Table of Contents

Joule: Distributed Application Foundations

300 Third Street
Los Altos, CA 94022
ph 415 941 8224
800 54 JOULE
fax 415 941 8225

Agorics Technical Report ADd003.4P

The Joule system is a foundation for building distributed applications. It combines many mechanisms already built and tested in existing products and systems. To encourage widespread acceptance and use of Joule, Agorics expects to release a public license implementation of Joule. Agorics also plans additional Joule development, to support the system as it grows, and to apply the ideas to other platforms.

Introduction

This is the technical manual for the Joule programming language. It is intended to familiarize the reader with the concepts underlying Joule, with Joule syntax, and with the fundamentals of a Joule programming environment. When you have finished reading this book, you should be able to read and create simple Joule programs.

The core of Joule is a new computational model for building distributed systems. Many of the ideas are distilled from existing systems, and could be applied at the language level, at the operating system level, or (as in CORBA) as extensions to existing languages. This manual describes the Joule programming language, a pure realization of these ideas that remains portable across all platforms (where an operating system would not). The Joule language is intended as a foundation for distributed systems, providing support in the language for many of the abstractions needed for network- or multiprocessor-based applications. Heretofore, it has been necessary to "reinvent the wheel" in many instances—to reimplement familiar techniques, tailoring them to the current special case. The goal of Joule is to provide the functionality required for distributed computing, in a straightforward and secure environment.

Chapter 1, *Foundations,* describes the intellectual origins of Joule and outlines the basic ideas on which the language is based.

Chapter 2, *Introductory Examples*, leads the reader through four simple Joule programs—the familiar factorial and compound-interest functions, plus two other servers that demonstrate some of the unique qualities of Joule.

Chapter 3, *Simple Execution Model,* describes the rules that all Joule computations must follow, and is intended to give the reader an intuition of how Joule computations could actually get work done; it is not intended to represent an efficient execution model.

Chapter 4, *Syntax,* presents an informal syntax for Joule. (For a formal syntax, see Appendix B.) Syntactic abstraction—the set of techniques for extending the Joule syntax—is discussed but not specified in this document.

Chapter 5, *Language Definition,* describes the present state of the Joule language design. It describes the computational primitives, along with typical techniques of their use, and provides a description of the syntac-

tic forms built from those primitives to directly support routine programming tasks.

Chapter 6, *Hierarchical Accounts Example*, presents a more complex Joule program, a hierarchical bank account. Hierarchical bank accounts provide a necessary component of *agoric resource management*—the use of market mechanisms to control allocation of computational resources like CPU time and network bandwidth (described more fully in Chapter 9). The program, and the underlying Joule concepts, are explained in detail.

Chapter 7, *Boundary Foundations,* describes the low-level foundations that support boundaries for creation and initiation of new programs in a running system, termination and resource management for existing programs, and access to foreign services. These foundations provide the mechanism on which the policies described in Section 5.10, *Module Programming,* are built.

Chapter 8, *Security*, introduces Joule's security foundations, many of which were drawn from or inspired by KeyKOS, a capability-based operating system, and by public-key security principles.

Chapter 9, *Resource Management*, describes managing resources in Joule. It first describes some underlying principles for resource management abstractions. It then describes facilities for resource encapsulation and transfer, the foundations for resource management. Finally, it describes market-based resource management mechanisms for making resource trade-offs in complex systems.

Chapter 10, *Distribution,* explores the issues affecting distributed systems and describes how Joule deals with them. This chapter describes support for the full spectrum of distribution regimes, from automatic distribution in which processes are automatically spread across multiple processors, to explicit distribution in which the programmer controls or influences the mapping from processes to processors, to untrusting distribution in which the programmer explicitly manages and adapts to trust boundaries and failure properties of the network.

Chapter 11, *Persistence,* describes possible implementations of persistence in Joule. The trade-offs between these implementations remain largely unexplored for Joule, though much of the territory is known for other related systems such as FCP, Actors, and KeyKOS.

Appendix A, *Language Comparison,* reviews other languages and systems relative to the requirements for robust servers and open distributed systems. It also compares the capabilities of Joule with those of its antecedents, Actors and concurrent constraint languages.

Appendix B, *BNF for Joule Syntax*, gives a description of the Joule syntax in Backus-Naur form.

The Energetic Secrets material appears in an appendix because it has not yet been integrated into the rest of the manual.

Appendix C, *Optional Arguments*, presents a proposal for managing optional arguments and "rest" arguments in messages.

Appendix D, *Energetic Secrets,* describes how SealedEnvelopes will replace Tuples in the Joule communication model, incorporating public-key semantics into the communication foundations.

Appendix E provides a bibliography of articles and books that influenced the design of the Joule programming language or that present background information on various aspects of the Joule design.

The Joule language is a work in progress, and pieces of this design will change as more experience with the syntax and computational model is gained. This book too is a work in progress; many sections remain unfinished. Some of the unfinished sections require incorporation of already developed techniques (such as the Security sections), others require significant design work (such as Agoric Resource Management).

Many thanks to the people who helped make this document and the technology behind it possible.

1.Foundations

This chapter presents the intellectual foundations for Joule: first, a brief synopsis of the history of programming languages, with emphasis on characteristics relative to distributed systems and Joule; second, a checklist of the criteria for distributed object programming languages (criteria which motivated the development of Joule); and third, a recap of some familiar, well-established design principles, showing how Joule embodies these principles in its design and supports the application of these principles to programs written in Joule.

1.1. Antecedents

1.1.1. The Rise of Modularity

From straight-line code to procedures to objects, the history of programming languages has been a history of increasing modularity to help solve increasingly complex problems. Modularity makes interfaces between pieces explicit, so that the extent to which the separate pieces interact can be controlled, then minimizes the dependencies required for a given level of cooperation. The more extreme the modularity, the more the unintended dependencies between the parts can be avoided. As systems get more complex, these interactions start to compound, placing an upper bound of complexity on the sophistication of programs and the size of a problem they can solve.

With procedures, programmers created boundaries around packages of behavior, allowing them to define procedures once and then not worry about the implementation when using those procedures. Factors such as data interactions in global environments still led to unintended interactions and a limit on the sophistication of programs.

With abstract data-types, programmers created boundaries around static packages of data and behavior, increasing the sophistication in each “black box”. Programs could now manipulate entities representing abstractions relevant to the problem being solved.

With objects, programmers created boundaries around dynamic packages of behavior and state. The polymorphism of object-oriented programming enables a much stronger separation between interfaces and implementations, allowing black boxes to hide not just the details of implementation, but also the details of which of many implementations the black box represents. The complexity limitations come from

the vestiges of global environments and the difficulty of synchronizing on shared data in a concurrent environment.

In the book *The Mythical Man-Month* **[14]**, Frederick Brooks described a study in which it was discovered that programmers wrote the same number of debugged lines of code per day no matter what programming language they were using. This observation motivated the drive towards higher-level programming languages that culminated in languages like APL and PL/1: the more and higher-level the abstractions they could make accessible, the more effective each line of code written by the programmers would be. At the same time, a much smaller thread was pursued in, for example, the Lisp community, leading through Simula, Smalltalk, and C++: the thread of abstraction languages. Instead of building in particular high-level abstractions, this new class of languages provided tools for programmers to build abstractions extending the language itself. In these languages, each line of code contributes to the level of abstraction of the next line of code. As a result, even though they start with much less capability, object-oriented systems have accumulated enough leverage that they are more effective tools for programming large systems than the high-level languages: programmers write the same number of lines of code, but they write more expressively.

By explicitly recognizing these threads of increasing modularity and abstraction power, Joule takes the next steps along both dimensions. Many of the capabilities of Joule were discovered by examining existing large systems such as networks and operating systems for the modularity tools and abstraction mechanisms which give them organization, and so make them manageable. The techniques that worked have survived into many existing systems; the techniques that failed have either fallen by the wayside or been entrenched in existing systems and demonstrate obvious failure modes.

1.1.2. Distributed Object Programming

Distributed object programming brings powerful new capabilities to computing. However, these capabilities demand the unlearning of some important paradigms of previous programming languages. The most important change required in the programmer's thinking is the abandonment of sequential call/return control flow. The sequential control flow and call/return is very natural for procedural divide-and-conquer programming in which each procedure calls other procedures in order to accomplish a specific task. The stacking behavior inherent in call/return is less appropriate for object-oriented systems in which the objects have invariants that need to be re-established before another call can be made to the object. In Smalltalk, for example, if a loop run on a collection of objects removes (or causes to be removed) an object from the collection, most times the loop operation will fail because it didn't expect the arrangement in the collection to change during the iteration.

The problems of sequential control flow and stack-style call/return are even worse in distributed object systems because such systems inherently provide concurrency and asynchrony. Sequential programming languages fundamentally cannot support distributed object systems; sequential programming languages plus external operating system support can, but the difficulty of developing applications, and the con-

tinued delicacy of communications between machines, suggest that existing tools are not well-suited to distributed object systems.

An analogy about stacks that is appropriate to objects is that stack-based programming is like a person whose work patterns are interrupt-driven. The introduction of a new task forces the current task onto the back burner. Interrupted tasks accumulate in the back of such a person's mind like calls on a stack. The mental model associated with a stack forces all operations into a LIFO queue regardless of the logical relationships between the tasks or their importance. A trivial but time-consuming task may be performed before a more important one simply because it was initiated later.

Distributed object programming is much more like managing one's time with a "to do" list. New tasks can be added to the task list without interrupting the execution of current tasks, and tasks can be interleaved without interfering with each other. Dataflow synchronization is like inter-task dependencies. It drives the ordering of tasks on the to-do list. "I have to cash my paycheck at the bank before I can buy this week's groceries, and I have to buy detergent at the grocery store before I can do the laundry." Tasks that are not logically dependent do not interfere with each other: "I can cook dinner regardless of whether I have done the laundry."

This property of distributed object programming derives from the defining characteristics of asynchrony, concurrency, and fundamental support for communications. Distributed systems are inherently asynchronous because they have to deal with events arising from multiple sources at spatially separated sites. The architecture of a distributed object programming system must be able to cope with this asynchrony (so that, for example, multiple clients can make requests of a service simultaneously). Distributed systems require concurrency because they operate on many machines simultaneously. Finally, because distributed systems must allow and encourage interaction between sites, they need to support communications abstractions. Although such support can be implemented between separate sequential "threads" at multiple locations, the sequential model adds nothing to the ease of implementing communications abstractions and distributed systems.

1.1.3. Server-Oriented Programming

Distributed object systems are rare today because building robustness on top of today's network software is difficult. *Server-oriented programming* (SOP) realizes the advantages of distributed object programming, by making the environment sufficiently resilient for distributed objects to survive. SOP applies intuitions about client/server systems to all levels of programming.

Take, as an example, a database system on a network. Multiple clients access the database across the network. These clients run concurrently with the database; they send requests, and occasionally wait for the answers, but otherwise remain responsive to the user. Such a client might access multiple databases on more than one machine.

Now, apply these same intuitions about the relationship between the clients and the database server, but within the database: there's a request-handling server for each client user, a disk subsystem, and an

indexing engine. The request-handling servers may perform translations on user queries before calling on the indexing engine with them. Within the indexing engine is a query optimizer, a B-Tree manager, and a transaction handler. Within the disk server is a process for each physical disk drive, a replication manager to protect against media failure, a transaction log, and a page reshuffler for grouping pages that should be clustered. Within each disk-drive process is a disk-arm scheduler, a disk cache manager, and a device controller.

Each of these units communicates, via well-defined protocols, with the other modules—and within each unit, sub-units communicate using other well-defined protocols. At each level of granularity, from the network application to the database to the disk handler to the device handler, the same client/server intuitions apply: separate servers can run concurrently and schedule and handle requests from other servers. Each server is a black box so far as its clients are concerned—the request handlers need not know which internal servers make up the indexing engine, so long as it responds appropriately to requests. The particulars of its internal structure are irrelevant, and may even change over time.

Proper encapsulation is a requirement for the creation of robust servers. A *robust server* is one which can guarantee continued correct service to well-behaved clients despite aberrant (that is, arbitrary or malicious) behavior from other clients. This is in contrast to fault tolerance, in which servers are able to operate continuously despite component failures (failure of other servers, or of hardware components). The kernels of traditional operating systems are designed to be robust—when one application misbehaves or crashes, the operating system is supposed to continue uninterrupted service to other applications.

The concept of robust servers is a very powerful tool with which to distinguish application platforms. They can't be built in most systems.

Attempting to implement distributed object programming over today's networks reveals problems that already exist, hidden, in single-machine systems. For example, a single misbehaving application can degrade the performance of other applications by disrupting services on which they both rely (causing "thrashing" of virtual memory, or allocating too much disk space). In a robust system, applications would be able to cope with temporary unavailability of those services and perform productively while waiting for them to return, rather than seizing up. Server-oriented programming builds tools to deal with the problems rather than just hide them.

Many of the principles of Joule were distilled from existing systems, and could be applied in more than just a language context, improving the robustness of more traditional applications.

The foundations of server-oriented programming enable extremely long-lived systems. These systems must meanwhile be able to grow and change, which motivates another defining characteristic of server-oriented programming, *open entry*—the capability of adding new components or replacing old ones, in a running system, with no interruption of service. This both requires and enables full encapsulation—a new server can replace an old one, despite having a completely different internal structure, if and only if its protocol is upward-compatible with that of its predecessor. To the clients, no change is visible.

These properties apply at all levels of a server-oriented system, enabling reliable construction of large and complex systems by assembly of well-behaved components. Properly robust servers in such systems could independently recover from failure, and communicate with each other through a well-defined interface such that they have no interactions beyond those that are explicit. The restriction of inter-

server interaction to explicit exchanges within a well-defined protocol forms the basis for real security in server-oriented systems. Security is discussed in more detail in Chapter 8.

1.1.4. Market-Oriented Programming

While server-oriented programming allows programs to guarantee the correctness and availability of computing services provided to clients, market-oriented programming enables systems to be adaptive to user and client needs and available resources by introducing *agoric resource management*. Agoric resource management uses market principles to dynamically allocate resources among software agents. By introducing the equivalent of money into the software resource management process, Joule takes advantage of the institutions and abstractions that have been developed for managing the allocation of physical goods.

For resource management issues to which markets are not well-suited, traditional centralized control (as demonstrated in the late USSR, for example) can also be constructed from the resource ownership and transfer foundations.

Markets work in the physical world because, in a sense, they already form a distributed computing system. Agents exchange goods for money, and in the process produce information about how valuable those goods are, in the form of prices. The role of money in a market system is as an abstraction which represents access to resources. Agents in a market make their decisions based on local knowledge of prices and the availability of resources. The information resulting from those decisions—what to buy and at what price—propagates through the market (the retail prices a consumer is willing to pay affect the prices which retailers are willing to pay wholesalers, which in turn affect deals between wholesalers and manufacturers). The communication of these price signals enables the whole system—the market and its participants—to allocate resources in a way that adapts to changing conditions and the different needs of different agents more effectively than could be done by any single allocating agent, based on more information than any such agent could access. (The costs of gathering and processing such information centrally would be prohibitive; much of the information would be out of date before it reached the central allocator; and in the context of mutually untrusting programs, such a central allocator might not be trusted by the participants.)

The introduction of market principles to server-oriented programming systems provides a necessary framework for efficient, decentralized management of computational resources. Local (in time or space) shortages of resources represent an opportunity for load-balancing agents—arbitrageurs—to correct the imbalance at a profit. Such agents need not violate the modularity of the programs they're helping—this resource allocation can be done through voluntary trade using client/server communications protocols, as will be seen in the next section.

Market-oriented programming relies on two concepts: the encapsulation of resources—that is, ownership by particular processes of access to blocks of, for example, memory or processor time—and the communication of access to those resources, making such ownership transferable in a flexible manner. This enables a simple initial allocation of resources among a set of providers (as described in Chapter 9) to evolve in complexity in response to the specific demands made on the system.

Encapsulation and communication of resources enable performance to be guaranteed by allowing programs to reliably purchase the rights to

particular quantities of resources at a future time. This enables servers to commit to deadlines for providing computational results to clients. Encapsulation of access (ownership) lets programs control resources; communication of access lets programmers build facilities that dynamically allocate those resources.

Object-oriented programming separated *what is to be done* (communicated by inter-object messages) from *how it is to be done* (the methods, invisible from outside the object, that are enacted in response to each message), so that the calling object need have no knowledge of the internals of the called object. A closely analogous benefit arises from the separation of resources from prices. The use of a medium of exchange (money) enables ready conversion between different kinds of resources—memory and processing cycles, for example—which would otherwise be incomparable.

The need for a particular resource varies with time and with the function being performed—suppose a particular 3D graphics rendering package is CPU-intensive. While it is running, the price of processor cycles goes up relative to memory, so other programs can adjust their budgets to rely more heavily on memory than on CPU by, for example, using more caches. If, instead, the system is currently dominated by a memory-intensive process like a drawing program, physical memory becomes expensive, making virtual memory more attractive. Resource management for complex systems needs to provide this same ability to allocate multiple kinds of resources among multiple users with diverse needs who contend for those resources.

1.2. Rules of the Game

In the context of market-oriented programming, we require a simple set of rules so that servers can interact with each other predictably. This is best illustrated by the observation that *a business can be open to the public because its cash register isn't*. Supporting such businesses requires strict, understandable rules so that participants can successfully protect their own interests while cooperating with other parties.

A computational foundation for supporting the interaction of diverse parties also defines the "rules of the game" by which those parties can interact. One never finishes learning the patterns which emerge from the rules of an interesting game, but it is important that the rules be simple enough to be understood completely, particularly if real interests are at stake.

The relevant systems are the frameworks for interaction. The C language, for example, does not support an open system because programs written in the same C address space can corrupt each other. C plus UNIX gives better support because it provides processes some measure of protection from each other. However, the continual security problems on the Internet (exemplified by the prevalence of "firewalls" that deliberately cripple insecure communications) demonstrate that C plus UNIX still does not support open systems because it is too insecure.

We define an open system as one which can continue to operate while allowing untrusted parties to "join the game," as opposed to the sense

of "open system" in which any server can get inside any other server, including its cash box.

The design process for Joule was based on finding a minimal set of rules that all processes could count on. Everything else necessary for large scale programming could be built in the framework of the fundamental rules. The computational model presented in Chapter 3 describes the rules for everything except resource management.

Joule is certainly not unique in this regard. It is unique in the set of constraints that were applied to guide the design process to a set of rules.

The checklist for Joule combines the checklists in **[65]** and **[89]**, the principles presented in Section 1.3 below, and practical issues from building large systems. Here is a partial informal checklist that drove particular aspects of the design of Joule:

- *Encapsulation and communication of information, access, and resources*

  Without this safety, businesses can't open their doors, users can't manage their resources, and groups can't cooperate.

- *Principles scale to arbitrarily large systems*

  There should be no inherent bottlenecks such as global state or inherent distributed transactions.

- *No global knowledge, control, or trust*

  These would all prevent the cooperation of agents that don't trust each other, and they are all single points of failure.

- *Robust servers*

  i.e., servers that can guarantee continued correct service to well-behaved clients despite aberrant (that is, arbitrary or malicious) behavior from other clients

- *Open entry*

  New services can be started, new customers can connect, and so forth.

- *Security*

  i.e., trust management so that services can interact while maintaining encapsulation boundaries

- *Composable correctness*

  It's possible to build something that fulfills its contract relying only on the contracts of other servers.

- *Separate resource management*

  This is the familiar principle of separation of concerns, but applied to a concern that most systems give users very little control over. This is the foundation out of which agoric resource management can be built.

- *Efficient execution*

  The model must be expressive, but must also map well to existing computer hardware architectures (for example, using message sending in Joule to implement procedures must be as fast as traditional procedure invocation).

- *Self-basis*

  It should be possible to build the distributed system in itself. If not, then the system doesn't provide sufficient functionality for managing distributed systems. Further, no single policy for how to distribute programs can be right for every application, so it

must be possible to express different distribution solutions in the language.

- *Concurrent and asynchronous*

  This is an inherent property of networks of machines that should be supported directly in the programming model.

1.3. Elements of Joule Style

This section restates some familiar, well-established design principles to show how Joule embodies these principles in its design and how it supports their application to programs written in Joule.

1.3.1. Recursive Abstractions

As described previously, the use of the client/server orientation at all levels of program design gives Joule many of the required characteristics for creating robust systems. This can be generalized to the principle of *recursive abstractions*—the use of similar organizing principles at different levels of operation. This property allows efficient scaling of code—the techniques learned for building small programs work equally well for large programs.

Another example is the principle of transparency—at all levels, Joule clients don't need to know the true nature of the server with which they're dealing; it may be a composite server or merely a transparent forwarder that chooses among several competing servers. This application of the same abstraction at different levels of granularity makes for more powerful and well-behaved Joule programs.

1.3.2. "What"/"How" Separation

This is a familiar application of the principle of separation of concerns— separating the interface of a service from its implementation to achieve better modularity. The interface specifies *what* is to be done—for example, what services a client requests from a server. The implementation—the *how*—provides the service, but the particulars of the implementation are not determined by what was requested; the internals of the server can be any implementation that conforms to the communication protocol and reveals correct results. An example of this was described above, in the discussion of separation of messages from methods in object-oriented programming. This is another organizing principle for programs which is well-suited to the capabilities of Joule.

Encapsulation is the property of Joule that hides the "how"—the limitation of interaction between processes to explicit exchanges prevents the calling process from discovering details of the implementation with which it is interacting. Polymorphism makes the "what" (the message, or the service requested) independent of a particular "how"—the server can choose internally among multiple techniques for doing the work itself, or even subcontract for the service elsewhere, with no difference apparent to the client.

1.3.3. Mechanism/Policy Separation

The *mechanism* of a particular function—the features present at the lowest level of abstraction to enable that function—should not inherently

impose unnecessary limitations on the range or application of that function. When there isn't a single "right" answer, Joule provides frameworks in which many policies can coexist. The restriction of the uses of a function—the *policies* governing its use—should instead be reserved for explicit definition at higher levels of abstraction. Caching strategies are a good example: the most effective strategy varies with how the cache is used.

The usefulness of this separation comes from abstracting from a set of desired capabilities the kernel capability which is most fundamental. An example of this is time-slicing. The fundamental capability is "determining who has control of the processor when." A system that dictates time-slicing at the kernel level is overdetermined; it rules out real-time applications, for example (as commonly defined). Joule instead treats ownership of the processor as a fundamental abstraction, allowing time to be sliced if and as needed, but also allowing for real-time applications to be built in Joule. This generality allows experimentation with other abstractions besides time-slicing, such as deadline scheduling.

Mechanism / policy separation is a way of separating things to create a new domain for distinctions. Putting the most general abilities at the bottom of a hierarchy of abstractions creates layers which can be used to determine the abilities of the layers built on them, as needed, rather than being inflexibly locked in from the very lowest layers on up.

1.3.4. Composable Orthogonality

The design of the Joule kernel is intended to separate functions along natural lines that allow the resulting abilities to be distinct, and to provide synergy when combined. The criterion for separation of functions is orthogonality: no function should partially duplicate the capability of another. This is akin to orthogonality in a mathematical coordinate system: from an orthogonal set of basis vectors, any vector in the space can be constructed more simply than from a non-orthogonal basis.

In Joule, this clean separation of powers results in smaller abstractions that give more power. Two structures that overlap in their abilities often conflict when used together in some ways. The lack of overlap between facilities in Joule prevents the elements of the Joule kernel from getting in each other's way—they can be sensibly combined in any way without conflicting. Also, if two structures overlap, it reduces the space of abilities that can be accessed by combining them—because they partially reproduce each other's abilities, less new function is discovered by using them together. The combination of two orthogonal functions, however, creates a space of new abilities inaccessible with just one of the components.

This partitioning of function is a design criterion at every level of Joule. For example, the **ForAll** statement (introduced in Section 5.3) implements multiple instantiation of code; the choose: message implements conditionals. There is no way to implement **ForAll** using choose:, or to implement choose: using **ForAll**, yet the two, combined, generate much of the Joule language. At a higher level of abstraction, resource management and concurrency are orthogonal facilities—neither can be used to generate the function of the other, but combined they give a whole new set of powerful abilities.

1.3.5. Complete Virtualizability

Anything virtualizable must be completely so, but not everything must be virtualizable. Numbers can be completely virtualizable without Tuples being so. However, in Joule, *everything* is completely virtualizable.

Complete virtualizability is one of the primary mechanisms for composable orthogonality. Because clients can only interact with servers by passing messages, other servers (middlemen) can be interposed between a client and a server to add functionality without requiring a change to either the client or the server. Virtualization only aids composability if it is complete: if programs behave differently in the presence of middlemen that weren't *intended* to change the behavior, then middlemen cannot be transparently added. Complete virtualizability enables *transparent* layering of functionality: servers only affect each other in intended ways.

One of the driving examples for transparent layering is a distributed version of Joule built in Joule. Each message from a client goes to a proxy for the intended server. The proxy turns the message into data which it sends to a handler on a remote machine. The handler on the remote machine turns that data back into an equivalent message and sends the message to the actual server with which the client wanted to communicate. Complete virtualizability means that neither the client nor the server can observe whether the network forwarder is there—if they could, then the client or the server could be written in such a way that it breaks in a distributed system but not on a single machine. The ability to observe the forwarder would require every program to take into account the implementation of the distributed system. With transparent layering, every Joule program can run across a network with *no change*.

Complete virtualizability is a very stringent requirement for the language: operations on numbers must succeed even if the "numbers" are forwarders to numbers on other machines, or user-defined servers for complex numbers, arbitrary-precision real numbers, or fractions. Message sending must work even if the "messages" are user-defined servers that merely act like messages but are actually implemented some other way. The techniques with which Joule satisfies these requirements while remaining efficient to implement have been designed. Some of these mechanisms will be presented in this document. A simple example is the primitive addition operation for Integers: if it is supplied with a non-Integer addend, it sends the +fromInteger message to the addend, supplying the original receiver (now known to be a primitive Integer) as an argument, along with the original result channel. The original addend (a complex number for instance) can then supply the behavior for adding itself to an Integer. This is not the complete story for bottoming out operations on numbers, but it demonstrates one of the simple techniques.

The completeness of virtualizability in Joule allows programmers to transparently extend functionality anywhere. They can build new transparent layers (such as the distributed system), or they can extend the functionality of any of the system abstractions (numbers, channels, messages, and so forth), while preserving the transparent layering properties of the system. Virtualizability is implemented largely through anonymity and polymorphism: servers can be distinguished only by their actions in response to messages. Security sometimes requires certification, however—you want to deposit only in your bank account, not some forwarder that might redirect your money. Joule

both supplies certification, which must be used carefully to preserve virtualizability of abstractions built with it, and provides abstraction to support virtuality in the presence of certification.

2.Introductory Examples

This chapter introduces the reader to the Joule language and its programming style and illustrates many of Joule's fundamental mechanisms in the course of explaining how some simple examples work. (Joule's underlying computational model is presented in Chapter 3.) The principles illustrated here apply at all levels of granularity—the message-plumbing techniques used here to interact with very simple servers are the same as those used with complex and versatile servers—so these examples can lead to understanding of how to construct large systems in Joule.

2.1. Forwarding and Expression Syntax

Joule objects, called *servers*, interact with each other by sending messages to *ports*. Ports can be extended transparently by *channels*. A channel is a unidirectional route originating at an *acceptor* port and terminating at a *distributor* port, each of which may be held by other servers. A server that holds the acceptor of the channel can send messages through it which can be received by any server holding the distributor.

The distributor can accept a special protocol of messages that instruct it where to forward the messages originating at the acceptor. One can think of the channel as a funnel pouring into a hose. One can pour messages down the funnel, and one can direct the hose to other funnels.

Messages are sent to a port by • (send) statements of the form • *port message*. Within some scope, the ports of a channel are named by identifiers. Typically, if A is the acceptor of a channel, A> will be its corresponding distributor. The ">" suffix is a convention used to distinguish the name of a distributor.

Messages sent to the distributor cause it to change its behavior in some way. For example, the statement

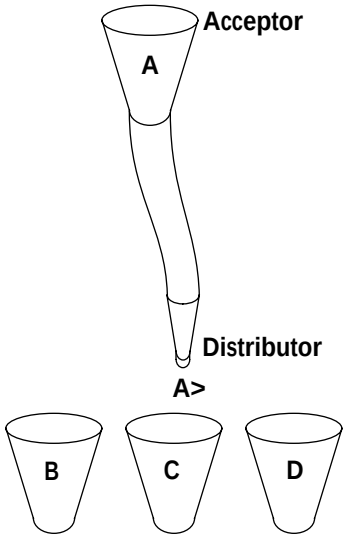
• A> → C

directs all arriving messages to the port C. This statement sends the forward message "→ C" to the distributor A>, instructing it to forward all messages received at A> to C.

Messages sent *through* the channel (via its acceptor) are forwarded to servers to which the corresponding distributor has been forwarded.

The same server can hold both ports of a channel—for example, in anticipation of passing one of them off to another server.

Fig. 2.1 • A> → C



Anything can be sent as a message, but most messages will be *Tuples* (ordered sets of ports). Tuples are the most common types of messages sent in Joule. The first element of a tuple is its name, known as the *operation*; the other elements are its *arguments*. The statement

- A oper: arg1 arg2

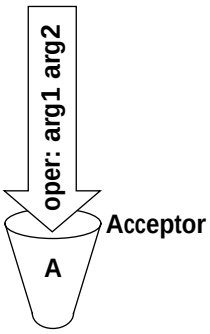


Fig. 2.2 • A oper: arg1 arg2

sends the tuple oper: arg1 arg2 to the port A (and, if A is an acceptor, through the channel to be delivered by the corresponding distributor). The colon (":") as a suffix distinguishes an operation.

Operators are a special class of operations that do not require the ":" suffix. The forward statement A> → C is actually the sending of the forward operation "→" to A> (presumably a distributor) with the port C as its argument.

In Joule, everything is a server. Numbers are also servers that respond to a set of messages. The expression

- 3 + 4 x>

Servers make use of results by sending them messages. A print server might send a number the operation meaning "give me an ASCII representation of yourself".

sends the operation named "+" to the server "3" with two arguments: the port to "4", and the distributor x> on which to reveal the result. "3" acts on the addition operation by forwarding the distributor x> to the server "7". Results in Joule are "revealed" on a distributor rather than "returned"—the calling server retains the corresponding acceptor, which now sends to the server that is the result.

Joule allows an expression-like syntax for operations like "+" that take as their last argument the distributor of a result channel. The correct way to think of Joule's expression-like syntax is to imagine an implicit intermediate result channel t1. Then

- sum> → 3 + 4

The use of "reveal" rather than "return" also reinforces the awareness of security in Joule. A Joule server need reveal only what it chooses to reveal about itself. This is discussed more thoroughly in Chapter 8, "Security."

Also, the use of "return" non-orthogonally mixes data issues with control issues. "Reveal" emphasizes that it deals only with data issues.

performs the same operations as

- 3 + 4 t1>
- sum> → t1

In practice, this looks as if "3 + 4" becomes a port to which messages can be sent. A tuple-sending statement like "3 + 4" can then be used as an argument to operations, including the forward operation:

- sum> → 3 + 4
- Fund withdraw: (3 + 4)

Precedence, in Joule, reads from right to left. In • sum> → 3 + 4, the tuple "+ 4", sent to "3", reveals as its result an acceptor to "7". The forward operation, with the result "7" as its argument, is sent to the distributor sum>, causing sum> to also deliver to "7". The forward operator "→" routes to the server "7" all messages arriving at the distributor sum>.

2.2. Dispatcher

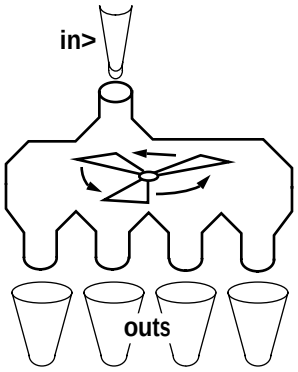


Fig. 2.3 Dispatcher

The Dispatcher server implements a simple statistical load-balancing algorithm for a number of identical servers on a network. Dispatcher receives incoming messages and forwards each one to one of the servers, chosen at random.

```
Server Dispatcher :: in> outs
    • in> → msgs
    ForAll msgs ⇒ message
        Define size
            • outs count: size>
        endDefine
        Define index
            • Random below: size index>
        endDefine
        Define out
            • outs get: index out>
        endDefine
        • out message
    endForAll
endServer
```

Dispatcher takes as arguments a distributor in> (on which the messages will arrive) and an array outs of ports to a set of servers that all provide identical services (outs is actually a port to the array, not the array server itself. For brevity, we will begin referring to acceptors interchangeably with the servers that they send to, except in cases where this could cause confusion).

Servers sending messages to outs don't know whether outs sends directly to the array, or to an array chosen randomly by another Dispatcher, and in most cases don't need to know.

Joule structures called *forms* begin with a keyword (in bold) which determines the syntactic type of the form. The **Server** form binds an identifier (in this case, Dispatcher) to a new server that executes the nested code block in response to messages from other servers. Dispatcher is a *procedural server*; it has a single method which is invoked by passing the "::" operation to the server with the appropriate number of arguments. The double colon is the simplest operation name possible in Joule; to a procedural server, it means "do what you do"—it tells the procedural server to perform its characteristic behavior.

Procedural servers are a special case of *methodical servers*. Methodical servers can accept a variety of operations (not just "::") and act on them in different ways. Random is a methodical server called by Dispatcher.

The **ForAll** form causes the nested block of code to be executed once for every message sent to the port defined by the **ForAll** as its first argument. Separate invocations are completely independent of one another and execute concurrently. The • in> → msgs forwards all messages received on in> to the **ForAll**'s input. The **ForAll** block is invoked for each message received, with message bound to that message.

msgs is a good example of a non-methodical server. As in this example, non-methodical servers typically provide message plumbing which is expected to terminate in methodical servers.

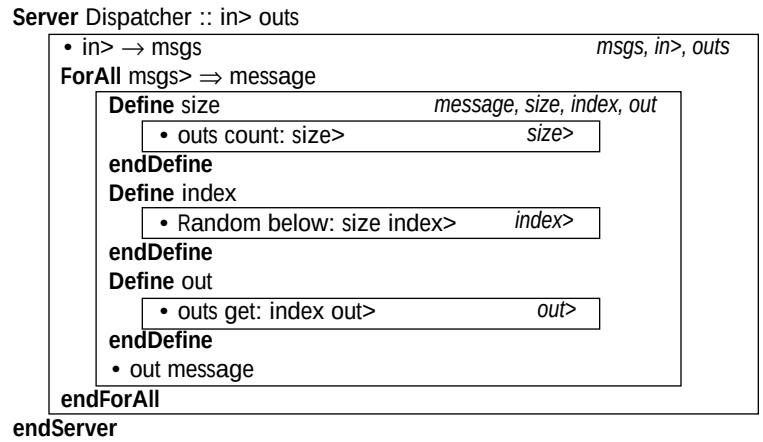
The inner scope of a Joule form consists of all lines of code between the first and last lines of the form (**Form** and **endForm**). Names defined in that block of code are visible anywhere inside that block (including the scopes of blocks nested within it), but not visible or accessible outside that block.

Lexical scoping of identifiers is discussed in detail in Section 5.3.0.

The scope of the **ForAll** form in Dispatcher includes all lines of code from **Define** size to • out message inclusive. The distributor in> is defined as a parameter by the **Server** statement and is available anywhere within the **Server** statement's scope (including inside the scopes of **ForAll** and the **Define** blocks).

The **Define** form creates a channel and binds its ports to identifiers, with the distributor bound only inside the scope of the define and the acceptor bound both inside and outside that scope. Again, distributors are distinguished by the ">" suffix. The distributor index> is defined inside the scope of **Define**; hence index> is invisible to statements at the scoping level of **ForAll**, **Server**, or • out message. The corresponding acceptor index is visible in the scope of **ForAll** because **Define** creates it in **Define**'s outer scope, but it is not generally visible in the scope of **Server**.

In the diagram below, each box represents the scope of the form surrounding it. At the upper right of each box are listed the identifiers that are visible only within that scope. Each scope also recognizes the identifiers that are visible in all boxes outside, so from the scope of the first **Define**, the visible identifiers are size> (visible only from within that **Define**); message, size, index, and out (since this **Define** is inside **ForAll**); in>, and outs (since **ForAll** is inside **Server**).



The nested blocks of code within the three **Define** forms do the work. First, the tuple count: size> is sent to the array of outputs outs. This is a standard operation for arrays; the number of elements in the array is revealed on the distributor size>—as a result, the acceptor size now sends to the number of elements in outs.

Next, the tuple below: size index> is sent to the server Random. Random is assumed to be an existing methodical server that provides random numbers. This tuple tells Random to forward the distributor index> to a random integer greater than zero and less than size. (Remember that the acceptor size is visible in the inner scope of **ForAll**, and hence also in the inner scope of this **Define**.)

Finally, the array outs is sent the tuple get: index out>. This is another standard operation accepted by arrays, telling outs to forward the distributor out> to the indexth element of the array.

More briefly, "The statement • out message sends the original message to the randomly-chosen server out." The ubiquity of transparent forwarding in Joule makes descriptions of programs extremely verbose (as you see) unless we refer to acceptors as if they were the servers to which they send.

Back in the outer scope of the last **Define**, the acceptor out is visible and now relays messages to an acceptor randomly chosen from those in outs. The statement • out message forwards message, the original message received by **ForAll**, to the randomly-chosen server at the other end of out.

Joule belongs to the class of programming languages in which statements execute concurrently, not sequentially. If a Joule statement relies on the output of another, the code expresses the dependency and that

statement will wait for the input it needs. Each line of code in Dispatcher executes concurrently.

Clearly, statements cannot execute concurrently on a single serial processor. In such an environment, the statements of a Joule program execute sequentially but in an order chosen by the compiler rather than in the order of their appearance in the source listing.

The statement Random below: size index> establishes communications links for messages to Random, and can execute whether or not size has yet been forwarded to its final value. When the **Define** size block completes, **Random** will use the result size to decide where to forward index>, but the **Define** index block has already done its part once the message plumbing to and from Random is established, and can cease execution.

Similarly, outs get: index out> could also execute, establishing the message plumbing to and from the array outs, without needing to wait for index> to be forwarded. The **Define** out block connects Random to outs, in a sense, and then can go away. The out message statement also executes concurrently, establishing message pathways in the same way.

The entire process "bottoms out" once outs reveals how many elements it has; then Random can generate a value for index, and the remaining forwards can take place, culminating in the final forwarding of message.

It's also important to note that, for every message sent to in, a separate **ForAll** is activated, and all of these activations of **ForAll** run concurrently, not sequentially. **ForAll** is not an iteration mechanism but a generator of multiple concurrent processes. See the next chapter for a more thorough introduction to the Joule computational model.

Dispatcher can be rewritten more concisely using Joule's expression-like syntax. The intermediate results channels size and out become implicit:

```
Server Dispatcher :: in> outs
    • in> → msgs
    ForAll msgs ⇒ message
        Define index = Random below: (outs count:) endDefine
        • (outs get: index) message
    endForAll
endServer
```

As a matter of programming style, this is a more attractive form of Dispatcher because the form of the code follows its function. Dispatcher does two things: pick a server at random from outs, and send message to it. This formulation has one line of code for each of these actions.

2.3. Continuous compound interest

This is a simple function that computes continuous compound interest using the formula $P + I = Pe^{\alpha t}$.

```
Reveal the interest generated by continually compounding 'principal' by 'rate' for 'time'
        time-units.
Server continuous-interest :: principal rate time total>
    • total> → principal * (e ^ (rate * time))
endServer
```

Code in italics represents comments. Like Dispatcher, the new server continuous-interest accepts the single operation "::" ("do what you do"), with arguments bound to the principal, the interest rate, the elapsed time, and a distributor for the result channel. The expression-like syn-

tax is used for brevity; assuming the intermediate channels have been defined, it could be written equivalently as:

- rate * time t1>
- e ^ t1 t2>
- principal * t2 t3>
- total> → t3

with the same result revealed on the channel total.

The continuous-interest server is invoked by statements like:

- continuous-interest :: 40000 0.15 5 result>
- result> → (continuous-interest :: 40000 0.15 5)

In either case, the distributor result> is forwarded to the number 84680.00068.

To forward a distributor is to forward all messages ever received on it.

2.4. Factorial

The next example is the familiar factorial function.

```
Reveal the factorial of the supplied number
Server Factorial :: number result>
    If number <= 1
        • result> → 1
    else
        • result> → number * (Factorial :: number - 1)
    endIf
endServer
```

Factorial takes two arguments: the number whose factorial is to be computed, and the distributor of the channel on which to reveal the result. Either of these statements:

- Factorial :: 5 foo>
- foo> → Factorial :: 5

means “calculate the factorial of 5 and reveal the result on the distributor foo>.”

The expression number <= 1 reveals an acceptor to either the true server or the false server. The server number forwards that distributor to either true or false when it receives the “<=” operation with 1 as its argument. More briefly, we may say that number <= 1 reveals either true or false.

Some extended forms like **If** have an additional layer of nested scoping between the keyword statement and the inner scopes of the nested blocks of code under the keyword and its extension keywords. The next example, Fund, discusses this aspect of extended forms more thoroughly.

The **If** and **else** statements and the nested blocks of code under each one are all part of the same **If**-**endIf** form. Such *extended forms* in Joule enable more complex program behavior, like conditional execution of code.

Depending on the value revealed by number <=1, Factorial forwards result> either to 1 or to the acceptor revealed by the expression number * (factorial :: number - 1). This recursive invocation of Factorial causes the activation of another instance of the Factorial server calculating the factorial of (number - 1).

2.5. Fund

Fund is a toy bank account used by Carl Hewitt to demonstrate properties of open systems. Fund is an example of a *methodical server*. Methodical servers are servers that respond to a fixed set of requests. Fund responds to deposit:, withdraw:, and balance:.

Fund is called a toy bank because it doesn't conserve money, nor does it prevent forging of money: the deposit: and withdraw: requests take a simple number as their argument. Fund is more like a rendezvous service that multiple cooperating agents could use to keep track of how much money had been used so far.

```
Server Fund
    var myBalance = 0

    return the current balance
    op balance: balance>
        • balance> → myBalance

    reduce the balance by an amount if that much is available
    op withdraw: amount flag>
        Define newBalance
            If amount > myBalance
                • newBalance> → myBalance
                • flag> → false
            orIf amount < 0
                • newBalance> → myBalance
                • flag> → false
            else
                • newBalance> → myBalance - amount
                • flag> → true
            endIf
        endDefine
        set myBalance newBalance

    increase the balance by an amount
    op deposit: amount flag>
        Define newBalance
            If amount < 0
                • newBalance> → myBalance
                • flag> → false
            else
                • newBalance> → myBalance + amount
                • flag> → true
            endIf
        endDefine
        set myBalance newBalance
endServer
```

Like procedural servers, methodical servers are defined using the **Server** form. There may be multiple **op** extensions to the **Server** form; each **op** statement defines one of the operations to which the server responds and specifies the arguments expected with that operation. The block of code under the **op** statement—the *method* corresponding to that operation—is executed whenever the server receives that operation; in this sense, each **op** statement is like a separate procedural server with a different characteristic operation. This entire program consists of one **Server** form, including its extension keywords and nested blocks of code.

The procedural form
Server Foo :: ...

is equivalent to
Server Foo
op :: ...

This is for the convenience of procedures, but any operation name could be used.

The term *method* comes from Smalltalk; it corresponds to the *member function* in C++.

The **var** extensions define state variables for the server. A **var** is an identifier which can be reassigned (using the **set** statement) to a different value. (It is thus unlike an acceptor—once the corresponding distribu-

tor has been forwarded, the server holding the acceptor has no control over where it sends.) A **var** is a local instance variable—it is defined only within the inner scope of the server which created it, and the value of the **var** is different in each instance of that server. In the Fund server, the **var** myBalance is set initially to zero.

The term *instance variable* comes from Smalltalk; *member variable* is the C++ term.

Joule **var**s are *not* globally-accessible locations. They are visible only to a single server and completely controlled by that server, so they do not create global synchronization problems. Furthermore, **Server** and **var** are not primitive to Joule, but are built out of more primitive constructs.

There are no global variables in Joule. Servers may interact only through explicit message passing.

The balance: operation instructs Fund to reveal the value of myBalance:

```
    return the current balance
    op balance: balance>
        • balance> → myBalance
```

The value of myBalance is revealed on the distributor handed to Fund as the argument of the balance: operation.

The withdraw: operation reveals a false result if amount is negative or greater than the available myBalance.

```
    reduce the balance by an amount if that much is available
    op withdraw: amount flag>
        Define newBalance
            If amount > myBalance
                • newBalance> → myBalance
                • flag> → false
            orIf amount < 0
                • newBalance> → myBalance
                • flag> → false
            else
                • newBalance> → myBalance - amount
                • flag> → true
            endIf
        endDefine
        set myBalance newBalance
```

Fund is a simplified version of the hierarchical bank account server Account presented in Chapter 6. In the hierarchical account, each guard of the **If** signals a different exception, rather than merely setting a success flag to false.

The **set** statement, which changes the value of a **var**, executes concurrently with the **If**. **Define** introduces the intermediate acceptor newBalance into its outer scope, so **set** can change myBalance to newBalance even though messages sent to newBalance will wait to be processed until the actual value is calculated.

This gives Joule compilers on sequential computers the option of converting the guarded **If** to a nested if—if x, then foo, else if y, then bar.

Each evaluation expression (or *guard*) of the Joule **If** form executes concurrently. However, only one of the guards that succeed gets to execute its nested block of code. If, as may happen on a sequential computer running Joule, one of the guards succeeds before another has begun executing, the system need not even bother to start up the evaluation of the second guard. Joule's **If** is a race—even if the conditions of the guards are not mutually exclusive, only a single guard out of those (if any) which reveal true gets its block of code run.

The result port flag> is used as a substitute for the normal action in such a situation, which would be to raise an exception. Exception handling is beyond the scope of this example; it is discussed in detail in Section 5.7.

If amount is negative or greater than myBalance, the result channel flag> is set to false, meaning that the attempted transaction did not succeed.

The deposit: request increases the value of myBalance by amount (if amount is not negative):

```
        increase the balance by an amount
        op deposit: amount flag>
            Define newBalance
                If amount < 0
                    • newBalance> → myBalance
                    • flag> → false
                else
                    • newBalance> → myBalance + amount
                    • flag> → true
                endIf
            endDefine
            set myBalance newBalance
    endServer
```

3.Simple Execution Model

This section describes a simple execution model for Joule. It is intended to describe the rules that all Joule computations must follow, and to give the reader an intuition of how Joule computations could actually get work done; it is not intended to represent an efficient implementation model. This simple execution model does not include resource management techniques or Domains (separately-executable pieces of code). Section 9.4 presents a more complete computational model after these concepts have been explained.

The execution model presented here owes much to the Actors execution model (**[2]**, **[65]**).

An executing Joule system consists of numerous servers sending messages to each other. Some of the servers, such as numbers, are *primitive servers*, built outside or underneath Joule; they include the basic servers from the kernel of Joule and foreign services provided externally. Execution bottoms out in these primitive servers. All other servers are *composite servers*, built in Joule from more primitive servers; they enact programmed flows of messages among primitive servers. To support the recursive abstraction of servers, primitive and composite servers have the same operational semantics so that clients cannot tell the difference.

All references to Joule servers are made via *ports*. Messages are never sent directly to a server, but rather to a port to that server. Each server can have multiple ports to it, each with a different behavior. These different behaviors are called *facets*.

The resource issue of who provides storage for messages will be dealt with in Section 9.4 in a future version of this manual. Strikingly, this particular kind of resource management problem is very similar to flow-control, which has been solved in the context of telecommunications: Tymnet and X.25 provide solutions, for example.

Messages sent to a port don't necessarily get to the server immediately. Instead, a *pending delivery* is made for the server receiving from that port. Execution proceeds by completing a pending delivery to the relevant server. The reception of the message by the receiving server and the ensuing computation in response to that message is called the *activation* of the receiving server.

Each composite server contains a collection of ports to other servers, and code to execute when activated with a message. The only ports accessible during the activation of a server by an incoming message are:

- the ports contained by the activated server
- the port to the incoming message
- ports to any servers created in the activation

The only actions a composite server can take when activated are:

- create new servers whose contained ports must be selected from the accessible ports

- send accessible port as messages to other accessible ports

These laws of computation restrict information flow among servers to message passing only; servers cannot, for example, compare ports for identity or side-affect global variables.

These laws do not directly provide the ability to change the collection of contained ports in the server; i.e., the server's state. The semantics of changeable state, so necessary for adequate modeling of real systems, is built in the language on top of the kernel semantics (and can be implemented efficiently). These abstractions (e.g., **Server** forms with **var** extensions) are described in Section 5.1.

When activated, primitive servers can perform arbitrary internal computation, so long as they respect these laws. They can contain ports to other servers, and can cooperate with each other (subject to the accessibility laws). They can change their internal state to include any other accessible servers. They cannot violate the modularity of the program by either reaching inside other servers (except cooperating primitive servers), or by referencing servers that were not explicitly accessible. Joule computation bottoms out by primitive servers cooperating with each other. For instance, integer addition bottoms out when the "+" operation is sent to a primitive integer with another primitive integer as its argument. The receiving integer gets the bits from the argument integer, computes a new result integer, and forwards the result channel to it; if the argument is not a primitive integer, then the receiving integer must send a message asking the argument to perform the addition.

The idiom of simple object-oriented message sending is as follows: the sending server, during some activation, creates a new tuple—the typical kind of server used for messages—and sends it to one of its other accessible ports. The delivery of that tuple is then pending for the server facet listening for messages on that port. When execution activates a server with the message, that server can then send to that message (considered as a tuple object). This allows it to extract the *operation* (the name of the tuple) and its arguments, for use in further computation. Because the tuple is a primitive server, when it receives messages to reveal internal parts of itself, it can do so immediately without spawning an infinite recursion of message sending.

An *operation*, in this execution model, is a unique token that can be compared with other tokens. They are not described in detail because they are replaced with public/private key pairs under the new regime described in the Energetic Secrets appendix.

Two other primitive server types, channels and arbiters, are used to interconnect servers into complex systems. Primitive Joule servers called *channels* have two facets, an acceptor and a distributor. The *acceptor* is for sending messages *through* the channel to other ports. The *distributor* is for controlling where messages sent to the acceptor get forwarded—messages sent to the distributor can forward the channel to other ports. The behavior of the channel is such that, for all messages sent to the acceptor port, a pending delivery of the message will be made for any port to which the distributor forwarded the channel.

To forward a channel is to forward all messages that have been or ever will be received on that channel.

Sending on the acceptor of the channel is equivalent to sending through the channel to each of the ports to which the channel is forwarded. This equivalence is *transparent:* sending to the acceptor of a channel is indistinguishable by the sender from sending directly to the ports of any servers to which that channel delivers. Messages sent through the channel are also preserved, so that if the distributor forwards the channel to any other ports, those ports will also get all the messages; a pending

delivery to the new destinations will be made for every preserved message.

Channels have private access to *arbiters* for choosing among messages received. Arbiters are primitive servers that are not directly accessible at the programmer level; they are implicitly accessed using the choose: operation of distributors. Supplied with a port for results and a distributor containing messages, an arbiter chooses one of the distributor's messages and forwards all of the others to a newly-created channel. It then sends to the result port a message that contains the chosen message and the distributor to the new channel. Arbiters provide the fundamental non-deterministic choice required for synchronizing access to resources. For example, in trying to model a bank account, if two clients try to withdraw the entire balance, only one can get it. Arbiters are the selection mechanism for ordering requests to provide synchronization for servers.

The **If** form is built from Arbiters.

Channels and Arbiters and the programming techniques using them are described in Section 5.1.

Servers perform the same role as objects in object-oriented programming languages; however, they differ in that they are implicitly *concurrent*, *ubiquitous* (everything is a server), and *uniform* (all behavior is in response to messages). Because all behavior is in response to messages, and because messages wait until their recipient can respond to them, Joule inherently provides data-flow synchronization. Any server can send messages on a channel even before the channel has been forwarded to any other servers. When receiving servers are created, they respond to all the pending messages.

Because delivery of messages is not immediate, any delay in the creation of the receiving server is not apparent to the client.

4.Syntax

This section presents an informal syntax for Joule. For a formal syntax, see Appendix B. Syntactic abstraction, the set of techniques for extending the Joule syntax, is discussed but not specified in this document.

In the Joule syntax presented here, typography is significant: whitespace delimits tokens and italics indicate comments. Boldface is used to denote syntactic keywords in the text, but is not semantically significant. Keywords occupy the same namespace as identifiers.

The Joule syntax presented in this document replaces a former one in which line indentation was significant.

4.1. Lexical Conventions

This section describes the token types for standard Joule programs. These include numerals, identifiers, keywords, labels, operators, special characters, whitespace, comments, literals and quasi-literals. Joule uses UNICODE for its character set.

When the UNICODE committee defines character categories such as numeric characters, identifier characters, and operator characters, Joule will adopt those distinctions. Until then, Joule will use the simplest possible distinctions.

4.1.1. Numerals

Numerals (the textual representations of the send ports to Joule numbers) are composed of the ASCII digits 0–9. No other UNICODE characters are considered “numerals” in Joule.

4.1.2. Identifiers

Identifiers are sequences of UNICODE letters, digits, and operator characters that begin with a letter, or sequences of any UNICODE characters (including whitespace) enclosed by either straight (' ') or standard (‘ ’) single quotes, with backslash as an escape character. The quotes and escapes are *not* considered part of the identifier. Case is significant. Some examples of legal identifiers are:

x	list	a>	‘an identifier’
question?	D→38a	δαιμον+	‘letter \’a\’ ’

4.1.3. Keywords

Keywords are identifiers that are treated specially. They are shown in the text as **bold**, but this representation is not syntactically significant. The syntax extension system to be described in future versions of the docu-

ment will cover this in detail. Keywords provide syntactic structure. Examples of keywords are:

Server	**If**	**•**	**Syntax+**
Case	**Define**	Δ	'Right Here'

(The send keyword •, which does not begin with a letter, is an exception to the rule that keywords must be identifiers.)

4.1.4. Operators

Operators are sequences of UNICODE letters, digits, and operator characters that begin with an operator character. Some examples of legal operators are:

+	→	!=	<-than-3

4.1.5. Labels

Labels are identifiers followed by colons (":"), or the double-colon by itself. Labels, along with operators, are used as operations (message names).

sort:	::	delta+3:	'there now\':'

4.1.6. Characters

When the UNICODE operator character declarations are finalized, the set of Joule operator characters will be extended to include additional characters like ± and ÷.

Operator characters include:
+ - * / < = > ! @ $ % ^ & | \ / ? ~ _ → ⇒

Some characters are treated specially. These include:
. ; , : # ' " ` { } [ ] ( )

"." and ";" are handled specially to support a syntax that doesn't require character attributes: identifiers that end in "." are considered keywords (without the "."), and ";" begins a comment that consumes the rest of the line. ":" is used generally as the indicator of an operation label. "#" introduces an arbitrary quasi-literal; "##" introduces an arbitrary literal.

4.1.7. Whitespace and Comments

Whitespace includes: space, tab, linefeed, CR, and form-feed.

Comments are arbitrary characters written with italic character attributes. Joule treats comments just like whitespace. Comments cannot be embedded within a single identifier.

4.1.8. Literals and Quasi-literals

Two special token types are *literals* and *quasi-literals*. Though directly supported by the syntax, these both represent expression values and are described in the next section.

Common literal types like numbers, strings, and characters are defined. Joule also supports general literals: arbitrary user-defined objects embedded in the source code by multi-media editors. The support for this is beyond the scope of this document.

4.2. Expressions

At the bottom, Joule syntax is imperative, and therefore statement-based. However, because expressions are used so frequently for math and comparison operations, a rich expression syntax is supported which transforms cleanly into the relational syntax underneath.

In brief, complex expressions become separate statements with the distributor of an implicit results channel. The site of the original expression is replaced by a reference to the acceptor of the results channel. This means that nested expressions still compute completely concurrently with their embedding statement. This transformation is described in detail in **Section 5.4**.

Below is a simplified BNF for expressions. Multiple lines in the production definition are disjunctive; thus, a simpleExpr is an Identifier, or a Literal, or a Quasiliteral, etc.

Production	Production Definition	Example
simpleExpr	Identifier Literal Quasiliteral tuple '(' nestExpr ')'	bank> 17.5 oper: arg
opExpr	simpleExpr simpleExpr *Operator* opExpr	17 3 + 17
nestExpr	simpleExpr simpleExpr opExpr	12 bank deposit: chk
tuple	*Operator* opExpr* *Label* opExpr*	+ b get: i - 1 result>

In BNF representations, foo* means zero or more instances of foo and foo? means zero or one instances of foo. Braces are used for grouping.

The apparent shift/reduce ambiguities in the grammar must be resolved by reducing (as YACC would).

Simple expressions designate particular values. These are:

4.2.1. Identifiers

Identifiers name communication ports on which messages can be sent to other Joule receivers, and which can be included in messages. These are just single tokens.

4.2.2. Literals

A literal expression statically designates a specific value that will be made available at run-time. It is represented as a single token to the compiler. Examples include numbers, shared immutable strings, and user-defined, embedded receivers (shared icons, shared print servers, etc.):

```
1234.5 ##"this is a test"
```

4.2.3. Quasi-literals

A quasi-literal expression designates a value which will be copied at run-time. These copies may incorporate literals, and run-time values. Examples include quasi-quoted lists as in Lisp, strings computed from formats as in C printf statements, and user-defined, embedded receivers.

For example, #"This is a $t1" where t1 is "Test" would result in a copy of "This is a Test".

4.2.4. Tuples

Tuples are used as messages in Joule. A tuple has a statically-available name, called an *operation*, and any number of arguments (including zero) which are other expressions. A tuple expression can be thought of as a special and extremely common kind of quasi-literal. *Operations* are either operators or labels:

```
+ 4 result>
req: arg1 arg2
```

Tuple expressions include all operator expressions following them, so they must be enclosed in parentheses (as a nested expression) in order for another expression to follow them in a line of source code.

4.2.5. Operator Expressions (opExpr)

Computation in Joule proceeds by sending and processing messages. The operator expression syntax supports conveniently sending messages commonly used in mathematical and relational expressions.

Operator expressions combine simple expressions into complex expressions using operators. Precedence is right-to-left—the first operator is applied to the first argument and the rest of the line, which will be the second operator applied to the second argument and the rest of the line.

The expression:	is interpreted as:
3 + 4 + 5 – 12	3 + (4 + (5 – 12))
a <= b * c	a <= (b * c)

4.2.6. Nested Expressions (nestExpr)

Nested expressions are enclosed by parentheses and can be used anywhere simple expressions are allowed. They support the explicit grouping as shown in the example above, and allow expressions that use label-based messages rather than just operator messages. For example, this statement

```
• x max: (y min: z)
```

sends the server x the max: operation with a single argument (the minimum of y and z). Without the parentheses, the max: request would be sent with two arguments: y and the tuple min: z.

4.3. Program Structure

Joule programs are composed of a sequence of forms. Each *form* starts with a keyword that identifies the syntactic type of that statement. Forms with the "•" keyword are used for the most frequent operation, message sending.

Syntactic extension tools allow users to associate new syntactic forms with keywords. The syntactic extension system is not presented in this

version of the manual, but many of the forms presented in later sections are actually syntactic abstractions built out of more primitive forms.

Forms typically have a **Keyword** followed by any number of operator expression arguments and ending with the corresponding **endKeyword**. In the interior of the form, underneath the keyword statement, is an optional block of more statements. Following the block are optional extension lines that have the same structure but whose keyword identifies them as part of the preceding statement. By convention, keywords that start with uppercase begin a form, keywords that start with lowercase begin extensions. A simple example is:

```
If amount <= balance
    • account withdraw: amount
else
    • account report-bounce:
endIf
```

The entire example is a single form; the **If** clause is the primary clause, the **else** clause is an extension. The **If** clause has one argument following, the operator expression amount <= balance. The nested form, • account withdraw: amount, uses the •-keyword statement form with a single argument, the tuple withdraw: amount. The nested form under the **else** extension is similar: it is a one form block using the •-keyword statement with a single argument, the tuple report-bounce:.

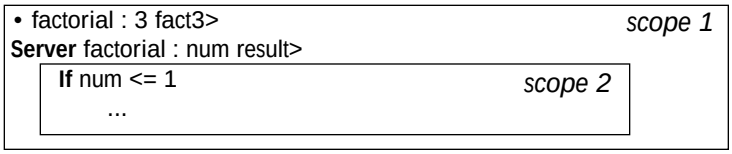
Note that tuple expressions include all operator expressions following them. For another expression to follow a tuple in a line of source code, the tuple must be enclosed in parentheses (as a nested expression).

4.4. Identifier Scoping

Any identifier may name a channel on which servers receive messages. An identifier that does so is said to be *bound* to that channel. Use of the identifier designates use of the channel to which it is bound.

Certain syntactic forms create new channels and bind identifiers to those channels. All bindings are *statically scoped*: the region of the source code in which the binding will be visible is an observable property of the source code. The same identifier may be bound to a different channel in an outer region that includes the inner region; the inner binding shadows the outer one, and is the only binding visible to code in the inner region.

Statements can create bindings in their *inner scope*—the statement itself and everything nested within it—or in their *outer scope*—the block that directly contains the statement, including all its sibling statements. The syntactic form of the statement determines where the statement makes bindings, and with what identifiers. **Server** is a simple construct that shows both kinds of binding:



Server makes a new channel for factorial requests named factorial and binds it in the outer scope (labeled *scope 1*). Each request sent to the factorial channel invokes the nested code with num and result> bound in the invoked code to the two arguments in the factorial request (roughly the channel for the argument and the channel for the revealed result). The two parameters are bound in the inner scope of the procedure statement.

Like Scheme, Joule is a statically scoped language with block structure. The block structure is represented by **Keyword**-**endKeyword** pairs; the binding site for a use of an identifier can be statically determined from the code. Unless hidden by the statement, bindings visible to a statement are also visible to statements nested within it. The multiple clauses of some statement forms may share the same inner scope, or may each introduce nested scopes only visible to that clause.

Strict static scoping allows visibility constraints to contribute to the modularity and security of the language. For instance, the **Define** construct makes a new channel and binds identifiers to its ports. The identifier supplied with **Define** is bound in the outer scope to the acceptor of the channel. This makes it visible both in the outer scope and the inner scope, since bindings in one scope are generally visible to all scopes within it. **Define** also binds a modification of the identifier (by convention, the identifier followed by ">") in the inner scope to the distributor of the channel. This distributor is visible only within the inner scope. The utility of this can be seen in:

```
Define bank
    implementation responding to messages on "bank>"
endDefine
bank clients that send messages to "bank"
```

Any number of clients can share the bank without trusting each other because their messages can only be received by the bank implementation (which they have to trust anyway). The visibility constraints of **Define** guarantee that the binding of the distributor of the channel is not visible outside the nested code within the **Define**-**endDefine** pair.

5.Language Definition

This chapter presents a complete description of the present state of the Joule language design. First, the computational primitives are described, along with typical techniques for their use. This is followed by a description of the syntactic forms built from those primitives to directly support routine programming tasks. The order was chosen to emphasize that the Joule computational primitives provide the entire rules for normal programming in Joule. Everything else described in this chapter can be built (at least semantically) on top of the primitive semantics, and can do nothing that the primitives could not do.

Resource management programming involves Boundaries as well. See Chapter 7, *Boundary Foundations*.

This follows the "hourglass" architecture principle: lots of functionality on top, lots of implementation tricks and machine specific adaptation on the bottom, and a very narrow "waistline" in the middle with a clear semantics that provides the rules that programmers must understand.

5.1. Message Plumbing

The first step in understanding the building of complex systems in Joule is to understand how programs are constructed. In Joule, programs literally get connected—much of the program design is the creation of the interconnections between servers. This section describes the mechanisms for managing message routing and response, and ways to program with these Joule mechanisms that are powerful equivalents to conventional programming techniques.

5.1.1. Sending Messages

A *simple expression* is an identifier, a literal, a quasi-literal, a tuple, or a nested expression within parentheses. See Chapter 4, *Syntax*.

The most common operation is sending a message; message sending is used for everything from adding numbers to bidding for remote database services. All values in Joule are servers, so message sending is ubiquitous. Because it is so frequent, message sending is represented in the syntax by juxtaposition after the • keyword. As a statement, • plus a simple expression, followed by any other expression, sends a message: the second expression is sent to the first expression.

Any server could be used as a message, but tuples are the mechanism to support normal object-oriented practice. Using other types of servers for messages is beyond the scope of this section.

The second expression, the message to be sent, will usually be a tuple. A tuple is an ordered list of arguments preceded by a statically-available name for the tuple, called an *operation*, which is either an operator or a label. The operation is followed by the argument list, an ordered sequence of zero or more ports to servers.

In the statement

```
• account withdraw: amount result>
```

the first simple expression is just the identifier account. The expression following it is the tuple withdraw: amount result>; the message sent to the receiver is an operation named withdraw: with two arguments. In Joule, any server can be sent as a message on a port; the expression following the receiver can be any expression. The expression withdraw: amount result> produces a tuple that is then sent to the receiver. The simple statement, • rcvr msg, sends msg—presumably a tuple—to rcvr.

In most cases, there will only be a single receiver for messages on a given channel. In this typical situation, the port to that single receiver can be treated like a pointer to that server in traditional object-oriented programming languages; that port is the capability to access the object.

Another pattern of use for channels is for the distributor of a channel to be shared. Such channels are multi-casting their messages to all the receiving servers. This works because at the distributor, messages are not removed from channels, but merely viewed, so all receivers see all messages. As result, there is no synchronization between receivers; they don't race to be the server that receives a given message. This is imperative for efficient distributed systems, because such races require expensive distributed coordination that should be written in the language rather than as a part of it.

Each receiver acts on every message in the channel. If a channel of five messages is received by six separate **ForAll** servers, thirty processes are initiated. There is no synchronization among these resulting processes; they will execute and finish in a non-deterministic order.

Also, multiple servers can send messages on the same port. Because these senders execute concurrently, messages sent to the same port by two different servers are completely unordered with respect to each other—either server could have sent its message first, so receivers of messages from that port might see the messages in either order. For example, suppose two clients of a database, A and B, send requests from their respective machines to the database server. Even if A's message is sent first in real time, B's message may arrive at the receiver before A's message does (perhaps B and the server are both in Dallas, and A is in Osaka). If A and B are not otherwise communicating about their interactions with the database, then the only ordering that any server in the system can observe is the ordering the database observes when it receives the messages in a particular order.

Because the messages are merely conveyed from the acceptor to the distributor, senders to a channel can only use it to communicate with servers at the other end of the channel, not with each other. This means that the semantics of each sender depends only on the semantics of the server at the receive side, not the other senders. This also means that senders need no expensive synchronization with each other.

An individual server can impose an order on the messages it sends. A single send statement can send several messages in a guaranteed order: receivers of the messages will see them in the order that the sender imposes. The form

```
• account deposit: amount, withdraw: amount result>
```

This simple example illustrates the need for ordered messages—there might not be enough in the account for the withdrawal until after the deposit.

sends two messages, a deposit: message and a withdraw: message, to the server named by account. The expression after each comma in a send statement is a further message that comes *after* the previous messages. This does not guarantee anything about how the messages from one

sender will be interspersed, if at all, with messages from any other servers.

Ordering messages among more than a single sender requires a different use of the same mechanism. The **then** extension to the send form forwards all the messages in a distributor to the receiver, with the guarantee that they will be received after the original messages.

```
(Assuming that the "from-seller" channel already exists:)
• account deposit: amount
then from-seller>
The seller's messages are received after the deposit operation.
• from-seller withdraw: amount result>
```

Ordered message sending is actually implemented using ordinary tuples to provide the order.

Any number of ordered messages can be sent before the **then** extension; the messages before the **then** are sent in the same order in which they appear in the program listing, while those in **then**'s distributor are guaranteed to be delivered after them.

5.1.2. Local Values and Channels

The **Define** form is the simplest mechanism for binding identifiers to ports locally. **Define** embodies two mechanisms: it binds identifiers to ports and, where necessary, it creates channels.

The simple statement

```
Define amount = 5 endDefine
```

binds amount to the port to the number 5 within the scope in which the **Define** statement occurs.

Define also creates a channel and makes the distributor of the channel visible in the inner scope of the **Define**. The name of the distributor is generated by appending ">" to the defined name. Thus, the distributor corresponding to amount above would be amount>.

As described in Chapter 3, messages sent on the distributor control the routing of messages sent through the channel (via the acceptor). The block nested within a **Define** form can forward the channel to some server and thus determine the server to which messages sent on the acceptor will be delivered, or it can pass the distributor to some other server to allow it to determine the value of the distributor. The above example, **Define** amount = 5 **endDefine**, can be rewritten to demonstrate local forwarding of the distributor.

```
Define amount
    • amount> → 5
endDefine
```

The existence of an intermediate channel is completely invisible to either the clients of amount or the server 5 because channels are transparent; messages sent through channels act exactly as if they had been sent directly to the servers to which the channel is forwarded. In efficient compilers, the two different definitions of amount will produce identical (and efficient) code.

The "→" operation with a single port as its argument tells the distributor to forward to the supplied port all messages ever received through

the channel. Since messages sent through a channel wait at the channel's distributor, messages can be sent before the channel has been forwarded to all its receivers. The delay in delivery is invisible to the message senders because message delivery is never immediate—a message sent from one machine to another takes time to cross the network. Message plumbing and dataflow synchronization allow programs to be built without immediacy requirements.

This simple example demonstrates passing the distributor to another server to allow that other server to determine the receiver of messages sent on the acceptor.

```
The following passes the distributor to another server
Define result
    • account balance: result>
endDefine
```

This example passes the distributor, result>, as an argument to the balance: operation to account. This allows account to forward the distributor to the server which is its balance, and so bind the result to the balance of the account. All messages, past or present, sent to result will arrive at the server that is account's balance.

The statements within a Joule form all execute concurrently. The linear form of the textual representation of a Joule program may give the appearance of sequential execution, but this is emphatically not the case. As a result, **Define** statements at the same level of scoping can use one another's acceptors in their definitions. This was used in the verbose form of the Dispatcher example in Section 2.2:

```
Define size
    • outs count: size>
endDefine
Define index
    • Random below: size index>
endDefine
Define out
    • outs get: index out>
endDefine
```

These three **Define** forms could be in any order without changing the operation of the program.

Also, because the outer bindings are visible to inner scopes, the name being bound in a single **Define** can be used in defining the binding of the name. This is commonly used in the definition of recursive functions. The following toy example makes the syntax clear:

```
Define ones
    • ones> → foo: 1 ones
endDefine
```

The acceptor ones now delivers to a tuple named foo: which has two arguments, namely the number 1 and a tuple named foo: which has two arguments, namely the number 1 and (effectively) a tuple named foo: which has two arguments...

Mutually recursive definitions are also supported: a single **Define** form can bind multiple names (separated by commas). The distributors for

those names are visible in the entire statement, so each definition can use the acceptors or distributors of the other definitions.

```
Define population, growth
    mutual-recursion :: population growth population> growth>
endDefine
```

The bindings of the distributors hide the bindings of the acceptors, so in

```
Define a, a>
    body
endDefine
```

the body will see two distributors, a> and a>>, corresponding to a and a> respectively.

The nested block of code under **Define** (and any further nested block within it) can use any or all of the ports thus **Define**d; this example uses them all.

Finally, the simple form of definition (**Define** acceptor = expression **endDefine**) and the more complex form (with a nested code block) can coexist. The distributor created, and thus all messages delivered to it, are forwarded to all of the results of the definitions. Logging is a simple example that starts to use the power of message plumbing:

```
Define amount = 5000
    Logger record: amount>
endDefine
```

The Logger here is just some server that records all the messages that arrive on amount>. The messages sent on amount are sent to both the number 5000 and whatever internal server the Logger server started.

5.1.3. Composite Servers

The **ForAll** form is the foundation for building composite servers. When a **ForAll** is executed, it creates a new composite server that will invoke the same block of Joule code for every message that it receives through a particular channel.

```
ForAll in ⇒ msg
    body
endForAll
```

The new server created by the above **ForAll** form will activate body once for every message sent through in. The identifier msg will be bound to the message in each independent activation. The new server will contain the ports bound to visible identifiers defined outside the nested body but used within it. When the new server is activated, it can create new servers (using **ForAll** for example), and send messages, following the accessibility restrictions described in Chapter 3, the execution model.

Because servers created with **ForAll** are not allowed to change the set of ports they can access, they cannot remember anything, and cannot implement (by themselves) servers with mutable state. They are the foundations for immutable servers such as complex numbers and procedures. For instance, the introductory example Dispatcher (Section 2.2) that randomly distributes messages among multiple output ports:

```
Server Dispatcher :: in> outs
    • in> → msgs
    ForAll msgs ⇒ message
```

```
            Define index = Random below: (outs count:) endDefine
            • (outs get: index) message
        endForAll
    endServer
```

could be expanded to:

This doesn't properly check that the operation is "::".

```
    ForAll Dispatcher ⇒ tuple
        • in> → msgs
        Define in> = (tuple get: 0), outs = (tuple get: 1) endDefine
        The above parameter extraction is merely suggestive.
        Now the body of the procedure, which also uses ForAll
        ForAll msgs ⇒ message
            Define index = (Random below: (outs size)) endDefine
            • (outs get: index) message
        endForAll
    endForAll
```

The first part of the example is an expansion of the simple use of the **Server** form—for every tuple, execute the nested body of code in a block with in> and out bound to the two arguments of the incoming tuple. This is clearly an oversimplified variant of the expansion; it has no error checking, it requires two operations to get all the parameters, and so forth. However, it does serve to show how the more powerful forms in the language can be semantically defined in terms of the communications primitives and the primitive servers.

5.1.4. Making Decisions

As mentioned in the execution model, decisions are made by Arbiters. An Arbiter chooses among the messages it receives. Supplied with an acceptor for results, and a distributor containing messages, it chooses one message, and creates a new channel to which are forwarded all the messages not chosen. It then sends to the supplied acceptor a message that contains the chosen message and the distributor to the new channel.

Decisions get made by Arbiters in several different circumstances. For **Server**, Arbiters select an ordering for unordered sets of messages—choose one message, process it, make another Arbiter, choose the next message, and so on. For the **If** form, Arbiters select the branch to take in an **If** form with multiple independent branches whose guards are all true. Other Joule forms use Arbiters similarly.

The Arbiter concept exists primarily as an aid to intuition about decisions—Arbiters don't necessarily ever exist in the implementation or as objects in the programming language. Arbiters are created by channel distributors when they receive the choose: message. The Arbiter chooses a message from the channel, and forwards the rest of the messages to a new channel.

The channel to which the choose: is sent may have multiple receivers; its content remains unchanged. Multiple Arbiters on the same channel choose independently: they could choose the same message or they could choose completely different messages. These two principles combine to avoid the need for any synchronization or coordination between multiple receivers. The multiple receivers can choose messages, forward messages, and so on, without needing to synchronize with other

receivers on the same channel. As a result, the only synchronization necessary and the only communication possible are between senders and receivers. This simplifies the semantics of the language and the implementation—particularly in distributed systems in which synchronization is expensive—without reducing the power at all. Programmers can build any synchronization mechanisms they desire from these primitives.

5.1.5. Making Decisions Easier

Though the choose: operation is the simplest mechanism, the **ForOne** form is a much easier way to understand Arbiter behavior. **ForOne** creates a new server that will execute a block of code for exactly one of the messages it receives. **ForOne** non-deterministically chooses one of the incoming messages (using choose:) and defines a new channel that contains the rest of the messages. (The original channel may have multiple receivers; its contents remain unchanged.) This "rest" subset can be redirected to other servers. The block of code in the **ForOne** server will have access to not just the message but also the distributor for a channel of the rest of the messages; i.e., all messages not chosen. As with **ForAll**,

```
ForOne in ⇒ one rest>
    block
endForOne
```

ForOne defines a port for incoming messages, while one and rest> are identifiers that will be bound in the nested block when that block is activated for a chosen message. The **ForOne** can expand to a use of the choose: message and a procedure

```
Define in
    • in> choose: choice
    Server choice :: one rest>
        block
    endServer
endDefine
```

The choice procedure is just to hide the extraction of the chosen message and the distributor with all the other messages. The choice procedure could expand further out to a **ForAll** and explicit extraction of one and rest>.

5.1.6. Receiving with ForOne

ForOne is suitable for the definition of mutable servers. The server chooses one message with **ForOne**, computes a new state based on that message, and recurs with that new state on the rest of the messages in the distributor. This use of **ForOne** imposes a particular full-ordering on the partial ordering of messages sent on the channel, properly synchronizing access to the mutable state of the server.

Receiving with choose: or **ForOne** subsumes the Actors computational model, which requires the semantically more complicated become operation.

Using choose: to provide a full ordering for a partially ordered set requires choosing a message, processing it, and then choosing further messages. The **ForAll** form is the primitive that allows code to be used more than once, so it will be used (inside of **Server**) to invoke the choose: as many times as necessary. When a **Server** form has mutable

state, it expands to a use of choose:. This can be seen by expanding a subset of the Fund example:

```
Server Fund
    var myBalance 0
    op balance: result>
        • result> → myBalance
    op deposit: amount success?>
    ...
endServer
```

The real expansion of the **Server** form is complicated by handling message ordering and multiple input channels. It will not be presented in this document.

which could expand to

```
Define Fund
    • Fund-recursion :: 0 Fund>
    Server Fund-recursion :: myBalance in>
        • in> choose: choice
        Server choice :: operation rest>
            Switch operation
                case balance: result>
                    • result> → myBalance
                    • Fund-recursion :: myBalance rest>
                case deposit: amount success?>
                ...
            endSwitch
        endServer
    endServer
endDefine
```

For a more detailed description of recursion, see Section 5.6.

The **Define** Fund creates the Fund channel. Inside the definition is the nested procedure, Fund-recursion, that implements the Fund server. The arguments for that procedure are the state variables for the server and the incoming request channel—myBalance gets initialized to 0, the input stream is initially Fund>.

The Fund-recursion procedure immediately waits (using choose:) for a tuple to be sent to Fund. When a tuple is received, the expansion here uses a **Switch** form to dispatch to appropriate code based on the name (or *operation*) of the incoming tuple and its arguments. The code for balance: (called the balance: *method*) is the supplied code plus a recursive call to the Fund-recursion procedure to process the rest of the messages sent to Fund. In each recursive call, the method could call Fund-recursion with a completely different amount, thus changing the state of the server Fund from the perspective of all its clients. A simple example is the following method that would zero the balance of the Fund.

```
op clear:
    set myBalance = 0
```

The clear: method just sets myBalance to 0. The expansion is just another clause of the **Switch** form used for dispatching on the incoming message.

```
case clear:
    Fund-recursion :: 0 rest>
```

In response to the clear: message, the call to Fund-recursion says to process further messages in a Fund that has a zero balance.

5.2. Methodical Servers

Methodical servers are like objects from traditional object-oriented programming languages: they respond to a specific repertoire of messages which each invoke a different behavior, called a *method*. For well-defined servers, the set of messages to which they respond, called their *signature*, satisfies a contract that clients can count on. Some elements of the contract can be specified in the language; these are captured in machine-verifiable ways in definitions of signatures called *Types*. Other elements are defined at a human level of understanding; these are currently relegated to comments. The **Type** form describes the specification to which methodical servers of that type must conform; the **Server** form describes a single implementation meeting that spec. This section presents the tools for defining servers and their types.

Put another way, **Type** forms describe the "what" of methodical servers, while **Server** forms describe the "how".

While many languages support the equivalent of methodical servers, few support non-methodical servers such as transparent forwarders. Non-methodical servers respond generically to messages, passing them through to other servers or processing them without regard to the particulars of each message. Chains of non-methodical servers typically terminate at a methodical server which provides the semantics to clients of the server. In addition to the general support for message plumbing (all of which produces non-methodical servers), the **Server** form supports the definition of methodical and partially methodical services.

The **Server** form supports several object programming techniques, many of which are abilities enabled by the flexibility of communication in Joule:

- immutable servers—servers that don't change state, such as procedures and complex numbers
- mutable state—standard mutable servers, but designed to work properly in a highly concurrent environment
- partially ordered messages—represents the potential concurrency among client requests
- multiple facets—Servers can have multiple input channels with different behavior on each. This supports private method groups and servers that present different facets to different clients (such as channels do).
- non-methodical servers—Servers can respond to messages generically, logging them, forwarding them, animating their delivery in a debugger, etc.

The **Type** mechanism supports

- compile-time implementation checking—Servers that claim to implement a type get checked at compile time.
- run-time checking—Servers that allege to be of a certain type can be verified to implement that type.
- default implementations—Types can describe parts of their interface in terms of other parts of their interfaces. These descriptions act as default implementations and provide further definition of

the contract that the type abstractly represents.

- inheritance—Types can inherit from other types. We believe this simpler mechanism, combined with object facets (see Section 5.2.2) will provide all of the power associated with implementation inheritance while avoiding some of its problems.

The syntax and type system will be extended to allow parameter types and local binding types to be declared.

5.2.1. Syntax

The **Server** form defines a single server. To create multiple instances of a particular kind of server, its definition can be nested inside another server; the containing server can then create a new instance of the contained server every time it is called. The syntax of the **Server** form is

Production	Production Definition
server	**Server** param {method}? {var}* ops {facet}* **endServer**
var	**var** {param | param = opExpr},* block
ops	{**implements** *Identifier*}? {**op** method}* {**otherwise** param block}
method	{pattern}**or**+ block {change block}*
change	**to** *Identifier* {opExpr},+ | **set** {*Identifier* = opExpr},+
facet	**facet** param ops

In BNF representations, the construct {bar}foo+ means "one or more instances of bar, separated by foos." For example, {pattern}**or**+ means one or more patterns separated by **or**s. The full BNF is presented in Chapter B, *BNF for Joule Syntax*.

The identifier following the **Server** keyword is bound in the outer scope to a port to the newly created server. Following the identifier is an optional method definition. This is primarily to conveniently support procedural servers. Method definitions will be described below. Following the optional method definition is the declaration of any mutable state for the server. The **var** declarations create the instance variables to represent this state. In this, **var** extensions act like the **Define** form—the identifier is bound either to the optional expression or to an acceptor through a channel that gets connected to an initial value in the nested body. The instance variables are only visible within the body of the server definition. Methods in servers with mutable state can rebind the identifier to other ports.

The instance variables in servers not actually locations like variables in C; for instance, **ForAll**s nested in a server method will see an unchanging snapshot of their containing server's state.

The King server might be an element of a Joule chess program, with instance variables describing its position and status:

```
Server King
    var myPosition = K1
    var check? = false
    op move: newPosition
        Define resultPosition
            If (rulebook allow: king myPosition newPosition)
                • resultPosition> → newPosition
                ...
        set myPosition = resultPosition
    op ...
endServer
```

5.2.2. Facets

Servers may have many *facets*—named channels on which they receive and respond to messages according to some contract. The identifier that follows the **Server** keyword names the primary facet of the server. Other facets, and thus other named input ports, are introduced with the extension keyword **facet**. The identifier following a **facet** keyword is like the identifier after the **Server** keyword: it names a newly defined port in the outer scope of the **Server** form. The method definitions for the primary facet appear after the instance variable declarations; for other facets they follow the facet declarations. Primary and secondary facets are semantically equivalent.

Most servers only have a primary facet. The most common kind of secondary facet is the private facet, a facet not exposed beyond the procedure that creates the server. Private facets are used for methods that should not be available to clients.

Before the method definitions in a facet there can be an **implements** declaration naming which **Type** the facet satisfies. The type named in an **implements** extension must be a type defined with the **Type** construct specified below. A facet with an **implements** declaration must implement all the methods specified by that type except for methods whose **Type**s define default implementations; the default implementation will be used if the method is not redefined by the facet. A facet may implement additional methods that are not part of the declared **Type**. Only one pattern is allowed for a given operation name.

For patterns that match a variable number of arguments, see Appendix C, *Optional Arguments.*

After the optional type declarations come the method definitions. Method definitions are introduced with the extension keyword **op**, except for the optional method definition immediately following the primary facet declaration.

A single method can actually respond to more than one message pattern using the **or** extension. Details of **or** will be presented in future versions of this document.

Method definitions begin with a pattern against which incoming messages are matched. Patterns are prototypes for the corresponding message: an operation name followed by named parameters which will be bound to the corresponding arguments in the message. For each message sent to the server, whichever method pattern is matched by that message is the one activated for it. Following the pattern in each method is the nested block of code to run for each activation.

5.2.3. State Change

Methods in servers with mutable state can include the extensions **set** and **to** which change the server's state. A method can have any number of these extensions, in any order.

The **set** extension designates an instance variable (previously declared with **var**) and an expression to which the instance variable should be rebound. The **to** extension designates an instance variable and a message to send to the current value of the instance variable. Messages sent with **to** are ordered sends: after the value of the instance variable receives the message, the instance variable is rebound to a new port containing messages guaranteed to be delivered after the message that was sent with **to**. As a result, messages sent to an instance variable during an activation are guaranteed to be delivered before messages sent to the same instance variable in a later activation.

The **to** extension can be defined in terms of the **set** extension and ordered message sending with **then**. The following example is part of

an account server that contains a Fund server and implements methods in terms of it.

```
Server account ...
    var myFund ...
    deposit amount and reveal the new balance.
    op deposit-balance: amount success?> balance>
        to myFund deposit: amount success?>
        to myFund balance: balance>
        ...
endServer
```

is equivalent to

```
Server account ...
    var myFund ...
deposit amount and reveal the new balance.
    op deposit-balance: amount success?> balance>
        Define fund'
            • myFund deposit: amount success?>
            then fund'>
        endDefine
        set myFund fund'
        to myFund balance: balance>
        ...
endServer
```

The first implementation just sends ordered messages to the contained Fund server. The second example takes exactly the same actions: the deposit: operation is sent to myFund with the amount, and an implicit channel of messages is created with **then**—messages guaranteed to arrive after the deposit: operation is received by myFund. Thus, it is guaranteed that the subsequent send of balance: to the fund is received by the fund after the deposit gets made.

The **to** extension is also used to send messages to the server itself. Joule supports a distinction between inner and outer selves that is not possible in sequential object-oriented languages. Sending to the inner-self means sending messages to a facet that will be processed before any further messages from the outside are processed. These get used when the messages to self are part of maintaining the invariants of the server. For example, the move-window: operation for a window in a windowing system might erase the window, change its coordinates, then draw the window again; no client messages should be able to get between the erase: and redraw: operations because they could easily break invariants that assume the window is currently displayed. A message to the inner-self is sent by using the **to** extension with one of the facet names as the designated receiver of the message.

Sending to the outer-self is accomplished by sending messages to a facet port just as if it were a regular port (without using the **to** extension). The outer-self is for the server to send messages to one of its facets that should be interpreted as if it came from a client. For example, deleting elements of a collection while iterating over its elements breaks most object-oriented systems (because most iteration schemes depend on representation details that are altered by deletion). If the deletion operation is sent to the outer self, it won't be received until the

iteration is finished, allowing the deletions to avoid interfering with the efficient implementation of iteration.

The optional block following a state-change extension executes with the server in the state it possesses after the variable has been rebound. In the nested blocks, passing an instance variable as an argument or sending it as a message refers to the new value, not the old value. A simple example is adding the deposit-balance: message to the Fund server:

```
Server Fund ...
deposit amount and reveal the new balance.
    op deposit-balance: amount success?> balance>
        to Fund deposit: amount success?>
        • balance> → myBalance
```

The myBalance reference that is revealed on balance> is the value of myBalance after the deposit has increased the balance.

Scoping is different for these state-change extensions. The entire method, including all the state change extensions, is a single scope, with the exception that instance variable definitions refer to different values after state change extensions. This is more consistent if state change extensions are viewed as extensions to the **op** extension rather than as extensions to the **Server** form itself.

The optional **otherwise** extension follows the method definitions for a facet. It supports non-methodical and partially-methodical servers. If an incoming message matches none of the methods for a facet, then, if that facet has an **otherwise** extension, it is invoked with the identifier bound to the unrecognized message. If there was no **otherwise** extension, the not-understood: exception is signalled. See Section 5.7 for details on exception handling.

5.2.4. Type

Future versions may extend the type system with multiple inheritance of types.

The syntax for declaring types is very similar to the syntax for defining servers. Types can not have variables or **otherwise** clauses. Types form an inheritance hierarchy, so they have the optional **super** extension to specify the parent type. The standard root of the type tree is Basic, a type that defines a very simple protocol appropriate for most servers; however, servers need not be subtypes of Basic. The syntax for **Type** declarations is:

The Basic type is defined along with other standard types in Section 5.8.

Production	Production Definition
type	**Type** param {**super** *Identifier*}? {**op** {pattern}**or**+ block {**to** name {ex},+ block}*}* **endType**

Everything after the pattern of a method is the optional default implementation. The messages defined in a type declaration are not messages that the type itself responds to; types respond to a fixed set of messages for asking about their protocol and such. The only change extension allowed in default implementations is the **to** extension. It can

be applied with either Self to send message to the inner-self, or Super to invoke overridden default implementations in the parent **Type**.

The following example defines the type for the simple Fund example presented in Section 2.5—an account in which money is not conserved, but with which trusting processes can keep track of money.

```
Type Fund
    super Basic
    op balance: balance>
    op withdraw: amount flag>
    op deposit: amount flag>
endType
```

The **Type** statement introduces the type named Fund into the type namespace. The **super** line says that facets that implement type Fund must also implement type Basic. Lines beginning with **op** declare messages to which instances of the type will respond. Following the **op** is the message pattern that will be matched, typically an operation followed by arguments.

5.2.5. Nested Servers

Servers can be nested. The simplest use of this is to make a procedure that will produce instances of a server implementation. The Fund example might instead be:

```
Server make-fund :: balance fund>
    • fund> → Fund
    Server Fund
        var myBalance = balance
        ...
    endServer
    …
endServer
```

Each invocation of make-fund with a balance produces and reveals a new and independent Fund server.

To the nested Server, the instance variables of the parent are unchanging; they remain bound to the same port as when the nested server was created. This is of course also true for nested **ForAll**s.

5.3. Procedures

Procedures in Joule are servers that respond to the procedure operation convention—any message named with a double colon ("::"), the shortest message label. The **Server** form supports the easy definition of procedures with the optional method definition immediately following the primary facet name. For procedures, the primary facet name, the identifier following the **Server** keyword, is the name of the procedure. Procedures can be defined using **op** extensions. Defining a method on the first line of the **Server** form is a syntactic convenience; a method so defined is no different than one defined using an **op** declaration.

Because Joule has true lexical scoping, all servers including procedures can be defined within other procedures. This allows the definition of

private helper procedures inside methods of a more complex server, for instance.

5.4. Functions and Expressions

The Joule syntax supports expressions, primarily as a convenience for common math expressions and tests for conditionals. This section describes how to implement result-revealing functions using the **Server** construct and passing in a distributor to return the result. It also describes how Joule expressions that resemble expressions in other languages ("3 + 5") expand into more primitive forms.

The "native" technique for using operators in Joule is the explicit sending of the operator request. For example, in the following statement

```
• 24 <= 60 small-enough?>
```

passes the distributor small-enough?> in a message to 24, which forwards small-enough?> to the result of the inequality (in this case, the Boolean server true). The distributor for the result was specified explicitly; the corresponding acceptor delivers its messages to the result.

However, operators may also be used in an expression context—that is, anywhere that a Joule expression would occur; for example, as the target of a send, or as the argument to a forward operation. Many examples of this type of usage have already been shown; consider the statement • sum> → 3 + 4. Any operator used in such an expression context is assumed to be sending an operation with two arguments: the expression immediately to the right of the operator, and an implicit distributor for the result. The operator expression is replaced with the acceptor for the implicit channel. One could accomplish the same thing by first defining an intermediate result channel t1 and executing the statements

```
• 3 + 4 t1>
• sum> → t1
```

In practice, this looks as if "3 + 4" becomes an acceptor to which messages can be forwarded. An operator expression like "3 + 4" can then be used as an argument to operations without the need to explicitly define channels for intermediate results. Parentheses can be used to force the expression-like evaluation of tuples that are named with labels instead of operators, as in

```
• should-be-120> → (Factorial :: 6)
```

The Factorial procedure was presented and explained in Section 2.4. The **Server** form for Factorial looks like this:

```
Server Factorial :: number result>
    If number <= 1
        • result> → 1
    else
        • result> → number * (Factorial :: number - 1)
    endIf
endServer
```

The final argument result> is a distributor for a result channel. Whenever the server is sent a message in an expression context, the Joule compiler automatically creates the implicit channel and supplies that distributor as the final argument. This is completely transparent so far as the called server is concerned; there is no difference between a distributor supplied explicitly by the programmer and one supplied implicitly by the compiler.

5.5. Conditionals

Joule supports several constructs for making decisions. The most familiar is **If**, similar to Dijkstra's "guarded-if" in which each condition expression, called a *guard*, is executed concurrently. The **If** construct is extended with generalized pattern matching, incorporating some ideas from logic languages. The second construct is **Switch**, much like C's switch statement, in which an input value (typically) is matched concurrently against a set of patterns, and the code associated with the matching pattern is executed. Finally, a process can decide among the results of several input processes by having their outputs race to be the first producer of a value. This is the fundamental semantics underlying all the conditional constructs in Joule, and is sometimes useful directly.

5.5.1. If

If is similar to Dijkstra's guarded-if construct. The **If** is a series of clauses which each have a guard and a block of code. The **If** construct executes all the guards concurrently. Of the guards that evaluate to true, the **If** construct executes the block of exactly one of them; the guards that evaluate to true race, and only one of them can win. The **If** construct also has some special clauses (like **else**) with implicit guards that participate in the same race. Here is a simple example:

```
A simple example of the If construct
If withdrawal > balance
    • withdrawal report-bounce:
orIf withdrawal < 0
    • account bad-arguments: withdrawal
else
    • account withdraw: withdrawal
endIf
```

An **If** form is a sequence of guarded clauses and a final optional **else** clause. Guards execute concurrently, and their execution is total; that is, they will either be executed to completion or not executed at all (that is, the compiler is allowed to rewrite the decision tree). The guards race to win the **If** and have their associated block of code run; the **If** will only run the block of code of the winner (if any) of the race.

There are two kinds of guards: expressions and pattern matches. An expression is just an operator expression that must evaluate to true to win the race. A pattern match is a simple expression, the target, followed by "~" followed by a pattern expression (a quasi-literal). If the pattern expression contains free identifiers, they will be bound to the corresponding part of the target in the associated block of code for the pattern match guard. The guard succeeds by successfully matching the pattern. The details of underlying pattern matching implementation

will not be discussed in this document. The basic implementation is that the pattern match syntax translates to sending the match: message to the target with an argument for each free variable and an extra result argument for the success flag that will participate in the race. Thus, pattern match guards also evaluate to true. Finally, the **else** clause is true only if all the guards are false. Therefore, the **else** doesn't need to participate in the race.

The **If** will commit to one of the guards that reveals true. It may not be the first guard because there isn't any well-defined notion of "first" except "the one chosen by the **If**": in a distributed system, the first guard to evaluate to true may be on a machine remote from the commit location, and by the time its success is communicated to the rendezvous site, a closer guard committed and won the race. If two guards evaluate to true simultaneously, the implementation will choose nondeterministically. In accord with totality, any guard computations not already started when the **If** commits may never be started by the implementation.

The BNF for the **If** construct is as follows:

```
If opExpr
      block
{orIf opExpr
      block}*
{elseIf opExpr
      block
{orIf opExpr
      block}* }*
{else
      block}?
endIf
```

Before the guards have computed enough to have revealed a value, the **If** is suspended. If all the guards evaluate to false, and there is no **else** clause, the failed-if: exception is signalled. See Section 5.7 on page 56 for details on exception handling.

As this example shows, the **elseIf** extension is exactly equivalent to an **else** extension containing a nested **If**:

```
If ex1
      block1
elseIf ex2
      block2
else
      block3
endIf
```

```
If ex1
      block1
else
      If ex2
            block2
      else
            block3
      endIf
endIf
```

5.5.2. Switch

The **Switch** construct is used to choose one of several blocks of code based on pattern matching against a single target. This is a convenience

for large scale pattern matching, and is used to dispatch on messages in the expansion of the **Server** form. The syntax for the **Switch** form is:

```
Switch opExpr
      {case pattern
      {or pattern}*
            block}*
      {otherwise param
            block}?
endSwitch
```

The argument of the **Switch** statement is an expression that reveals the target of the pattern matches. The argument of each **case** or **or** extension is the pattern to be matched. The pattern is a quasi-literal, just as in the pattern-match **If** guard. Any free identifiers in the pattern will be bound to the corresponding substructure of the target in the block of the pattern that wins the switch. The **otherwise** clause will be run if all of the pattern matches fail. As with **If**, if no pattern matches and no **otherwise** extension is supplied, the failed-switch: exception is signalled.

This example of the **Switch** statement is from the meta-interpreter for Joule. A meta-interpreter is an interpreter for the language written in the language.

```
Switch statement
      case define: names block
            • interpret :: block (env attach: names)
      case send: recT tupleT
            • (env lookup: recT) (env lookup: tupleT)
...
endSwitch
```

The interpreter uses tuples to represent program structure, and a **Switch** form to match the incoming tuple and extract the arguments. It then takes the appropriate interpreter action to execute the particular statement type.

5.5.3. Race

Race isn't a construct, but rather a way of using the primitive choose: facilities for making decisions in a program. Several clients can send their results to the same acceptor. The choose: message, sent to the corresponding distributor, then picks exactly one of the incoming results so that it can be operated on. This is the primitive in the language for choosing among alternatives. The other decision constructs are implemented with it, but it is sometimes directly useful.

5.6. Iteration

Joule supports iteration through recursion. Simple functions can recur by calling themselves with other arguments. If a result argument is passed through the recursion, then the nested call can determine the result for the computation.

```
      Reveal the new total after interest on 'principal' accumulates at 'rate' for 'units' time
            units.
Server acc-interest :: principal units rate total>
      If units > 0
```

```
            Define sofar = principal * rate + 1 endDefine
            • acc-interest :: sofar (units - 1) rate total>
        else
            total> → principal
        endIf
    endServer
```

This acc-interest server first tests to see whether any more time-units of interest accumulation are necessary. If so, it computes the principal plus interest for one time-unit. It then calls itself with the new intermediate total, with one less time-unit to compute, and with the original arguments of the interest rate and the result port total>.

If no more time-units need be computed (because units is zero), then the principal accumulated so far is the total accumulated for the supplied number of time-units. The result is revealed by forwarding total> to the accumulated amount. The total> argument was passed unchanged through all the recursions; it still represents the distributor to be forwarded to the answer to the computation.

A more complicated example demonstrates using recursion inside of another computation to provide all the facilities of loops. This style of recursion is similar to named-let in Scheme.

```
    Reveal the interest and total after interest on 'principal'
    accumulates at 'rate' for 'units' time units.
    Server interest :: principal units rate interest> total>
        • loop :: principal units
        Server loop :: sofar units
            If units > 0
                • loop :: (sofar * rate + 1) (units - 1)
            else
                • total> → sofar
                • interest> → sofar - principal
            endIf
        endServer
    endServer
```

For efficient iteration-by-recursion, Scheme specifies that implementations must be tail recursive. This optimization happens naturally in the Joule semantics: the recursive call is simply passed the result port; no stack is ever created.

This example server reveals two results: the interest, and the principal plus the interest. The trick that will become familiar is the invocation of the loop followed by the definition of the loop. The statement loop :: principal units sets up the initial values for the changing parameters of the loop. As in acc-interest, the loop procedure checks to see if any more iterations are necessary. If so, it computes the new principal and remaining iterations and calls the loop procedure recursively, not the outermost procedure. Unchanging parameters like rate are just used freely in the loop. When the loop has been called recursively once for each time-unit, the **else** clause is called (because units will be 0) and total> gets forwarded to the accumulated principal (named sofar), and interest> gets forwarded to the total minus the principal. Because both total> and interest> are lexically visible from the original context of the interest procedure, they don't need to be passed through each iteration of the loop.

5.7. Exception Handling

KeyKOS distinguishes between errors that would be wrong merely on the basis of the operation and server type, and those errors due to the current state of the server.

This section provides a high-level description of how Joule handles exceptions. Many exceptions are the result of improper arguments or unusual but semantically sound conditions. An example already shown includes signalling an exception when an attempt is made to withdraw too much money from an account. Exceptions are largely used for error reporting, but they can also be used for reporting conditions that are not errors, but are merely sufficiently unusual to warrant attention.

5.7.1. Normal Exceptions

Exception handling is a complicated business in sequential languages because it combines communication about the state of the computation with a transfer of control. The problems arise from the transfer of control. Being concurrent, Joule does not experience these problems: exceptions are reported and execution of other branches of the same computation continues. This is appropriate because those other branches might already have completed by the time the exception was raised (they can't be dependent on the exceptional computation or they would have suspended waiting for its result); terminating them would merely result in more confusion.

The exception port is implicit and dynamically bound: it follows the message-sending path, so raising an exception will report the exception to the caller of a server. The syntax for raising an exception is very much like message sending:

```
    Raise an exception named 'overdrawn' with balance as an argument. Raising
        exceptions is like sending messages to a Signal server. Any message can be
        sent.
    Signal overdrawn: balance
```

The exception can be any message. Exceptions are caught by a construct that rebinds the implicit exception port to a new port. As a result, the redirector can do anything with the exceptions, including drop them, terminate computations because of them, compute the failed computation another way, or pass the exception to further exception handlers. Here is an example of redirecting the exception port:

```
    Recover from a a failed money transfer
    Handler bounce?
        • myAccount deposit: customer-payment
    endHandler
    • service provide: customer

    Here's the exception handler to suspend service and get the money from the customer
        some other way in the case of a bounce. Otherwise it just forwards the
        exception back to the customer and goes on.
    Server bounce?
        op insufficient-funds: amount
            Define continue? = service suspend: endDefine
            • finance collect: customer amount continue?
        otherwise exception
```

```
        Signal exception
    this signal will reraise the exception in the handler outside this code example. The
        server 'bounce?' is defined outside the above Handler, so its signals are not
        intercepted.
endServer
```

In the example above, the **Handler** statement redirects all exceptions occurring within its nested body to the server named bounce?. The new exception handler is defined below the provision of the service to the customer. In this contrived example, if the customer payment is an account with too little money, the insufficient-funds: exception is sent to the bounce? server which suspends the service (returning a continue? flag), and initiates collection processes on the customer. All other exceptions are just passed through to the next outer exception handler because that's the dynamic context of the signal statement. Only the statements contained within the **Handler** form have their exceptions intercepted.

This structure of exception handling takes advantage of all the other tools built to manage messages: Server-based message dispatch, message plumbing to allow supplied handlers, etc. Types are even quite useful in this scheme, as every operation could declare a Type for the set of exceptions that might be raised. Typed exception handlers would then guarantee that they caught all the appropriate exceptions.

The syntax for the exception handling tools is:

Production	Production Definition
signal	**Signal** opExpr
handler	**Handler** opExpr block **endHandler** **HandlerTap** opExpr block **endHandlerTap**

A **HandlerTap** is like a **Handler** except that all exceptions are also automatically forwarded to the containing exception channel. **HandlerTap** is used for the concurrent equivalent of unwind-protection in sequential languages.

This issue needs further exploration because many of the unwind-protection problems go away in the course of solving other concurrency problems.

5.7.2. Keepers

Normal exception handling proceeds with the above constructs, but larger programs have many levels of exceptions: a server might raise some exceptions in response to user requests, but it might raise others because its algorithms broke or its data was corrupted. Keepers are lexically nested exception managers. They intercept the dynamically raised exceptions of any nested computation in order to decide whether the exception should be signalled to a client or acted upon internally. An example is if a database gets a disk checksum error, it shouldn't report bad-page errors to its caller, it should gracefully shutdown and recover the page from backups. The syntax for **Keeper** is similar to that of **Handler**:

```
A simple Keeper example:
Keeper ex
    Server database ...
```

```
            some service that raises exceptions
        endServer
    endKeeper
    The handler for the keeper.
    Server ex
        op disk-crash: device page
            backup recover:
        otherwise exception
            Signal exception
    endServer
```

We are currently revisiting the distinction between keepers and handlers to clarify when to use one and when to use the other. For now we recommend using **Handler**. We are exploring the idea of designating an error scope when signalling an exception. If the exception is not handled within that error scope, it becomes an error. This satisfies much of the need for keepers: a disk-crash error would be signalled as an error within the server, not a client error.

If the above had been a **Handler** form, any exceptions raised in the database server would have been raised directly to the database caller. The **Keeper** form intercepts any exceptions that escape its lexical scope whereas the **Handler** form just captures any that escape the dynamic scope (the innermost call). Thus this keeper can make sure that the exceptions reported out are client-worthy.

Keepers are also used with the Boundary facilities defined in Chapter 7 to provide debugger access when particular exceptions occur. Granting of debugger access is a carefully managed capability, and seems to correspond well with the keeper model of exceptions.

5.8. Standard Protocol

The standard protocol is the small set of messages to which all servers should respond. These are for purposes such as type queries and server comparison.

```
Type Basic
    op = other flag>
    op hash: hash>
    op type: type>
    op prove-type: type token>
    op respond: to
endType
```

The directionality of "=" prevents spoofing: a server can't claim to substitute for another server, it can only claim that another server can substitute for it.

A server should reveal true in response to an "=" operation if it considers the other to be a suitable representative of itself. This is only appropriate if the other can continue to represent the receiver forever. Therefore, servers with independently mutable state which happen to currently have the same state must reveal false. The hash: operation reveals a hash value for use in equality tests, Therefore, the hash must never change (or it can't support hash tables) and the hashes of equal servers (servers that reveal true to the equals message) must be equal.

The type: and prove-type: operations support type checking and type dispatch. The type: operation reveals the type the server claims to implement. The prove-type: operation is used by **Type** servers to verify that the server in fact implements their behavior. It is for internal use and should be ignored.

The respond: operation asks the server to send itself to the port. This allows clients of a server to delay executing code until the server actually starts responding to messages. The respond: operation also allows clients of a channel with several receiving servers to separate them.

5.9. Standard Servers

Standard servers are the server types that normal implementations require. This section documents those standard servers that are familiar to traditional programmers. Other standard servers such as verifiers for security will be documented in the appropriate sections.

5.9.1. Number

This section describes the user level protocol for Numbers (integers, etc.). The particulars of representation restrictions and interaction between number types will not be documented. The contract is not defined here.

```
Type Number
    super Basic

relational operators
    op = num flag>
    op != num flag>
    op < num flag>
    op > num flag>
    op <= num flag>
    op >= num flag>
    op min: num min>
    op max: num max>

arithmetic operators
    op + num sum>
    op – num difference>
    op * num product>
    op / num dividend>
    op % num remainder>
    op / / num intDividend>
    op / /% num div> rem>
    op negated: result>
    op abs: result>

extended math operators
    op ceiling: result>
    op floor: result>
    op truncated: result>
    op rounded: result>
    op log: n result>
    op ln: n result>
    op exp: n result>
endType
```

These Types specify the required behavior for the specific Integer and IEEEFloat number types:

```
Type Integer
    super Number

bit-wise Boolean operators
    op | num result>
    op & num result>
    op ^ num result>
    op complement: result>

bit representation operations
    op << bits left>
    op >> bits right>
    op precision: bits>
    op highBit: index>
endType
```

```
Type IEEEFloat
    super Number
    op mantissa: result>
    op exponent: result>
    op precision: bits>
endType
```

5.9.2. Tuple

Tuples are the primitive construct for messages. This section describes their protocol.

```
Type Tuple
    super Basic
tuple access protocol
    op count: count>
    op name: name>
    op get: arg# arg>
endType
```

5.9.3. Channel Distributor

Distributors are the ports that talk to the channel itself; operations include forwarding the channel, choosing an element of it, and the standard primitive operations.

```
Type Distributor
    super Basic
forward all messages ever received to 'port'
    op → port
send a pair with a message and the distributor to a newly created channel which will
        contain all the contents of the server except the separated message.
    op choose: choice
endType
```

5.9.4. Boolean

Booleans respond to standard Boolean logic messages as well as a few messages for control structures.

```
Type Boolean
    super Basic
relational operators
    op = bool flag>
    op != bool flag>
Boolean operators
    op | bool flag>
    op & bool flag>
    op ^ bool flag>
    op complement: flag>
Control structure operations
    op ifTrue: trueThunk race
Send trueThunk to race if the server is True. The race will invoke the first thunk sent to
        it.
    op ifFalse: falseThunk race
Send falseThunk to race if the server is False. The race will invoke the first thunk sent
        to it.
op if: trueThunk falseThunk race
Send trueThunk to race if the server is True, or falseThunk if the server is False. The
        race will invoke the first thunk sent to it.
endType
```

5.9.5. Array

Arrays are primitive servers that are used as the basis for collection classes, strings, etc. These are generally recommended against for pro-

grammers: they implement side-effects that lead to synchronization bugs. Arrays are to support efficient implementation of safer collection structures.

```
Type Array
    super Basic
    op count:
    op get: key# value>
    op store: key# value
endType
```

Array will also support a variety of group operations like copying and searching. This allows range checking to preserve memory safety, but allows extremely efficient operations (copying devolves to memory block transfer operations, for instance). There will also be subtypes that efficiently support primitive representation types like characters.

5.9.6. Types

Types are the runtime servers that can be queried about instances.

```
Type Type
    op isTypeOf: candidate yes?>
endType
```

The isTypeOf: operation uses the prove: operation to verify that the candidate server implements the type that the receiver represents. Further messages for types will exist to query them about the protocol that they require, and about default implementations.

5.10. Module Programming

This section will describe a module system to support the creation and maintenance of complex programs, once the design settles. The module system shares many characteristics with configuration version management systems.

5.10.1. Module Interface Definitions

Module Interface Definitions are like abstract types for modules; they allow the use of a module to be independent from the definition of the module. This supports complex systems with more than one implementation of a module coexisting. This is required for simultaneously running a system while testing a new version of it.

5.10.2. Export/Import/Open

This section will describe the syntax and semantics for managing modules. These are the static definitions in a module for connecting it to the rest of the computational universe. They will include which interfaces to import and how to import them, what authentication to require and how to negotiate it, and so on.

5.10.3. Module Namespace

This section will describe the module naming scheme. Several running Joule systems may exist for years before becoming connected. No glo-

bal naming scheme could possibly work. In addition, this scheme must integrate with existing file systems for early versions of Joule.

5.10.4. Module Versions

In a continuously running system, modules of code need to be replaced, upgraded, and patched. This section will describe a proposal for version and configuration management of modules. The current model for the module system is based on configuration versions management with nesting scopes of modules.

5.10.5. Naming People

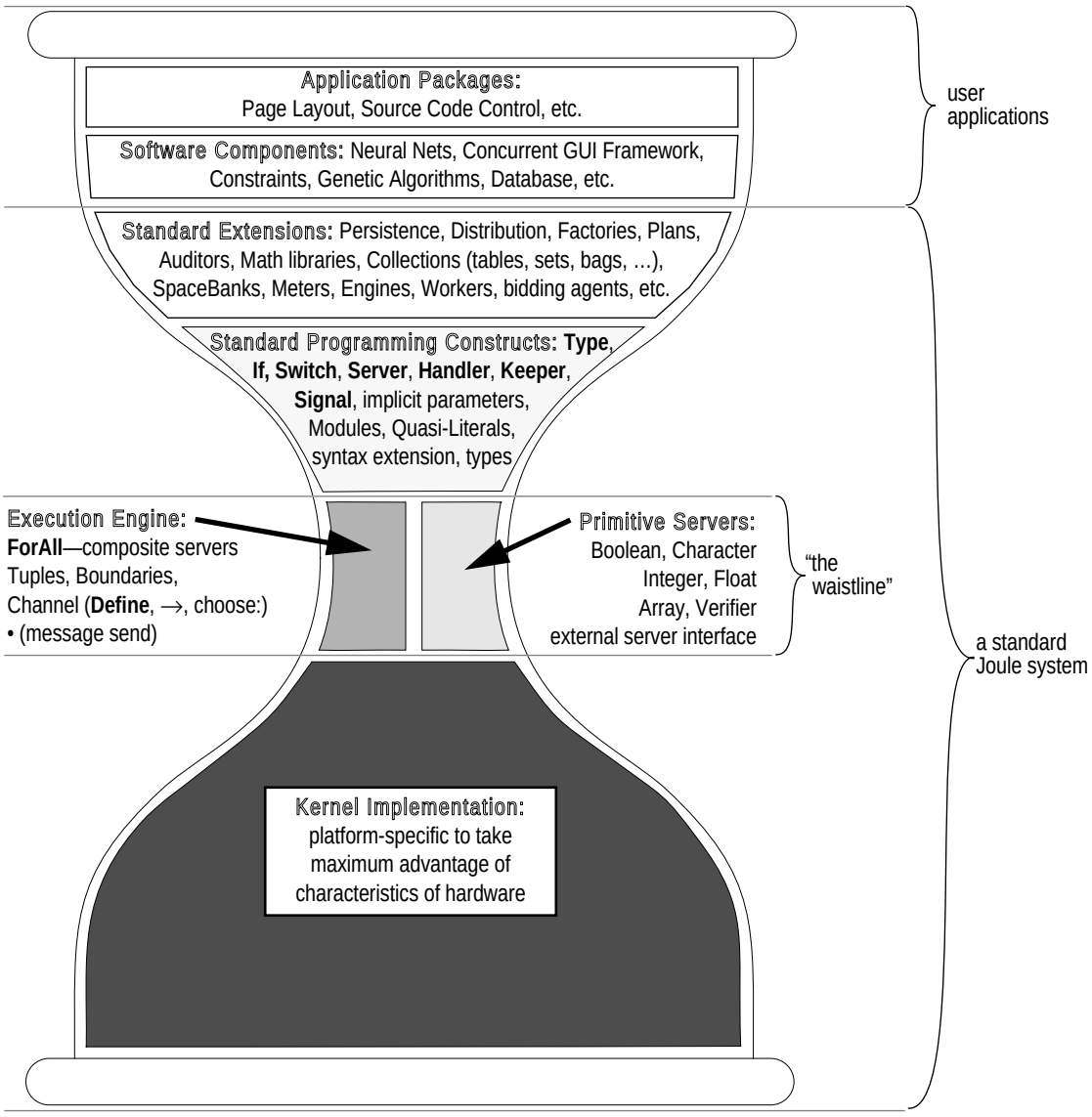
Systems of software grow in a context of interacting groups of people. There must be some way to represent appropriate information about the connection between pieces of software, live objects, and the people or companies responsible for or in control of them.

5.11. Parts of a Joule System

Figure 5.1 shows how the individual parts of a production Joule system fit into the architecture. The bottom half of the hourglass represents the Joule kernel implementation. The “waist” of the diagram is the primitive semantics and the compositional semantics needed to compose the rest of the Joule system from them. Above the “waist” are the components of a full Joule system (standard libraries, the distributed Joule layer, and so forth); specialized libraries supporting tools such as neural networks and genetic algorithms; and applications packages.

Some components in the top half of the hourglass are directly supported by companions in the bottom half; for example, the math libraries will to some extent rely on the default behaviors of some of the primitive servers such as Integers.

Fig. 5.1 The "hourglass"

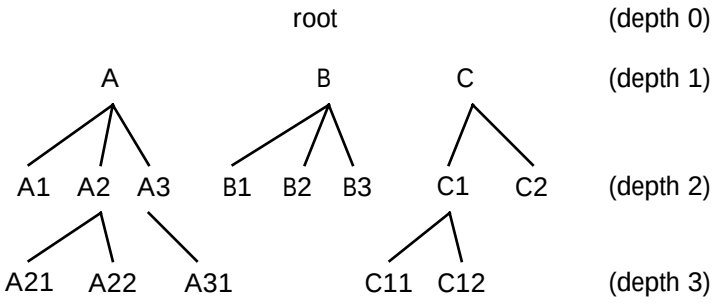


6.Hierarchical Accounts Example

This chapter presents a more complex example of Joule programming, a hierarchical bank account. Hierarchical bank accounts are part of agoric resource management; they implement hierarchical ownership and drawing authority. The account is hierarchical because it can have multiple sub-accounts, each of which is budgeted drawing power on the parent account (and each of which is itself a hierarchical account).

The importance of hierarchical ownership and drawing authority is explained in Section 9.1.

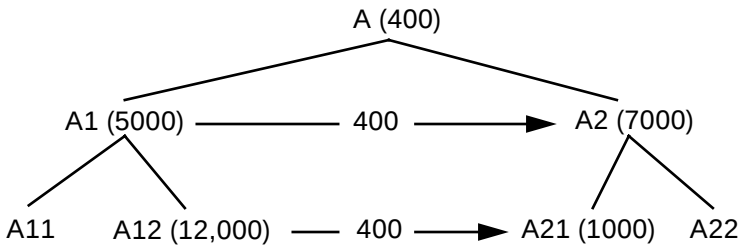
Fig. 6.1 Tree of hierarchical accounts



The root server is not an account but the environment in which top-level accounts are created. Each top-level account can be thought of as the supply of a single currency. In this model, there is no exchange between currencies; each is completely separate.

A hierarchical account can create sub-accounts with arbitrary balances. The balances an account may assign to its subaccounts are unlimited. When a sub-account within that account needs to transfer funds outside of the parent account, however, the amount is limited by the balance of the parent account. This is because the balances of their

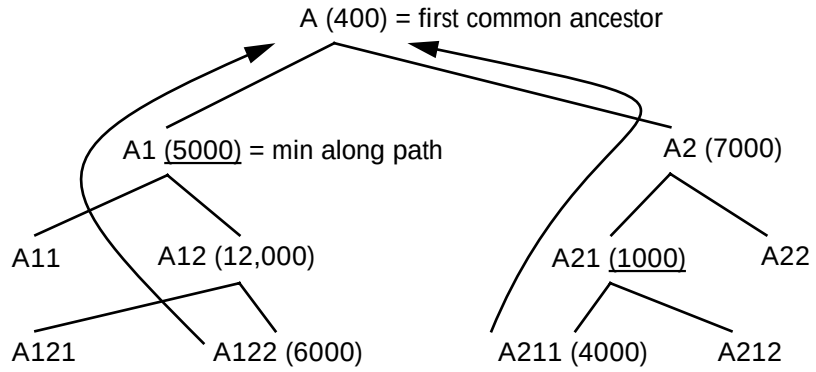
Fig. 6.2 Transfer of funds



respective parent accounts must be balanced as well. In Figure 6.2, any amount (up to the balance of A11) can be transferred from A11 to A12, because these are totally internal to the A1 parent account; however, the transfer of 400 credits from A12 to A21 must be covered by a corresponding transfer from A12's parent A1 to A21's parent A2. The maximum amount for such a move is A1's balance of 5,000 tokens.

In general, the amount that can be transferred from one account to another anywhere in the hierarchy is the minimum of the local balances of the accounts on the path from the donor account to the nearest ancestor it has in common with the recipient account (not including the common ancestor account itself).

Fig. 6.3 Nearest common ancestor for two accounts



For example, in Figure 6.3, the most that could be transferred from A122 to A2 or any of its descendants is 5,000 tokens, the minimum among the local balances of A122 and its ancestors A12 and A1. The most that could be transferred from A211 to A1 or any of its subaccounts is 1,000, the minimum of the balances of A211, A21, and A2.

The term *fractal reserve banking* has been applied to this hierarchical system of accounts. The system is "fractal" because it applies the device of fractional reserve banking recursively. The logical relationship of pieces to wholes does not change at different levels of granularity—the system exhibits the fractal property of *self-similarity*.

6.1. Hierarchical Accounts Components

6.1.1. Type Definitions

To program such a system of Account servers in Joule, we first define the type Account. Any server claiming to be of type Account must accept the set of requests specified by this **Type** form:

```
Type Account
    super Basic
    op split: amount account>
    op deposit: account amount deposited?>
    op budget: amount account>
    op balance: max account balance>
    op private: priv>
endType
```

The split: request will instruct the account to create a sibling account and transfer amount from its own balance to the new account. The result revealed is the public channel to the new account. Because this new account is created by its sibling, its balance must be deducted from the balance of the existing sibling account; money is conserved among sib-

lings. The budget: request instructs the account to create a new subaccount, with an initial balance of amount, which (since it is internal money) can be arbitrary. The deposit: request transfers amount from an existing account to the account receiving the request.

The balance: request takes three arguments: an amount, another account, and a result channel. The balance: request addresses the question "Could this account transfer max tokens into account?" The result revealed is the minimum of max and the maximum amount available for such a transfer (which is the minimum of all the balances of ancestors from the queried account to the ancestor it has in common with account). The candidate amount max is present to avoid infinities in the protocol.

The second **Type** form defines the *private requests* any Account should accept:

```
Type AccountPrivate
    super Basic
    op public: pub>
    op depth: depth>
    op parent: parent>
    op balance: max ancestor balance>
    op reserve: amount ancestor commit? success>
endType
```

Private methods can only be activated by requests received on the server's private channel. A server can receive from any number of channels; *private channels* are closely held because they accept messages with special capabilities. The same message, received via private and public channels to a server, might produce completely different behavior. The private requests to an Account server are used for special functions which should be kept secure.

The public: request reveals the acceptor for a public channel to the server. This ensures that any server which has access to the private channel of an Account can send messages to its public channel as well.

The depth: request reveals how far down in the account hierarchy this account is. It is used only for finding the first common ancestor of two accounts. The parent: request reveals an acceptor for the private channel of this account's parent. The private balance: requests are used to implement the public balance: requests.

The reserve: request instructs the account to adjust its own balance to reflect an impending withdrawal. This adjustment is conditionally based on the commit? flag passed to it. The success> distributor is used to signal success or failure to the server which sent the reserve: request.

The "?" suffix is conventional in Joule to indicate a flag, a port to a Boolean value (true or false).

6.1.2. The make-account Server

The procedure make-account creates new Account servers. Nested within it is the **Server** Account form that defines the behavior of the created accounts.

```
Server make-account :: amount parent account>
    • account> → account
    Server account
```

```
Server account
    var myLocalBalance = amount
    var myParent = parent
    var myDepth = (parent depth:) + 1
    implements Account
```

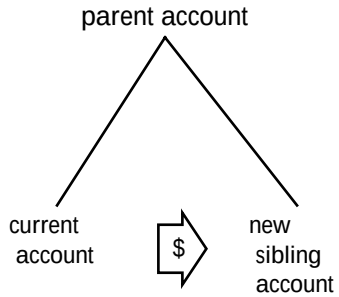
The new server account is created using the parameters passed with the "::" request to make-account. The result distributor account> corresponds to an acceptor held by the server that called make-account; that server can thus send to the new account.

The new account is created with three instance variables. myLocalBalance has the initial value amount provided in the call to make-account. The parent myParent of the new account is specified by the supplied acceptor parent. This acceptor must be for the private channel of the parent account because of the special information subaccounts need about their ancestors (for example, the depth: request, needed to determine common ancestors, is private).

6.1.2.1. The split: Request

The **op** extensions to the **Server** form define the methods of the account. The split: request creates a sibling account:

Fig. 6.4 split: creates new sibling account




```
op split: amount account>
    Define balance
        If amount < 0
            • balance> → myLocalBalance
            Signal positive-amount-required: amount
        orIf amount > myLocalBalance
            • balance> → myLocalBalance
            Signal insufficient-funds: myLocalBalance
        else
            • balance> → myLocalBalance - amount
            • make-account :: amount myParent account>
        endIf
    endDefine
    set myLocalBalance balance
```

The **Define** statement creates a channel balance which can be used immediately by the **set** statement to change (if necessary) the account's local balance. The statements of a Joule program execute concurrently. The instance variable myLocalBalance can be set to balance before the server that will receive from balance is known. If some other computation sends to myLocalBalance before balance is defined, those messages

will wait in the channel until the server that should receive them is determined.

Meanwhile, the **If** guards race to evaluate. If the creation of the account fails because a negative initial balance was specified for the new account, or because the current account does not contain enough tokens to provide the requested initial balance for the sibling, then balance sends to myLocalBalance (meaning that myLocalBalance ends up unchanged), and the appropriate exception is **Signal**ed.

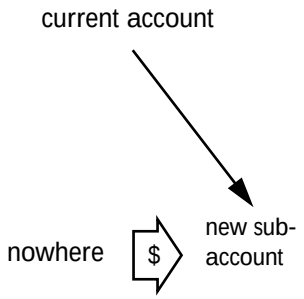
Since accounts happen to never have negative values, the **If** is actually a determinate choice.

Otherwise, the initial balance of the new account is deducted from the present balance of this account, and make-account is sent the "::" request to create the new account. Since it is a sibling of this account, it has the same parent (and is passed the private channel to that parent).

6.1.2.2. The budget: Request

The method for the budget: request is even simpler. Creation of a subaccount has no effect on the local balance of the current account, so we

Fig. 6.5 budget: creates a new subaccount, with any balance



only need to check that the initial balance requested is non-negative. The request to make-account is straightforward:

```
op budget: amount account>
    If amount < 0
        Signal positive-amount-required: amount
    else
        • make-account :: amount Private account>
    endIf
```

Because the new subaccount must have private access to its parent (the current account), the acceptor for this account's Private channel is passed in the request to make-account.

6.1.2.3. The balance: Request

The public balance: request "passes the buck" to its private counterpart.

```
reveal the balance of the receiver with respect to the ancestor in common with supplied
    account.
    op balance: max account balance>
        Define
            common =
                common-ancestor :: Private (private :: account)
        endDefine
        to Private balance: max common balance>
```

6.1.2.4. The private: Request

The private: request instructs the server to reveal its private channel.

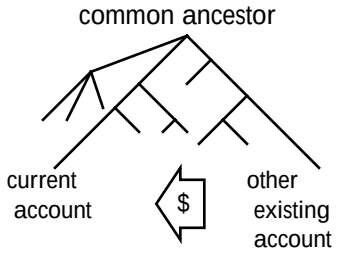
```
op private: priv>
    • priv> → Private
```

Sending the private: request to the public channel forwards the supplied distributor to the private channel. This implementation is clearly insecure. The methods enabling a server to decide securely whether or not to reveal its private channel (using a SealedEnvelope) will be discussed in Section 8.2.1.

6.1.2.5. The deposit: and reserve: Requests

The deposit: request transfers an amount from another account to this account. Whether or not the deposit attempt succeeded is revealed on

Fig. 6.6 deposit: transfers money from another existing account



the result channel deposited?>. Before the deposit: method can proceed with the transfer, it needs to ensure that the donor account is actually able to transfer that amount. It does this by sending the reserve: request to the private channel of the donor account.

```
op deposit: account amount deposited?>
    Define
        accPriv = private :: account,
        common = common-ancestor :: Private accPriv,
        withdrawn? =
            accPriv reserve: amount transferAmt common
    endDefine
    • deposited?> → withdrawn?
    Define transferAmt
        If withdrawn? & amount >= 0
            • transferAmt> → amount
        else
            • transferAmt> → 0
        endIf
    endDefine
    Define ignore> endDefine
    to Private reserve: 0 (transferAmt negated:) common ignore>
```

The deposit: method accepts three arguments: account, from which the deposit is being transferred; the amount of the deposit, and a result flag deposited?, letting the depositor know that the deposit succeeded.

The first **Define** statement calls the private server to get the private channel of the depositing account. Again, this version of private does not implement real Joule security techniques.

```
Reveal the private channel of account. This procedure will be substituted later for one
        that is secure.
Server private :: account priv>
      • account private: priv>
endServer
```

In response to the "::" message, private sends the request private: priv> to account's public channel; account then forwards priv> to account's private channel.

Back in the **op** deposit: block, accPriv is an acceptor for the depositor's private channel. The next **Define** block calls the common-ancestor procedure. In response to the "::" request, common-ancestor forwards the

```
Server common-ancestor :: acc1 acc2 ancestor>
    Define d1 = acc1 depth: , d2 = acc2 depth: endDefine
    If d1 < d2
          • common-ancestor :: acc1 (acc2 parent:) ancestor>
    orIf d1 > d2
          • common-ancestor :: (acc1 parent:) acc2 ancestor>
    orIf (d1 = d2) & (acc1 != acc2)
          • common-ancestor :: (acc1 parent:) (acc2 parent:) ancestor>
    orIf acc1 = acc2
          • ancestor> → acc1
    endIf
endServer
```

distributor ancestor> to the closest common ancestor of the acc1 and acc2 accounts. It does this by calling itself recursively: if the depth of one account is greater than the other, it recurs with the shallower account and the parent of the deeper account as arguments. If both accounts are of the same depth, it recurs with their two parents. This continues until the new arguments are (acceptors for) the same account.

The **facet** Private extension to the **Server** form introduces the private methods of the Account server. All **op**s following **facet** Private, are private methods.

The next **Define** statement sets withdrawn? to the success flag of the statement accPriv reserve: amount transferAmt common. This statement sends the private request reserve: to the private channel of the depositing account, asking it to verify that it can in fact transfer the amount requested.

The **set** statement can immediately tell myLocalBalance to deliver to newBalance—that is, either the same value it presently has, or its present value minus transferAmt. Again, messages to myLocalBalance will be held and delivered after the new value of myLocalBalance is determined.

What is the value of transferAmt? It is set in the deposit: method—if the flag withdrawn? indicates that the money was reserved as requested, transferAmt is set to the amount specified in the original deposit: request. If withdrawn? indicates that the depositor was unable to reserve the amount requested, then transferAmt is set to zero, and the depositor's local balance does not change.

This is one of the powerful benefits of Joule's inherent concurrency. The deposit: method of the server receiving the deposit sends the reserve:

```
facet Private
    type AccountPrivate
    op reserve: reserveAmt transferAmt ancestor success?>
        Define newBalance, parent'
            If (reserveAmt <= myLocalBalance) &
                    (Private != ancestor)
                • myParent reserve: reserveAmt transferAmt
                                    ancestor success?> then
                                    parent'>
                • newBalance> → myLocalBalance - transferAmt
            else
                • parent'> → myParent
                • newBalance> → myLocalBalance
                • success?> → Private = ancestor
            endIf
        set myParent parent'
        set myLocalBalance newBalance
```

request to the depositing server with an argument transferAmt that does not yet have a value. The depositing server can determine, based on the other arguments of the request, whether or not the request can succeed, and can inform the receiver of this (via the result channel success?>). Based on this go/no-go result flag, the server which sent the reserve: request can now supply the value of transferAmt. Meanwhile, both servers have already used transferAmt to adjust their own local balances.

If the depositor is an ancestor of the receiver, then the depositor does not adjust its own balance—the transfer is entirely internal to the ancestor and does not affect the ancestor's local balance. The withdrawn? flag is set to true, but no money is subtracted from the ancestor's balance.

Both deposit: and reserve: recur to the respective parent accounts, because those balances must also be adjusted by the amount of the transfer, up to but not including the common ancestor of the two accounts. To that common ancestor, the transfer of monies is completely internal, but to every intermediate account, the transfer is real money.

6.1.2.6. Other Private Requests

The other private methods of Account are fairly straightforward:

```
op public: pub>
    • pub> → account
op depth: depth>
    • depth> → myDepth
op parent: parent>
    • parent> → myParent
```

Any server holding the private channel to this account should presumably be allowed to hold the public channel as well; the private request public: reveals it. The depth: and parent: requests are used only by common-ancestor.

6.1.2.7. The Private balance: Request

The private balance: request reveals the balance of the receiver with respect to an account known to be its ancestor. (Normally, this will be called with the result revealed by common-ancestor.)

```
        op balance: max ancestor balance>
            Define parent'
                If ancestor = Private
                    • balance> → max
                    • parent'> → myParent
                else
                    Define
                        localBal = myLocalBalance min: max
                    endDefine
                    • myParent balance: localBal ancestor balance>
                        then parent'>
                endIf
            endDefine
            set myParent parent'
    endServer
endServer
```

The **If** guard ancestor = Private halts the recursive passing of balance: requests up the tree when they reach the ancestor itself. The **then** extension to the sending of balance: to myParent is there to ensure that messages from an account to its parent arrive in the order in which they were sent. (If you deposit a sum of money into an empty account, then try to withdraw some of it, the withdrawal attempt will fail unless the order of the requests is preserved.)

The **set** and **Define** statements are running concurrently. The **set** reassigns myParent to the acceptor parent' created by **Define**. All messages sent to myParent are forwarded into the channel parent' and held there. The **then** statement is an extension to the message-send statement. It takes as its argument a distributor whose messages (if any) will be forwarded to the target of the send, guaranteed to arrive *after* the one sent in the original message. In this case, the target is myParent, and the distributor is parent'>, which is holding the messages meant for myParent that piled up behind the privileged message balance: localBal ancestor balance>. If the other clause of the **If** wins and the **Define** is never executed, then parent'> and all the messages in it are forwarded directly to myParent in the ordinary Joule fashion, without any ordering.

6.1.3. The root Server

Recursive requests that are passed all the way up the "money tree" bottom out at the server root, which is the "parent" of the top-level

```
Server root
    op mint: amount account>
        If amount < 0
            Signal positive-amount-required:
        else
            • make-account :: amount Private account>
        endIf
    facet Private
    type AccountPrivate
```

```
        op public: pub>
            Signal not-a-currency:
        op depth: depth>
            • depth> → 0
        op parent: parent>
            Signal broken-recursion:
        op balance: max ancestor balance>
            Signal different-currencies:
        op reserve: amount ancestor commit? success>
            Signal different-currencies:
            • success> → false
    endServer
```

accounts. Except for the mint: request, it accepts only private messages—the same set of private messages as Account, so its private facet is also of type AccountPrivate. The public mint: request creates a new currency (a top-level account), with the money supply amount, and reveals that account's public channel on account>. (Note that root signals an exception to the reserve: request—once a currency is created, its total money supply cannot be increased.

Here are uninterrupted program listings for the make-account, common-ancestor, private, and root servers:

6.2. Program Listings

6.2.1. make-account

```
Server make-account :: amount parent account>
    • account> → account
    Server account
        var myLocalBalance = amount
        var myParent = parent
        var myDepth = (parent depth:) + 1
        type Account
        op split: amount account>
            Define balance
                If amount < 0
                    • balance> → myLocalBalance
                    Signal positive-amount-required: amount
                orIf amount > myLocalBalance
                    • balance> → myLocalBalance
                    Signal insufficient-funds: myLocalBalance
                else
                    • balance> → myLocalBalance - amount
                    • make-account :: amount myParent account>
                endIf
            endDefine
            set myLocalBalance balance
        op budget: amount account>
            If amount < 0
                Signal positive-amount-required: amount
            else
                • make-account :: amount Private account>
            endIf
        op balance: max account balance>
            Define
                common =
```

```
                    common-ancestor :: Private (private :: account)
            endDefine
            to Private balance: max common balance>
        op deposit: account amount deposited?>
            Define
                accPriv = private :: account,
                common = common-ancestor :: Private accPriv,
                withdrawn? =
                    accPriv reserve: amount transferAmt common
            endDefine
            • deposited?> → withdrawn?
            Define transferAmt
                If withdrawn? & amount >= 0
                    • transferAmt> → amount
                else
                    • transferAmt> → 0
                endIf
            endDefine
            Define ignore> endDefine
            to Private reserve: 0 (transferAmt negated:) common ignore>
        op private: priv>
            • priv> → Private
    facet Private
        type AccountPrivate
        op reserve: reserveAmt transferAmt ancestor success?>
            Define newBalance, parent'
                If (reserveAmt <= myLocalBalance) &
                        (Private != ancestor)
                    • myParent reserve: reserveAmt transferAmt ancestor
                                success?> then parent'>
                    • newBalance> → myLocalBalance - transferAmt
                else
                    • parent'> → myParent
                    • newBalance> → myLocalBalance
                    • success?> → Private = ancestor
                endIf
            set myParent parent'
            set myLocalBalance newBalance
        op public: pub>
            • pub> → account
        op depth: depth>
            • depth> → myDepth
        op parent: parent>
            • parent> → myParent
        op balance: max ancestor balance>
            Define parent'
                If ancestor = Private
                    • balance> → max
                    • parent'> → myParent
                else
                    Define
                        localBal = myLocalBalance min: max
                    endDefine
                    • myParent balance: localBal ancestor balance>
                        then parent'>
                endIf
            endDefine
            set myParent parent'
    endServer
```

```
endServer
```

6.2.2. common-ancestor

```
Server common-ancestor :: acc1 acc2 ancestor>
    Define d1 = acc1 depth: , d2 = acc2 depth: endDefine
    If d1 < d2
        • common-ancestor :: acc1 (acc2 parent:) ancestor>
    orIf d1 > d2
        • common-ancestor :: (acc1 parent:) acc2 ancestor>
    orIf (d1 = d2) & (acc1 != acc2)
        • common-ancestor :: (acc1 parent:) (acc2 parent:) ancestor>
    orIf acc1 = acc2
        • ancestor> → acc1
    endIf
endServer
```

6.2.3. private

```
Server private :: account priv>
    • account private: priv>
endServer
```

6.2.4. root

```
Server root
    op mint: amount account>
        If amount < 0
            Signal positive-amount-required:
        else
            • make-account :: amount Private account>
        endIf
    facet Private
    type AccountPrivate
    op public: pub>
        Signal not-a-currency:
    op depth: depth>
        • depth> → 0
    op parent: parent>
        Signal broken-recursion:
    op balance: max ancestor balance>
        Signal different-currencies:
    op reserve: amount ancestor commit? success>
        Signal different-currencies:
        • success> → false
endServer
```

7.Boundary Foundations

Modules are built on low-level foundations that support boundaries for creation and initiation of new programs in a running system, termination and resource management for existing programs, and access to foreign services. These foundations provide the mechanism on which the policies described in the Module Programming section are built. By separating policy from mechanism, we enable multiple programmer-level module systems to co-exist.

7.1. Domains

Domains are the foundational primitive for separately-executable pieces of code. They represent the boundaries needed for modules, security, and resource management. This section will describe them in detail.

7.2. Initiation

7.2.1. Necessity of Initiation

Initiation is the ability to start newly generated programs and connect them into an already-running system. This is a fundamental requirement for open systems.

7.2.2. Layers of Initiators

Programs can be initiated at many levels of abstraction. Machine code programs, Joule abstract machine programs, and Joule parse trees are all program representations that could be initiated. Initiators at each of these levels can be built on the initiator for the next level down.

7.3. Export/Import Issues

This section will describe how an initiated process gets properly connected to the rest of the universe of services.

7.4. Debugging Issues

Each separate domain is debugged independently. The typical model of systems that provide general debugger access violate encapsulation in a distributed system with untrusted clients. This section will describe

how debuggers are implemented while maintaining modularity and respecting trust boundaries.

7.5. Interoperability

Domains are the boundaries at which Joule communicates with foreign services (services written in other languages). To Joule, a foreign service looks like an independent Domain with which Joule engages in message communication. The Joule semantics could actually manage entire populations of external programs as if Joule were an operating system.

8.Security

Many of Joule's security foundations were drawn from or inspired by KeyKOS, a capability-based operating system that provides NCSC (National Computer Security Center) B3-level security. Security can be thought of as the extreme of modularity: truly independent programs can only interact with each other through explicit and controlled boundaries.

Security, like modularity, is first supported by negative capabilities: operations that are prevented. After insecure abilities have been removed (such as the ability to write any file or write to any memory location), and the system has been reduced to secure foundations, one then builds abstractions that provide all the standard functionality of insecure systems without exposing programs to risk. Finally, one establishes tools and methodologies with which programs can securely engage in otherwise risky activities. This methodology for security is really a methodology for managing and arranging trust relationships.

This section first describes encapsulation, the enforcement of rules for accessibility and visibility. Encapsulation makes each server inviolable by other servers—the only thing a client can do to a server is send it messages to which the server explicitly decides how to respond.

Encapsulation brings with it polymorphism and anonymity: servers can only be distinguished by how they behave, so servers could be written that pretend to be other servers (for instance, money). How does one build trust relationships in such a system? We describe the technique used in Joule with which servers can prove their identity to each other, allowing the establishment and extension of trust relationships between servers. This supports the creation of extended networks of servers that cooperate and subcontract with each other.

With the establishment of these networks, the encapsulation and trust issues arise all over again: does the original client trust a subcontractor? There are many properties of a server that are composed from their subcontractors. The two we describe here are discretion ("Will the server keep secrets?") and durability ("Will the server still be in operation in the future?"). Other such properties, like timeliness and robustness, are not explored here.

8.1. Encapsulation

Encapsulation is what people commonly think of when they think about security. It is the enforcement of rules for accessibility and visibility. Languages like C and C++ provide weak modularity because any program in the same address space can convert a number to a pointer and violate the integrity of other parts of the code. Encapsulation allows programs to run without interference or corruption from other programs.

8.1.1. Capability Security

Capability security is the foundation for good encapsulation. This section describes capability security. Certain powerful capabilities, particularly those that violate encapsulation, such as the ability to read and write any section of memory, are closely held by severely restricted service providers.

8.1.2. Accessibility Relationships

The semantics of the accessibility relationships from the abstract execution model constrain the kinds of communication and access that can happen among programs in a Joule system. The rules guarantee that all communication is by message passing, and that all access to a receiver is only through message passing. This guarantees the encapsulation of Joule programs.

8.2. Certification

To build trust among unknown servers requires the ability to prove their identities to each other. The identity might be of a type: servers that charge money want to be paid with *real* money (also implemented as a server), not a forged server that responds to the same messages as money. The identity might also be of a particular server: when sending to requests to a bank, a server wants it to be the bank at which it has its account, not just any bank (even though all banks run the same code).

8.2.1. Verifiers

Joule provides the Verifier abstraction, a mechanism for certification built using only encapsulation and message passing. Unlike KeyKOS brands, Verifiers require no support in the computational model.

Verifiers provide the services of a single-key encryption scheme using encapsulation. Given a value to be sealed, a Verifier will create and reveal a SealedEnvelope containing the supplied value that can only be opened by the Verifier that sealed the envelope. That SealedEnvelope can then be passed through insecure channels to some other server which has access to the same Verifier. That other server can open the SealedEnvelope and use the contained server.

For more about SealedEnvelopes, see Appendix D, *Energetic Secrets*.

Type Verifier
super Basic
seal the supplied contents in an envelope that can only be opened by the receiving Verifier, and reveal that sealed envelope.
op seal: contents enveloped>

```
    given an envelope sealed by the receiving Verifier, reveal the contents of the envelope.
    op unseal: envelope content>

    The type for envelopes used by Verifiers. SealedEnvelopes have no other behavior
        (beyond Basics) than the secure access that Verifiers use to get at the
        contents. Verifiers are built using more primitive Verifiers, so no user program
        can get at the contents of a SealedEnvelope; only the proper Verifier can.
    Type SealedEnvelope
    super Basic

    Only Verifiers can get at the private channel of a SealedEnvelope (because they prove
        their identity using other Verifiers).
    op private: private>
```

Verifiers rely on encapsulation for their certification properties: unsealing a message proves that the originating party had access to the same Verifier as the receiver; with encapsulation, the receiver can know the extent in which the Verifier is visible, and so can know what code generated the message.

Here is the replacement code for the hierarchical bank account example (Chapter 6) that uses Verifiers to allow accounts access to each other's Private channel without exposing the Private channels to outside code. This line of code is added within the definition of the make-account server; it creates a new Verifier named AccountPrivate within the scope of the Account implementation:

```
Define AccountPrivate (make-verifier ::)
```

The remainder of the code replaces the corresponding insecure implementations in the original Account implementation code:

```
Reveal an Envelope containing the private channel for the receiver.
    op private: priv>
        AccountPrivate seal: Private priv>
Reveal the private channel of another account by asking it for a sealed Private channel
    and unsealing it.
    Server private :: account priv>
        Define box account private:
        AccountPrivate unseal: box priv>
```

The redefinition of the private: method now reveals a SealedEnvelope containing the Private channel rather than revealing the actual Private channel. The redefined private server (the internal server that an account calls to access the Private channel of another account) also uses that Verifier: it asks the desired account for an envelope containing the account's Private channel, then unseals that envelope using the AccountPrivate Verifier (the one shared by all accounts made by this make-account server.)

Both the account requesting a private channel and the account providing the private channel can be assured that the other is a real account, and can be assured that the Private channel is secure (not exposed to eavesdroppers or forgers). The type-authenticity (proving the other is a real account) is guaranteed because both parties know the other party has access to the AccountPrivate Verifier (or the seal/unseal wouldn't have worked), both parties know that the accessibility rules of the language guarantee that unless the make-account code reveals the

AccountPrivate Verifier, then only the body of that code can use it, and finally both parties know that their implementation in make-account doesn't reveal the AccountPrivate Verifier. The combination of these means that only an account could have provided a SealedEnvelope openable by AccountPrivate, and only an account could open that envelope. This means that messages sent on any account Private channel are sent by an account server, and so are part of the proper implementation of the Account abstraction.

An eavesdropper is a forwarder that wraps the SealedEnvelope in order to get access to the contained Private channel when the envelope is unsealed, or to engage in the protocol with which envelope is unsealed. At no time is the Private channel of an account exposed outside of the context of the account implementation, except in a SealedEnvelope that can only be opened in the Account implementation context. The only potential for exposure of the contents of the SealedEnvelope occurs during the unsealing operation; during the sealing, the account has an internal (i.e., no eavesdroppers) channel to the Private channel.

The unsealing operation uses the same kind of protocol in order to unseal the envelope. All Verifiers and SealedEnvelopes within a trust boundary share access to a single Verifier (the implementation of which distributes efficiently) through which they can securely connect with no eavesdroppers. As a result, during the unseal operation, the AccountPrivate Verifier and the SealedEnvelope containing the desired private channel make a secure connection through which the Envelope reveals the contained Private channel to the Verifier which then reveals it to the caller of unseal:.

8.2.1.1. Unique Tokens

In the private protocol, an implementation could define messages to reveal information unique to each instance in order to prove identity. This code implements a simple unique token scheme in which the clients can only compare unique tokens, but cannot otherwise find out *anything* about them:

```
Server make-token
var myNext 0
var TokenPrivate (make-verifier :)
this should be a const declaration, but we don't have those yet.
Define the message used for procedure call (to make a unique token). This could also
        have been part of the first clause, but this is a better style for procedures with
        changing variables.
op : token>
    token> -> token
Here is the behavior for an instance made by make-token.
    Server token
    var myID myNext
Get the number of the other guy and then compare it against the number of the
        receiver.
    op = other equal?>
        Define hisNum (TokenPrivate unseal: (other private:))
        myNum = hisNum equal?>
 Reveal an Envelope containing the number unique to each instance.
    op private: envelope>
        TokenPrivate seal: myNum envelope>
set myNext myNext + 1
```

Tokens each contain a number guaranteed to be unique with the simple expedience of allocating them sequentially. The sequential ordering won't reveal anything about the running of the program even if the token creator is shared among untrusting programs because the numbers are never revealed except inside the actual implementation (just like account above).

8.3. Discretion

As described in the introduction to security, once a system has certification and encapsulation, it is possible to build large networks of servers that subcontract with each other to provide services to their clients. The naïve expansion to these large networks of servers leaves systems with the same precarious lack of robustness characteristic of networks today. Joule supports these large networks by allowing servers to establish and verify transitive properties of these subcontracting relationships so that a server can ensure its robustness. Many of these properties are managed through the same general mechanisms, but they will first be explored in the context of discretion. The techniques we describe here are called assurance by construction, assurance by auditing, and assurance by special execution.

Discretion control is a generalization of the better-known confinement problem. Successful *confinement* allows untrusted code to be run on private data without the untrusted code being able to communicate any secrets to the outside world. As the name implies, solutions to the confinement problem typically rely on running the untrusted code in a box out of which it cannot communicate. Successful *discretion control* allows that same untrusted code to communicate with other services, but still prevents secrets from being leaked.

8.3.1. Assurance by Construction

Assurance by construction means having mutually trusted third-parties build services for each other. The clients of such a construction service can then know that no other server has access to their private services, and so even if those services cheat, they still can’t communicate secrets to the outside world. This section will describe the process in more detail, including how Factories can provide assurance by construction.

8.3.2. Assurance by Auditing

Auditors might be equipped with an abstraction-breaking tool which can examine an existing server for capabilities to steal secrets. This tool is very closely held. This is assurance by *auditing.* An auditor is able to audit a service for connections that could lead to data leaks or for dependencies on facilities that may be insufficiently permanent.

Modules pass an audit if they are functionally discreet. Factories produce modules that are structurally discreet—structural discretion can only be supplied by construction because a module that would otherwise be discreet might be shared with an indiscreet observer (a discreet database, for instance).

8.3.3. Assurance by Special Execution

Another technique involves an elaboration of the execution model with which the owner of a secret can send messages involving the secret inside special query messages. The ultimate effects of a query are solely upon those things named in the message. This may sound like a very limiting style of programming but such a query may, for instance, create a database and reveal access to it while guaranteeing that no one else has access. That the database was created with a query ensures that secrets entrusted to it are safe from disclosure. This is assurance by *special execution*.

8.4. Durability

Durability is the property of a service that it can guarantee its own survival and continued function. It is *durable* if no untrusted authority has the ability to destroy it or other services on which it depends. This can extend into very physical realms of assuring that communication links are independently redundant so that a single failure doesn't partition the network.

8.4.1. Implementation

Support for durability relies on the same foundations as discretion, but uses the mechanisms somewhat differently. This section will describe the differences.

8.4.2. Requirements

Where discretion is a correctness issue, durability brings performance into the correctness domain. Thus, a program is not durable unless it has access to enough resources to guarantee that it will run; durability requires concurrency so that the process can run, and resource management so that it can guarantee that it will.

9.Resource Management

This section first describes some underlying principles for resource management abstractions in Joule. It then describes abstractions for resource encapsulation and ownership, the foundations for resource management. Finally, it describes market-based resource management abstractions for making resource trade-offs in complex systems.

9.1. Resource Management Fundamentals

This section describes some underlying principles in the design of resource management abstractions. The first two principles of hierarchical ownership and drawing authority are demonstrated in Chapter 6, the hierarchical accounts example.

9.1.1. Hierarchical Ownership

Hierarchical ownership makes Joule's resource management abstractions recursively applicable. It permits reuse of mechanisms within an entity without the entity losing control of its pieces. This section will explain in detail why hierarchical resource ownership is important, and gives real-world examples of its use (renters and landlords).

9.1.2. Drawing Authority

The naive way of sharing a resource among consumers is to divide it up (to allocate it). This allocation assumes prior knowledge of how the resource will be used. Allotting budgets to the consumers allows the programmer the same control over the limits of consumption, without requiring prior knowledge of resource utilization. This section will describe budgeting drawing authority in more detail, describe how it subsumes allocation, and give examples that motivate the shift to drawing authority.

9.1.3. Quantity vs. Territory

Quantity and territory are two extremes for measuring or representing access to resources. *Quantity* represents an amount of some *fungible* resource (a resource whose units are all equivalent). *Territory* represents a particular piece of some resource, analogous to real estate. Many computational resources can be represented both ways (most memory pages are fungible, for instance), and these representations are useful

for different things. This section will describe the distinction and give examples.

9.2. Primitive Resources

The two fundamental computational resources are execution time and memory. Management of other resources can be built in the language, but these require support in the language *implementation*. This section will describe the primitives for reifying and encapsulating these two primitive resources, and give examples of using them.

9.2.1. Meters and Engines

Meters and Engines are two tools for encapsulating execution time. Meters support ownership of quantities of compute time; Engines support ownership of "territories" of compute time. Engines are provided to support real-time applications.

9.2.2. Space Banks

Space Banks encapsulate computer memory. This section will describe the interface to them.

9.3. Agoric Abstractions

Agoric resource management is the use of markets and prices to manage resources. This section introduces a simple system design and default strategies such that the emergent behavior of such a system is understandable; then presents mechanisms for adding programmer-defined strategies and policies for dynamically adapting to resource availability.

9.3.1. Example System Design

This section will describe a particular system for market-based resource management. It will include the definition of Workers, the virtual machine that runs on money instead of CPU cycles.

9.3.2. Default Strategies

This section will describe the default strategies from which price signals emerge. Properties of default strategies are well described in **[89]**. These policies must:

- result in the emergence of typical system behaviors, such as fairness among processes
- produce price information that reflects resource costs
- remain strategically robust against gaming by non-default strategies
- be reasonably computationally efficient

9.3.3. Using Default Strategies

This section will describe the tools for using the default strategies, including Workers (virtual machines that run on money) and Expense Accounts (budgets for Workers to draw upon). These tools make it easy

to divide resources among many services, and call upon existing services that require resources.

9.3.4. Building New Strategies

This section will go under the hood of the system to describe how to build and use more sophisticated strategies for adapting resource usage to the demands of the rest of the computation. These tools are for processes that use price information, and include Agents (strategy elements) and the interfaces to standard resource Providers.

9.4. Improved Computational Model

Here we will present an abstract computational model that includes resource management and a correct state of execution even in the absence of sufficient resources.

10.Distribution

This chapter explores the issues affecting distributed systems and describes how Joule satisfies them. The solutions discussed assume the availability of the resource management tools described in the preceding chapter. By providing mechanisms to support process migration, as well as default policies, Joule supports the full spectrum of distribution regimes, from automatic distribution in which processes are automatically spread across multiple processors, to explicit distribution in which the programmer controls or influences the mapping from processes to processors, to untrusting distribution in which the programmer explicitly manages and adapts to trust boundaries and failure properties of the network.

10.1. Transparency

Transparency is the ability of a program written in Joule to function unaffected as it is stretched out across machines. This section will first describe the importance of transparency, then examine the primitives and the computational model to show how machine boundaries and communication lags can be invisible to an executing program.

10.1.1. Separation of distribution from correctness

Transparency allows programs to first be built to work, then be distributed without breaking the logic of the program. Because the assignment of processes to processors doesn't change the program, transparency also increases reliability and maintainability.

10.1.2. Adaptive distribution requires transparency

Adaptive distribution is the ability to write programs that migrate other, already-running programs to improve performance or adapt to the changing topology of a network. This adaptability requires that machine boundaries move in relation to the underlying program, without the program being changed. This transparency enables adaptive and automatic distribution, and also enables applying all the abstraction power of the language to the problem, including price-based competition among processors.

10.1.3. Channels stretch across wires

The semantics of Channels is such that they can be stretched across wires (with the inherent delays, etc.) without breaking.

10.1.4. Trust relationships are the same

The security system provides programs with ways of managing trust boundaries. Distributed systems simply introduce more trust boundaries, so the nature of the system stays the same, and programs will already be built to deal with the security problems revealed by distributed systems.

10.2. Failures in Distributed Systems

A continuously-operational open, distributed environment must remain robust in the face of many failure modes that are either not present or not obvious on single machines. This section explores many of them, and briefly describes Joule's solutions.

10.2.1. Node Failures

Node failures occur when a machine on the network fails. This section will describe the Unavailable exception which reports the failure and describes how to handle this exception. It will also describe using message plumbing to acquire control over the raising of this exception.

10.2.2. Network Partitions

A network partition is like a multiple-node failure except that the machines may return to service. Many applications can withstand the wait, so handling the return of access to a service is important. This section will describe the handling of the Available exception, which is reported when a service returns, and give examples.

10.2.3. Aberrant Behavior

Because of the Joule computational model, malicious and arbitrary behavior in a distributed system creates no new problems. Therefore, the security support deals with aberrant behavior of nodes in the distributed system. Further, the virtualizability of Joule channels allows them to be transparently encrypted between sites, so they can remain secure from eavesdroppers.

10.2.4. Node Amnesia

Since Joule is a persistent system, a particularly difficult form of failure is for a node to fail, and then revive in a previous state (from a checkpoint or backup). The issues here are complicated and subtle and will not be dealt with in detail in this document.

10.3. Explicit Distribution

This section will describe how a programmer can explicitly distribute a Joule computation. It will describe one particular distribution infrastructure, and how programs should interact with it.

10.3.1. Migration

This section will describe how to migrate processes between virtual processors in order to modify or improve the topology of a network.

10.4. Frameworks for Automatic Distribution

This section will build on the previous section to describe a framework in which programs can be automatically distributed (though not as well as a programmer might do), with no change to the program.

10.4.1. Simple Mechanisms

This section will describe a minimal strategy for automatically distributing processes to processors.

10.4.2. Stochastic / Heuristic

This section will describe stochastic and heuristic methods for load-balancing and distribution of processes to processors.

10.4.3. Agoric-Driven

This section will describe price-based strategies for load-balancing between processes using some of the agoric resource-management foundations.

10.5. Off-line Distribution

Occasionally-connected networks are those whose sites rarely talk to each other. This definition applies primarily to laptops and the networks to which they connect, and to networks that connect periodically to transmit updates (for example, USENET links). This section will describe how Joule distributes successfully over occasionally-connected networks.

11.Persistence

This section will describe possible implementations of persistence in Joule. The trade-offs between these implementations remain largely unexplored for Joule, though much of the territory is known for other related systems such as FCP and Actors.

11.1. Page-Based Persistence

This section will describe a persistence implementation at the level of pages of virtual memory. The design of this system is based on the persistent virtual memory system in KeyKOS. In page-level persistence, changed memory pages get checkpointed to persistent store at regular intervals, saving the entire execution state of the machine.

11.2. Server-Based Persistence

This section will describe a persistence implementation at the level of servers. In server-level persistence, changed servers write themselves to persistent store at regular intervals.

11.3. Replay-Based Persistence

This section will describe a persistence implementation in terms of logging messages passed and the internal non-deterministic choices made by servers in order to replay the actions of the system and reconstitute its state.

A. Language Comparison

This section reviews other languages and systems relative to the requirements for robust servers and open distributed systems. It also compares the capabilities of Joule with those of its antecedents, Actors and concurrent constraint languages.

Joule is designed to be a foundation for distributed applications. It is largely a language foundation because a language can be layered on top of any operating system, enabling the foundation to be made extremely portable. Since the design captures many of the ideas from good operating systems, the semantics also match well with network and machine operating systems.

Many people consider language and system comparisons a soft and subjective art. The discussion of server and market-based computation in Section 1.1 provides the foundation for a hard-edged discriminant for comparing languages and systems: Can they build robust servers? This requires encapsulation, concurrency, and resource management.

The present comparison only applies the robust servers criterion. Future versions of this section will also document the large contributions from previous languages. In the present comparison, however, we consider only those features which are lacking in a particular language but present in Joule. We offer our apologies if this narrow focus makes the comparison seem invidious, and hope to amend this lack in future versions.

A.1. Language Comparison

Languages like C and C++ do not have true encapsulation; any program in the same address space can violate the modularity of objects by using casts. Systems that glue together separately-written C or C++ programs to attempt to support distributed programs are considered separately. They don't provide C or C++ any more support than they provide to assembly language programs.

Even if these programming languages provided modularity, they still lack concurrency. Again, concurrency tools like threads that are provided by the operating system are really a property of the operating system, and are considered in the next section.

The Smalltalk language and system are extremely good, but all implementations provide hooks for the debugger that any object can use to violate modularity; the language semantics provide encapsulation but

the environment throws it away. Smalltalk is also a sequential language. The standard support for concurrency is a Semaphore mechanism with non-preemptive multi-tasking; this is not sufficient for reactive, concurrent systems.

The Linda language and system can be considered together because they share the same semantics. Linda assumes a global tuple space for communication. In a distributed system, the existence of that global structure is untenable—machines fail, networks partition, etc. The reliability of a robust server cannot depend on the reliability of machines on which the robust server is not running. Further, the tuple-space of Linda is insecure: tuples are just placed in the space and other processes pattern match against the tuple-space to extract *any* tuple they recognize. Proposals have been made for M-Linda, a Linda with multiple tuple-spaces that was moving towards the Joule communication semantics, but progress on that front stopped.

Languages like Actors, ABCL/1, and Hermes, and concurrent logic programming languages like FCP, FGHC, and Janus all have the requisite encapsulation and concurrency properties. They do not have sufficient resource management capabilities to implement robust servers well, but they come the closest of existing programming languages. Many of the logic programming languages suffer from the additional disadvantage of a global environment for procedure definitions; such global constructs break in large, distributed systems. The Joule computational model is a direct combination of features from the Actors computational model and the concurrent logic programming model.

A.2. Operating Systems

Operating systems on consumer platforms (Macs and PC) often provide no encapsulation, so they cannot be a foundation for robustness. More sophisticated operating systems like Windows NT and UNIX have more encapsulation—they provide inviolable accessibility boundaries between applications—but hidden in their complexity they throw away the security in the abstractions that they provide to programmers (in UNIX, every program that runs with root permissions is a potential hole through which the entire system could become vulnerable). The resulting systems have holes that are fatal flaws when connected into a network with untrusted clients.

Micro-kernel operating systems like Mach start with capability security and provide a clean enough environment that they can be secure. They also clearly provide concurrency. The mechanisms for resource management don't provide much flexibility, but they are capable of supporting robust servers.

KeyKOS goes one step further. KeyKOS is a capability-based, secure operating system that provides hooks for explicitly managing computational resources such as processor time and memory. The only remaining lack is that it does not provide transparent forwardability of message passing between operating system objects (called Domains). As a result, the KeyKOS model cannot be extended to a network transparently.

B. BNF for Joule Syntax

This chapter presents, in Backus-Naur form, a grammar for the Joule language forms and expression syntax. Lexical conventions will appear in a later version of this Appendix.

B.1. BNF Conventions

In the BNFs in this appendix, the following conventions apply:

- Italicized names indicate terminals. The terminals are not presented in this Appendix. See **Section 4.1:** *Lexical Conventions* for an informal presentation.
- Verticals (“ | ”) are used to separate alternative components that may be used in the same place.
- A question mark (“?”) following a component means exactly zero or one instance of the component is allowed.
- An asterisk (“*”) following a component means zero or more instances of the component are allowed.
- A plus sign (“+”) following a component means one or more instances of the component are allowed.
- Braces (“{ }”) are used to indicate grouped components, to which one of the preceding allowance indicators applies as a unit. {fee fie}* means zero or more instances of the series fee fie are allowed.
- A component followed by some delimiter foo and an asterisk means that zero or more instances of the component may be present, separated by foo. For example, “{bar},*” means that any number of bar components may be present, separated by commas.
- A component followed by some delimiter foo and a plus sign means that one or more instances of the component may be present, separated by foo.
- A production name for which multiple definitions are given means that any one of the definitions may be used where that token appears.
- The indentation describes the indentation rules that were generally used throughout this manual, but has no semantic significance.

B.2. Forms

Production	Production Definition
block	{form}*
form	• simpleExpr {opExpr},+ {**then** opExpr}?
	Define {param | param = opExpr},* block **endDefine**
	ForAll param ⇒ param block **endForAll**
	ForOne param ⇒ param param block **endForOne**
	Handler opExpr block **endHandler**
	HandlerTap opExpr block **endHandlerTap**
	Keeper opExpr block **endKeeper**
	Signal opExpr
	If opExpr block {**orIf** opExpr block}* {**elseIf** opExpr block {**orIf** opExpr block}* }* {**else** block}? **endIf**
	Switch opExpr {**case** pattern {**or** pattern}* block}* {**otherwise** param block}? **endSwitch**
	Type param {**super** *Identifier*}? {**op** {pattern}**or**+ block {**to** *Identifier* {opExpr},+ block}*}* **endType**
	Server param {method}? {var}* ops {facet}* **endServer**
var	**var** {param | param = opExpr},* block

Production	Production Definition
ops	{**implements** *Identifier*}? {**op** method}* {**otherwise** param block}
method	{pattern}**or**+ block {change block}*
change	**to** *Identifier* {opExpr},+ | **set** {*Identifier* = opExpr},+
facet	**facet** param ops

B.3. Expressions

opExpr	simpleExpr | simpleExpr *Operator* opExpr
simpleExpr	*Identifier* | *Literal* | *Quasiliteral* | tuple | '(' nestExpr ')'
nestExpr	simpleExpr | simpleExpr opExpr
tuple	{*Operator* | *Label* } {opExpr}*
param	*Identifier*
pattern	tuple | *Quasiliteral*

C. Optional Arguments

This proposal for managing optional arguments and "rest" arguments in messages is consistent with the existing syntax of the language, Energetic Secrets (see Appendix D), and efficient implementation.

C.1. Overview

Optional arguments and rest arguments are both extremely useful facilities. They are realizable (more or less conveniently) in languages such as C++ and Scheme. This proposal supports them both directly. It starts by allowing only one method to respond to a given selector or unsealer; that one method can supply optional parameters to handle multiple cases. This keeps a method name associated with a single semantics.

C.2. Receiving Messages

Here is a template that illustrates all the argument-passing idioms:

```
Server serverName
    op sel1: arg1 arg2 arg3
        body...
    op sel2: arg1 arg2 optional arg3 = exp1, arg4 = exp2
        body...
    op sel3: arg1 rest num fn
        body...
endServer
```

The first operation, sel1:, is the standard message passing pattern: several argument names following the selector that will be matched against the incoming arguments.

The second operation, sel2:, illustrates handling of optional arguments. It includes the keyword **optional**, which signals that the pattern following is for optional arguments only. Each optional argument name is followed by "=" and an expression specifying a default value to be used if that argument is not supplied in a message.

The third operation, sel3:, illustrates handling of "rest" arguments, for use when any number of supplied arguments are to be handled generically. The identifier num is bound to the number of arguments remaining. The identifier fn is bound to a function which can reveal the arguments to the message. Any of the "rest" arguments can then be revealed by calling the argument function fn with the index of the argu-

The current definition for "rest" arguments is in terms of a function and arguments. This may be changed to make use of a virtual collection type.

ment to reveal and a distributor on which to reveal it. fn can only be called once per index, with the calls in any order. (On further calls with the same index, it must either return the same result or signal an exception.)

C.3. Sending Messages

Sending messages with optional arguments is just like normal message sending. Each optional argument can be either supplied or not; the receiving server accommodates either case. Similarly, messages can be sent to servers that will treat all the arguments generically. Special handling is needed to forward messages generically when manipulating the arguments. This requires the support of Energetic Secrets (see Appendix D).

Sealers for Energetic Secrets support direct protocol that can seal using the same kind of argument count and argument function that the "rest" arguments mechanism supplies. Programs can provide a function directly that supplies the arguments dynamically to the message send. Thus:

- receiver (msg:sealer seal*: num fn arg1 arg2)

would supply 3 static arguments and any number of dynamic arguments at call time (determined by the combination of num and fn).

C.4. Other Changes

To support the implicit result argument convention in the presence of optional and "rest" arguments, the implicit result argument will be the first argument in a message. (Previously, it was assumed to be the last argument.) Thus, the "plus" operation would be defined with:

op + result> addend

This allows operations that are used in a functional style to also use optional and rest arguments.

D. Energetic Secrets

The technique of Energetic Secrets replaces Tuples with SealedEnvelopes as messages in the Joule communication model, incorporating public-key semantics into the communication foundations. This change simplifies the Joule semantics (answering questions like, "What is a message selector?"), incorporates the authentication and other security properties of Verifiers into the foundation, and improves the potential efficiency of an untyped Joule implementation by enabling C++-style dispatch. This Appendix introduces the Energetic Secrets concepts and uses. In later versions of this document, the concepts in this Appendix will be integrated into the main body of the text and specified in more detail.

In using Energetic Secrets, each potential operation (message selector) is represented by a pair of a Sealer and an Unsealer (which we will call an Un/Sealer pair), roughly corresponding to send and receive rights for messages of that operation. When a Sealer is applied to arguments, it seals them in a new SealedEnvelope that can only be opened by the corresponding Unsealer. The Unsealer is used by receivers of the SealedEnvelope to recognize the message and extract the arguments.

D.1. Sending Messages

Sealers and Unsealers are typically used implicitly: what had formerly been a Tuple expression (foo: arg1 arg2) implicitly applies a Sealer to arguments to produce an Envelope; **Switch** and **Server** constructs implicitly extract using Unsealers; and the **Type** form creates Sealer/Unsealer pairs. The statement

- receiver anOperation: arg1 arg2

which sends an envelope sealed with the Sealer for anOperation: containing arg1 and arg2, is equivalent to

- receiver (anOperation:sealer seal: arg1 arg2)

in which the seal: operation, sent to anOperation:sealer, produces an Envelope which is then sent to receiver.

Energetic Secrets introduces a new syntactic convention, shown in the example above: labels implicitly refer to sealers and unsealers with a naming convention of appending sealer or unsealer to the end. (Operators append :sealer or :unsealer.) The identifier, anOpera-

tion:sealer, is just a normal identifier, which is bound to the Sealer for anOperation:. Note that the sending of the seal: message to the Sealer similarly invokes a Sealer and produces an envelope; seal:sealer is a primitively supplied Sealer (along with a few others like ::sealer) which serves to bottom out the mechanism for envelope creation.

D.2. Receiving Messages

Messages are received using **ForAll** and choose: as before. Recognizing and parsing are different for envelope messages, however. The code below shows the expansion of a **Switch** form, which is part of the expansion for a **Server** form.

```
Switch envelope
    case foo: a b
        scope in which a and b are visible
    case bar: c
        scope in which c is visible
endSwitch
```

The above **Switch** construct semantically expands into code involving the unseal: method, as shown in the following fragment for the foo:unsealer branch of the **Switch**:

```
Define num, fn
    • foo:unsealer unseal*: envelope num> fn>
endDefine
If num = 2
    Define a = fn :: 0, b = fn :: 1 endDefine
    scope in which a and b are visible
endIf
```

The num and fn revealed by the unseal*: operation are just like the num and fn used for "rest" arguments in Appendix C, *Optional Arguments* (and are in fact used to implement "rest" arguments).

The invocation of the foo:unsealer takes an envelope (received as a message to a **ForAll**, for instance) and, if the envelope really is an envelope sealed by the corresponding foo:sealer, reveals num which is the number of arguments in the envelope (presumably 2 in this case), and a server that will reveal each argument when called with an integer index and a result port. The remainder of the code invokes the supplied argument function to bind the arguments and executes the nested body. If the unseal failed, the revealed num would be –1.

D.3. Sealer and Unsealer Types

Sealers and Unsealers are methodical servers that respond to the protocol below.

```
There are no operations on Envelopes beyond the basic ones.
Type SealedEnvelope
    super Basic
endType
Type Unsealer
    super Basic
    Given a SealedEnvelope sealed by the corresponding Sealer, reveal the number
        of arguments in the envelope and a server that will reveal the arguments. The
        'fn' may only be invoked once per argument.
    op unseal*: envelope num> fn>
endType
```

```
Type Sealer
    super Basic
    Given any number of args, create a SealedEnvelope that encapsulates them,
        and which can only be opened by the corresponding Unsealer.
    op seal: arg... envelope>
    Like seal: except the num and fn arguments are required and are a number and
        function so that users can supply a dynamic set of arguments computed at run
        time.
    op seal*: arg... num fn envelope>
endType

Type make-un/sealer
    op :: sealer> unsealer>
endType
```

D.4. Types and Virtual Un/Sealers

The Un/Sealer pairs are typically generated by the **Type** form. The straightforward expansion is to generate a different Un/Sealer pair for each selector. Instead, the **Type** form can expand to virtual Un/Sealers (unsealers implemented in Joule) to enable the use of a C++-style vtable implementation of message dispatch. Where the code:

```
Type T
    op foo: a b
    op bar: c
endType
```

would normally expand to include un/sealer creation as in:

```
Define    foo:sealer, foo:unsealer,
          bar:sealer, bar:unsealer
    • make-un/sealer :: foo:sealer> foo:unsealer>
    • make-un/sealer :: bar:sealer> bar:unsealer>
endDefine
```

it would expand instead to create a single Un/Sealer used for the whole type and would create virtual Un/Sealers that encode a vtable index (for example the operation's number in the Type) for each operation.

```
Define    foo:sealer, foo:unsealer,
          bar: sealer, bar:unsealer
    Define T:sealer, T:unsealer
        • make-un/sealer :: T:sealer> T:unsealer>
     endDefine
    • virtual-sealer :: T:sealer 0 foo:sealer>
    • virtual-unsealer :: T:unsealer 0 foo:unsealer>
    • virtual-sealer :: T:sealer 1 bar:sealer>
    • virtual-unsealer :: T:unsealer 1 bar:unsealer>
endDefine
```

When sealing, a virtual sealer would take all the supplied arguments, prefix the vtable index (0 for foo in the above case) and then seal with T:sealer. Servers without T:unsealer would be unable to open the envelope, so they couldn't discover the virtualization. The virtualized foo:unsealer would attempt to unwrap with T:unsealer, then check to see whether the first argument is 0 (the vtable index for foo), and only if

both tests pass would it reveal the arguments of the envelope. The advantage of this scheme is that a compiler (or even a smart server) could expand the **Switch** statement shown in Section D.2 into a single unseal operation using T:unsealer (instead of one per case alternative) followed by an indirect jump through a vtable using the index. This will be described in detail (along with Joule implementations of virtual sealers and the fast-dispatch Type expansion) in future versions of this manual.

D.5. Certifying Requests

The last example application of Energetic Secrets presented in this Appendix is certifying properties of requests. A virtualized sealer can dynamically type-check arguments and refuse to produce a sealed Envelope unless the arguments type-check correctly. It can check other properties as well, including relations between the arguments, and extending to full pre-condition checking of the arguments. It can ensure durability by wrapping up, not the arguments, but rather the results of verifying the arguments, potentially reproducing them so that the receiver will be assured that the arguments will remain available.

Finally, modules can export different sealers for the same operation that implement different checks. The exporting module could give the sealers with fewer checks to clients who could prove they passed some trusted analysis that statically checks preconditions (such as argument types).

E. Bibliography

[1] Abelson, Harold, and Sussman, Gerald Jay, *Structure and Interpretation of Computer Programs.* Cambridge, MA: MIT Press, 1985.

[2] Agha, Gul, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA: MIT Press, 1986.

[3] Alchian, Armen A., and Allen, William R., *University Economics.* 2nd Ed. Belmont, CA: Wadsworth, 1968.

[4] Ames, Bruce N., Magaw, Renae, and Gold, Lois Swirsky, "Ranking Possible Carcinogenic Hazards," in *Science* (17 April 1987) Vol. 236.

[5] Artsy, Y., and Finkel, R. "Simplicity, Efficiency, and Functionality in Designing a Process Migration Facility," in *Proceedings of the Second Israel Conference on Computer Systems and Software Engineering* (IEEE, Tel Aviv, Israel, May 1987).

[6] Artsy, Y., Chang, H-Y, and Finkel, R., *Processes Migrate in Charlotte.* Computer Sciences Technical Report #655. Madison: University of Wisconsin, August 1986.

[7] Artsy, Yeshayahu, and Livny, Miron, *An Approach to the Design of Fully Open Computing Systems.* Computer Sciences Technical Report #689. Madison, WI: University Of Wisoconsin, 1987.

[8] Axelrod, Robert, *The Evolution of Cooperation.* New York: Basic Books, 1984.

[9] Barak, A., and Shiloh, A., "A Distributed Load-Balancing Policy for a Multicomputer," in *Software Practice and Experience* (September 1985) 15.

[10] Barstow, David R., Shrobe, Howard E., and Sandwall, Erik (eds.), *Interactive Programming Environments.* New York: McGraw-Hill, 1984.

[11] Barto, Andrew G., "Game Theoretic Cooperativity in Networks of Self-Interested Units," in Denker, John S. (ed.), *Neural Networks for Computing.* New York: American Institute of Physics, 1986.

[12] Birrell, Andrew D., Levin, Roy, Needham, Roger M., and Schroeder, Michael D., "Grapevine: an Exercise in Distributed

Computing," in *Communications of the ACM* (April 1982) Vol. 25, No. 4.

[13] Bishop, Peter B., *Computers with a Large Address Space and Garbage Collection*. MIT/LCS/TR-178. Cambridge, MA: MIT Press, 1977.

[14] Brooks, Frederick P., Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[15] Buchanan, James M., and Tullock, Gordon, *The Calculus of Consent: Logical Foundations of Constitutional Democracy*. Ann Arbor, MI: University of Michigan Press, 1965.

[16] Chaum, David, "Design Concepts for Tamper Responding Systems," in *Advances in Cryptology: Proceedings of Crypto '83*. New York: Plenum Press, 1984.

[17] Cheriton, D. R., "The V Kernel: A Software Base for Distributed Systems," in *IEEE Software* (April 1984) Vol. 1, No. 2.

[18] Clinger, Will, *Foundations of Actor Semantics*. MIT AI-TR-633. Cambridge, MA: MIT Press, 1981.

[19] Coase, R. H., "The Nature of the Firm," in *Economica: New Series* (1937), Vol. IV, reprinted in Stigler, G. J., and Boulding, K. E. (eds.), *Readings in Price Theory*. Chicago: Richard D. Irwin, Inc., 1952.

[20] Conway, M. E., "How Do Committees Invent?" in *Datamation* (April 1968) Vol. 14, No. 4.

[21] Cox, Brad J., *Object Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, 1986.

[22] Davison, A., "POOL: A PARLOG Object-Oriented Language," Dept. of Computing, Imperial College, 1987.

[23] Dawkins, Richard, *The Extended Phenotype*. New York: Oxford University Press, 1982.

[24] Dawkins, Richard, *The Selfish Gene*. New York: Oxford University Press, 1976.

[25] Demers, Alan, Greene, Dan, Hauser, Carl, Irish, Wes, Larson, John, Shenker, Scott, Sturgis, Howard, Swinehart, Dan, and Terry, Doug, "Epidemic Algorithms for Replicated Database Maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (ACM, Vancouver, BC, 1987).

[26] Denning, Peter J., "The Working Set Model for Program Behavior," in *Communications of the ACM* (May 1968) Vol. 2, No. 5.

[27] Dijkstra, E. W., "Co-operating Sequential Processes," in Genuys, F. (ed.), *Programming Languages*. New York: Academic Press, 1968.

[28] Drexler, K. Eric, "Molecular Engineering: An Approach to the Development of General Capabilities for Molecular Manipulation," in *Proceedings of the National Academy of Sciences USA* (Sept. 1981) Vol. 78, No. 9.

[29] Drexler, K. Eric, "Rod Logic and Thermal Noise in the Molecular Nanocomputer," in *Proceedings of the Third International Symposium on Molecular Electronic Devices.* Amsterdam: Elsevier Science Publishers, 1988.

[30] Drexler, K. Eric, and Mark S. Miller, "Incentive Engineering for Computational Resource Management," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[31] Drexler, K. Eric, *Engines of Creation*. Garden City, NY: Anchor Press / Doubleday, 1986.

[32] Drexler, K. Eric, *Hypertext Publishing and the Evolution of Knowledge.* Palo Alto, CA: Foresight Institute, 1986.

[33] Epstein, Richard A., *Takings: Private Property and the Power of Eminent Domain*. Cambridge, MA: Harvard University Press, 1985.

[34] Ferguson, D.F. "The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms" (doctoral dissertation).

[35] Friedman, Daniel, "On the Efficiency of Experimental Double Auction Markets," in *American Economic Review* (March 1984) Vol. 24, No. 1.

[36] Friedman, David, *The Machinery of Freedom: Guide to a Radical Capitalism*. New York: Harper and Row, 1973.

[37] Friedman, Milton, and Schwartz, Anna, "The Great Contraction," in *A Monetary History of the United States, 1867-1960.* Princeton, NJ: Princeton University Press / National Bureau of Economic Research, 1963.

[38] Gehringer, Edward F., *Capability Architectures and Small Objects.* Ann Arbor, MI: UMI Research Press, 1982.

[39] Goldberg, Adele, and Robson, Dave, *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.

[40] Granovetter, Mark, "The Strength of Weak Ties," in *American Journal of Sociology* (1977) Vol. 78.

[41] Gregory, S., *Parallel Logic Programming in PARLOG: The Language and Its Implementation.* Reading, MA: Addison-Wesley, 1987.

[42] Haase, Kenneth W., Jr., "Discovery Systems," in *ECAI '86: The 7th European Conference on Artificial Intelligence* (July 1986), Vol. 1.

[43] Hamming, R. W., "One Man's View of Computer Science," in Ashenhurst, Robert L., and Graham, Susan (eds.), *ACM Turing Award Lectures: The First Twenty Years 1966-1985.* Reading, MA: Addison-Wesley, 1987.

[44] Hanson, Robin, *Toward Hypertext Publishing: Issues and Choices in Database Design*, in press. Draft available from Foresight Institute, Palo Alto, CA, 1987.

[45] Hardin, Garrett, "The Tragedy of the Commons," in *Science* (13 December 1968) Vol. 162.

[46] Harris, Jed, Yu, Chee, Harris, Britton, *Market Based Scheduling* (1987) in preparation.

[47] Hayek, Friedrich A., "Cosmos and Taxis," in *Law, Legislation, and Liberty, Vol. 1: Rules and Order.* Chicago: University of Chicago Press, 1973.

[48] Hayek, Friedrich A., "Economics and Knowledge," from *Economica, New Series* (1937), Vol. IV.; reprinted in Hayek, Friedrich A. (ed.), *Individualism and Economic Order.* Chicago: University of Chicago Press, 1948.

[49] Hayek, Friedrich A., *Denationalisation of Money*, 2nd Ed. London: The Institute of Economic Affairs, 1978.

[50] Hayek, Friedrich A., *New Studies in Philosophy, Politics, Economics, and the History of Ideas.* Chicago: University of Chicago Press, 1978.

[51] Hayek, Friedrich A., *The Constitution of Liberty.* Chicago: University of Chicago Press, 1978.

[52] Hayek, Friedrich A., *The Counter-Revolution of Science: Studies on the Abuse of Reason*. Indianapolis: Liberty Press, 1979.

[53] Hayek, Friedrich A., *Unemployment and Monetary Policy: Government as Generator of the "Business Cycle."* San Francisco, CA: Cato Institute, 1979.

[54] Hewitt, Carl, "Concurrency in Intelligent Systems," in *AI Expert*, No. 1, 1986.

[55] Hewitt, Carl, "Offices are Open Systems," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[56] Hewitt, Carl, "The Challenge of Open Systems," in *Byte* (April 1985).

[57] Hirsch, M., Silverman, W., and Shapiro, E., *Layers of Protection and Control in the Logix System.* Weizmann Institute Technical Report CS86-19.

[58] Hirsh, Susan, Kahn, Kenneth M., and Miller, Mark S., *Interming: Unifying Keyword and Positional Notations.* Palo Alto, CA: Xerox PARC, 1987.

[59] Hoare, C. A. R., *Communicationg Sequential Processes.* New York: Prentice-Hall, 1985.

[60] Hofstadter, Douglas R., "Dilemmas for Superrational Thinkers, Leading Up to a Luring Lottery," in *Metamagical Themas: Questing for the Essence of Mind and Pattern.* New York: Basic Books, 1985.

[61] Hofstadter, Douglas R., "The Prisoner's Dilemma Computer Tournaments and the Evolution of Cooperation," in *Metamagical Themas: Questing for the Essence of Mind and Pattern*. New York: Basic Books, 1985.

[62] Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R. *Induction: Processes of Inference, Learning, and Discovery.* Cambridge, MA: MIT Press, 1986.

[63] INMOS Limited, *Occam Programming Manual.* London: Prentice-Hall International, 1984.

[64] Jacobson, Gary, and Hillkirk, John, *Xerox: American Samurai.* New York: Macmillan, 1986.

[65] Kahn, Kenneth M., and Mark S. Miller, "Language Design and Open Systems," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[66] Kahn, Kenneth M., Tribble, Eric Dean, Miller, Mark S., and Bobrow, Daniel, "Vulcan: Logical Concurrent Objects," in Shriver, B., and Wegner, P. (eds.), *Research Directions in Object-Oriented Programming* and in Shapiro, E. (ed.), *Concurrent Prolog.* Cambridge, MA: MIT Press, 1987.

[67] Kahn, Kenneth, *A Partial Evaluator of Lisp Written in a Prolog Written in Lisp Intended to be Applied to the Prolog and Itself which in turn is Intended to be Given to Itself Together with the Prolog to Produce a Prolog Compiler.* UPMAIL Tech. Report No. 17. University of Uppsala, Sweden, 1983.

[68] Keynes, John Maynard, *The General Theory of Employment, Interest, and Money.* San Diego, CA: Harcourt Brace Jovanovitch, 1964.

[69] Kornfeld, William A., *Using Parallel Processing for Problem Solving.* MIT-AI-561. Cambridge, MA: MIT AI Lab, 1979.

[70] Kornfeld, William A., and Hewitt, Carl, "The Scientific Community Metaphor," in *IEEE Transactions on Systems, Man, and Cybernetics* (IEEE, 1981) SMC-11.

[71] Kowalski, R., "Logic-based Open Systems," Dept. of Computing, Imperial College, September 1985.

[72] Kurose, James F., Schwartz, Mischa, and Yemini, Yechiam, "A Microeconomic Approach to Decentralized Optimization of Channel Access Policies in Multiaccess Networks," in *Proceedings of the Fifth International Conference on Distributed Computing Systems*, Denver, CO, May 1985.

[73] Leach, P.J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., and Stumph, B. L., "The Architecture of an Integrated Local Network," in *IEEE Journal on Selected Areas in Communication* (IEEE, November 1983).

[74] Lenat, Douglas B., "The Role of Heuristics in Learning by Discovery: Three Case Studies," in Michalski, Rysznard S., Carbonell, Jaime G., and Mitchell, Tom M. (eds.), *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA: Tioga Publishing Company, 1983.

[75] Lenat, Douglas B., and Brown, John Seely, "Why AM and Eurisko Appear to Work," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[76] Levy, Henry M., *Capability-Based Computer Systems.* Bedford, MA: Digital Press, 1984.

[77] Lieberman, Henry, and Hewitt, Carl, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," in *Communications of the ACM* (June 1983) Vol. 26, No. 6.

[78] Lindstrom, G., "Functional Programming and the Logical Variable," *12th ACM Symposium on Principles of Programming Languages* (New Orleans, 1985).

[79] Liskov, Barbara, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[80] Liskov, Barbara, Herlihy, M., and Gilbert, L., "Limitations of synchronous communication with static process structure in languages for distributed computing," *Proceedings of the Thirteenth Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986.

[81] Malone, Thomas W., "Organizing Information Processing Systems: Parallels Between Human Organizations and Computer Systems," in Zacharay, W., Robertson, S., and Black, J. (eds.), *Cognition, Computation, and Cooperation*. Norwood, NJ: Ablex, 1986.

[82] Malone, Thomas W., Fikes, R. E., and Howard, M. T., "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[83] Malone, Thomas W., Yates, Joanne, and Benjamin, Robert I., "Electronic Markets and Electronic Hierarchies," in *Communications of the ACM* (June 1987) Vol. 30, No. 6.

[84] March, J. G., "Footnotes to Organizational Change," in *Administrative Science Quarterly* (1981) 26.

[85] McClelland, James L., Rumelhart, David E., and PDP Research Group, *Parallel Distributed Processing. Volumes 1 and 2.* Cambridge, MA: MIT Press, 1986.

[86] McDermott, Drew, "A Critique of Pure Reason," in Levesque, Hector (ed.), *Computational Intelligence.* National Research Council of Canada, 1987.

[87] McGee, John S., "Predatory Price Cutting: The Standard Oil (N.J.) Case," in *Journal of Law and Economics* (October 1958) 1.

[88] Miller, Mark S., and K. Eric Drexler, "Comparative Ecology: A Computational Perspective," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[89] Miller, Mark S., and K. Eric Drexler, "Markets and Computation: Agoric Open Systems," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[90] Miller, Mark S., Bobrow, Daniel G., Tribble, Eric Dean, and Levy, Jacob, "Logical Secrets," in Shapiro, Ehud (ed.), *Concurrent Prolog: Collected Papers.* Cambridge, MA: MIT Press, 1987.

[91] Minsky, Marvin, "Steps Toward Artificial Intelligence," in Feigenbaum, Edward A., and Feldman, Julian (eds.), *Computers and Thought*. Malabar, FL: Robert E. Krieger, 1981.

[92] Minsky, Marvin, *The Society of Mind*. New York: Simon and Schuster, 1986.

[93] Nelson, B., *Remote Procedure Call*. CSL-81-9. Palo Alto, CA: Xerox PARC, 1981.

[94] Nelson, Theodor, *Literary Machines*. Available from the author.

[95] Nisbett, Richard, and Ross, Lee, *Human Inference: Strategies and Shortcomings of Social Judgment*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[96] Ohki, M., Takeuchi, A., and Furukawa, K., "An Object-Oriented Programming Language Based on the Parallel Logic Language KL1," in *Logic Programming: Proceedings of the Fourth International Conference*. MIT Press.

[97] Organick, Elliott I., *A Programmer's View of the Intel 432 System*. New York: McGraw-Hill, 1983.

[98] Popper, Karl R., O*bjective Knowledge: An Evolutionary Approach*. London: Oxford University Press, 1972.

[99] Pountain, D. *A Tutorial Introduction to Occam Programming*. INMOS, 1986.

[100] Quarterman, John S., Silberschatz, Abraham, and Peterson, James L., "4.2BSD and 4.3BSD as Examples of the UNIX System," in *ACM Computing Surveys* (December 1985) Vol. 17, No. 4.

[101] Raffia, Howard, *Decision Analysis: Introductory Lectures on Choices under Uncertainty*. Reading, MA: Addison-Wesley, 1970.

[102] Rao, Ramana Balusu, *Toward Interoperability and Extensibility–in Window Environments via Object-Oriented Program-ming*. Masters thesis, MIT Press, 1987.

[103] Rashid, Richard, "From RIG to Accent to Mach: The Evolution of a Network Operating System," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[104] Rees, Jonathan A., and Adams, Norman I., IV, "T: A Dialect of Lisp or, Lambda: The Ultimate Software Tool," in *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming* (August 1982).

[105] Rivest, R., Shamir, A., and Adelman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," in *Communications of the ACM* (Feb. 1978) Vol. 21, No. 2.

[106] Safra, S., and Shapiro, Ehud, "Meta-Interpreters For Real," in *Proceedings, IFIP-86* (1986).

[107] Shapiro, E., *Algorithmic Program Debugging*. Cambridge, MA: MIT Press, 1982.

[108] Shapiro, E., and Takeuchi, A., "Object-Oriented Programming in Concurrent Prolog," in *New Generation Computing* (July 1983) Vol. 1, No. 1.

[109] Shapiro, Ehud (ed.), *Concurrent Prolog: Collected Papers.* Cambridge, MA: MIT Press, 1987.

[110] Shapiro, Ehud, "Concurrent Prolog: A Progress Report," in *Computer*, IEEE, August 1986.

[111] Shapiro, Ehud, "Systolic Programming: A Paradigm for Parallel Processing," in *Proceedings of the International Conference on Fifth Generation Computer Systems* (1984).

[112] Shrager, Jeff, and Klahr, David, "Instructionless Learning about a Complex Device: The Paradigm and Observations," in *Int. J. Man-Machine Studies* (1986) 25.

[113] Smith, Maynard J., and Price, G. R., "The Logic of Animal Conflicts," in *Nature* (1973) 246.

[114] Smith, Vernon L., "Experimental Methods in the Political Economy of Exchange," in *Science* (10 October 1986) Vol. 234.

[115] Stamos, James W., *A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance.* SCG-82-2. Palo Alto, CA: Xerox PARC, 1982.

[116] Star, Spencer, "TRADER: A Knowledge-Based System for Trading in Markets," in *Economics and Artificial Intelligence First International Conference* (Aix-En-Provence, France, September 1986).

[117] Stefik, Mark, "The Next Knowledge Medium," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[118] Stefik, Mark, Foster, Gregg, Bobrow, Daniel G., Lahn, Kenneth, Lanning, Stan, and Suchman, Lucy, "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," in *Communications of the ACM* (January 1987) Vol. 30, No. 1.

[119] Strom, R., and Yemini, S., "NIL: An Integrated Language and System for Distributed Computing," *Proceedings of SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983.

[120] Sutherland, I.E., "A Futures Market in Computer Time," in *Communications of the ACM* (June 1968) Vol. 11, No. 6.

[121] Tanenbaum, Andrew S., and van Renesse, Robbert, "Distributed Operating Systems," in *ACM Computing Surveys.* NewYork: ACM, 1985.

[122] Terry, Douglas Brian, *Distributed Name Servers: Naming and Caching in Large Distributed Environments.* CSL-85-1. Xerox PARC, February 1985.

[123] Theriault, D., *Issues in the Design and Implementation of Act 2.* AI-TR-728. Cambridge, MA: MIT AI Lab, 1983.

[124] Tribble, Eric Dean, Miller, Mark S., Kahn, Kenneth M., Bobrow, Daniel, Abbott, C., and Shapiro, Ehud, "Channels: A Generalization of Streams," *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press.

[125] Tullock, Gordon, *The Organization of Inquiry*. Durham, NC: Duke University Press, 1966.

[126] Tullock, Gordon, *The Vote Motive*. London: The Institute of Economic Affairs, 1976.

[127] Ueda, K., *Guarded Horn Clauses*. Cambridge, MA: MIT Press, 1987.

[128] Ungar, David Michael, *The Design and Evaluation of a High Performance Smalltalk System*. Cambridge, MA: MIT Press, 1987.

[129] Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J.O., and Stornetta, W.S. "Spawn: A Distributed Computational Economy." *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, February 1992.

[130] Wallace, C.S. and Pose, R.D. "Charging in a Secure Environment" *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence*, Bremen, FRG, 1990. (A revised version has been published in *Security and Persistence*, Bremen 1990. J. Rosenberg and J.L. Keedy (Editors) Springer-Verlag Workshops in Computing Series. ISBN 3-540-19646-3, pp. 85-96.)

[131] Wickler, Wolfgang, *Mimicry in Plants and Animals*. New York: World University Library / McGraw-Hill, 1968.

[132] Williamson, Oliver, *Markets and Hierarchies: Analysis and Anti-Trust Implications*. New York: Free Press, 1975.

[133] Wilson, Edward O., *Sociobiology*. Cambridge, MA: Belknap Press / Harvard University Press, 1975.

[134] Winograd, Terry, and Flores, Fernando, *Understanding Computers and Cognition*. Norwood, NJ: Ablex, 1986.

[135] Xerox, *Courier: The Remote Procedure Call Protocol*. Stamford, CT: Xerox Corp., 1982.

Index