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.

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.

```
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
```

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"