing space costs that we decided against implementing.

3.1.1 Marshalling Strategies

The root problem in marshalling code for transmission is to generate a representation that the recipient can understand. This representation can be one that is independent of the target architecture or one that is specifically tailored to it (i.e., machine code).

Architecture-Independent Representations

The first approach, using architecture-independent representations, is perhaps the more obvious one. At the source level, code is architecture independent. (Many high-level languages allow architecture-dependent source to be written, but this capability is oriented towards specialized programming, and we will ignore it.) It is only as source is translated successively into native machine code that it becomes more and more processor specific. Therefore, we can simply use source as our transmissible representation, or we can use a lower-level but still architecture-independent representation.

The recipient of an architecture-independent representation must either compile or interpret it. Because the compilation must be performed at run time, we want it to be as fast as possible. A lower-level representation is likely to be faster to compile than a higher-level one such as source, because fewer compilation steps must be performed. However, to generate a low-level representation originally that is still architecture independent we may have to make certain worst-case assumptions about the target, such as how many registers it has. Such assumptions may allow the low-level representation to be compiled faster, but the cost may be that the resulting machine code executes more slowly than machine code generated from source. Thus there is a potential trade-off between compilation time and subsequent execution speed.

For interpretation, source is once again a possible representation. However, byte-coded virtual machine code is likely to execute faster. A single representation may also support both compilation or interpretation, in which case the recipient must decide which to use. Interpreted code is likely to run slower than compiled

code, but it can be used immediately upon receipt; there is no delay while compilation is performed. The choice of whether to compile or to interpret is affected by the demands of the application (e.g., short response times) along with the expected running time of the agent.

Machine-Code Representations

We can sidestep the preceding trade-offs if we take the second approach to marshalling. In this case we use machine code as our transmissible representation. To support heterogeneity, we precompile for multiple architectures. At run time we select the proper machine code representation to transmit based on the recipient's architecture. The received code can be executed immediately. Moreover, it executes at maximum speed, as we can optimize it fully during the compilation.

The disadvantage to this approach is that each sender must generate machine code for all potential recipient architectures and must store all these multiple representations. Adding a new architecture to the distributed system involves modifying every sender to handle that architecture. Storage costs increase as well. Another problem is that communication patterns in the system are more restricted: A recipient cannot forward code to another architecture if it has received the code only in the recipient's native format.

A Hybrid Approach

For Facile, we use a marshalling strategy that combines the best elements of both of the preceding approaches. There are two components to the strategy:

- We support more than one transmissible representation.
- We allow several representations to be transmitted *together*.

The reason to support more than one representation is that it allows the programmer of an agent to pick the representation most suited for the execution characteristics of that agent. If the agent is a batch job being sent to a CPU server, for example, the programmer might pick a high-level representation. Such a representation could be compiled with as many optimizations as possible on the server; the cost of performing these optimizations would presumably be outweighed by the faster execution speed during the long run of the agent. In contrast, an interpreted representation or a lower-level compiled representation might be more

suitable for an interactive agent that performs little or moderate computation. Representations may also vary on other dimensions that affect performance. One in particular is compactness of the representation: For a large agent sent over a slow link, transmission time may be a significant. A programmer targeting users with dial-up or wireless links would then need to consider which code representations are smallest.

The primary reason for allowing several representations to be transmitted together is to allow a machine-code representation to accompany an architecture-independent one. In this way we gain the benefits of each approach. If the recipient shares the architecture of the sender, the machine code can be used, eliminating the start-up delays of compilation and providing maximum performance. Otherwise, the architecture-independent representation can be used, either with interpretation or with compilation.[2]

Transmitting machine code optimistically in this way has a cost: Transmission times will be higher. This strategy may therefore not be appropriate for slow links. On the other hand, an agent programmer may assume that users on slow links are probably using Intel x86-based computers. She may therefore decide that the potential improvement in execution speed by sending Intel machine code justifies the extra transmission costs.

Note that giving the recipient the choice of which architecture-independent representation to use is not a good reason to allow several representations to be transmitted together. A recipient is unlikely to know the execution characteristics of an agent, and therefore would not have the information to pick the best representation.

3.1.2 Reducing Space Costs with Annotations

Compile-time marshalling of function code offers a run-time performance advantage. Each time a function is transmitted, no time must be spent on marshalling because the marshalled representation of the function's code is already available.

[2] This strategy is similar to that used for transmitting values in Network Data Representation [Zahn et al. 1990]. NDR provides several standard representations for each data type, any of which may be transmitted. For example, both big- or little-endian formats are legal representations for integers. Therefore, a sender can transmit an integer in its local native format. If the recipient happens to use the same representation, the integer can be used directly, and the cost of marshalling and unmarshalling is saved. If the recipient uses a different local representation, it does the work of converting the integer itself. In effect, marshalling is performed lazily.

However, this availability of the representation does not come for free. Storing the representation generated at compile time imposes a space cost that continues while the program executes.

Keeping one or more transmissible representations for a function as well as its local machine code representation increases the space required to store it. The size of the increase depends on the relative sizes of the transmissible and local representations, but it could easily double the space required for a function. For a large program, doubling the run-time size of every function would bloat program images unacceptably. We must therefore reduce the needed space.

The simplest and most effective way to reduce the space cost of a transmissible representation is not to generate one at all. If there is no transmissible representation for a function, it occupies no additional space. We cannot do better than zero overhead!

We expect the average program will only contain a small proportion of functions that might ever be transmitted. We call such functions *potentially transmissible.* For the remaining "normal" functions, there is no need to generate and store a transmissible representation. Although we must still generate and store representations for potentially transmissible functions after identifying them at compile time, across the whole program we can realize significant savings. In addition to reducing run-time space costs, generating fewer transmissible representations also saves compilation time.

Identification of Potentially Transmissible Functions

For our extension to Facile, we rely on the programmer to *annotate* potentially transmissible functions so that the compiler knows to generate transmissible representations for them. Because control-flow analysis is a difficult problem in higher-order languages [see Shivers 1991], it may be hard to identify these functions.

The criteria we developed for determining whether a function is potentially transmissible, and therefore should be annotated, are as follows:

- Any function that is passed directly to the `send` function is potentially transmissible.
- Any function that is referred to by a potentially transmissible function and is not defined within it is potentially transmissible.

A function is passed directly to `send` if it is the only argument to `send` (other than the channel argument) or if it is contained within a language-level data structure passed to `send`. For example, in both `send(chan,f)` and in `send(chan,("abc",12,f))`, we consider the function `f` to be passed directly to `send`. In the latter case, `f` is a member of a tuple.

We can illustrate the recursive nature of the definition with a simple example. Consider the following code:

```
let
    xfun square x = x * x
    xfun cube x = x * square x
in
    send(chan,cube)
end
```

First we note that `cube` is potentially transmissible because it is passed directly to `send`. Because `cube` refers to `square`, `square` is also potentially transmissible. We must therefore annotate both. (The actual annotation is to use the keyword `xfun` rather than `fun`.) All built-in library functions (the *pervasives*) are assumed to be potentially transmissible and are already internally annotated, so we need do nothing for the multiplication function, `*`, which is referred to by both `cube` and `square`.

Note that if we change the example slightly, the set of potentially transmissible functions changes (and thus our annotation requirements as well):

```
let
    xfun cube x =
        let fun square x = x * x
        in x * square x
        end
in
    send(chan,cube)
end
```

Only `cube` is potentially transmissible and must be annotated. Because `square` is defined within it, it is not potentially transmissible and need not be annotated. In effect, `square` is treated as if it is just part of the code for `cube` (which it is), and not as a separate, distinct entity.

Determining which functions are potentially transmissible becomes harder when higher-order functions are used, though the definition continues to apply. Consider this example:

```
fun makeCube square =
    let xfun cube x = x * square x
    in send(chan,cube)
    end
```

In this case we must annotate `cube` as before because it is potentially transmissible. In addition, all functions in the program that may be passed as arguments to `makeCube` are potentially transmissible. We must find these functions and annotate them as well. Though Shivers [1991] shows the difficulty of this problem in general (it is undecidable), we can expect that an agent programmer would normally structure her code so that such functions could be readily identified by her.

The most difficult case occurs when we transmit a higher-order function. Here is an example:

```
let
    xfun makeAdder x =
        let xfun add y = x + y                    (* xfun or fun *)
        in add
        end
in
    send(chan,makeAdder)
end
```

As in the other examples, `makeAdder` is potentially transmissible. The question is whether `add` is potentially transmissible. It is not potentially transmissible by the second part of our definition because it is defined within `makeAdder`. However, the first part of the definition *may* apply. It depends on what is done with `add`. If it is passed directly to `send` by some other code, then it must be annotated because it is potentially transmissible. Otherwise, it is not potentially transmissible.

The code that manipulates `add` may not be known to the programmer. For example, `makeAdder` may be sent to a remote site, and the code there may or may not transmit `add`. In cases where it is not possible to determine whether `add` is potentially transmissible, the conservative approach is to annotate it.

Potentially transmissible functions may be anonymous or occur in mutually recursive blocks. To annotate functions in the former case, the keyword `xfn` is used instead of `fn`. Mutually recursive functions are introduced in Facile by a single `fun` declaration with function definitions separated by `and`; changing

the `fun` to `xfun` serves to annotate all the functions at once. (Functions defined in this way are not required to be mutually recursive by Facile. The single annotation will still serve to annotate all the functions in the block, however.) Facile allows curried functions such as `fun f x y = ...`, which is equivalent to `fun f x = fn y => ...`; annotating the single `fun` is equivalent to annotating the `fun` and `fn` in the expansion.

An Alternative to Annotation

Relying on programmer annotations for identifying potentially transmissible functions, while easy to implement, has disadvantages common to most annotation-based systems. The primary drawback is that annotation has no connection to the normal language semantics. That is, from the perspective of the high-level language, it does not matter whether a function is annotated or not. Annotations only affect how the compiler generates code. Their underlying purpose is to enable the transmission of functions across heterogeneous architectures, whereas the language hides the existence of different architectures. Because of this separation from the concepts and principles of the language, programmers may forget needed annotations, leading to programs that raise exceptions at run time. Programmers may also choose to risk wasted space over faulty programs and annotate more than is necessary.

As an illustration, consider a variation on our earlier example with `cube` and `square`:

```
fun square x = x * x
xfun cube x = x * square x
```

The code is faulty because `square` is missing an annotation. An attempt at run time to marshal `cube` for transmission to another architecture would fail. Such mistakes are surprisingly easy to make, particularly as code is modified.

A preferable approach is for the compiler to detect automatically which functions are potentially transmissible, making conservative judgments when it is not clear. One method under investigation by other researchers is *effect-based analysis*. In this analysis, expressions (including communications) are assigned "effects" as well as types, and this static information can then be used to predict the dynamic behavior of the program [see Lucassen and Gifford 1988; Talpin and Jouvelot 1992; Nielson and Nielson 1994a,b; Thomsen 1994]. With such analysis it may be possible to identify sets of functions that include all the potentially transmissible ones.

3.1.3 Other Space Reduction Methods

Generating fewer transmissible representations through annotations is not the only way to save space. We can trade space for time by *compressing* transmissible representations. Because the compression would be performed at compile time, the only run-time cost would be decompressing at a recipient. Besides reducing space requirements at the sender, the smaller representations would take less time to transmit, improving performance in this part of the system.

However, the incremental improvement in space savings is not likely to be large in comparison to eliminating generation of unnecessary transmissible representations. Even though decompressing is normally faster than compressing, we do not expect the space savings and shorter transmission times to outweigh the additional run-time costs of performing it. The exception is in transmissions over very slow links. However, slow links often integrate compression at a lower layer in the protocol stack. For example, many modems support compression in hardware.

Another way to reduce space is to eliminate local machine code representations for potentially transmissible functions. Instead, the compiler would only generate and store intermediate representations. Note that we do not lose the ability to run functions locally. To run a function, the unmarshalling machinery can simply translate the intermediate representation into an executable one.

An argument for this approach is that most transmissible functions will never be executed locally. That is, the programmer would probably not write an agent to carry out a task remotely if the task could be performed locally, so we can expect the functions making up the agent will always be sent elsewhere for execution.

However, there are two drawbacks. First, should we need to execute the function locally, we pay the costs of interpretation or compilation. These time costs may not be justified by the incremental space savings, particularly if native code is compact relative to transmissible representations (and it is for Facile, as we shall see in Chapter 6). Second, the machine code representation may be needed even if the function is not executed locally: As discussed in Section 3.1.1, we may want to transmit the machine code with other representations. We therefore avoid this strategy.

3.2 Typed Remote Resource Access

In providing typed remote resource access to agents, there are several problems we must solve. The programmer of an agent needs a way to specify which remote resources the agent will access. The compiler of the agent needs to know the types of those remote resources so that it can type-check the agent code. Finally, the run-time system must be able to link a received agent with the resources that the agent needs.

For flexibility, we want a naming and linking scheme that allows an agent to be linked and executed on multiple sites. That is, the name of a resource should not be tied to a particular implementation on a particular machine. An agent should be able to move around the network and to use the local implementation of the resource wherever it is.

At the same time, we want to preserve strong typing across the distributed environment. Although an agent may use different implementations of a resource, that resource should have the same type on every site. The type should be the same as the one assigned to the resource when the agent code was type-checked.[3]

To accomplish these goals, we introduce a new language construct, *proxy structures,* which can be used to specify remote resources by their types and the libraries in which they are located. We see that proxy structures can be used to introduce new type abbreviations into a distributed environment, but not abstract types, a restriction which stems from our desirable properties for the implementation. We then discuss the use of dynamic linking to resolve proxy structures at run time. Finally, we note rules on the use of proxy structures.

3.2.1 Proxy Structures

Our approach to naming and typing remote resources is based on the concepts already found in the Facile module system. Although the module system incorpo-

[3] A stronger property to maintain is to ensure that the resource has equivalent *functionality* on every site as well as the same type. Although maintaining this property is not necessary for providing safety from run-time type errors, it would prevent the case of two implementations of a resource performing different actions even though the types are identical. For example, a function to delete a file could have the same type as a function merely to "touch" a file, but the two functionalities are quite different. Work on implementing such properties is based on specification languages; for example, see [Rollins and Wing 1991]. We have not attempted to extend that work into our system for identifying resources.

```
signature ADDER1 =
  sig
    val plus1 : int -> int
  end

structure Adder2 =
  struct
    fun plus1 x = x + 1
    fun plus2 x = x + 2
  end
```

```
signature ADDER2 =
  sig
    val plus1 : int -> int
    val plus2 : int -> int
  end

structure Adder3 =
  struct
    fun plus1 x = x + 1
    fun plus2 x = x + 2
    fun plus3 x = x + 3
  end
```

Figure 3.1: Signatures and structures in Facile. Either signature can be used to specify the interface to either structure. If `Adder3` and `ADDER1` are matched, the values `plus2` and `plus3` are hidden. If `Adder2` and `ADDER2` are matched, no values are hidden.

rates a number of sophisticated features, for our purposes we can focus on its most basic components. These components share concepts with the module systems of other languages, such as Ada.

In Facile, value and type definitions can be grouped into modules known as *structures.* For any structure we can create an interface that specifies the types (by name) and values (by name and type) that are available in the structure. Such a specification is called a *signature.* A signature need not be complete; it can omit types and values that are private to the structure and should not be publicly accessible. A structure can be specified by more than one signature, and a signature can be a specification for more than one structure. Figure 3.1 provides some examples.

We can view a signature as a specification for a set of structures that define the same types and values. In the case of mobile agents, we need to be able to specify a set of resource implementations (i.e., an implementation on each site) that share the same name and type. There is a clear parallel between this need and the model provided by signatures and structures. With some changes, we can reuse this model for the new purpose.

Resources are simply values with names and types. We can describe any given set of resources with a signature. Any site that provides the resources must implement a structure the matches the signature. The signature is common to all providers and users of the resources.

An agent writer who wants to program an access to a remote resource starts with the signature (called the *remote signature*) that specifies *remote structures* providing the resource. The programmer then creates a *proxy structure* that introduces a placeholder name for structures that match the signature. Access to the remote resource is then programmed just like access to members of a normal structure. Because the signature describes the type of the resource, the compiler can also type-check uses of the resource in the agent program.

On transmission to a remote site, an agent carries with it the signatures for the structures containing needed resources. At the site, the agent is linked to the local structures that match those signatures. (In the implementation the signatures are not actually transmitted; see Section 4.1.)

As an example, consider the following fragment of an agent program that makes use of a remote `gettimeofday` function:

```
xsignature TIMESIG =
  sig
    val gettimeofday : unit -> int * int
  end
xstructure RemoteTime : TIMESIG

xfun agent () = ...RemoteTime.gettimeofday () ...
```

The signature `TIMESIG` specifies a remote collection of resource values, as indicated by the keyword `xsignature` rather than `signature`. In this case, there is just one value in the signature, the function `gettimeofday`. Following the signature definition a proxy structure is created using the special `xstructure` keyword. No implementation is given for the structure; the code simply introduces a name (`RemoteTime`) that serves as a placeholder for the remote structures specified by `TIMESIG`. The agent uses the remote value `gettimeofday` by way of the proxy structure. When the agent is shipped to a remote site, the recipient must resolve `RemoteTime` by linking with a structure that implements `TIMESIG`.

Using proxy structures builds on the general name management scheme provided by the module system. Remote values are encapsulated into modules with abstract interfaces just like the programmer's own code. The module system allows the programmer to keep collections of values separate by giving those

```
signature STACK =
  sig
    type 'a stack
    val empty : 'a stack
    val push : 'a * 'a stack -> 'a stack
    ... other operations on stacks ...
  end
```

Figure 3.2: A signature for polymorphic stacks. The values in the signature are declared in terms of the abstract type `'a stack`.

collections names, and the use of proxy structures is a natural way to integrate additional collections of values into the programming environment. In effect, we provide a form of lexical scoping for remote resource access; the types of remote resources are bound statically, although the final instantiation of a resource is deferred until the value is linked in at run time.

3.2.2 Types

In addition to values, a signature can declare new types. In keeping with the principle of abstraction, type declarations in signatures need not contain any information about how the types are actually implemented. Figure 3.2 shows a fragment of a signature for polymorphic stacks.

Abstract types pose a problem to our remote access method because the same type can have different implementations on different machines. A value with one implementation could be transmitted to a machine with a different implementation of the value's type, which would cause a run-time failure. The following example shows how such a failure can occur with two agents that exchange integer stacks via a channel:

```
xstructure Stack : STACK                    (* A proxy structure *)

val chan : int Stack.stack channel = channel()

xfun Agent1 () = send(chan, Stack.push(5, Stack.empty))
xfun Agent2 () = Stack.push(6, receive(chan))
```

Assume the two agents are sent to different sites. `Agent1` creates a stack using the local implementation of `Stack` and sends it through the channel. `Agent2` receives the stack and uses the implementation of `Stack` at its site to push another value. However, there is no assurance that the `push` function on `Agent2`'s site uses the same underlying representation for stacks as is used on `Agent1`'s site. As a result, the operation may fail catastrophically.

If we changed the example so that a *single* program contained two implementations of `Stack` (with each agent using a different one), the type checker would be able to detect the inconsistencies and issue an error message at compile time. The preceding problem arises because the type-checker does not know how stacks are implemented and cannot know because those implementations are distributed on other machines. Our agent language property of independent compilation prevents us from requiring that a single compilation be used to check all components of the distributed system.

There is a basic incompatibility between abstraction and our implementation goals for remote resource access. We want to link agents to the local implementations of resources and types on their execution sites, rather than transmitting those implementations with the agents. To perform this linking safely, we need sufficient information about what is being linked. Abstraction hides the needed information.

We therefore prohibit abstract types from signatures used for proxy structures. Instead, such signatures must provide the definition of each type (these definitions are known as *type abbreviations*). In the `STACK` signature, we might now have

```
type 'a stack = 'a list
```

The implementation of the type is explicit and based on a standard abstract type provided by Facile (`'a list`). We rely on Facile to provide identical implementations of its standard abstract types across machines. These abstract types are based on primitive built-in types such as `int` and `string`, and Facile properly handles conversion of the latter in transmissions between machines. Thus, when type abbreviations are used in the signature, we know that all implementations of the type in remote structures will be equivalent.

A second form of abstraction is possible with *datatypes.* A datatype definition in Facile introduces a new type and constructors for creating instances of that type. Here is a polymorphic datatype for a binary tree with entries at the leaves:

```
datatype 'a tree = NODE of 'a tree * 'a tree | LEAF of 'a
```

Datatypes are *generative* in Facile: Each datatype definition generates a new, unique type. The constructors for the type (`NODE` and `LEAF` in our example) are unique to it. A second, identical datatype definition may introduce constructors with the same names, but they cannot be used to create or deconstruct instances of the first type.

Identical datatype definitions thus produce distinct types, even though the compiler implements them in exactly the same way. If in our stack example we had used a datatype for stacks, the two agents would be able to exchange a stack without any danger of a run-time failure, because the datatype would expose the structure of the stack type. Nevertheless, the operation would be a type error according to the Facile type discipline.

We therefore eliminate generativity for datatypes in the following case: All identical datatype definitions of structures specified against remote signatures are equivalent. They remain distinct from other identical datatype definitions that happen to be found in the program, but not from each other.

The place of this new kind of datatype in Facile's type system can be clarified by considering a two-dimensional categorization of types. On one axis we have generativity versus nongenerativity, and on the other we have product (nondatatypes) versus coproduct (datatypes). Facile supplies only nongenerative product types and generative coproduct types. We are introducing a nongenerative coproduct.[4]

Sacrificing the abstraction provided by generativity is necessary for safe dynamic linking. In practice, the loss goes unnoticed, because the abstraction provided by generative datatypes is neither desired nor expected in programming with proxy structures and remote resources. There was no need for generative datatypes in the agent applications we constructed (see Chapter 5).

3.2.3 Linking

When an agent is received at a site, it must be linked dynamically with modules that supply the values it needs. Linking instantiates the proxy structures in the agent and allows it to use local implementations of resources.

Dynamic linking should not be confused with full dynamic binding. Our dynamic linking is performed on modules, with static typing provided through lexical scoping. Furthermore, linking is performed only when a new function value (an agent) is introduced into the local environment.

[4] I am indebted to Tsung-Min Kuo for this categorization.

```
functor Agents (Stack : STACK) =
  struct
    val chan : int Stack.stack channel = channel()

    xfun Agent1 () = send(chan, Stack.push(5, Stack.empty))
    xfun Agent2 () = Stack.push(6, receive(chan))
  end
```

Figure 3.3: A functor parameterized by one argument.

Dynamic linking of proxy structures is not very different from other linking mechanisms built into Facile. In Facile, a structure can be parameterized by other structures. Such a parameterized structure is known as a *functor*. When the functor is applied to actual structure arguments at link time, the parameters are instantiated and the functor produces a new structure.

Figure 3.3 provides an example. When the functor `Agents` is applied to a structure that matches the `STACK` signature, a new structure will be created containing the channel and agents. The parameter `Stack` will be instantiated with the structure supplied as the argument. Note that in contrast to our example on page 35, if these agents are transmitted the local implementation of the stack will be transmitted with them.

Proxy structures and functor parameters play essentially the same roles. Both make it possible to program references to values from a structure given only the signature for that structure. We can think of a function that uses proxy structures as existing inside a functor parameterized by those proxy structures. Sending the function translates into sending the functor. When it is received, the functor is applied to the appropriate local structures, but the result is a function value, not a structure.

The key difference between the two linking mechanisms is their relative explicitness. Functors in Facile must be explicitly applied to a fixed set of structure arguments. In our dynamic linking approach, the "functor" for an agent knows what structures it needs to be applied to, and these are automatically supplied to it by the recipient without any action by the programmer.

The difference is more than a matter of simplifying the programmer's work. Implicitness is better suited to the realities of distributed systems by supporting

extensibility and backwards-compatibility. Open distributed systems evolve in a very loosely coupled way. When one component is extended and upgraded, we cannot expect that all the others will be as well. If additional resources are made available on a server in new structures, newly written clients can send functions that make use of them. Meanwhile, functions from old clients will be handled as before. The underlying machinery takes care of linking the right structures to each incoming function, whether it is old or new.

If the language were to require that all structure instantiations be explicitly programmed, we would have the problem of needing to supply one set of structure arguments to old functions and a different, expanded set to new ones. Solutions to this problem are certainly possible (e.g., a new version of the service could be offered in parallel, or received functions could specify an interface version), but they lack simplicity.

Implicitness also provides some convenience in that it preserves the property that agents are constructed from normal functions. The functions do not need to be treated specially by the programmer (other than having annotations, which serve a different purpose); they may be received and executed directly. In an explicit approach, functions could not be received as functions. They would be some other kind of value that the programmer would have to handle specially.

Another explicit approach is simply to transmit functors. Because functors are modules and are not part of Facile's expression language, supporting their transmission would change the Facile model of communication profoundly and require a completely new approach to the modules facility.

Dynamic linking differs from static linking in that it may fail. A received agent may require resources that are not available locally. The recipient must be able to detect this exceptional condition, or at the very least be unaffected by it. In Facile we can handle missing resources by raising a special exception when the recipient attempts to execute the agent. The recipient can choose what action to take at this point; for example, it can either discard the agent, attempt to contact the sender, or send the agent to a different server. The point to raise the exception is on an execution attempt rather than as soon as the agent is received, since a recipient may have no intention of executing an agent (it may just be a front-end that will dispatch the agent elsewhere). The exception must be handled by whoever tries to execute the agent.

To complement the ability of the recipient to detect agents that require unavailable resources, the sender may need the ability to detect when recipients are unable to provide the necessary environment for its agents or to determine what

resources are indeed available. The former capability can be provided by sending a *trial agent* that, when executed, retrieves the real agent from the sender, attempts to execute it, and reports back to the sender if a linking failure occurs. The latter can be provided by sending an *inspection agent* that uses a pervasive function to get the list of locally available library modules and sends it back.

3.2.4 Access Context

Although proxy structures allow the programmer to reference remote resources as if they were in local modules, these structures are just proxies: There is no local implementation for the value names that they make available. We therefore have the question of what it means to refer to such a value within an expression that is outside of an agent.

Rather than define a general semantics for such cases, we simply eliminate the problem by introducing the following restriction: The members of a proxy structure that are values may only be referenced from within the body of a potentially transmissible function. References anywhere else in the program result in a compile-time error. (References to *types* from proxy structures are legal everywhere.)

This restriction is not a complete solution, however, because we can create expressions that execute a potentially transmissible function. For example,

```
xstructure Remote : REMOTE
xfun f () = ... Remote.x...          (* Reference to a remote resource *)
val y = f ()                         (* Call to f requires a value for x *)
```

To handle this case, we treat potentially transmissible functions defined locally exactly the same as those received from channels: If a function requires resources that are not locally available, a special exception is raised.

3.3 Ubiquitous Values

The kinds of values accessible to agents that we have seen so far can be divided into two groups: *singular* and *remote*. Singular values are assumed to be available only on the agent's originating site. When an agent is transmitted, they must be transmitted along with it so that the agent can execute without callbacks. In

contrast, remote values are defined to exist only at other sites. They are linked to an agent at its destination.

Between these two categories we can define a third: *ubiquitous values.* A ubiquitous value exists on the agent's originating site just like a singular value. Unlike a singular value, we further assume that an "equivalent" implementation of the value also exists on sites to which the agent may be transmitted, just like multiple implementations of remote values. As a consequence, an agent that refers to a ubiquitous value need not take the value with it; the agent will be linked to an identically typed and named replacement at other sites.

The most likely candidates for ubiquitous values are library functions (in Facile, the pervasives) and values storing system state. Defining these values as ubiquitous gives us two main advantages. First, they do need not be transmitted with agents, so transmissions are smaller and fewer values must be marshalled and unmarshalled. Library functions are used frequently in Facile programs, and the savings in not transmitting them are significant. For example, the function to print "hello, world" marshals to 790 kilobytes if the standard library containing the `print` function is pulled in but to just a few hundred bytes if it is not.

The second main advantage is that agents can access *nontransmissible values.* Though Facile can transmit any value, some values incorporate system state or system-specific functionality and therefore are meaningless to transmit. An example is a file descriptor. The descriptor type is implemented as an integer in Facile, which can certainly be transmitted, but a descriptor is of no use outside of its home address space. We want agents to be able to access such values both on their originating site and after transmission. If these values are defined as ubiquitous, agents will always use the locally available implementation.

Ubiquitous values need not come only from standard libraries. Consider a server that supports an interface to special image processing hardware on its host. There may also be a local software implementation of the values in the interface. Given a function that performs image processing through use of these ubiquitous values, we could either run the function locally or transmit it to the server, depending perhaps on the server's reported load.

3.3.1 Ubiquitous Structures

Programming with ubiquitous values is similar to programming with remote values. The same naming, linking, and typing problems must be solved to ensure that a transmitted agent safely accesses the resources on a remote site. The one new

feature that ubiquitous values introduce is that they have local implementations.

Ubiquitous structures are an extension of proxy structures. A ubiquitous structure can be created by using a remote signature to declare a structure name (just as with proxy structures) and also providing a definition for the structure. To distinguish the declaration from a normal structure definition, we use the `xstructure` keyword. The compiler can distinguish between proxy and ubiquitous structures because the former have no definition, while the latter do.

Because they are available locally, the values in a ubiquitous structure may be accessed from both within and outside of potentially transmissible functions. An agent that only uses singular and ubiquitous values can always be executed locally. However, the special treatment for ubiquitous values at transmission time (i.e., not sending them) applies only to those that are directly referenced from within the bodies of potentially transmission functions. Ubiquitous values that appear in all other contexts are transmitted just like singular values.

The reason we do not extend the no-transmission property to ubiquitous values outside of agents is that it would require tagging all such values in the run-time system so that they could be detected by the marshaller at transmission time. However, we are not willing to make such a comprehensive change to the Facile run-time system, as much of Facile's existing functionality would have to be completely reimplemented.

Note that ubiquitous values are not the same as global values. Ubiquitous values have limited scopes (their structures) and implementations that are restricted to their sites. Their only "global" aspect is that ubiquitous values on different sites can share the same name and type, but not necessarily the same implementation. Thus a received agent can link to the local version of a ubiquitous value on each site.

3.3.2 Closure Trimming

Supporting the no-transmission property of ubiquitous values inside potentially transmissible functions requires two operations. The first is active at compile time, and the second during marshalling at run time.

When an annotated function is compiled, we scan its body for references to ubiquitous values. We then store with the function a map recording which elements of the function's run-time closure are ubiquitous.

During marshalling, we retrieve the map and use it to reduce the closure to only the singular values. We call this operation *closure trimming*. We marshal

the singular values and add placeholders for the trimmed ubiquitous values. The placeholders contain the information for linking at the recipient.

Closure trimming is a key performance optimization. We analyze its effect on agent programs in Chapter 6.

3.4 Lazy Run-Time Compilation

Before a received agent can be run, it must be converted to an executable form. The first step in this process is compilation, which can be eliminated if the agent's transmissible representation is directly executable. The second step is to resolve remote and ubiquitous value references in the agent through linking. For simplicity, in this section we will refer to these operations collectively as compilation.

We need to establish how compilation is initiated. There are two main choices. Compilation may be explicitly performed by the programmer, or it may be implicitly performed by the run-time system.

In the former approach, we prohibit function channels. Functions must first be explicitly passed to a marshalling function that produces objects of a new type, e.g., `'a transrep`. These objects may be transmitted over an appropriately typed channel. After receipt, the objects are explicitly passed to a compilation function that converts from `'a transrep` back to `'a`. The need for the explicit marshalling function is because a Facile channel has only one type. The types of values sent into a channel must be the same as the types of values received, so we cannot send functions and receive `transrep` objects.

Though this approach is workable, it increases the work on the programmer and adds complication to the language. The alternative approach is to compile implicitly. We can then continue to marshal implicitly, and functions are received as functions from the programmer's perspective.

With implicit compilation we have a new choice of when to perform compilation. The *eager* strategy is to compile a function as soon as it is received. The *lazy* strategy is to defer compilation until the function is called.

The eager strategy is easier to implement, but the lazy one offers a potential performance benefit. If an agent is never used at a site, perhaps because it will simply be forwarded to another destination, we save the work of compiling it. Avoiding needless compilation improves the performance of the forwarder.

This argument is not the only one for lazy compilation. An agent will typically consist of many functions, which are used at different times and in response to

different events. Some of these functions may not be used at all during a run, and some may be used for the first time only later in the run. By compiling each annotated function of the agent separately and on demand, we can further eliminate unnecessary compilation and also amortize compilation over the run of an agent. Amortization may be particularly attractive for interactive applications, where it decreases the start-up delay.

Note that adding this level of compilation control to the explicit approach would be very complex. Rather than having a handle only on the top-level function of an agent, the programmer would need handles to all the other functions making up the agent.

For Facile we therefore chose to implement lazy run-time compilation.

3.5 Related Work

Other researchers have examined problems related to the ones we have addressed in the modification of Facile. In this section we review the work relevant to our major design areas.

3.5.1 Transmissible Representations

The problem of sending code between heterogeneous architectures has been widely addressed. In languages that support code transmission, we can identify several approaches:

Source. Although source code takes longer to process at run time in comparison to lower-level representations, it is a popular choice as the transmissible representation for languages based on Lisp or Scheme. The `read` and `eval` primitives of these languages make it easy to compile or interpret source code at run time. Relevant implementations include REV [Stamos 1986; Stamos and Gifford 1990a,b], Avalon/Common Lisp [Clamen et al. 1990], Partridge's late-binding RPC [Partridge 1992], Rosette [Tomlinson et al. 1992, 1993], Dreme [Fuchs 1993], and Messenger Scheme [Di Marzo et al. 1995].

Languages where source code is interpreted have also been used for agent experiments, mostly by researchers focusing on communication models rather than language design. Tcl has been the basis for Wit [Watson 1994,

1995], TACOMA [Johansen et al. 1995a,b], and Safe-Tcl [Borenstein 1992, 1994]. Postscript is the basis of Tps [Heimbigner 1995], and the Unix pattern-scanning language awk has been used for Transportable Agents [Kotay and Kotz 1994].

Interpreted intermediate representations. Several languages transmit code in an intermediate representation that is interpreted by the recipient. Obliq [Cardelli 1994] and a derivative, Phantom [Courtney 1995a,b], use abstract syntax trees for the transmissible representations. One drawback of these trees is that they are rather large, increasing storage costs and transmission times.[5] Compact byte-coded representations are used in Telescript [White 1994] and Java [Gosling and McGilton 1995].

Compiled intermediate representations. Though it is not an agent language per se, Omniware [Colusa 1995] provides the necessary infrastructure to receive, compile, and execute code at run time. Omniware compiles a RISC-like virtual machine representation to native code; the generality of the virtual machine allows any language to be targeted to it. Run-time compilation of virtual machine code has also been investigated for heterogeneous process migration [Attardi et al. 1988; Theimer and Hayes 1991].

The thesis work of Franz [1994] has shown that loading an abstract program representation and compiling it on the fly can be almost as fast as loading binary code. His abstract representations are half the size of binary code, and the time of compilation is offset by the faster loading. Moreover, the representations are architecture independent.

Machine code. Recent work on Emerald [Steensgaard and Jul 1995] implements code transmission by sending the appropriate machine code for the recipient's architecture. Code must be compiled in advance for multiple targets and stored for use at run time. This approach has also been used for heterogeneous process migration [Dubach et al. 1989; Shub 1990].

Compilation and interpretation from intermediate representations have been investigated in many other contexts as well. However, the preceding survey covers the major techniques. The primary distinction of our work in this area is support for *multiple* transmissible representations, which permits us to choose

[5] Personal communication with Luca Cardelli, 1994.

the representation based on the application and to send machine code with an intermediate representation. Within our implementation, both of these capabilities offer potentially improved performance. We investigate them further in Chapter 6.

3.5.2 Typed Remote Resource Access

Our use of lexically scoped remote resource identifiers combined with dynamic linking is new. However, other researchers have considered separately the transmission of functions with their lexical environments and the static type-checking of remote resource accesses. We examine three representative systems: REV, Obliq, and Avalon/Common Lisp.

REV

REV [Stamos 1986; Stamos and Gifford 1990a,b], for *remote evaluation,* provides code transmission in an RPC context. A client can relocate a procedure call $P(x,y,z)$ to a server S with the language construct **at** S **eval** $P(x,y,z)$. P and its arguments are transmitted to S for execution, and any results are returned to the client via a reply message.

A transmitted procedure may access resources on the server. REV is the only code transmission system other than our own that provides strong static typing for these accesses. *Service interfaces* are similar to signatures and specify the names of abstract types and their associated operations. Interfaces are assumed to have globally unique names and to exist in some global database available to the compiler. An REV procedure can access the operations in an interface, and these uses are statically checked. At run time, the procedure can only be transmitted to servers that implement an instance of the interface.

By allowing remote resources to have abstract types, REV appears to offer a significant advantage over our system. In our design, the only legal abstract types for remote resource are built-in ones such as `list`; for user-defined types, the structure of the type must be exposed. However, REV's approach has a serious limitation. Because the implementations of the types in an interface are private to each service instance, values of those types can only be manipulated on the site where they were created. We allow values to be transmitted through the network and manipulated at different sites regardless of where they were created.

To be fair to REV, its goals and communication model make it reasonable to use abstract types for remote resources. The purpose of REV is to extend the

functionality of the server interfaces, so values with abstract types are expected to stay where they are created. Support for general agent programming was not a target.

REV includes a feature somewhat analogous to our ubiquitous values. When each REV request in a program is compiled, the interfaces required by the transmitted procedure are compared against those provided by the destination service. If an interface required by the procedure is locally implemented and not known to be provided by the service, the code for the interface will be transmitted with the procedure. However, if the service does provide the interface, the extra code will not be sent and the service's implementation will be used. (If the interface is neither implemented locally nor known to be provided by the service, the compilation fails.)

The compiler's knowledge of what interfaces are provided by a service comes from service binding statements in the client program. To retrieve and bind a service id from the global name server, the programmer must specify statically what interfaces a retrieved service has to provide. Furthermore, so that the compiler can build the call graphs to determine what interfaces a procedure uses, REV prohibits the use of first-class functions and procedures. Transmissible procedures also may not refer to any values in their lexical environment except for other procedures.

Though in our system ubiquitous structures must be explicitly identified by the programmer, we allow first-class functions and do not rely on a global interface database. We consider both of these features an important advantage.

Obliq

Obliq [Cardelli 1994] is a distributed object-oriented programming language. The language is lexically scoped and untyped. Obliq supports first-class functions, and functions as well as other values may be transmitted over the network.

Obliq distinguishes between values that embody state (such as arrays) and those that do not. The former values are transmitted as *network references,* while the latter are simply copied. On a remote site, use of a network reference invokes an implicit callback to retrieve or update the value. A transmitted function closure may contain both kinds of values.

Because of its strict lexical scoping, Obliq functions cannot access remote resources directly. For a function to use any values local to a recipient, those values must be passed explicitly to it as arguments. In Section 3.2.3 we considered remote resource access based on parameterization, but we rejected it; in a loosely coupled

distributed system, fixing the set of resources available to received functions creates problems of extensibility and backwards compatibility.

The use of implicit callbacks is another important distinction between our system and Obliq. An Obliq programmer can create a stand-alone function, but she must take care not to refer to any nontransmissible values. In Facile there is never a danger of agents including implicit callbacks because all values are copied on transmission, including state-containing ones.

Avalon/Common Lisp

Avalon/Common Lisp [Clamen et al. 1990] allows the transmission of arbitrary Lisp expressions across the network. Each expression is accompanied by its lexical environment, which is used by the recipient to resolve references in the expression. If a value is not bound in this environment, it is resolved using the dynamic environment of the recipient. This technique allows remote resources to be accessed by the expression.

As an example, consider the following Avalon/Common Lisp code:

```
(let ((a 5)) (remote (+ a b)))
```

The `remote` special form causes the addition to be performed remotely. On the remote site, `a` will have the value 5. The values for + and `b` will be obtained from the recipient's dynamic environment.

The weakness in this approach to remote resource access is its reliance on dynamic scoping. There are no static guarantees for how a reference will be resolved. In Facile we have only dynamic *linking;* we use lexical scoping to guarantee that remote values have a specific type and come from a module with a specific signature.

Avalon/Common Lisp avoids implicit callbacks. Explicit callbacks can be easily programmed by prefacing an expression inside a `remote` with the special form `local`. This construct causes the expression to be evaluated on the original site.

3.5.3 Lazy Compilation

Our use of lazy compilation for transmitted agents is a unique feature of our design. However, the general technique of lazy program generation has been investigated in other contexts. Heering et al. [1994] developed a syntax-oriented

editor where the user is permitted to change the syntax interactively. Changes to the syntax require the generation of new parsers, but this operation takes too long for interactive performance. Parsers are therefore generated lazily; when part of the parser is needed that does not exist, the parser generator creates it. Another feature of their system is incremental generation: After a syntax change, as much of the previously generated parser is retained as possible.

Brown [1976] developed a lazy compiler for BASIC to solve the problem of an entire program not fitting into memory. Statements are compiled as they are encountered, and when memory is filled, all object code generated so far is discarded. This approach required several innovative addressing methods.

Chambers and Ungar [1991] introduced lazy compilation into SELF, a "pure" object-oriented language (i.e., every type, including primitive ones such as integers, is implemented as an object). SELF provides an interactive loop where programs can be compiled, executed, and then changed. However, compilation times were unacceptably long. The time was primarily spent generating code for uncommon actions, so by deferring compilation of these actions to the rare times they occurred at run time, interactive performance could be significantly improved.

Run-time code generation has been used as an optimization technique in the Synthesis operating system [Massalin and Pu 1989], which generates specialized code for frequently used kernel calls. It has also been used to specialize programs to data only available during execution [Keppel et al. 1993; Leone and Lee 1994].

In general, lazy compilation of programs has received little attention to date because there have been few reasons to defer compilation until run time. For mobile agents, compilation is possible only at run time, because the code is not available until then.

3.6 Summary

With the Facile language as a starting point, providing support for agents breaks down into four key areas. The first, transmissible code representations, is essential for handling heterogeneity. Adding transmissible representations to a language and compiler can dramatically increase the space requirements of programs, however, and this increase must be mitigated. Additionally, each transmissible representation has different implications for run-time performance. We balance some of the trade-offs by supporting the transmission of multiple representations.

The second area is preserving Facile's strong typing while allowing agents to access values on remote, separately compiled systems. Typing is normally enforced through a final check by a single, "focal" compilation, but in a distributed setting this option is not available. By using names *and* types to identify remote values, we ensure that agents are linked properly and that run-time type errors cannot occur.

Ubiquitous values and closure trimming form a third area closely tied to remote value access. Agents must be able to access values locally but should not have to carry them along when transmitted if those values will be available at the destination. This capability is particularly important for library and system-specific values.

Finally, compilation and linking of agents is performed lazily. To avoid potentially wasted effort, the functions making up an agent are only compiled as they are needed.

Chapter 4

The Agent Language Implementation

Our extensions to Facile consist of transmissible code representations, remote and ubiquitous values, and lazy compilation and linking. At the implementation level, they correspond roughly to handling the code in closures, handling the values in closures, and rebuilding executable closures on receipt. Several key problems are associated with implementing these extensions:

- For a recipient to link an agent, it must know which structures supply the remote and ubiquitous values the agent needs. The "names" of these structures are the signatures that were used to specify them. Agents should carry these names in a concise format that can be resolved quickly by the recipient.
- After a transmissible representation is generated at compile time, it must be stored in a location that can be found at run time. That is, at run time the `send` function will be given only the closure of the locally executable function; from this it must be able to find the transmissible code.
- Closures must be trimmed of ubiquitous values before transmission. To enable the marshaller to do this, the compiler must determine which entries in the closures of potentially transmissible functions are ubiquitous. This information must also be stored for access at run time.
- From the programmer's perspective, a received function should look and behave like any other function value. However, the recipient should not

compile or link it until it is called for the first time. At that point the transmitted representation must be replaced seamlessly by a locally executable one.

We begin this chapter by addressing the implementation of signature "names" in Section 4.1. In Section 4.2 we look at the generation of transmissible representations during compilation, the choices for those representations, and their storage and marshalling. We then examine closure map generation as a means for closure trimming in Section 4.3. In Section 4.4 we show how to defer compilation on a recipient transparently. We summarize the chapter in Section 4.5.

4.1 Naming via Signatures

According to the design in Chapter 3, agents only refer to remote and ubiquitous values as members of remote or ubiquitous structures. These structures are named globally by their signatures. A recipient uses the signature information transmitted with an agent to link it with the proper structures.

Although signatures serve as the names of structures, the actual signatures do not need to be transmitted with an agent. If we give each signature a unique identifier, we can associate that identifier with structures that match the signature. Agents then only have to carry the ids for the structures with which they need to link. An id can be much more concise than a full signature, which contains the names and types of all the values in a structure. Using ids reduces the size of agents and therefore their transmission and marshalling times.

We can generate a unique id when compiling a signature by including a timestamp and information from the local compilation environment. During front-end processing, remote or ubiquitous structures specified with the signature inherit its id. When an annotated function is compiled, the id for each referenced structure can be looked up and stored with the function's transmissible representation.

The only difficulty is that we must be able to associate the same id with separately compiled structures. Two different structures specified by the same signature but compiled on different sites must each inherit the same id. Because structures inherit ids from their signatures, the problem becomes to ensure that the signatures used to specify each structure have the same id.

We can solve this problem by making all the signatures copies of the same compiled signature. We compile each signature just once on one site and then copy it to all other sites that need it. The copying and distribution can be done

by conventional means (e.g., FTP), or it can be done with Facile's preexisting signature server, accessed via the keywords `demanding` and `supplying`. For example, a defining site for a signature could have the following code:

```
xsignature MYLIB = sig ...body of signature... end
supplying signature MYLIB as MYLIBKEY
```

When compiled, the signature `MYLIB` will be placed on the system-wide signature server in an architecture-independent representation. A copy of the signature will be downloaded when the following code is compiled at another site:

```
xsignature MYLIBCOPY = demanding signature MYLIBKEY
```

The implementation includes one special case where signature retrieval is not necessary. The standard library in Facile, called the *pervasives,* is assumed to be available on every site. For programmer convenience, the compiler automatically treats the pervasives as ubiquitous values. The pervasives have fixed ids that are everywhere the same.

4.2 Transmissible Code Representations

Perhaps the most obvious requirement for supporting agent transmission in a heterogeneous environment is one or more architecture-independent code representations. Incorporating such representations into Facile raises three basic problems:

- Transmissible representations should be generated only for annotated functions. To do this, these functions must be isolated from the surrounding code so that they may be processed separately. At the same time they should be converted into stand-alone "packages" that can be linked on other sites.
- The choice of transmissible representation affects the performance of function transmission, from the size required to store potentially transmissible functions to the speed with which they can be compiled and executed. We must decide which representations to provide.
- Transmissible representations for normal data are generated by the marshalling routines at run time. In contrast, transmissible representations for functions are generated at compile time. We need a location where the compiler can store them and which the marshaller can find.

In creating stand-alone packages for annotated functions, we must handle any singular or ubiquitous values that they reference. (Recall that singular values are available only on the originating site, ubiquitous values are available at all sites, and remote values are available at recipients.) Mutually recursive and higher-order functions add some special cases to consider. For the choices of transmissible representations, the intermediate representations from the Facile compiler offer several possibilities. Finally, for storing transmissible representations we look at how the run-time system stores and manipulates compiled code.

4.2.1 Stand-Alone Functions

In Facile, source is compiled in chunks called *compilation units.* A compilation unit usually contains multiple functions and other values. Each top-level structure becomes a compilation unit. Given a unit, we want to generate transmissible representations only for the annotated functions in it, not for the rest of the code.

The first step we take is to extract annotated functions from the surrounding code. Once extracted, they can be placed in their own compilation units and processed separately. To simplify the code manipulation necessary for extraction, we want to perform this step after the code has been processed by the front end. At the same time, the code must still be in an architecture-independent representation so that we can generate transmissible representations for the functions. In the Facile compiler, a representation that satisfies these two constraints is the *lambda language.* The lambda language is a simple lambda calculus variant that consists of about twenty constructs (Figure 4.1 on page 58 provides an example). By the time code is transformed into this representation, all syntax- and type-checking has been performed.

After a compilation unit has been converted to the lambda language, we scan it for annotated function definitions. When we find an annotation, we extract the function in several steps:

1. We check for mutual recursion. A single annotation can apply to several functions if they are defined together with the `and` keyword. For example,

   ```
   xfun f x = ... and g x = ...
   ```

   Functions defined together may be mutually recursive. If they are, they must be extracted together. Functions which are not mutually recursive should

be separated into their own compilation units so that when we transmit one, the transmissible representation does not contain other, unnecessary code.

Straightforward analysis of the lambda code shows exactly whether such functions are mutually recursive. In the first step, we check the bodies of functions defined together for references to each other. Using these references we build a directed graph with the functions as nodes and the references as edges and then find the strongly connected components. The members of each component correspond to the mutually recursive functions and are extracted together.

2. We find all the free variables in the function. In the lambda language, variables may represent either normal Facile values or they may represent structures. Structures are implemented as records in the lambda language, so a value in a structure is just a particular element of a record. For each variable, we look in the compilation environment and check whether the variable represents a structure, and if so, whether it is a ubiquitous or remote structure (this information is recorded when structures are compiled). We classify variables for ubiquitous or remote structures as *external.* We classify all other variables as *singular;* they are assumed only to be available on this site and their values must be transmitted with the function.

3. We make a *transmissible* version of the function. The transmissible version is built by wrapping the function with two custom linking functions. One of these takes as arguments, and thus binds, the singular free variables and the other binds the external free variables.

   On a recipient, applying the linking functions to the singular values transmitted with the function and to the appropriate remote and ubiquitous structures produces a record containing a pointer to the executable function closure. For normal functions, the record is a singleton. However, for mutually recursive functions, which in Facile share a single closure at run time, the record contains multiple pointers with different offsets into a single closure. Each pointer corresponds to one of the mutually recursive functions.

   As an example of how the linking functions are constructed, consider the following function (written in Facile source rather than lambda language for clarity):

```
xfun f x = x + a + b + C.c + D.d
```

Suppose that `a` and `b` are singular and that the structures `C` and `D` are external. Then the transmissible version would be

```
fn (a,b) => fn (C,D) =>
    let fun f x = x + a + b + C.c + D.d
    in (f)                    (* Record consisting of one element *)
    end
```

4. We make a version of the function for *local* execution. The local version is also built by wrapping it with another function, but this wrapper binds only the singular free variables in the function. If the function refers to any remote variables, it cannot be executed locally, and in this case we replace the body of the function with code to raise a special exception. Otherwise the body is unchanged. Continuing the previous example, if either of the external structures `C` or `D` is remote, we get:

   ```
   xfun wrapper (a,b) =
       let fun f x = raise NotLocal
       in (f)
       end
   ```

   If all of the external structures are instead ubiquitous, we get:

   ```
   xfun wrapper (a,b) =
       let fun f x = x + a + b + C.c + D.d
       in (f)
       end
   ```

   When the Facile compiler is given this compilation unit, it will place its own standard linkage wrapper around it. The standard wrapper will bind and resolve any ubiquitous variables (including the pervasives). The variables bound in this case will be `C` and `D`.

5. We patch the original compilation unit. The original unit needs a replacement for the function that has been extracted from it. We give it this replacement by inserting a call to the wrapper of the locally executable version of the function. The values of the singular free variables in the function are passed as arguments. The result of the call is a record containing the extracted function.

Suppose that our example function `f` originally appeared in the following fragment:

```
val a = 10 and b = 20
xfun f x = x + a + b + C.c + D.d
val e = f 5
```

At the lambda level, the definition of `f` appears as

```
val f = xfn x =>  x + a + b + C.c + D.d
```

The definition of `f` is replaced with code to call the wrapper with `a` and `b` as arguments:

```
val a = 10 and b = 20
val f = #1(wrapper(a,b))          (* Extract first element of result *)
val e = f 5
```

The `#1` is necessary to extract the first element of a record; recall that the wrappers place their results in records to handle the case of mutually recursive functions.

Note that because a definition for `f` still appears in the compilation unit, none of the code that uses `f` has to be changed, as the third line shows.

The wrapper functions are responsible for creating the closures that pair code with singular, remote, and ubiquitous values. In Section 4.3 we see how marshalling of closures is tied to the structure of the wrapper functions.

Each annotated function is extracted into its own stand-alone package. For an annotated function nested within another, the extraction proceeds recursively just as described above; no special actions must be taken. Functions without annotations inside an annotated function are not extracted. They remain part of its compilation unit.

4.2.2 Representations

Once a potentially transmissible function has been extracted into its own compilation unit, we generate transmissible representations for it. Representations differ in such characteristics as their conciseness, how high-level they are, whether they are

```
FIX(fact:
  FN(n, SWITCH n
     of 0 => t1 = RECORD()
               1
        _ => t2 = n
             APP(PRIM *, RECORD(t2, APP(fact,
                 APP(PRIM -, RECORD(t2,1))))))
IN  RECORD(fact))
```

Figure 4.1: The factorial function in lambda language. Function application and record creation are represented with the `APP` and `RECORD` constructors. For this simple function, the lambda language is very similar to the source.

interpreted or compiled, and what machine-specific optimizations a recipient can perform on them before execution. All of these characteristics affect performance.

Generating a machine-code representation is straightforward, but we also need architecture-independent representations. The Facile compiler already uses several such representations internally that vary over the preceding characteristics. Transmitting these internal representations is convenient because we can reuse the existing compilation machinery to generate them at the sender and to translate them at the recipient. There are two natural choices to use for transmission:

Lambda language. Because we do not create the stand-alone package for a function before it is converted to lambda language, this representation is the most high-level that we can transmit. The compiler uses no information about the target architecture in generating the lambda language representation. A recipient can therefore apply the full range of machine-specific optimizations in compiling it to machine code. Facile also contains a built-in interpreter for the lambda language. By using the interpreter, a recipient can start executing code without doing any translation.

Figure 4.1 shows the lambda language code for the following recursive factorial function:

```
fun fact 0 = 1 | fact n = n * fact(n-1)
```

```
fact(code,closure,n,kont,c1,c2,c3) =
   (L)fact'(n,kont,c1,c2,c3)

fact'(n,kont,c1,c2,c3) =
   if ineq((I)0,n) then
      n,kont,c1 -> c1_new
      -(n,(I)1) -> t1
      (L)fact'(t1,(L)domult,c1_new,c2,c3)
   else
      kont(kont,(I)1,c1,c2,c3)

domult(code,partial,c1,c2,c3) =
   c1.0 -> n
   *(n,partial) -> result
   c1.2 -> c1_orig
   c1.1 -> kont
   kont(kont,result,c1_orig,c2,c3)
```

Figure 4.2: The factorial function in final CPS. The parameters of the functions correspond directly to registers. In continuation-passing style, functions do not return; instead, they call their continuation (`kont`) with their result. The registers `c1`, `c2`, etc., are *callee-save registers.* These are used to pass values through to a continuation without saving them in a record on the heap. Note that `fact'` uses one of these to pass a record to its continuation `domult`, and must therefore save the original value in the record along with the information it wants to pass.

CPS. After lambda language, the compiler converts code into *continuation-passing style,* or CPS [Appel 1992]. Most of the optimization and code manipulation work in the compiler is done with this representation; each of the different phases generates a new form of CPS. The final CPS representation is then translated into machine code. Figure 4.2 shows the factorial function in its final CPS form.

The optimizations on CPS are parameterized by the number of general registers on the target machine. To preserve the architecture independence of CPS, we can set this number to some low, standard value while processing the transmissible version of a function. Our assumption is that every recipient

will have at least this many registers and will therefore be able to use the CPS representation.

As compared to the lambda language, the final CPS representation is produced after the completion of many more compilation steps. Run-time translation into machine code will therefore be faster. However, the code will not be optimized for the full number of registers on some machines. The trade-off is that translation will be faster, but execution might be slower.[1]

There is one problem with using CPS as an architecture-independent representation. Facile normally uses nine registers for special purposes and additional registers for general computation. However, the Intel 386 architecture, one of our targets, has a very limited number of general purpose registers available for both of these uses—only seven. To be able to run on the Intel, the Facile compiler sacrifices use of callee-save registers (refer to Figure 4.2) and simulates some registers on the stack. The elimination of callee-save registers makes CPS generated for the Intel incompatible with code (including normal, nontransmissible code) generated for any other architecture.

The problem cannot be solved by setting the number of registers on every architecture to a sufficiently low value while generating transmissible code. In that case, transmissible CPS generated on a Sparc, say, would be unable to call or be called by normal Sparc code because of the difference in callee-save registers. The only solution would be to make every architecture give up all use of callee-save registers, but this would hurt performance everywhere, not just with transmissible functions.

The final CPS representation is therefore not truly architecture independent. We can however use a CPS form from an earlier phase of the compilation before the choice is made to use callee-save registers or not. This *middle CPS* can be targeted towards the small number of registers available on the Intel while remaining architecture independent. Figure 4.3 shows the factorial example in this representation.

The Intel 386 is anomalous in having so few registers; modern architectures (including more recent Intel processors) can be expected to have more. By supporting final CPS, we can get a better picture of the performance of low-level, architecture-independent representations. The implementation therefore supports both middle and final CPS.

[1] [Appel 1992, p.189] notes one benchmark that executed *faster* in SML/NJ when the number of registers used was artificially restricted.

```
fact(n,kont) =
  fact'(n,kont)

fact'(n,kont) =
   if ineq((I)0,n) then
      domult(partial) =
         *(n,partial) -> t1
         kont(t1)
      -(n,(I)1) -> t2
      fact'(t2,domult)
   else
      kont((I)1)
```

Figure 4.3: The factorial function in middle CPS. Conversion to closure-passing style [see Appel 1992], introduction of callee-save registers, and register spilling have not been performed.

Finally, we also support transmission of optimized machine code. Such code can be executed immediately and at maximum speed by a recipient. We can send this representation along with an architecture-independent one for a function, allowing a recipient to proceed at once if it has the same architecture as the sender.

4.2.3 Storage

Once the transmissible representations for a function have been generated, they need to be associated with the locally executable code so that they may be found at run time. A normal function closure only contains a pointer to the machine code and the values for the free variables. The marshaller needs to be able to find the transmissible representations from this limited information.

We can take advantage of how Facile stores compiled machine code to solve this problem. In Facile, the machine code for all the functions in a compilation unit is kept in one long string, which allows relative addressing between functions. The individual functions are accessed by pointers into the string. During garbage collection, the collector must be able to find the start of the string given a pointer to its interior. This is made possible by *back pointers,* offsets to the front of

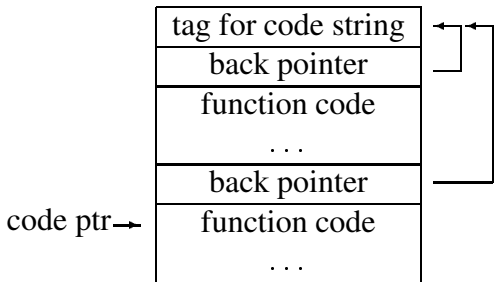

Figure 4.4: Layout of a compiled code string for a compilation unit. From the pointer to the code for the second function (shown left), the garbage collector can use the preceding back pointer to find the start of the code string.

the string stored just before each individual function. The collector follows the back pointers recursively until it reaches the tag marking the start of the string.[2] Figure 4.4 shows a sample code string with back pointers.

The algorithm used by the garbage collector allows us simply to prepend transmissible representations onto a code string and to add an additional back pointer so that the collector will skip over them. Nothing else need be changed. Figure 4.5 shows an example of such a modified code string.

With this storage strategy, marshalling now proceeds as follows. In the run-time system, closures are simply records. To distinguish that a record is a closure, we check its first element to see if it is a pointer, and if so, whether the pointed-to object is preceded by a back pointer (objects are always preceded by tags or back pointers, and the two are easily distinguishable). We then follow the chain of back pointers to find the transmissible representations stored at the start of the code string. If no representation corresponding to the current function is found, we raise an exception, signalling the presence of a nontransmissible function.

Note that it is not sufficient to check simply for the existence of transmissible representations in the code string. The function being marshalled could be a non-transmissible one that was nested inside a transmissible function, and is therefore in the same compilation unit.

[2] Full details of how the run-time system represents and manipulates code and other data are provided in [Appel 1990].

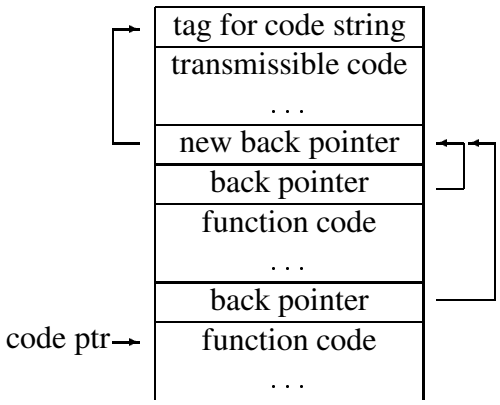

Figure 4.5: A code string prepended with transmissible representations. The existing back pointers (shown right) point to the previous start of the code string. A new back pointer at that location redirects the garbage collector to the new start of the string.

In Facile, compiled code strings contain no absolute addresses; only relative addressing is used, and linking is accomplished by applying the linking function wrapped around the compilation unit to the appropriate arguments. The lack of absolute addresses in code greatly simplifies the work of the garbage collector as it relocates code strings. We preserve this simplicity by using relative addressing to find transmissible representations from their associated functions.

4.3 Closures

The closure of a potentially transmissible function consists of a code pointer and some collection of singular and ubiquitous values. When the closure is marshalled, we must determine which values are singular and which are ubiquitous. The singular values should be transmitted with the function because they are only available at this site, while the ubiquitous ones should be trimmed away. There can be no remote values in the closure, because they do not exist locally.

At run time, ubiquitous values do not look any different from singular ones. To

distinguish them, we need a *closure map* generated by the compiler that identifies which values in a closure are which.

In principle we could generate closure maps when the compiler converts CPS code to use closures. However, this conversion occurs very late in the compilation process, and modifying it to generate closure maps would require carrying annotations through all the preceding phases and intermediate forms. Because variables are renamed through the compilation process, we would also need to track these changes so that variables could still be identified as ubiquitous or singular.

Fortunately, we can generate closure maps without making any of these modifications. Consider our earlier function where `a` and `b` are singular and `C` and `D` are ubiquitous:

```
xfun f x = x + a + b + C.c + D.d
```

In step 4 on page 56, we saw that after extracting `f` we create a locally executable version of it by wrapping it with a function that binds its singular free variables:

```
xfun wrapper (a,b) =
    let fun f x = x + a + b + C.c + D.d
    in (f)
    end
```

We call a wrapper that binds only singular free variables a *singular wrapper*. The singular wrapper is wrapped in turn with Facile's normal linkage function, which binds any remaining free variables. These remaining free variables (`C` and `D` in this case) are precisely the ubiquitous ones.

As the last step of compilation, the Facile compiler executes the standard linkage function, which looks up the variables it binds in the dynamic compilation environment. The result produced is a closure containing the singular wrapper and the ubiquitous values, e.g., `(<wrapper>,C,D)`.

The singular wrapper is intended to be applied to the singular values at run time, when it will return a closure for the target function containing both singular and ubiquitous values, e.g., `(<f>,a,D,b,C)`. Note that we do not know how the singular and ubiquitous values will be ordered in the closure; for our example, we have simply picked a random ordering. It is the wrapper that encodes where the singular values are placed in the closure for the target function. We can extract this encoding at compile time with the following trick:

1. We copy the closure for the compiled and linked singular wrapper. In the copy, we replace the values (which are all ubiquitous) with 0's. In our example, this gives us `(<wrapper>,0,0)`.

2. We apply the copy to a record consisting of 1, 2, ..., *n,* where *n* is the number of singular values. Normally the result would be the closure for the target function. However, now it will contain just a code pointer and the various integers. In our example, we would apply `wrapper` to the record `(1,2)`, where 1 becomes the value for `a` and 2 becomes the value for `b`. Given the previous ordering of elements in the result closure, we would then get `(<f>,1,0,2,0)`. The 0's represent the ubiquitous values, while the other integers are the singular values.

3. We scan the result closure and find the correlation between each nonzero value (the position of a singular value in the wrapper argument) with its position in the closure. This is the closure map. For our example, we see that `a`, represented by 1, can be found in position 1 in the closure, while `b`, represented by 2, can be found in position 3.

The closure map is stored with the transmissible code representations in the header of the compilation unit.

On marshalling we use the closure map to extract the singular values from the closure. The values are placed in the outgoing message in the order that they should be given to the wrapper for the transmissible version of the function (which is the same order as that used by the singular wrapper). We also place in the message the list of ids for remote and ubiquitous values, again in the proper order for the wrapper. This list can be generated at compile time, as its contents do not depend on any run-time information, and stored in the header of the compilation unit as well.

At the recipient, the list of ids can be instantiated with the proper remote and ubiquitous structures. These structures and the singular values sent in the message can be supplied to the wrapper once it has been made executable. The wrapper then produces the closure for the target function.

4.4 Run-Time Compilation

Before a function received from a channel can be used, its closure must be rebuilt to include any needed remote or ubiquitous values and its code must be made locally executable. If native code or an interpretable representation is available, we skip the latter operation. For other representations, however, we must compile the code.

Compilation can be a time-consuming process, and because the recipient must perform it at run time, it plays an important role in the performance of function transmission. We can potentially improve performance in two ways by *deferring* compilation until a function is called for the first time:

- If the function is never called, we avoid the cost of compilation altogether. This case may occur if the function is in a part of a larger agent that does not always get used or if the recipient will simply forward it elsewhere.
- For an agent consisting of multiple functions, deferral can reduce the latency between when an agent is received and when it starts executing, an important factor in interactivity. Functions are only compiled as their part of the agent is activated. Through deferral, the compilation latency is amortized over a greater portion of the agent's execution.

When the compilation does occur, we must consider the trade-offs in performing optimizations. Optimizing increases compilation time but may produce code that runs faster, so it may be desirable for code that runs for a long time.

As a final step, we must link the code with locally available values to resolve remote and ubiquitous references.

4.4.1 Laziness

Deferring compilation until a function is needed (i.e., making compilation lazy) can be done either explicitly or implicitly. If done explicitly, a function is received as a special object which the programmer must then pass to a compilation routine to produce an executable form. If done implicitly, a received function looks no different from a locally defined one and can be called normally. In the design, we opted for the implicit approach.

We implement implicit lazy compilation in Facile by creating a special *trigger closure* for each function when we receive it. The trigger closure encapsulates the transmissible version of a function. When the trigger is called, it compiles the function and overwrites itself with the new closure. By overwriting the closure, rather than creating a new one in memory, we ensure that any existing pointers to the closure remain valid. The details are as follows:

1. When a function closure is marshalled, a record representing it is placed in the outgoing message. The new record is larger than the original closure

record. All of its entries are null except for the last. The last entry points to a *transmission record* containing the function's code string, singular values extracted from the closure, ids for the remote and ubiquitous structures, and other information. The new record is large enough to later contain all the functions of the original closure record (normally only one, but more for mutually recursive functions), the singular, ubiquitous, and remote values used by the functions, and pointers to Facile's three libraries of pervasives, as well as the transmission record.

Thus, a marshalled closure record looks like

```
(null,null,...,null,<info>)
```

2. Closure records are detected on unmarshalling through a special tag. After copying the record out of the message, the unmarshaller replaces the null entries by pointers to the code string for the local trigger function. However, the last null pointer (the second-to-last entry in the record) is replaced by a pointer to the recipient's compilation function. The unmarshalled record looks like

   ```
   (<trigger>,<trigger>,...,<trigger>,<compiler>,<info>)
   ```

   Note that there will be more triggers than functions in the original closure. Pointers to trigger code are only needed at those locations in the original closure where there were function pointers. However, on unmarshalling it is convenient to fill in all but the last two slots with triggers; the extra triggers do no harm, and we are guaranteed that all function pointer locations have been properly filled.

   In Facile, a function value is really a pointer into a closure record. The location pointed to is itself a pointer to the function's code string. When the code is executed, it retrieves values it needs from the closure record using addressing relative to its own entry in the record.

   The trigger code is specially written so that it contains no free variables and thus requires no values from its closure. It is this property that allows pointers to the trigger code to be placed anywhere in the closure.

3. When the closure is applied to an argument and called, the trigger code scans forward until it finds the beginning of the record and the run-time tag with

the record's length. Using this information, it finds the second-to-last entry of the closure (the compilation function) and calls it. The trigger passes the argument it has been passed (which is intended for the target function) and the closure itself to the compilation function.

4. In Facile, multiple processes run within a single address space. Because more than one process could call an uncompiled function simultaneously, the compilation function first seizes a lock shared by all processes in the address space and checks if the target function has been compiled since the trigger was called. If not, the compilation function looks in the closure record passed to it and finds the transmission record in the last entry. It compiles the transmissible code string, producing the wrapper around the target function. It looks up the remote and ubiquitous structures required by the function, and then applies the wrapper to these structures and to the singular values transmitted with the function. The resulting closure is copied onto the initial trigger closure, which is large enough that the transmission record in the last field is left untouched (it will be used if the function is retransmitted).

   The compilation function releases the lock, calls the new executable closure with the argument originally passed to the trigger, and finishes by returning the result. On further calls to the closure, the target function will be executed directly; the triggers have been overwritten.

In addition to receiving function closures, we can also receive *wrapper closures* when functions contain nested transmissible functions. Consider the following code:

```
xfun g y =
    let xfun f x = x + y
    in 10 + f y
    end
```

When this code is compiled, the function `f` will be extracted into its own compilation unit separate from `g`. The body of `g` will be converted into the following (written in Facile source for clarity):

```
let val f = #1(wrapper (y))
in 10 + f y
end
```

The extraction process introduces the new free variable `wrapper` into `g`. Therefore `wrapper` will appear in `g`'s closure at run time. When `g` is transmitted, `wrapper` will be treated as a singular value and will be transmitted as well.

Wrapper closures that are transmitted as singular values require different treatment to ensure that lazy compilation is preserved. We handle them in the following way:

1. At compile time, we record the sizes of the back pointers that precede the code strings for the function and the function's wrapper. This information is stored along with the closure map and transmissible representations.

2. During marshalling, when a pointer leads us into a closure, we compare the size of the back pointer preceding the code we have been pointed to with the stored back pointers. We save the result of the comparison (i.e., whether we have found a wrapper or a normal function) in the transmission record. The stored back pointers also allow us to detect attempts to marshal nested, nonannotated functions that appear in an annotated function's compilation unit.

3. On unmarshalling we construct a closure for the wrapper consisting of the trigger function pointers, the compilation function, and the wrapper's transmission record, just as for normal functions. When the closure is called (from the body of `g` in our example), the trigger function is activated and calls the compilation function.

4. If the compilation function has been called on a wrapper closure (as determined from the information in the transmission record), we want to defer compilation further. The reason is that the wrapper and the functions it wraps are in the same code string, so compiling the wrapper implies compiling the functions. We do not know if the functions will actually be called, even though the wrapper is being called. For example, suppose the body of `g` were

   ```
   let val f = #1(wrapper (y))
   in if someCondition then f y else y
   end
   ```

   Here it is clear that calling the wrapper does not imply that `f` will be called.

We therefore overwrite the trigger entries in the wrapper's closure with pointers to the code for a general *substitute wrapper*. Like the trigger function, the substitute wrapper contains no free variables, allowing a pointer to its code to be placed anywhere in a closure. We overwrite the entry for the compilation function with a record containing information for the substitute wrapper to use.

We now have an executable closure for the wrapper. The compilation function will call this closure with the argument passed to the trigger and return the result. For subsequent calls (e.g., when the body of `g` is executed again), the compilation function will not be involved; the substitute wrapper function will be called directly.

5. When the substitute wrapper is called, its argument consists of the singular values needed to create the target function closure (in our example, the argument is `y`). It is still not known at this point whether the target function will be called; at this point we are only being asked to create its closure. Therefore, the substitute wrapper creates a closure record for the function, but it fills it with the normal trigger function pointers, the compilation function, and the transmission record. In the transmission record it updates the field for singular values with the argument it has been passed. We have still not compiled the function.

6. When a closure created by the substitute wrapper is called, the same actions occur as with lazy compilation of unnested functions. The trigger launches the compilation function, which overwrites the closure and calls the function.

   There is one further subtlety. As explained so far, the substitute wrapper creates a new closure each time it is called. When one of those closures is called for the first time, it will compile the transmissible code string and overwrite the closure. However, it is only necessary to compile the code string once for all the generated closures. Therefore, we have the substitute wrapper place a reference cell containing the potentially compiled code string in each closure in generates (i.e., all the generated closures have pointers to this cell). The compilation function always checks the cell first and retrieves the already compiled code if it exists. If not, the cell is updated with the newly compiled code.

The preceding strategy works for arbitrary levels of nested functions.

A function or wrapper can be retransmitted from a recipient before or after it has been compiled. In any of these cases, all that we need to do is find the transmission record stored at the end of the closure. However, we must be able to recognize that the closure is transmissible. For the case of compiled functions, we have the compiler prepend an extra backpointer and a special marker to the code string. The marshaller can identify these in the same way as transmissible functions originating on the site (Section 4.2.3). To handle functions that have not been compiled and to handle wrappers, the marshaller checks specifically for the code pointers of the trigger function and the substitute wrapper.

4.4.2 Compilation and Linking

Depending on the transmissible code representations available for a function, it may be interpreted or compiled with various levels of optimizations. In general, we expect that the more time that is spent converting the code into an executable form (e.g., by applying more optimizations), the faster it will execute. The exception is for native machine code, which requires no conversion but executes at maximum speed.

In the implementation we provide a number of flags that the recipient can use to control what compiler optimizations are applied and whether to interpret. We can use these to examine the trade-offs for each compiler setting.

After compilation, a function must be linked by applying its wrapper to the singular values sent with it and to the ubiquitous and remote values from the local environment. Each id for a ubiquitous or remote value is compared to the list of ids for locally defined ubiquitous structures. If a needed id is not found, the compilation function raises the special `NotLocal` exception. For the programmer, it appears as if the received function raises this exception when called.

The recipient can take whatever action it deems appropriate if an agent raises the `NotLocal` exception. For example, it could send a message back to the sender, send the agent elsewhere for execution, or simply discard it. A properly structured agent can even catch the exception itself, as the following shows:

```
xfun doWork () = ...agent code that might not link...
xfun mainAgent () = doWork() handle NotLocal => ...recovery...
```

For this example to work, `mainAgent` must not require any values from the local environment that might not be available, otherwise it could raise `NotLocal`

itself. Once the failure of `doWork` to link has been detected, `mainAgent` might send a message back to the sender over a channel brought with it.

4.5 Summary

The conversion of Facile into an agent programming language is based on support for transmissible representations, typed access to remote and ubiquitous values, and lazy compilation. In this chapter we have seen the primary methods for implementing these features. To summarize:

Signature ids. Signatures serve as the keys that identify remote and ubiquitous values. The drawback is that signatures are not very concise. To avoid the overhead of storing and sending full signatures with transmissible functions, we associate globally unique ids with each signature. When a signature is compiled an id is associated with it, and the compiled signature is then exchanged with other sites that need it. We use a globally known "signature server" for this purpose, though any form of file exchange would be sufficient.

Stand-alone function packages. To make an annotated function transmissible, we extract it into its own compilation unit. The free variables in the function are bound through wrappers that serve as the means for linking the function either locally or at another site.

Transmissible representations. To allow evaluation of the trade-offs in different architecture-independent code representations, we support lambda language (relatively high-level) and two forms of CPS (relatively low-level). These representations can be optimized to a recipient's architecture to varying degrees. The lambda language can also be interpreted. We support transmission of native code as well.

Storage of representations. A function's transmissible representations are generated at compile time but must be found at run time. By storing the representations in part of the function's code string, we can find them at run time using relative addressing. This technique sidesteps problems that might arise because of code relocation during garbage collection, for example.

Closure maps. Closure trimming is straightforward once the locations of singular and ubiquitous values in a function's closure are known. The closure is created by the wrappers around the function. By running the wrappers on specially constructed dummy arguments at compile time, we can generate a map of where the different values lie.

Triggers and substitute wrappers. For implicit lazy compilation we create temporary trigger closures for functions when we unmarshal them. When a trigger closure is executed, it compiles its function and overwrites itself with the result. Overwriting ensures that any references to the function will remain valid and will use the newly compiled code. Nested transmissible functions complicate the approach but it remains fundamentally the same; we use substitute wrappers as an extra level of indirection before creating trigger closures.

Although these methods were developed for the Facile agent language implementation, they represent solutions to problems that are likely to be encountered in other systems with similar design goals. We have not proved the correctness of the implementation, but it has been subjected to significant testing, as we see in the next chapter.

Chapter 5

Agent Applications

Our goal in the design and implementation has been to satisfy all of the agent language properties described in Chapter 2. Our implicit assumption is that these properties are the right ones; in satisfying them, we produce a language well-suited for agent programming.

To validate our assumption, we need experimental evidence that agent applications can be constructed simply and naturally in our extended Facile. By building real applications, we can learn if the language provides the necessary features and abstractions. Real applications also allow us to test the performance of the implementation.

In this chapter we explore several applications that share a common framework known as *mobile service agents* (MSA). For each application, we examine the crucial code involved in establishing communication links and transmitting and executing agents. Our purpose in doing so is twofold: We gather subjective evidence of whether Facile's primitives make agent programming easy, and we see how agent applications can be structured. We defer performance issues until Chapter 6.

We start by introducing the mobile service agents framework in Section 5.1. In Section 5.2 we then examine the components of the framework, following in Section 5.3 with a more detailed look at the implementation of specific services. We summarize the chapter in Section 5.4.

5.1 An Overview of Mobile Service Agents

In the first chapter we saw how mobile agents can be used as a way to provide services. Agents allow parts (or all) of a service to be downloaded to a client. Using agents, we can ease loads on servers while also taking advantage of local resources. We can hide the effects of slow or broken communication links by decreasing or eliminating communication needs. We can also use agents to project a custom interface for a service onto a client's machine.

In the MSA framework, services are based on agents. When a user accesses a service, an agent that represents the service is downloaded to the user's computer and starts running there. No special software is needed on the user's side other than the run-time system for receiving, compiling, and executing code, which remains the same for all agents and services. Requiring only one piece of software to access multiple services is a principle that also underlies the World Wide Web: With one browser, a user can view any page. The difference with agents, of course, is that they can carry out actions not previously encoded in the browser.

The MSA framework allows collections of agent-based services to be provided by different administrative *domains.* Like a Web site, a domain might be a city, a company, or a university department. Rather than a home page, a domain offers a *directory service* that provides access to other services on that domain. Each service, including the directory, is represented by an agent. As services are accessed, their agents are automatically downloaded to the user's machine.

I developed a design for the MSA framework that specifies how the directory and other service agents are requested from the client, transferred to it, and then executed. Other members of the Facile project and I then conceived several different MSA services. These were implemented by one member, Pierre-Yves Chevalier, to construct a full demonstration where a user with a portable computer travels between two domains, each providing various services and running on a different machine architecture. This demonstration has been successfully presented in and outside of ECRC since November 1994.

5.2 Design of the MSA Framework

There are three key steps involved in providing a user with access to domain services:

1. The services on the domain register with the directory service.

2. The user accesses the directory service and receives its agent via a generic program not specific to any one domain.

3. The directory service, under control from the user, accesses other services and receives their agents.

We will concentrate on the second step; the first step is straightforward, and the third can be performed in the same way as the second.

Each service on the domain has a primary server. When the directory server is started, it registers the domain name and two contact channels with the MSA name server. (In the current implementation of the MSA framework, there is only one name server that must be used by all domains.) One of these channels is intended for service requests from clients. The other is used to receive registrations from other servers on the domain. As each server starts, it finds the registration channel for the directory and sends to it the server's own service request channel.

For a user to access domains, she must have the Facile run-time system and a small program known as the *connection bootstrapper*. When the user starts the run-time system on her machine, the bootstrapper runs automatically. The user can specify the name of a domain to which she wishes to connect, and the bootstrapper will access that directory service.

The first step taken by the bootstrapper is to look up the service request channel for the directory on the specified domain. To signal a request for service, it sends over the channel a new, locally created channel on which it then waits. The directory server receives this new channel and sends through it a directory agent containing all the service request channels for the domain. The bootstrapper receives the agent from the channel, spawns it, and quits. The directory agent can now interact directly with the user.

The Facile code to accomplish the bootstrapping, only slightly simplified from the actual implementation, is shown in Figure 5.1. Given the name of a domain (`domainName` in the figure), the bootstrapper queries the name server for the service request channel of that domain's directory server (lines 3 and 4). The bootstrapper creates the new channel on which to receive the directory agent, which it sends to the directory server on line 6.

The directory server on the domain side is in a continuous loop in which it receives channels sent to it and sends the directory agent back over them (lines 12 and 13). After the directory server processes its request, the connection bootstrapper receives the directory agent in line 7 and finishes by spawning it in a new thread. Figure 5.2 shows how such an agent may appear to the user.

```
 1  (* Connection bootstrapper (client side) *)
 2  let
 3      val dirServiceChan : scr channel channel =
 4          FacileExt.find(domainName)
 5      val dirAgentChan : scr channel = channel()
 6      val _ = send(dirServiceChan, dirAgentChan)
 7      val dirAgent = receive dirAgentChan
 8  in
 9     spawn dirAgent
10  end
11  (* Directory server (domain side) *)
12  while true do
13      send(receive dirServiceChan, dirAgent)
```

Figure 5.1: Bootstrapping the connection to a directory service. The first code fragment runs on the user's machine, while the second runs on the domain's.

The directory agent in the bootstrapping code is transmitted as a *process script* rather than as a function (`scr` is the type of scripts). The reason is that the `spawn` function, which forks execution of its argument in a new thread, only accepts scripts. The distinction is a minor point, however, because any function can be readily converted into a script. To create an equivalent script for a function `f` of type `unit` $\rightarrow$ `'a`, we use the Facile idiom `script(f();terminate)`.[1] Both `script` and `terminate` are keywords. For our purposes, the argument of `script` is a series of expressions that are executed in order when the script is spawned, finishing with the special `terminate` action. The result of `script` is a value of type `scr`. Though it looks like a function, `script` is a syntactic construct. The distinction between scripts and functions completely disappears during compilation, when all scripts are represented as potentially transmissible functions.

The directory agent retrieves and launches service agents using code almost identical to that of the connection bootstrapper. The only difference is that it does not have to query the name server for the contact channels, because it has these

[1] The use of this idiom by the directory server to create its agent is not shown.

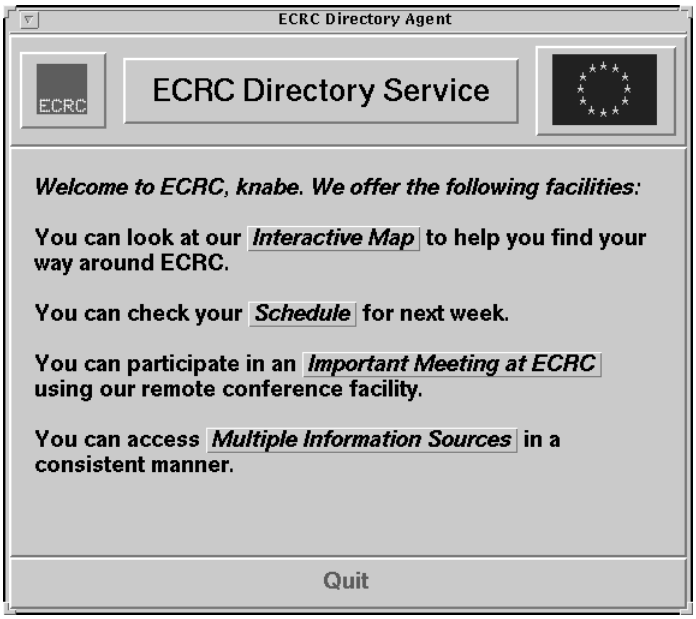

Figure 5.2: The user interface of a directory agent.

already in its closure.

5.3 MSA Applications

Although services in the MSA framework are always accessed via downloaded agents, there are no further constraints on how they may be structured. Once the agent is on the client, the service can create any network of communicating processes that it desires.

Because MSA applications may be accessed by a user with a portable computer, an important discriminant between different services is whether their agents require communication with their servers after they have been downloaded. When the user has a high-speed, low-latency connection to the domain, these differences may not matter. However, if the link is poor (e.g., via cellular telephone) or nonexistent, the communication requirements of agents can determine whether services remain

usable or not.

The MSA applications that have been implemented can be classified into three categories based on their communication needs:

No communication. The downloaded agent does not communicate with the domain.

Required communication. The downloaded agent must communicate with its domain to provide the service.

Preferred communication. The downloaded agent normally communicates with the domain to provide the service. However, if the communication link is broken, service degrades gracefully until it is reestablished.

We consider examples from each of these categories in turn and then examine how the implementations of the MSA applications use our extensions to Facile.

5.3.1 No Communication

The motivation for the *interactive map* service is to help a user who visits a domain in person to navigate through unfamiliar surroundings. An interactive map allows the user to type in a nearby room number and to see where she is, to find the office of a particular person, and to locate resources such as printers. Figure 5.3 shows a snapshot from a map service in use.

The information provided by an interactive map changes slowly compared to the expected time any one visitor will use it. In addition, the information set (consisting of building geography and office assignments) is small enough to download. For these reasons it is possible to encapsulate completely the information provided by the service into the service's agent. We then remove the need for the agent to communicate with its domain. The advantage is that the service can be used without a network connection, making it suitable for a visitor with a portable computer.

The agent is implemented as a set of annotated functions collected into a single module. One function is the entry point to the agent, and it is this function that the map server transmits in response to requests. When the function is marshalled, all the other functions making up the agent are marshalled as well.

Because Facile lacks a graphical user interface library, the GUI for the map agent is programmed in Tcl/Tk [Ousterhout 1994]. The agent carries with it a

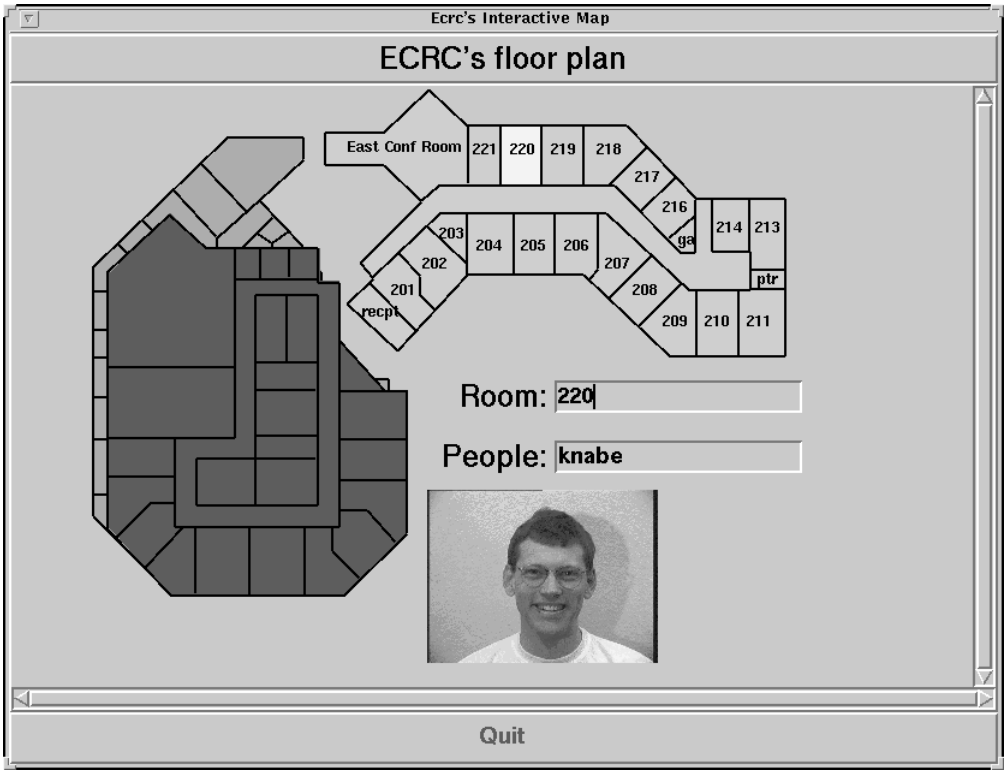

Figure 5.3: An interactive map agent.

Tcl/Tk code string for the interface. On the client, the agent uses Facile's Tcl/Tk connection library to execute and communicate with the interface.

In essence, the map agent wraps a static set of information with a custom access method. The same model can be used for other types of information. For example, a railway schedule could be coupled with route-planning software in an agent. The agent could be downloaded from a domain, used during a trip, and then discarded.

5.3.2 Required Communication

The *directory* is an example of a service where communication between the agent and the domain is required. Note, though, that the directory agent does not communicate with its own server. When a user requests a service, the agent uses the channels it contains to communicate directly with the corresponding service. It then receives and launches the appropriate agent as described in Section 5.2.

The *file system browser* is another service that must communicate with the domain, but it has a different communication structure from the directory. This simple service was originally developed as the "hello, world" program for mobile service agents. It allows the user to list directories on the domain-side server's file system. Though the action that the service performs is simple, its communication architecture is interesting.

For each request that the browser server receives, it creates a new agent and a new server dedicated to that agent. The agent is sent to the client, while its dedicated server is spawned in a new thread. Facile's very lightweight threads make it feasible to create a new thread for every client. The resulting agent–server relationship is one-to-one, which is simpler to program than many-to-one. Moreover, the main server is insulated from the connection between the client and the domain. If this connection should be lost just as the dedicated server tries to send a listing to the client, it is the dedicated server that will block (and eventually time out), not the main server. The main server can therefore be more responsive to new service requests.

A client requests service when its directory agent sends a newly created channel to the browser server. Figure 5.4 shows the code executed by the main server in response (`replyChannel` is assumed to be the channel received from the client). Two channels, `inc` and `outc`, are created for use as the communication links between the agent and its new, dedicated server. A script that calls the main function for the browser agent (`browserAgent`) with the channels as arguments is then created and sent to the client on `replyChannel`. A second script that calls the main function for the dedicated server (`browserServer`), again with the channels as arguments, is spawned locally. The channels are automatically shared between the scripts; in the run-time representations, the channels appear in each script's closure.

The code shown has a flaw: If the connection to the client is lost while this code executes, the main server could be blocked. We could insulate ourselves from this risk by spawning yet another process that simply performs the send

```
let
    val inc = channel ()
    val outc = channel ()
in
    (* Send the file browser agent to the client *)
    send (replyChannel,
          script (browserAgent(inc, outc);
                  terminate));
    (* Start the file browser for this agent *)
    spawn(script (browserServer(inc, outc);
                  terminate));
end
```

Figure 5.4: Establishing the file system browser service.

through `replyChannel`, or by timing out on the send (Facile provides versions of `send` and `receive` that take a maximum-time-to-wait parameter).

For both the directory and file browser agents, if the connection to the domain is lost while they are running, the agents can no longer provide their services. This situation is handled gracefully: Communication attempts time out, and the agents remain running. The user can shut them down explicitly or wait until the communication link has been reestablished to repeat her request.

5.3.3 Preferred Communication

If a service agent provides information from its domain that changes during the agent's lifetime, the agent needs a network connection to receive the updates. For some services, such as a stock ticker, a continuous network connection allowing immediate updates may be necessary for the service to be useful. For other services, however, intermittent connections allowing occasional updates may suffice.

Services that only need occasional updates are well-suited for use on a portable computer that may often be disconnected from the network. A service agent can use the latest version of the information that it has, receiving updates when possible. For the highest quality service and the most up-to-date information, communication is *preferred;* however, it is not critical to providing service.

We describe two services based on preferred communication: a dynamic conference schedule and a domain monitoring tool.

Conference Schedule

The purpose of the *conference schedule service* is to provide users with a shared schedule of events that may be updated dynamically. During a conference, information such as the times or locations of events change frequently. The service allows a user to make changes to the schedule that will then be propagated to other attendees whenever they connect to the network.

On initially accessing the schedule service, the user receives an agent with which she can browse the schedule even after disconnecting. When the network connection is reestablished, the agent automatically receives updates to the schedule that have been made since the last connection and displays an alert. The user can also make her own updates to the schedule while connected or disconnected, and these will be communicated to the central server when possible.[2] A snapshot of a schedule agent is shown in Figure 5.5.

The implementation of the conference schedule service is more complex than the others, primarily because of the need to handle schedule updates. The server waits for service requests on the channel previously registered with the directory. At the same time, it waits on another channel for any updates sent from clients. To perform two waits simultaneously, we use Facile's *alternative* construct:

```
alternative
    [recvguard(serviceChannel, handleServiceRequest),
     recvguard(serverUpdateChannel, handleUpdate)]
```

The alternative allows several different actions to be guarded by either send or receive attempts with only one succeeding (if more than one guard is true, the choice is nondeterministic). The function `alternative` takes a list of guard values as its arguments. The guard values are created with the `sendguard` and `recvguard` functions. This code fragment only uses `recvguard`, which takes a channel to try to receive on and a function to call with the received value.

[2] We do not currently take any special actions to resolve conflicts. For example, if an entry on the central server has been updated while a user has been disconnected, and the user has also updated that entry, the user's update is simply sent through on reconnection. It would not be difficult to change this behavior so that the user is first shown the server's update and queried whether her update should still be sent. However, further difficulties remain (e.g., races between users).

Figure 5.5: A conference schedule agent.

Depending on whether it is possible to receive first on `serviceChannel` or on `serverUpdateChannel`, the functions `handleServiceRequest` or `handleUpdate` will be called, respectively.

When a service request is received, the server creates a communication chain between itself and the client. The chain consists of the service agent, which is sent to the client in response to the request, a *proxy process,* and the server, all linked by two new channels. The proxy is a newly created process dedicated to its particular agent. Like the dedicated server of the file browser service, the proxy insulates the main server from a lost connection to the agent. In addition, the proxy stores schedule updates distributed by the server. Whenever it can, the proxy forwards

```
let
    val serverToProxyChannel = channel()
    val proxyToAgentChannel = channel()
in
    send(replyChannel,
         script (scheduleAgent(serverUpdateChannel,
                               proxyToAgentChannel,
                               eventList);
                 terminate));
    spawn(script (scheduleProxy(proxyToAgentChannel,
                                serverToProxyChannel);
                  terminate));
    proxies := serverToProxyChannel :: (!proxies)
end
```

Figure 5.6: Establishing the conference schedule service. The server, proxy, and agent are linked together by two new channels. The agent also has a direct channel to the server in `serverUpdateChannel`.

these updates to the agent, which can then incorporate them into its copy of the schedule.

The code to create the communication chain is shown in Figure 5.6. The first action is to send the new agent script to the client over `replyChannel`, which once again is the channel created by the directory agent on the client and sent to the server as the service request (we do not show the receipt of the service request). The free variable `eventList` is assumed to contain the current version of the schedule. Because `eventList` is referenced from within the script, it will appear in the agent's closure and will be transmitted with it. The closure will also contain several other values: `serverUpdateChannel` is assumed to be a channel shared by all schedule agents that the agent can use to send back schedule changes made by the user, and `proxyToAgentChannel` is the newly created channel from which it will receive updates forwarded by the proxy.

Once the agent has been sent, the server spawns the proxy for it. The proxy uses `proxyToAgentChannel` to forward updates to the agent and `serverToProxyChannel` to receive them from the server. The server stores

this latter channel in the `proxies` list.

To distribute an update, the server sends it down all the channels in its `proxies` list. Each proxy maintains a queue of updates to be sent to its agent when possible. If its queue is empty (no updates are pending), a proxy simply waits to receive and enqueue an update from the server. If updates are pending, the proxy simultaneously waits on the server and attempts to send the update at the head of the queue to the agent. These two communication attempts are programmed with an alternative, which ensures that the proxy will always be ready to receive communications from the server and will not block it:

```
if updatesPending() then
    alternative
        [recvguard(serverToProxyChannel, enqueueUpdate),
         sendguard(proxyToAgentChannel, getUpdate(),
                   dequeueUpdate)]
else
    enqueueUpdate(receive serverToProxyChannel)
```

The `sendguard` function is used to construct a guard that attempts to send the value in the second argument (here the result of `getUpdate`) through the channel, calling the function specified in the third argument on success. If the agent is connected to the network and the send guard succeeds, the successfully distributed update is removed from the queue.

The agent itself consists of three processes. One of these waits for updates to arrive from the proxy and modifies the local copy of the schedule appropriately. Another handles changes made to the schedule by the user of the agent. These changes are passed to the third process, which like the proxy maintains a queue of updates to send up to the server when it is reachable. The details are similar to the domain-side code shown already.

Agent Tracking System

The *agent tracking system* is a monitoring tool for the MSA framework. It provides a graphical view of domains with their servers, clients with their agents, service requests and other communication, and agent downloading. Figure 5.7 shows it in use. Because the tracking system exists outside the MSA framework, it is structured quite differently from the other services. Like the schedule service, though, it uses the preferred communication model for information updates.

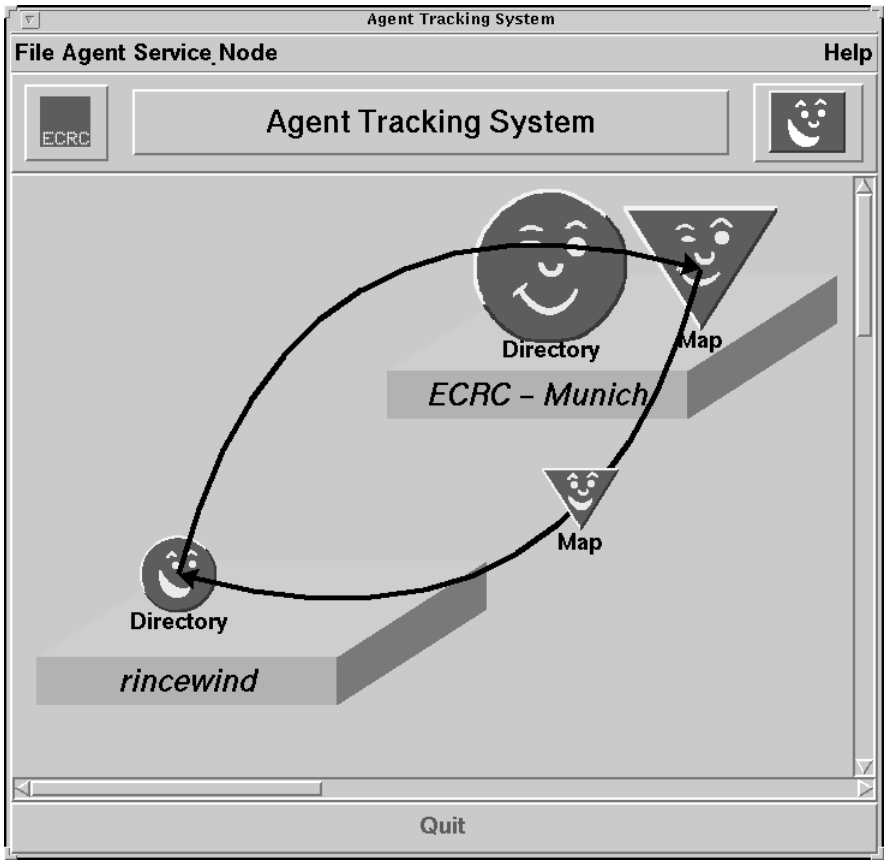

Figure 5.7: The agent tracking system in use. The snapshot shows one domain, *ECRC – Munich,* with one client, *rincewind.* The large shapes represent servers on the domain; the small ones are their agents. The directory agent on the client has requested service from the map server, as shown by the upper arrow. The small triangle representing the map agent moves on the screen from the domain to the client along the lower arrow. The arrows disappear after it reaches the client.

The tracking system consists of a stand-alone *monitor* that receives and displays information from the servers and agents in various domains. The servers and agents must be instrumented by hand to send this information. The instrumentation consists of calls to certain tracking system functions inserted at key points in the code (e.g., when contacting a server or dispatching an agent). These functions are transmissible and get sent along with an agent just like other referenced functions.

At start-up, the monitor registers a single contact channel with the name server. As each instrumented agent or server starts running, it calls a tracking system initialization function that looks up the contact channel on the name server. All subsequent calls to tracking system functions use this channel to send information to the monitor.

After obtaining the contact channel, the agent or server sends a message to register itself with the tracking system, supplying its name, associated service, current site, etc. The tracking system records this information and sends back a unique identifier that the agent or server then uses to tag all further messages. The tags on messages allow the monitor to distinguish between multiple simultaneous events.

A sample tracking function called by an agent is `CallMySvr`, which the agent uses to indicate that it is about to call its associated server. The body of this function is

```
if !ATSisUp = false then ()
else timeout_send(!WatchChn, CALL_SVR(my_id), 2)
     handle EXN_TIMEOUT => ()
```

The free reference variable `ATSisUp` is shared by all the tracking functions; it is set to true if the initialization function succeeds in obtaining the contact channel for the monitor.[3] The transmission is performed with `timeout_send`, a variant of `send` which takes an extra argument specifying the milliseconds to wait before raising the exception `EXN_TIMEOUT`. The reference variable `WatchChn` contains the contact channel for the monitor, and `CALL_SVR` is a datatype constructor that is applied to the id of the agent.

To avoid blocking a server or agent, the tracking system functions time out quickly. Normally these time-outs only occur with agents that are disconnected from the network. Communications that time out are not repeated; the information is lost. The monitor cannot update its information about disconnected agents until

[3] Note that `!` is the dereferencing operator in Facile, not boolean negation.

they reconnect and perform an action. For an agent with a proxy, the proxy may detect that the agent is unreachable and notify the monitor, which then displays the agent with a special icon.

Although both the agent tracking system and the conference schedule service are based on preferred communication, the two differ in how they handle loss of communication. The schedule service delays the transmission of information until communication is possible, while the tracking system discards the information after time-outs. Because the purpose of the tracking system is to provide a real-time view of the MSA framework, there is no use in retaining old information for later transmission.

5.3.4 A Sample Implementation

In this section we sketch out some of the details of how an MSA application is implemented. Our purpose is to show how the Facile language extensions are used in agent programming.

We will use the file system browser as the example application because of its relative simplicity. The browser is implemented in two structures, one for the server and one for the agent. We will concentrate on the latter.

The agent structure defines several functions and values, but only one function, `browserAgent`, is externally accessible. This function is the agent's entry point. It is shown in Figure 5.8 in a stripped-down form. The function accepts two channels as arguments, as well as several other values that are not shown (the function will be applied to these values in the script transmitted by the server). Its first action is to start a Tcl/Tk interpreter by executing a shell command. It then calls the internal function `pickup`, which repeatedly attempts to establish contact with the new interpreter through the library function `lconnect`. The last argument to `lconnect` is a callback function for strings received from the Tcl/Tk interpreter.

After `pickup` returns, the function `Create` is called. This function sends the Tcl/Tk user interface code sent with the agent to the interpreter. Once this action has been accomplished, `browserAgent` exits. The other functions in the agent will all be called via requests passed from the user interface to the agent's callback function.

The function `browserAgent` contains several free identifiers. Of these identifiers, `ConnectFailed`, `HandleClbk`, `TclSvr`, and `Create` are all defined elsewhere in the agent's structure, and their values are transmitted along

```
xfun browserAgent (BrwChn, AgtChn) =
    let fun pickup (n) =
        if n > 10 then raise ConnectFailed
        else (Threads.sleep(10);
              TclSvr := TkConnect.lconnect(...,
                  fn s => HandleClbk(BrwChn, AgtChn, s)))
             handle _ => pickup(n+1)
    in
        System.system("tcltk ...");
        pickup(0) handle ConnectFailed => ...;
        case !TclSvr of
            SOME(theSvr) => Create(theSvr, ...)
          | NONE => ...
    end
```

Figure 5.8: Entry point to the file system browser agent.

with `browserAgent`. The structures `System` and `Threads` are part of the pervasives and their members are treated automatically as ubiquitous values. The structure `TkConnect` is a ubiquitous library developed for the MSA applications. The file system browser includes the following code for it (outside of the agent and server structures):

```
signature TK_CONNECT = demanding signature TK_CONNECT

xstructure TkConnect : TK_CONNECT =
  struct
    ... Complete implementation ...
  end
```

`TkConnect` does not need to be ubiquitous because it is not used on the server side where the agent is defined. A remote definition would be sufficient. However, the preceding code is in a single file used by each of the MSA applications as well as the bootstrapper, so it is more convenient to treat it as ubiquitous.

The code for the agent's callback function is shown in Figure 5.9. Like `browserAgent`, this function is annotated (there are a total of seven annotated functions in the agent structure, all at the top level). The first `case`

```
xfun HandleClbk (BrwChn, AgtChn, theArg) =
    case !TclSvr of
        SOME theSvr =>
            let val Cmd::Args =
                StringUtil.tokenize (...,theArg)
            in case Cmd of
                "Quit" => Destroy(theSvr,BrwChn)
              | "List" =>
                    (send(BrwChn,SOME(hd Args));
                     case receive AgtChn of
                         ...)
              | ...
            end
        NONE => ...
```

Figure 5.9: The callback function for the file system browser agent.

statement checks that the reference variable `TclSvr` has been initialized by `browserAgent`. The argument string `theArg` received from the Tcl/Tk interpreter is then parsed and a central `case` statement is entered, with different actions being taken depending on the command selected by the user. For example, if the user wants a listing of a particular directory, the agent uses `send` to transmit the directory name through `BrwChn` back to the server, does a `receive` on `AgtChn` to await the results, and then processes them appropriately.

Of the free names in this extract of the callback function, `TclSvr` and `Destroy` are part of the agent. `TclSvr` is a reference variable defined at the top of the agent's structure with

```
val TclSvr : TkConnect.con option ref = ref NONE
```

When the agent is transmitted, a copy of `TclSvr` is sent along which remains shared by all the functions in the agent that refer to it. `Destroy` is one of several functions called by `HandleClbk` to carry out the agent's actions (the calls to the other functions are not shown). These functions are all annotated and at the top level of the structure. `HandleClbk` also refers to the structure `StringUtil`, which is a ubiquitous library like `TkConnect`.

5.4 Evaluation

We can use the implementation experience from the preceding applications to evaluate our language modifications to Facile. The two additions we made were annotations and remote and ubiquitous structures.

For Chevalier, the main MSA implementor, following the rules for potentially transmissible functions and annotation were straightforward. In each of his applications, he gathered all the functions making up an agent into a single structure. Only the top-level functions in the structure then had to be annotated, and keeping track of these annotations was relatively easy.

However, we have also seen sample agents written by other team members where every function is annotated, even though many functions are nested and do not need annotations. There are two penalties from these extra annotations. Storage requirements are somewhat increased because more functions now have wrappers, closure maps, and other marshalling information associated with them. Execution times are also increased because the extra annotations mean that the recipient must compile and link each function separately and perform more closure overwriting operations.

Despite these errors, we have not seen the mistake of annotating functions that will not be transmitted at all. Programmers maintain a distinction between the mobile parts of their code and the static parts. Thus our main goal for introducing annotations continues to be satisfied: Transmissible representations are generated only for agents, with the result of significant savings in space.

The MSA agents make extensive use of ubiquitous values. Most of these values are pervasives that are ubiquitous by default and therefore require no action from the programmer to be treated as such. Considering the frequency of the pervasives in the agent applications, we believe this default is a good choice.

In addition to using the pervasives, Chevalier also built several ubiquitous libraries specifically for the MSA framework. These libraries provide functions for interacting with Tcl/Tk, which was used to build the user interfaces for the applications. The servers and bootstrap program were all constructed using the same implementation of these libraries. The only weakness seen with the ubiquitous structures was in the exchange of compiled signatures. Using the `demanding` and `supplying` constructs and the signature server for this purpose made compilation clumsy. Saving the signatures to disk and then exchanging the resulting files would have been simpler.

Though they are not directly the subject of this dissertation, the MSA frame-

work and the original Facile language can also be evaluated using the agent applications. Not surprisingly for an initial experiment into a new area, some weaknesses exist from which we can draw lessons for future work.

One general weakness can arise from the very flexibility of mobile service agents. While navigating domains, the user is deluged with new software, much of which is intended for short, one-time use. In this setting it is easy to overwhelm the user with too many things to learn. Standard interfaces that make it immediately obvious how to use an agent are an important need. Conceptually simple services should be simple to learn and use.

Persistence would be a useful feature for agents. The need for persistence is most obvious when using a portable computer: As soon as the computer is powered off, any agents running on it are lost. An easy fix for this problem in the MSA framework would be to modify the connection bootstrapper so that on start-up, it retrieves saved agents from disk and spawns them. Persistence could than be added on a per-service basis by having agents periodically save "state-preserving" agents to disk in marshalled form. However, it would be up to the service programmer to determine what structure these state-preserving agents should have and what recovery actions they should take. Adding a more general notion of persistent processes to Facile might be a preferable solution.

A running Facile system cannot change the name server it uses, so to allow clients to access multiple domains, all domains must use a single name server. Using a single name server does not scale. Moreover, it does not allow domains to be completely independent administrative entities. Facile should be extended so that IP addresses and port numbers can be used to specify a name server.

The implementation of channels in Facile assumes that all communicating participants have fixed IP addresses. In mobile computing, this assumption does not hold; when a portable computer is reconnected to the network, it may be assigned a new IP address. If the IP address changes, Facile can no longer transparently reconnect broken channels. For the MSA demonstration the implementation was extended to handle reconnections within a single subnet. The long-term solution is to await the widespread adoption of a mobile IP standard on the Internet; in the meantime, a level of indirection above IP addresses could be incorporated into Facile.

Despite these deficiencies, the implementation of the MSA applications demonstrates many strengths of Facile. The simplicity of the communication constructs, the lightweight threads, and the higher-order properties of the language all aid agent programming. Most striking was that Chevalier was able to imple-

ment the MSA applications (several thousand lines of code) with little difficulty though he had only minor previous experience with Facile or related languages. The development time for the first prototypes was approximately two and a half man-months, including GUI development.

Finally, our modifications to the Facile language appear to be usable in practice. In the next chapter we evaluate the implementation from the perspective of performance.

Chapter 6

Performance

Performance has been the motivating reason behind many of our design decisions for the agent language. To find out whether those decisions were correct, as well as which were most important, we measured the performance of real programs implemented in our extended version of Facile. There were several key questions we wanted to answer:

- How should the function transmission facility be configured? Is it better to compile or to interpret code? Should compilation be optimized? What representations should be transmitted?
- Does closure trimming significantly improve the performance of agent transmission in real applications?
- What are the sizes of the transmissible versions of functions? Do the sizes justify not generating transmissible code for all functions?

We expected the best configuration for the function transmission facility to depend on the characteristics of the application being tested. Different applications represent different sets of trade-offs in compilation time, execution time, etc. For this reason, we tested a broad range of facility configurations on both the MSA applications and synthetic benchmarks.

Agent-based applications incur costs of marshalling, transmission, and compilation. We wanted to see whether these costs were acceptable for the MSA applications. The data collected from comparing facility configurations can also be used for this purpose.

We used the MSA applications again to assess the effect of closure trimming on performance. Synthetic benchmarks are not suitable for this purpose, because it is easy to create programs that benefit either greatly or not at all from closure trimming. These extremes would tell us little about whether trimming closures is worthwhile. Measurements of real programs provide a sounder basis for evaluating this feature.

Finally, we examined the relative sizes of the native code and transmissible versions of functions. This data can show whether we were justified in distinguishing potentially transmissible functions from nontransmissible functions to reduce run-time space costs.

We start this chapter in Section 6.1 by analyzing the performance of different configurations for the facility. We then examine closure trimming in Section 6.2 and the relative sizes of functions in Section 6.3. We evaluate our design decisions in light of the performance data in Section 6.4.

6.1 Facility Configuration

Our first goal was to determine what configuration of the function transmission facility minimizes the cost of transmitting and executing functions. Some definitions will clarify this goal.

The facility consists of three main components. The first is the compile-time marshaller that generates transmissible versions of functions. The second is the run-time data marshalling and transmission subsystem that is used to transmit normal Facile data as well as the transmissible representations for functions. The last component is the run-time compiler and linker for received functions.

The facility allows a function to be transmitted to a remote machine and executed there. We wanted to assess the performance of this service. We only considered outcomes of using the service where the function was successfully transmitted, compiled, and then executed. We ignored outcomes where transmission or linking failed or where the recipient chose not to execute the function.

The performance of the service depends on the characteristics of the workloads tested. We hypothesized that some facility configurations would handle some workloads better than others and vice versa. We did not expect a single configuration to minimize cost in all situations. However, we wanted to rule out some configurations altogether and to learn which ones we would want to use in practice.

6.1.1 Performance of Benchmarks

We began the performance study using synthetic, noninteractive benchmarks. These have the advantages that we could make them easy to measure and that we could vary their characteristics in a controlled way. Using the benchmarks we could test many workloads and configurations and repeat tests to generate statistically significant data. We were then able to proceed with more focused tests for the MSA applications.

The MSA applications all concentrate on quick, interactive response and perform minimal computation. We designed the synthetic benchmarks to be more computation intensive in comparison, thus rounding out our analysis of the different configurations.

Metrics

To measure the performance of transmitting and executing a given function, we used the sum of the following metrics:

- The time to marshal the function at run time
- The time to transmit the function
- The time to unmarshal the function
- The time to compile and link the function
- The time to execute the function

If a function is transmitted as native code, the compile-time portion of the compile and link metric measures only the time for the trigger function to extract the code string from the transmission record and to overwrite the function closure (see Section 4.4). Similarly, if the function is to be interpreted, the "compile time" is the time to create a specialized interpreter for the function by applying Facile's general interpreter to the extracted code.

Parameters

The parameters affecting the performance of the facility can be split into those characterizing the *system* and those characterizing the *workload.* In the former category, we have the following:

- The set of representations transmitted for a function
- Interpretation or compilation of functions on the recipient

- Optimization of CPS when compiling lambda code on the recipient
- Instruction scheduling during compilation on the recipient
- Overhead of the data marshalling and transmission subsystem
- Garbage collection overhead

The parameters characterizing the workload are as follows:

- Size and running time of the transmitted function
- Speed and architecture of the sending CPU
- Speed and architecture of the receiving CPU
- Speed (bandwidth and latency) of the network link
- Other loads on the CPUs
- Other loads on the network

To focus the study on the compile-time marshalling and run-time compilation components of the function transmission facility, we left the data marshalling and transmission subsystem unchanged. The overhead of this system only varies as a factor of the other parameters; for example, its overhead increases when larger functions are transmitted.

We also kept the overheads from other loads on the CPUs and network links as constant as possible. We performed tests only during off-peak times on lightly loaded machines.

We did not change the garbage collector. Like the data marshalling and transmission overhead, the total cost of garbage collection in a test depends on the other parameters. In running tests one after another, though, we needed to ensure that the cost of collecting garbage generated by one test was not paid by the succeeding test. So that all tests started with a clean slate, we preceded each with a garbage collection.

Factors

We treated all the remaining parameters as factors to vary during the performance study. Each factor can range over several levels.

The first factor is the set of representations transmitted for a function. There are four possible representations:

- Lambda code
- Middle CPS

- Final CPS
- Native code

There are $2^4 - 1 = 15$ possible sets of representations. However, for testing it is sufficient to send single representations. The reason is that even if multiple representations are sent, only one is used by the recipient. Any other representations sent along only increase the marshalling and transmission times. Moreover, because we controlled the sender and the recipient in each trial, we knew in advance which representation would be used. By measuring the marshalling and transmission times individually for each representation, we could collect data which would allow us to assess whether sending multiple representations might be worthwhile.

The other factors that determine the facility configuration are not all independent. Optimization of CPS is only relevant when the transmitted representation is lambda code, because the CPS representations are already optimized (though to a predetermined number of registers). Interpretation is also only possible with lambda code. Instruction scheduling does not apply to interpretation or to native code. These constraints yield the following ten facility configurations:

- Native code
- Interpreted lambda code
- Compiled lambda code
- Lambda code compiled with CPS optimization
- Lambda code compiled with instruction scheduling
- Lambda code compiled with CPS optimization and instruction scheduling
- Compiled middle CPS
- Middle CPS compiled with instruction scheduling
- Compiled final CPS
- Final CPS compiled with instruction scheduling

We also needed levels for the factors determining the workload. Varying the sending and receiving architectures can show whether the choice of architecture affects the relative performance of different facility configurations. We used a Sparc Classic with 32 megabytes of memory and an Intel 486/66 with 32 megabytes of memory as the possible CPUs for sender and receiver. Not all combinations of these machines can be used with all the facility configurations. Native code can only be sent between identical architectures, and the final CPS representation is not used with the Intel. Also, because the Intel is a CISC architecture, instruction scheduling does not apply to it.

Agents may be retrieved over a LAN or across the Internet. The speed of the communications link is therefore an interesting factor to consider in studying performance. We performed tests on both the LAN and with transatlantic communication between Pittsburgh and Munich. The data transmission subsystem delivered bandwidths of approximately 550 kilobytes per second for the former case and 40 kilobytes per second for the latter.

The last factors are the characteristics of the transmitted functions. We used functions that sort the same random array of real numbers using Quicksort. The code size of the functions could be *large* or *small,* while the running time could be *long* or *short.* For each of these factors, the difference between the two levels was approximately an order of magnitude. We made the code larger by inserting numerous copies of the core sorting routines. We increased the running time by repeating the sort on copies of the input.

From preliminary tests we adjusted the sizes large and small and the times long and short so that we had different relationships between the compilation and execution time metrics. For example, when using the middle CPS representation, the compilation and execution times were roughly equal both for the large, long-running function and for the small, short-running function, because compilation time is proportional to code size. However, for the large, short-running function the execution time was an order of magnitude less than the compilation time, while the reverse was true for the small, long-running function. The purpose of varying the relationships between the metrics was to expose the trade-offs of different facility configurations.

The different function characteristics reflect different kinds of agents. For example, an agent that presents a three-dimensional view of a real-time feed might be large and run for a long period of time. A mobile service agent such as the directory may be small and short running. An agent that filters a data set on a remote site might also be small but run for a comparatively long time.

The sorting functions all had empty closures; in particular, the input array was not transmitted with them. By organizing the functions in this way, we avoided skewing the marshalling and transmission times with a potentially large constant overhead and hiding the effects of the different representations on these times.

Hypotheses

We expected the following general results from testing the benchmarks on the different configurations:

- Interpretation should be best for the short-running functions, especially for the large one, which would have longer compilation times. Compilation should be better for the other cases, particularly for the small, long-running benchmark.
- Lower-level representations should be more verbose and have longer transmission times than higher-level ones. They should compile faster, but because of having fewer architecture-specific optimizations, also execute slower. Lambda code, our highest-level representation, should be favored in some tests and final CPS, our lowest, in others, depending on the relative contribution of these factors.
- Optimizations during compilation should be worthwhile for long-running functions.

Experimental Design

A full factorial design over the factors and levels yields a total of 184 experiments, each with five metrics, for a total of 920 data points. However, many of the metrics are identical between experiments. For example, the time to marshal and transmit a lambda representation is unchanged whether the representation is compiled or interpreted. By eliminating such duplicates and timing some metrics together we could reduce the number of data points that needed to be collected to just 208.

For each data point we repeated the test to generate it 50 times. Preliminary tests showed that with this many samples we were able to generate narrow intervals for the mean at 95% confidence. The high number of repetitions also allowed us to detect outliers, which is particularly important for timing WAN transmissions (one to two large outliers were not uncommon in these samples).

To be able to time tests separately, the error of our measurement tools needed to be small relative to the durations of the tests. We used calls to the Unix `gettimeofday` function. On the Sparc we achieved an accuracy of ± 10 microseconds using `gettimeofday`, and on the Intel ± 100 microseconds. The latter figure is coarse enough that timings of unmarshallings on the Intel could have errors greater than 30%. Unmarshallings are fast because they consist only of copying the transmissible code string out of a network message and setting up the trigger closure. However, the contribution of the unmarshalling times and their high errors disappeared when we summed the metrics for an overall performance figure.

Results

Table 6.1 shows the results of testing the different facility configurations on the workloads. The values are the sum of the marshalling, transmission, unmarshalling, compilation, and execution times for the given configurations and workloads. Each value is shown to the number of significant figures corresponding to its error range at the 95% confidence level.

The table only includes data for cases where the sender has the same architecture as the receiver. The results when the sender has a different architecture are statistically indistinguishable at the 95% confidence level from those when it has the same architecture, with the exception of seven cases. For the seven cases, the differences between corresponding data points are less than 1%. Therefore, we do not consider the architecture of the sender in the analysis.

The low significance of the choice of sending architecture is not because marshalling the benchmarks takes the same time on the Sparc and the Intel; the data show that the Sparc uniformly outperforms the Intel. The explanation is that marshalling is a short operation compared to transmission, compilation, and execution. Because the benchmarks have empty closures, marshalling consists simply of finding the transmissible representations and placing them in the outgoing message buffers, primarily a copying activity.

We can observe several trends in the data:

- *Compilation is always better than interpretation for these computation-intensive benchmarks.* The interpreter lags the fastest compiled representations by one to two orders of magnitude for the long-running functions and similarly for the small, short-running function. For example, the large, long-running benchmark takes 17.05 seconds on the Sparc with LAN when the final CPS representation is compiled, but 348.1 seconds when it is interpreted. The large, short-running benchmark, where compilation times are longest relative to execution times, shows the smallest gap between interpretation and compilation (e.g., 22.2 seconds for Sparc with WAN and middle CPS, compared against 47.3 seconds interpreted). Nevertheless, even for this benchmark interpretation is still slower than the middle CPS representation by a factor of two.

- *It only pays to perform CPS optimizations on lambda code for small functions.* The time to optimize the CPS when compiling large functions from lambda code outweighs any improvement in their execution times. If we

Table 6.1: Performance data (in seconds) for the sorting function benchmarks.

sender	Sparc	Sparc	Intel	Intel	Sparc	Sparc	Intel	Intel
net	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
recipient	Sparc	Sparc	Intel	Intel	Sparc	Sparc	Intel	Intel
	large–long				**large–short**			
$\text{lambda}_{\text{int}}$	348.1	360.0	396.6	408.2	35.44	47.3	41.79	53.4
lambda	38.7	50.6	44.13	55.8	24.90	36.8	26.97	38.6
$\text{lambda}_{\text{opt}}$	45.95	57.8	58.49	70.1	38.9	50.8	49.35	61.0
$\text{lambda}_{\text{sch}}$	43.0	54.9			30.36	42.2		
$\text{lambda}_{\text{opt,sch}}$	49.6	61.5			42.9	54.8		
middle	24.11	**29.0**	**28.45**	**33.11**	17.33	**22.2**	**18.68**	**23.34**
$\text{middle}_{\text{sch}}$	27.84	32.7			21.38	26.2		
final	**17.05**	30.0			**10.77**	23.8		
$\text{final}_{\text{sch}}$	21.97	35.0			14.35	27.4		
native	7.05	9.2	10.54	12.42	0.91	3.0	1.37	3.25
	small–long				**small–short**			
$\text{lambda}_{\text{int}}$	668.2	669.4	782.0	783.2	33.01	34.3	39.35	40.57
lambda	33.51	34.8	40.83	42.04	3.642	4.92	4.24	5.45
$\text{lambda}_{\text{opt}}$	10.73	12.0	18.68	19.89	3.20	4.5	4.07	5.28
$\text{lambda}_{\text{sch}}$	30.89	32.2			4.00	5.3		
$\text{lambda}_{\text{opt,sch}}$	10.37	11.7			3.365	4.65		
middle	**9.36**	**9.87**	**17.69**	**18.21**	1.665	**2.17**	**2.42**	**2.94**
$\text{middle}_{\text{sch}}$	9.39	**9.90**			1.970	2.47		
final	12.11	13.5			**1.400**	2.8		
$\text{final}_{\text{sch}}$	12.07	13.5			1.741	3.2		
native	7.65	7.82	15.94	16.11	0.379	0.551	0.809	0.980

Boldface entries are the smallest for a given workload, excluding the native code configuration. Multiple boldface entries mean the values are not statistically distinguishable (their error ranges overlap). The subscripts *int, opt,* and *sch* refer to interpretation, CPS optimization, and instruction scheduling, respectively.

Table 6.2: Sizes (in bytes) of the marshalled transmissible representations for the large and small sorting functions.

	large	small
lambda	533760	51416
middle	217264	20600
final	560416	55956
$\text{native}_{\text{Intel}}$	82496	7112
$\text{native}_{\text{Sparc}}$	90260	7668

compare the lambda and $\text{lambda}_{\text{opt}}$ rows in the upper half of the table, the latter is always larger than the former.

For the small functions, the overhead of CPS optimization is sufficiently small that faster execution times make up for it. Looking at the same two rows but now in the bottom half of the table, we see that the improvement in overall performance is marginal in the case of the small, short-running function, but it increases to a factor of two to three for the long-running one.

- *Instruction scheduling is not worth performing on the Sparc.* A comparison of the middle and $\text{middle}_{\text{sch}}$ rows, and likewise for the final and $\text{final}_{\text{sch}}$ rows, shows the increased time for scheduling. Only for the small, long-running benchmark, where the time for scheduling is shortest relative to execution time, does scheduling offer a marginal improvement for some configurations.

- *The relative advantages of the final and middle CPS representations depend on the transmission times.* On the LAN, the final CPS representation offers better performance than the middle one because of its shorter compilation times (with one exception). However, for transatlantic communication, the middle CPS representation performs better. The bold numbers in the first two columns of each table quadrant show the difference.

  To understand why middle CPS outperforms final CPS on the WAN, we need to look at the sizes of the different representations. Table 6.2 shows the sizes of the marshalled large and small sorting functions (the sizes are independent

of whether they run long or short). The final CPS representation is more than twice the size of the middle CPS for both functions (e.g., 560 kilobytes versus 217 kilobytes). On the WAN, our slow communications link, this factor translates into several-second differences between transmission times for the two representations. The more compact middle CPS representation ends up with better overall performance.

The exception to the better performance of final CPS on the LAN is in the small, long-running benchmark. In this case, the final CPS representation is slower than both optimized lambda and middle CPS. Examination of the underlying metrics shows that final CPS has longer execution times than these other two.

The explanation is that in this benchmark the long execution times relative to compilation offer the greatest reward for taking extra time to generate higher-quality code. The code generated from final CPS is of lower quality than for the other representations because it does not take advantage of all the registers available on the recipient. Recall from Section 4.2.2 that the sender specializes final CPS to a small, worst-case number of registers during closure conversion and register spilling. For the lambda and middle CPS representations, these steps are performed on the recipient and use the full number of available registers. The cost of performing these steps on the recipient is repaid by the increased speed in using more registers, but only for this benchmark.[1]

- *The native code representation offers the best performance in all cases.* The speedup factor over the other representations is largest when execution times relative to compilation times are shortest. The speedup is least for the small, long-running function, where the reverse relation is true. The native representation is also the most compact one that we transmit (less than half the size of the next smallest, middle CPS), so it has the added advantage of shorter transmission times in WAN communications.

- *The choice of architecture has no effect on the relationships between the*

[1] The sender also specializes middle CPS to a worst-case number of registers during earlier compilation phases. So that middle CPS can also be used on CISC machines, this number is even smaller than the worst case assumed for final CPS. Because middle CPS nevertheless has better performance in this benchmark, we can conclude that it is more important to have more registers in the closure conversion and register spilling phases than in the earlier ones.

data. The times for the Sparc are shorter than the times for the Intel, but for each architecture the relationships between the data remain the same.

Discussion

The benchmarks suggest that lambda should never be used as a compiled representation; it does not offer significantly improved performance in return for the extra compilation time. We thought lambda, because of its greater possibility for machine-specific optimizations, would at least outperform the lower-level representations for the small, long-running benchmark (it did come close: 10.37 seconds versus 9.36 for middle CPS), so this result surprised us.

For compilation we should use either final CPS with no scheduling or middle CPS with no scheduling. Thus, we would choose middle CPS for WAN communications or in cases where the execution time is long relative to compilation, otherwise we would choose final CPS.

Because the benchmarks emphasize computation, we cannot rule out interpreted lambda code as a good choice for functions that run for much shorter periods than the benchmarks. We use the performance of the MSA applications to evaluate this possibility in Section 6.1.2.

The sizes in Table 6.2 are surprising in that the intermediate representations are so verbose in comparison to native code; we expected the opposite, with the lowest-level representation requiring the most space. The reason for the large sizes is that these intermediate representations are stored and transmitted as Facile data structures. For example, an operation such as adding 5 and 4 becomes `APP(PRIM +,RECORD(5,4))` in the lambda language, a data structure requiring 30 words of memory (garbage collector tags use 10 words and pointers account for another 9). In contrast, the equivalent RISC code might be only 2 or 3 words.

The lambda representation is larger than middle CPS because many constructs are simplified or eliminated in generating the latter. In generating final CPS from middle CPS, closure conversion is performed. This step adds new code to create and manipulate closures, which drives the size back up.

The small relative size and high performance of native code suggest that it may be worthwhile to send it along optimistically with the middle or final CPS representations. The small size means that the penalty from higher transmission times should be low if the code cannot be used. Otherwise, we expect a sizable benefit by eliminating compilation time and maximizing execution speed.

Table 6.3: Combined marshalling and transmission times (in seconds) for the large and small sorting functions over LAN and WAN links.

	large		small	
	LAN	WAN	LAN	WAN
lambda	0.948	12.8	0.0991	1.38
middle	0.383	5.2	0.039	0.545
final	0.989	14.0	0.094	1.5
$\text{native}_{\text{sparc}}$	0.0134	0.185	0.167	2.31

The sending architecture is Sparc. Significant figures correspond to the 95% confidence level.

To test this hypothesis, we can calculate the performance of sending along native code and then either using it or not using it. Let p_r be the overall performance from Table 6.1 for representation r. Let t_r be its marshalling and transmission time (see Table 6.3). Then for the performance of sending native code with middle CPS (for example) but not using the native code, we calculate $p_{\text{middle}} + t_{\text{native}}$. If we do use the native code, the performance is $p_{\text{native}} + t_{\text{middle}}$. We can view this calculation as adding in the penalty for needlessly sending an intermediate representation along with native code.

Table 6.4 shows the computed values along with the original performance figures for middle and final CPS. For simplicity, we consider only the Sparc architecture.

If we send the combined representations and the recipient is able to use the native one, we obtain speedup factors ranging from 1.2 to 13. The biggest gains occur with the large, short-running benchmark on a LAN. The compilation time, which normally dominates this benchmark, is eliminated, while the overhead of sending the extra code is minimized by the fast network connection. The speedups on the WAN are less than those on the LAN because the overhead of sending the extra code is proportionately higher on the slower network connection.

The speedups are greater for middle CPS than for final CPS. The reason is that middle CPS normally compiles longer than final CPS, so the savings when compilation is eliminated are larger. The small, long-running function is an exception. In this case final CPS gets a bigger speedup because its execution time

Table 6.4: Calculated performance (in seconds) for the sorting function benchmarks using multiple representations.

	large–long		large–short		small–long		small–short	
	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
middle	24.11	29.0	17.33	22.2	9.36	9.87	1.665	2.17
middle+native	24.27	31.3	17.50	24.5	9.37	10.05	1.679	2.36
middle+**native**	7.44	14.4	1.29	8.3	7.69	8.36	0.418	1.10
final	17.05	30.0	10.77	23.8	12.11	13.5	1.400	2.8
final+native	17.21	32.4	10.93	26.1	12.12	13.7	1.413	3.0
final+**native**	8.04	23.2	1.90	17.0	7.74	9.3	0.474	2.1

Representations in boldface are used by the recipient.

without the native code is longer than middle CPS's.

When the recipient cannot use the native code, the penalty for having sent it depends on whether the LAN or WAN was used. On the LAN, the performance penalty is 1% or less. On the WAN, the greater contribution of transmission times leads to penalties from 7 to 10%, except for the small, long-running benchmark once again. For that benchmark, the extra transmission time is mitigated by the long execution time, so the penalty is only 1 to 2%.

The choice of whether to send native code or not must weigh the potential speedup against the potential penalty. If the probability of an agent executing on particular architectures can be estimated, this information can be used to help quantify the decision. We can also consider the real time cost of the penalty. For example, we may decide that in a particular application we can always afford penalties of less than 1 second.

6.1.2 Performance of MSA Applications

The sorting benchmarks show how the function transmission facility should be configured for agents that perform heavy computation. However, the MSA applications consist of functions that run for much shorter periods relative to their compilation times. For these applications, different trade-offs may apply. We wanted to find the configuration that delivers the best performance with these

workloads.

In addition to the relative performance of different configurations, we were interested in the actual performance numbers. The MSA applications are intended to be interactive, and we wanted to see whether the overheads of downloading, compiling, and executing code were compatible with that goal.

Methodology

With the exception of the unmarshalling time, we used the same metrics as before for comparing the performance of the MSA applications on different facility configurations. The unmarshalling time is negligible compared to the others, so we ignored it.

The four facility configurations we tested are native code, interpretation of lambda code, and compilation of final and middle CPS without scheduling. Our previous results served to rule out the other configurations.

For the workloads we had four agent applications: the directory, file browser, interactive map, and schedule (all of which are instrumented with calls to the agent tracking system). We used the same choices for the network link: LAN and transatlantic WAN. We fixed the sending and receiving CPUs to Sparc; the benchmarks showed that cross-platform transmissions had virtually identical performance to homogeneous ones, and the relations between performance figures for the Intel were the same as those for the Sparc.

For the marshalling, transmission, and compilation times we repeated tests 50 times as before. For the execution times we repeated tests only three times. The reason for the small number of repetitions is that the service agents of the MSA applications are interactive, and in the absence of special test harnesses we had to exercise each agent by hand. These tests consisted of an interaction sequence that exercised each user operation once.

Results

Table 6.5 shows the performance data for the MSA applications. The values are the sum of the marshalling, transmission, compilation, and execution times for the given configurations and workloads. The significant figures are derived from error ranges for these times at the 95% confidence level, except for the execution time. Because of the small number of repetitions, we computed its error ranges at the 90% confidence level. The execution times are only for the portions of

Table 6.5: Performance data (in seconds) for the MSA applications.

	directory		browser		map		schedule	
	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
$\text{lambda}_{\text{int}}$	0.32	2.8	0.23	1.70	0.25	1.7	0.6	4.7
middle	3.75	5.15	2.42	3.28	2.35	3.22	6.67	9.0
final	2.30	5.9	1.53	3.65	1.48	3.6	4.08	10.5
native	0.17	0.85	0.16	0.66	0.19	0.68	0.26	1.5

the agents written in Facile; the performance of the Tcl/Tk user interface was not measured.

In contrast to the sorting benchmarks, interpretation offers better performance than compilation, and by a substantial margin: more than five times faster with the LAN link and twice as fast with the WAN (the difference reflects the greater significance of WAN transmission times to overall performance). Taking the time to compile code does not pay for these short executions.

As with our previous experiments, the final CPS representation outperforms the middle one with the LAN link. The reverse is true with the WAN. (Compare the second and third entries in each pair of columns.) The reason again is that the final CPS representation is substantially larger and therefore takes longer to transmit. Table 6.6 shows the different sizes.

Because execution times are so short, sending native code with the lambda representation is not worth the overhead of higher transmission times. Table 6.7 shows that the performance of the combined representations is almost the same whether the recipient can use the native code or not. Therefore it is more efficient to send simply the lambda representation.

Interaction Times

Though the overall performance times are useful for comparing the facility configurations under different workloads, they do not show how the times are spread out over an interactive run. We wanted this latter information to assess whether the function transmission facility is suitable for real interactive applications.

When an agent is first requested, there is a delay of up to several seconds while

Table 6.6: Sizes (in bytes) of the transmissible representations for the MSA service agents.

	directory	browser	map	schedule
$\text{lambda}_{\text{int}}$	102472	64364	58248	176460
middle	60232	36744	35592	103300
final	153764	93760	93552	277324
$\text{native}_{\text{Sparc}}$	27528	17532	17164	49260

Table 6.7: Calculated performance (in seconds) for the MSA service agents using multiple representations.

	directory		browser		map		schedule	
	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
$\text{lambda}_{\text{int}}$	0.32	2.8	0.23	1.70	0.25	1.7	0.6	4.7
$\text{lambda}_{\text{int}}$+native	0.38	3.5	0.27	2.24	0.29	2.2	0.7	6.0
$\text{lambda}_{\text{int}}$+**native**	0.36	3.6	0.28	2.25	0.29	2.25	0.57	5.9

Representations in boldface are used by the recipient.

it is marshalled, downloaded, and then unmarshalled. Because this delay occurs before the start of execution, we did not include it as part of the interaction times.[2] Note that our design decision to avoid implicit callbacks means that once an agent has started execution, no further components must be retrieved or awaited from the sender.

For assessing interactive performance we therefore concentrated on compilation and execution times. Compilation times are dispersed through the run, but because each function is only compiled once, these times may tend to be concentrated in the early part of the run or in clusters. We wanted the combined compilation and execution times between user interactions to be as small as possible.

[2] We can rationalize this decision in part by observing that users of the World Wide Web experience (and tolerate) similar delays when waiting for images or video clips to download. However, once a download has completed, users expect quick response.

Table 6.8: Times (in seconds) between a series of user interactions with the MSA agents.

$\text{lambda}_{\text{int}}$	0.26	0.16	0.07	0.04	0.18	0.042	0.46
middle	4	2.24	1.0	0.52	1.678	1.21	3.95
final	2.5	1.20	0.5	0.29	1.02	0.76	1.98
native	0.13	0.12	0.049	0.0231	0.16	0.0315	0.18

For the MSA applications, there was one execution time we ignored. When an MSA agent starts executing, its first action is to spawn a Tcl/Tk user interface. This action can take from 2 to 12 seconds to complete, though the long time is mitigated by the incremental appearance of the interface (the user can see that something is happening). Because it occurred at the start of execution, we placed this delay in the same category as the downloading time. It is also unrelated to the performance of our facility.

Table 6.8 shows the combined compilation and execution times after each user input during a run of the MSA agents. Each run was repeated three times, and significant figures correspond to the 90% confidence level. The interpreted lambda representation is the only architecture-independent one that delivers subsecond response times, with each action taking less than 0.2 seconds to complete. This level of performance is suitable for interactive use [see Stallings 1992, pp. 409–411].

For the CPS representations, the combined times were long enough to impose a noticeable drag during interaction. Optimistically transmitting native code with these representations would significantly improve performance on homogeneous architectures, especially because the extra cost of downloading the native code is paid before execution starts.

If we extended the run and began to repeat calls to functions, the times for the CPS representations would fall to levels comparable with that of native code. However, this change in running times implies that interactive performance is inconsistent, which can mislead users. The first time a user selects a particular functionality, she builds a mental model of the time it takes to execute. Based on this model, the user may decide not to select the functionality a second time because it is too slow. However, the model is incorrect: The second and subsequent

uses of a functionality will be much faster than the first. A tension therefore exists between lazy compilation and interactive environments.

6.2 Closure Trimming

In Section 3.3 of the design we introduced ubiquitous values. One important feature of these values is that they can be trimmed from closures during function marshalling. Use of ubiquitous values can improve performance by cutting transmission and compilation overheads.

We assessed the improvement offered by two categories of ubiquitous values. The first consists of the pervasive library values. These are ubiquitous by default and no action is required from the programmer to gain the benefits of trimming them. The second category consists of values from ubiquitous structures. These structures must be explicitly defined by the programmer.

Synthetic benchmarks are not a good choice for evaluating closure trimming because they are unlikely to reflect how ubiquitous values are used in real programs. We used the MSA applications instead. These applications use pervasive values and include some programmer-defined ubiquitous structures.

The key metric for assessing the effect of closure trimming is the size of a marshalled function. The more values from a function's closure that are marshalled together with it, the larger the size. Transmission and compilation times are proportional to this size.

We expected the use of closure trimming to offer some benefit, but we were unsure of the magnitude of that benefit. Closure trimming adds complexity to the language implementation and, outside of the pervasives, requires actions from the programmer. We wanted to see if these costs were justified.

Table 6.9 shows the sizes for the four MSA agents marshalled in the middle CPS representation. Three levels of closure trimming are represented: full trimming of programmer-defined ubiquitous values as well as the pervasives, trimming of just the pervasives, and no trimming at all. We present results for only one code representation because the sizes simply scale for the others; for example, the native code sizes are one-half the middle CPS ones. Tables 6.10 and 6.11 show how the sizes translate into transmission and compilation times on the Sparc. These times are for a LAN, so the contribution of transmission times is minor.

Overall, the results show that closure trimming substantially improves performance. In the MSA applications, trimming the pervasives alone reduces mar-

Table 6.9: Effect of closure trimming on the sizes (in bytes) of marshalled agents.

	directory	browser	map	schedule
trim all	60232	36744	35592	103300
trim pervs	108456	80540	79492	151384
no trim	247100	200348	200264	274692

Table 6.10: Effect of closure trimming on transmission times (in seconds).

	directory	browser	map	schedule
trim all	0.110	0.069	0.068	0.18
trim pervs	0.196	0.146	0.15	0.27
no trim	0.441	0.365	0.360	0.50

Table 6.11: Effect of closure trimming on compilation times (in seconds).

	directory	browser	map	schedule
trim all	3.54	2.16	2.15	6.38
trim pervs	4.95	3.50	3.51	7.69
no trim	12.5	9.94	10.11	14.8

shalled sizes by 45 to 60% and combined transmission and compilation times by 48 to 65% (compare the first and second entries in each column). Closure trimming therefore offers a benefit even if the programmer does not define any ubiquitous structures herself. The improvement is greatest for the browser and the map because the pervasives add about the same amount to each agent (between 120 and 139 kilobytes), and this amount is proportionately larger for these smaller agents.

Further benefits can be realized after the pervasives have been trimmed by explicitly using ubiquitous structures. The data show that the sizes of the MSA agents drop by 32 to 55% and the associated times by 18 to 39% (the difference between the second and third entries in each column).

The times do not drop as much as the sizes do in these latter figures. The reason is that compilation times are affected by the complexity of code as well as its size, so we cannot expect the decreases to be identical. If the transmission times were a larger factor in the times, we would expect the changes in times to reflect the changes in sizes more closely.

Closure trimming is therefore an important optimization in supporting agent programming, and it is worthwhile for programmers to define and use ubiquitous structures.

6.3 Function Sizes

In Section 3.1.2 we decided not to generate transmissible representations for every function in a program on the grounds that the space increase would be unacceptable. We introduced annotations for those functions for which representations should be generated at compile time.

Tables 6.2 and 6.6, showing the sizes of different transmissible representations, allow us to assess this decision. Our most compact representation, middle CPS, is twice the size of native code. Lambda code is at least three times larger, and final CPS at least five times. If every function in a program were increased by these factors, overall image sizes would be unacceptably large (many megabytes). We are therefore justified in limiting the generation of transmissible representations to only those functions that are potentially transmissible.

At the same time we must acknowledge that our transmissible representations are exceedingly verbose. As we stated before, the reason is that we simply marshal the Facile data structure for each representation. By using byte encodings specially designed for the representations, we would be able to reduce the space required to that of native code or less.

6.4 Evaluation

From the performance data of the preceding sections, we can evaluate our agent language design and implementation:

- Supporting both compilation and interpretation allows the function transmission facility to deliver good performance for different kinds of workloads. Using the right technique for the task dramatically improves performance.

Computation-intensive tasks were an order of magnitude faster when compilation was used rather than interpretation, while the reverse was true for the interactive applications.

- Our decision to implement support for more than one intermediate representation was correct. Supporting lambda code alone would have been sufficient to provide both compilation and interpretation. However, we saw that lambda is best interpreted and that the CPS representations offer better performance when compilation is used. Moreover, of the two CPS representations, middle CPS is better for slow network connections (WAN) and final CPS better for fast ones (LAN).

  In our implementation, the sender uses compiler settings to control which representations are generated for potentially transmissible functions. It is even possible to control the choice of representation on a function-by-function basis. If recipients only interpret lambda code and compile the other representations, the sender can then control through the representation it sends whether a recipient will compile or interpret the agent. This feature is a good one because it is the sender who knows if an agent is best compiled or interpreted; the recipient does not have to guess.

- Allowing multiple representations to be transmitted together is another strong point of the design. The main use of this feature is to send native code optimistically with other representations. For computation-intensive tasks, the small potential performance penalty if the native code cannot be used is offset by substantial speedups if it is.

- Closure trimming is effective in reducing the size of marshalled agents and their associated transmission and compilation times.[3] This was a difficult feature to design and implement, but we feel justified by the benefits it offers.

- Generating transmissible representations selectively, rather than for every function in a program, is important for avoiding blow-ups in run-time memory requirements.

[3] If an agent is interpreted, closure trimming also improves execution times by allowing local, compiled versions of library functions to be used. However, this improvement is likely to be minor because we generally reserve interpretation for agents that have short execution times.

- Our transmissible representations are too large. More concise representations would improve performance over WAN connections and decrease memory requirements. Some of our trade-offs would shift as well; for example, the cost of transmitting native code optimistically with an intermediate representation would be proportionately more expensive.

Chapter 7

Conclusions and Future Work

In this dissertation we examined the design and implementation of an agent language. Our language satisfies a number of essential and desirable properties for agent programming. We evaluated the language in light of the construction and performance of real applications based on mobile agents.

In Section 7.1 we review the contributions made by this dissertation. In Section 7.2 we then explore future work based on our design and implementation. We discuss the integration of security into agent languages in Section 7.3. We conclude in Section 7.4 with several key lessons for further work on mobile agent languages.

7.1 Contributions

As we saw in Section 3.5, there has been much recent work on languages supporting code transmission. The growth of the Internet and the potential of mobile agents have spurred much of this interest, though work on code transmission goes back to at least the 1960s. Our work makes several contributions to the areas of code transmission and agent languages:

- *We developed a set of essential and desirable properties for agent languages.* For a code transmission system to be considered an agent language, it must at least provide a means to manipulate code (including the ability to incorporate received code into an existing address space), support transmission across heterogeneous architectures, and deliver performance suitable for its target applications. Desirable properties include language abstractions for code

manipulation, remote resource access, strong typing, automatic memory management, stand-alone execution, independent compilation, and security.

These properties serve as a means to guide and evaluate current and future work on agent languages. They are specifically motivated by the needs of mobile agent programming.

- *We integrated all of these properties (save security) into a single language implementation.* As we have seen, tensions exist between several of the properties. An implementation must balance and resolve these tensions. Our work is evidence that such a balance may be achieved, opening the way to the integration of further desirable features.

- *We developed a way to combine strong typing, remote resource access, and independent compilation.* Our design and implementation permits lexically scoped access to the names and types of remote resources. The use of signatures as names for remote resource collections provides the ability to type-check agents at compile time. We showed how to support independent compilation and type-safe dynamic linking of resources by placing careful restrictions on the use of abstract types with remote resources.

- *We added dynamic unlinking to dynamic linking.* With closure trimming and the concept of ubiquitous values we developed a means for values to be left behind when an agent is transmitted. In addition to improving performance, we gained the ability for agents to access the site-specific implementations of resources.

- *We introduced the use of multiple representations for transmitted agents.* With our system we can choose more efficient representations to send depending on the characteristics of the workload. We are also able to send two or more representations together, which can significantly improve performance for homogeneous transmissions.

- *We implemented lazy compilation for transmitted functions.* Each of the annotated functions making up an agent is only compiled as necessary by the recipient. By delaying compilation of code as late as possible, we avoid compiling code that is never used. The same technique helps to amortize compilation times over the run of an agent. Our implementation of

lazy compilation is transparent to the programmer and preserves the Facile language model.

- *We developed a programming framework for mobile agents and explored its use.* Programming paradigms for mobile agents are still emerging. The mobile service agent applications show how servers and clients can be structured to use agents. Moreover, the ease with which these applications were implemented provides a strong case in favor of agent language properties such as abstractions for manipulating code (in Facile, first-class functions), stand-alone execution, and automatic memory management.

- *We assessed the performance of the language implementation and applications.* Through performance analysis we exposed the cost and value of the major design decisions taken in our extension of Facile. The performance data allowed an evaluation of these decisions and also provide a sample of the real costs of using agents in applications.

7.2 Future Work

From our experience in adding support for mobile agents to Facile, we can identify several areas where more work or different approaches would be useful. We can summarize these areas in the following goals:

- *Detect potentially transmissible functions automatically.* Programmer annotations are not an ideal solution to the problem of identifying the potentially transmissible functions in a program. As we noted in Section 3.1.2, effect-based analysis offers one possibility for performing this task automatically. Another approach is to add a special transmissible subtype into Facile introduced by annotations and to modify `send` to accept only values with this type; the type-checker could then detect missing annotations. This approach would also be useful in distinguishing between transmissible and nontransmissible values (i.e., those that have only a site-specific meaning).

- *Simplify the publication of signatures.* In our system we distribute remote signatures in a compiled representation generated by a single site. It would be better to allow signatures to be distributed in source form. Source provides much more information to programmers than binary forms and

it is easier to exchange (a signature could be published in a manual, for example). To create effectively unique ids for signatures (and thereby keep dynamic linking fast), we could use a hashing scheme such as the one for canonical type ids in LYNX [Scott and Finkel 1988] or the one for persistent ids in Standard ML of New Jersey [Appel and MacQueen 1994].

- *Extend ubiquitous values beyond functions.* In our implementation, only those ubiquitous values referenced inside annotated functions are trimmed from closures on transmission. A more uniform approach is for ubiquitous values in any context to be trimmed from transmissions. Special run-time tags could allow detection of such values at marshalling time (though at the cost of a more complex implementation).

- *Evaluate lazy compilation further.* As we noted in Section 6.1.2, lazy compilation can lead users to form incorrect mental models of an application's speed. This effect is a potentially significant deficiency for lazy compilation in interactive applications. If the performance demands of an interactive application are low enough, interpretation is sufficient, but for higher demands (for example, with graphics) compilation may be necessary. The usability of lazy compilation in such cases remains to be evaluated.

7.3 Security

In addition to the goals from the preceding section, another important area for future work is to satisfy the security property for agent languages. Calling it "the" security property is perhaps misleading, because there are several different security problems to solve in the context of agents. We briefly discuss some of them here.

Because received agents are executed within the recipient's existing address space, one immediate question is how to preserve the privacy and integrity of that address space. We need to prevent an agent from accessing and changing the recipient's data and from triggering faults (through illegal instructions, bad memory accesses, etc.). Agents may also try to consume too much memory or CPU time, leading to denial of service.

Several languages oriented towards code transmission address these problems, including Safe-Tcl [Borenstein 1992, 1994], Tps [Heimbigner 1995], Java [Gosling and McGilton 1995], Omniware [Colusa 1995], Telescript [White 1994],

and Safe-Python [Majewski 1995]. One approach is to transmit agents in a language with tightly constrained actions, where, for example, random memory accesses may not be possible because the language does not provide pointers. Another approach is to use a general language but then to check or modify received agent code so that it meets various constraints. For example, Omniware uses *software fault isolation* [Wahbe et al. 1993] to run untrusted code in a separate portion of the address space. The code is modified so that it cannot access or jump to addresses outside its domain. Of course, even with such techniques many traditional vulnerabilities remain (bugs in interpreters or code checkers, for example).

Completely sealing off an agent is not desirable, however, because we want the agent to be able to access resources on the recipient. We therefore need *safe libraries* that agents may use. These libraries must carefully control access to resources, so that, for example, an agent can use a temporary file without being able to fill or search the file system.

Safe-Tcl, Telescript, and Java attempt to provide safe libraries. A major difficulty in developing such libraries is designing security models that allow sufficient access to resources while preventing malicious actions. For example, it might be useful for agents to be able to send email, but how can we judge whether an individual email is "safe" to send? Furthermore, each library adds another security model and more trusted code to the environment, increasing the likelihood of bugs or holes that an attacker could exploit. The Internet worm [Spafford 1989] provides a lesson in how the combination of mobile code and security weaknesses can enable large-scale attacks on the Internet.

In addition to protecting recipients from agents, we may also want to protect agents from recipients. A provider may only be willing to distribute agents if the privacy and integrity of agent code can be guaranteed. Conventional encryption methods can protect the agent during transmission, but the harder problem is to enable the recipient to execute the agent without being able to examine it. *Secure coprocessors* [Yee 1994; Yee and Tygar 1995] are one solution. A secure coprocessor is a physically protected device in the recipient's machine that can perform computation and store data. An agent can be transmitted to a recipient in an encrypted form that can only be decrypted by the secure coprocessor. The agent executes within the coprocessor and communicates with the untrusted environment of the recipient.

Secure coprocessors can be used to provide copy protection for agents. However, in the agent framework we may prefer to allow copies to be made as long as

it is still possible to bill for the use of agents. In a *superdistribution* architecture [Mori and Kawahara 1990], agents may be distributed and copied by anybody without prior negotiation with the agent provider. However, agents may only be executed within secure coprocessors containing previously installed billing software. The billing software debits the user's account on each use of an agent.

Another possibility for an agent provider is to register agents to particular users, to bill for their use through conventional means, and then to attempt to detect illegal copying and distribution. One means to detect illegal copies is through *digital watermarking* [Komatsu and Tominaga 1989]. In this approach a secret label is encoded into a copy by making small changes to redundant information. The label identifies the registered owner, so the provider can check that the registered owner and actual holder of a copy are the same. Standard cryptographic techniques can assure that the watermark, even if it is detected, cannot be replaced with another valid watermark. Although watermarking does not provide privacy or copy protection (it allows detection of misuse rather than preventing misuse), it needs no special hardware.

Integrating solutions for security problems into an agent language implementation supporting the other properties raises questions in such areas as performance, strong typing, and remote resource access. Though our solutions in these areas for Facile do not address security, for future work they provide an initial set of approaches against which the needs of security may be assessed and compared.

7.4 Lessons Learned

Reflecting on our experience in developing and using a mobile agent language, we can draw several broad lessons that may serve as advice to others working in this area. These lessons are also a guide to what we would and would not do differently if approaching this problem again.

Our first point is that *intermediate representations matter.* When building an agent language, the choice of representation for transmitting code is one of the earliest and most important decisions to make. The intermediate representation affects both performance and functionality. For performance, the affected metrics include transmission, compilation, and running times as well as storage costs. Compact representations can reduce both transmission times and storage costs. Low-level representations can be compiled faster and require less compilation machinery to be available at run time. They can also be designed to permit both

efficient interpretation and compilation to native code (the Java virtual machine is an example). A sufficiently general intermediate representation can serve as a common target for multiple compilers, allowing agents to be exchanged between programs written in different languages (an approach taken in Omniware). As we saw in the preceding section, the design of the intermediate representation may also be tied to providing some of the security needs of mobile agents.

In our work we let the source language and its existing implementation drive the choice of intermediate representations. This decision led to such problems as large transmissible code sizes and long running times when interpreting. In the future we would start by choosing a good intermediate representation and then examining anew the usefulness and applicability of techniques such as multiple transmissible representations.

A second lesson is that *languages with first-class code-containing objects have natural advantages for agent programming.* Languages that provide constructs and abstractions for manipulating code are well-suited for developing agent applications. Such languages provide a tight integration between code and data, allowing the dynamic creation of specialized agents. The programmer can easily write agents at different granularities, from as small as a single line of code to as large as desired. Another advantage of having the constructs for code manipulation incorporated into the language is that the semantics of agent creation and execution become part of the language semantics. The programmer thus has a single coherent framework in which to work.

In our work we used a language with first-class functions as the foundation for our agent language. Structuring and building agent applications in this language was quite simple, as our MSA experiments showed. Moreover, we were able to leverage off other features of the language such as strong typing and an advanced module system to strengthen the agent programming environment.

Our third lesson is that *the characteristics of the target application environment should be used to define the requirements for the agent language.* Though this statement may seem obvious, it is important to realize that the target environment is characterized both by run-time features and by the way in which applications are developed for it. Different environments may require different properties from the agent language.

For example, applications and infrastructure in the Internet are developed and upgraded in a loosely coupled manner. As the components of a distributed application are gradually changed, backwards compatibility may need to be preserved, which in turn may have implications for how distributed types should be handled

in the agent language. In contrast, if the application environment is a local area network, the working assumption may be that all parts of the infrastructure are changed simultaneously, and the language designer may be able to require that all components of an application be generated by a single compilation. As another example, if portable computers are the target environment, the communication channels between senders and recipients of agents cannot be relied upon to persist. Eliminating implicit callbacks may then be a design goal for the language.

If we were repeating our work, we would define the target application environment more carefully. Though we stand by our existing essential and desirable agent language properties in the context of Internet applications, we initially selected these properties before characterizing the target environment. A more rigorous approach would lead to a more defined and focused set of requirements.

To conclude, mobile agents are poised to make dramatic changes in how distributed systems are structured and how the Internet is used. We firmly believe that the potential of this technology is only just being recognized, and that the uses of mobile agents will increase dramatically in coming years.

The interesting challenge raised by mobile agents is the new demands they place on programming languages, compilers, and run-time systems. We have defined these demands, designed and implemented innovative solutions to them, and evaluated our work in practice. The contributions of this dissertation serve to advance the understanding of this field.

Bibliography

Klaus H. Ahlers, Andre Kramer, David E. Breen, Pierre-Yves Chevalier, Chris Crampton, Eric Rose, Mihran Tuceryan, Ross T. Whitaker, and Douglas Greer, September 1995. Distributed augmented reality for collaborative design applications. In *Proceedings of EUROGraphics '95*. Also available as Technical Report ECRC-95-03 from ECRC, Arabellastr. 17, 81925 Munich, Germany.

Kristy Andrews and Duane Sand, October 1992. Migrating a CISC computer family onto RISC via object code translation. In *Architectural Support for Programming Languages and Operatings Systems (ASPLOS V)*, pp. 213–222, Massachusetts. ACM.

ANSA, July 1986. *ASN.1 Notation: A Tutorial*. Architecture Projects Management Limited. ANSA Number AO.11.02.

Andrew W. Appel, 1990. A runtime system. *Lisp and Symbolic Computation*, **3**:343–380.

Andrew W. Appel, 1992. *Compiling with Continuations*. Cambridge University Press.

Andrew W. Appel and David B. MacQueen, August 1991. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, number 528 in Lecture Notes in Computer Science, pp. 1–13, Passau, Germany. Springer-Verlag.

Andrew W. Appel and David B. MacQueen, June 1994. Separate compilation for Standard ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13–23, Orlando, FL.

G. Attardi, A. Baldi, U. Boni, F. Carignani, G. Cozzi, A. Pelligrini, E. Durocher, I. Filotti, Wang Qing, M. Hunter, J. Marks, C. Richardson, and A. Watson, November 1988. Techniques for dynamic software migration. In *ESPRIT '88: Proceedings of the 5th Annual ESPRIT Conference*, volume 1, pp. 475–491, Brussels.

Nathaniel S. Borenstein, 1992. Computational mail as network infrastructure for computer-supported cooperative work. In *Proceedings of CSCW '92 (Computer-Supported Cooperative Work)*.

Nathaniel S. Borenstein, May 1994. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *Proceedings of the 1994 IFIP WG6.5 Conference on Upper Layer Protocols, Architectures, and Applications*, Barcelona. North-Holland.

P. J. Brown, 1976. Throw-away compiling. *Software—Practice and Experience*, **6**(3):423–434.

Luca Cardelli, May 1994. *Obliq: A Language with Distributed Scope*. DEC Systems Research Center.

Craig Chambers and David Ungar, 1991. Making pure object-oriented languages practical. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 1–15, Phoenix, Arizona. ACM.

Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing, March 1990. Reliable distributed computing with Avalon/Common Lisp. In *International Conference on Computer Languages*, pp. 169–179. IEEE.

Colusa, 1995. Omniware: A universal substrate for mobile code. Colusa Software white paper. Available at `http://www.colusa.com/`.

Antony Courtney, May 1995a. *Phantom: A Language Overview*. Department of Computer Science, Trinity College Dublin, Ireland.

Antony Courtney, June 1995b. Phantom: An interpreted language for distributed programming. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, Monterey, CA.

Michael A. Dean, Richard M. Sands, and Richard E. Schantz, March 1987. Canonical data representation in the Cronus distributed operating system. In *Proceedings of the Sixth Annual Conference on Computer Communications (IEEE INFOCOM 87)*, pp. 814–819, San Francisco, CA.

Giovanna Di Marzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms, 1995. The messenger paradigm and its implication on distributed systems. In *Workshop on Intelligent Computer Communication (ICC)*. Also available at `http://cuiwww.unige.ch/tios/msgr/msgr.html`.

F. Brent Dubach, Robert M. Rutherford, and Charles M. Shub, February 1989. Process-originated migration in a heterogeneous environment. In *Seventeenth Annual ACM Computer Science Conference*, pp. 98–102, Louisville, KY, USA.

Joseph R. Falcone, November 1987. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, **5**(4):330–351.

Michael Steffen Oliver Franz, 1994. *Code-Generation On-the-Fly: A Key to Portable Software*. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, Switzerland. Dissertation ETH No. 10497.

Matthew Fuchs, September 1993. Dreme, for life in the net. Author's email address: `fuchs@cs.nyu.edu`.

Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad, April 1989. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, **18**(2):121–160.

James Gosling and Henry McGilton, May 1995. The Java language environment. White paper. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA. Available at `http://java.sun.com/`.

James Gosling, David S. H. Rosenthal, and Michelle J. Arden, 1989. *The NeWS Book: An Introduction to the Network/extensible Window System*. Springer-Verlag, New York.

Robert Harper, David MacQueen, and Robin Milner, March 1986. Standard ML. LFCS Report Series ECS-LFCS-86-2, LFCS, University of Edinburgh.

J. Heering, P. Klint, and J. Rekers, May 1994. Lazy and incremental program generation. *ACM Transactions on Programming Languages and Systems*, **16**(3):1010–1023.

Dennis Heimbigner, March 1995. The Tps reference manual Version 2.1a. Arcadia Technical Report CU-ARCADIA-104-95, Department of Computer Science, University of Colorado, Boulder, Colorado 80309, USA.

ISO, July 1995. MPEG-4 Proposal Package Description (PPD) — Revision 3 (Tokyo Revision). International Organisation for Standardisation document ISO/IEC JTC1/SC29/WG11 N0998. Produced by the working group on Coding of Moving Pictures and Associated Audio Information.

Dag Johansen, Robbert van Renesse, and Fred B. Schneider, June 1995a. An introduction to the TACOMA distributed system Version 1.0. Technical Report 95–23, Department of Computer Science, University of Tromsø, Norway.

Dag Johansen, Robbert van Renesse, and Fred B. Schneider, May 1995b. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, USA. Also available as Technical Report TR94-1468, Department of Computer Science, Cornell University, USA, November 1994.

David Keppel, Susan J. Eggers, and Robert R. Henry, November 1993. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, University of Washington Department of Computer Science and Engineering.

Frederick Knabe, 1993. A distributed protocol for channel-based communication with choice. *Computers and Artificial Intelligence*, **12**(5):475–490.

N. Komatsu and H. Tominaga, March 1989. A proposal on digital watermark in document image communication and its application to realizing a signature. *Transactions of the Institute of Electronics, Information and Communication Engineers*, **J72B-I**(3):208–218.

Keith D. Kotay and David Kotz, November 1994. Transportable agents. Available from authors via email to `dfk@cs.dartmouth.edu` at `http://www.cs.dartmouth.edu/papers/Kotz/agents.html`.

Mark Leone and Peter Lee, June 1994. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 97–106. Available as University of Melbourne Department of Computer Science Technical Report 94/9.

J. M. Lucassen and D. K. Gifford, 1988. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, pp. 47–57.

Steven D. Majewski, 1995. Safe-Python. `http://minsky.med.virginia.edu/sdm7g/Projects/Python/SafePython.html`. Overview of the Safe-Python project.

Henry Massalin and Calton Pu, December 1989. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelth Symposium on Operating System Principles*, pp. 191–201. ACM.

Scott McCartney and Jonathan Friedland, 29 June 1995. Computer sales hot as poorer nations try to string PC gap. *The Wall Street Journal*.

Robin Milner, 1980. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag.

Robin Milner, Mads Tofte, and Robert Harper, 1989. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

Ryoichi Mori and Masaji Kawahara, July 1990. Superdistribution: The concept and the architecture. *Transactions of the Institute of Electronics, Information and Communication Engineers*, **E73**(7):1133–1146. ISSN 0913-574X.

H. R. Nielson and F. Nielson, 1994a. Constraints for polymorphic behaviours of Concurrent ML. In *Proceedings of CCL '94*, number 845 in Lecture Notes in Computer Science, pp. 73–88. Springer-Verlag.

H. R. Nielson and F. Nielson, 1994b. Higher-order concurrent programs with finite communication topology. In *ACM Symposium on Principles of Programming Languages*, pp. 84–97.

Birger Olofsson, March 1985. *SOFTNET Programming Guide (Linköping Implementation)*. Department of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden.

John K. Ousterhout, 1994. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA.

Craig Partridge, March 1992. *Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment*. Ph.D. thesis, Harvard University, Cambridge, Massachusetts.

J. Reynolds and J. Postel, August 1987. The Request For Comments reference guide. ARPANET Request For Comments 1000, available at `ftp://ds.internic.net/rfc/rfc1000.txt`.

Eugene J. Rollins and Jeannette M. Wing, June 1991. Specifications as search keys for software libraries. In *Proceedings of the Eighth International Conference on Logic Programming (ICLP 91)*, pp. 173–187, Paris, France.

Jeff Rulifson, June 1969. DEL. ARPANET Request For Comments 5, available at `ftp://ds.internic.net/rfc/rfc5.txt`.

Michael L. Scott and Raphael A. Finkel, August 1988. A simple mechanism for type security across compilation units. *IEEE Transactions on Software Engineering*, **14**(8):1238–1239.

Olin Shivers, May 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Technical report CMU-CS-91-145.

Charles M. Shub, February 1990. Native code process-originated migration in a heterogeneous environment. In *1990 ACM Eighteenth Annual Computer Science Conference Proceedings*, pp. 266–270, Washington, DC, USA.

E. H. Spafford, June 1989. The INTERNET worm: Crisis and aftermath. *Communications of the ACM*, **32**(6):678–687.

William Stallings, 1992. *Operating Systems*. Macmillan, New York.

James W. Stamos, January 1986. *Remote Evaluation*. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology. Technical report MIT/LCS/TR-354.

James W. Stamos and David K. Gifford, July 1990a. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, **16**(7):710–722.

James W. Stamos and David K. Gifford, October 1990b. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, **12**(4):537–565.

Bjarne Steensgaard and Eric Jul, December 1995. Object and native code thread mobility among heterogeneous computers. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado.

Sun, 1992. *Sun Network Programming Guide*. Sun Microsystems. Chapters 5 (External Data Representation: Sun Technical Notes) and 6 (External Data Representation Standard: Protocol Specification).

J.-P. Talpin and P. Jouvelot, 1992. The type and effects discipline. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pp. 162–173.

Jean-Pierre Talpin, 1994. The Calumet experiment in Facile – A model for group communication and interaction control in cooperative applications. Technical Report ECRC-94-26, European Computer-Industry Research Centre, Arabellastr. 17, 81925 Munich, Germany.

Marvin M. Theimer and Barry Hayes, May 1991. Heterogeneous process migration by recompilation. In *11th International Conference on Distributed Computing Systems*, pp. 18–25, Arlington, TX, USA. IEEE.

Bent Thomsen, 1994. Polymorphic sorts and types for concurrent functional programs. In John Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, University of East Anglia, Norwich, United Kingdom.

Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, André Kramer, Fritz Knabe, and Alessandro Giacalone, December 1993. Facile Antigua Release programming guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Arabellastr. 17, 81925 Munich, Germany. Available at `http://www.ecrc.de/`.

Christine Tomlinson, Phil Cannata, Greg Meredith, and Darrell Woelk, April 1993. The Extensible Services Switch in Carnot. Technical Report Carnot-058-93, Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, Texas 78759.

Christine Tomlinson, Greg Lavender, Greg Meredith, Darrell Woelk, and Phil Cannata, June 1992. The Carnot Extensible Services Switch (ESS) – Support for service execution. Technical Report Carnot-129-92, Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, Texas 78759.

Steven R. Vegdahl, September 1986. Moving structures between Smalltalk images. In N. Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pp. 466–471, Portland, OR, USA.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, 1993. Efficient software-based fault isolation. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, November 1994. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, California 94043, USA.

Terri Watson, November 1994. Wit: An infrastructure for wireless palmtop computing. Technical Report UW-CSE-94-11-08, University of Washington.

Terri Watson, May 1995. Effective wireless communication through application partitioning. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*.

James E. White, 1994. Telescript technology: The foundation for the electronic marketplace. General Magic white paper, 2465 Latham Street, Mountain View, CA 94040.

Xerox, December 1981. *Courier: The Remote Procedure Call Protocol*. Xerox Corporation. Xerox System Integration Standard 038112.

Bennet Yee, May 1994. *Using Secure Coprocessors*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Technical report CMU-CS-94-149.

Bennet Yee and J. D. Tygar, August 1995. Secure coprocessors in electronic commerce proceedings. In *1st USENIX Workshop on Electronic Commerce*, New York. USENIX.

Lisa Zahn, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant, 1990. *Network Computing Architecture*. Prentice–Hall.

Jens Zander and Robert Forchheimer, 1984. High-level packet communication. *FORTH Dimensions*, **6**(5):32–33. The year of this reference may be incorrect.