ActorScript™ extension of C#®, Java®, Objective C®, C++, JavaScript®, and SystemVerilog using iAdaptive™ concurrency for antiCloud™ privacy and security:¹

One computer is no computer in IoT

Carl Hewitt

This article is dedicated to Alonzo Church, John McCarthy, Ole-Johan Dahl and Kristen Nygaard.

ActorScript™ is a general purpose programming language for efficiently implementing robust applications using iAdaptive™ concurrency that manages resources and demand. It is differentiated from previous languages by the following:

- Universality
  - Ability to directly specify exactly what Actors can and cannot do
  - Everything is accomplished with message passing using types including the very definition of ActorScript itself.
  - Messages can be directly communicated without requiring indirection through brokers, channels, class hierarchies, mailboxes, pipes, ports, queues etc. Programs do not expose low-level implementation mechanisms such as threads, tasks, locks, cores, etc. Application binary interfaces are afforded so that no program symbol need be looked up at runtime. Functional, Imperative, Logic, and Concurrent programs are integrated.
  - A type in ActorScript is an interface that does not name its implementations (contra to object-oriented programming languages beginning with Simula that name implementations called “classes” that are types). ActorScript can send a message to any Actor for which it has an (imported) type.
  - Concurrency can be dynamically adapted to resources available and current load.

¹ C# is a registered trademark of Microsoft, Inc.
Java and JavaScript are registered trademarks of Oracle, Inc.
Objective C is a registered trademark of Apple, Inc.
² with no single point of failure
• Safety, security and readability
  o Programs are extension invariant, i.e., extending a program does not change the meaning of the program that is extended.
  o Applications cannot directly harm each other.
  o Variable races are eliminated while allowing flexible concurrency.
  o Lexical singleness of purpose. Each syntactic token is used for exactly one purpose.
• Performance
  o Imposes no overhead on implementation of Actor systems in the sense that ActorScript programs are as efficient as the same implementation in machine code. For example, message passing has essentially same overhead as procedure calls and looping.
  o Execution dynamically adjusted for system load and capacity (e.g. cores)
  o Locality because execution is not bound by a sequential global memory model
  o Inherent concurrency because execution is not limited by being restricted to communicating sequential processes
  o Minimize latency along critical paths

ActorScript attempts to achieve the highest level of performance, scalability, and expressibility with a minimum of primitives.

**Message passing using types is the foundation of system communication:**
• Messages are the unit of communication
• Types enable secure communication with Actors

*Computer software should not only work; it should also appear to work.*

---

1 Performance can be tricky as illustrated by the following:
• “Those who would forever give up correctness for a little temporary performance deserve neither correctness nor performance.” [Philips 2013]
• “The key to performance is elegance, not battalions of special cases” [Jon Bentley and Doug McIlroy]
• “If you want to achieve performance, start with comprehensible.” [Philips 2013]
• Those who would forever give up performance for a feature that slows everything down deserve neither the feature nor performance.

II Each type is an Actor. However, it may be the case that a type will work some places and not others. For example, to be used in message passing, the type of an address may require access to particular hardware.
Introduction
ActorScript is based on the Actor mathematical model of computation that treats “Actors” as the universal conceptual primitive of digital computation [Hewitt, Bishop, and Steiger 1973; Hewitt 1977; Hewitt 2010a]. Actors have been used as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems.

ActorScript
ActorScript is a general purpose programming language for implementing massive local and nonlocal concurrency.

This paper makes use of the following typographical conventions that arise from underlying namespaces for types, messages, language constructs, syntax categories, etc.¹
- type identifiers
  - blue for types in general (e.g., Account)
  - green for the special case of implementation types (e.g., SimpleAccount)
- program variables (e.g., aBalance)
- message names (e.g., withdraw)
- reserved words² for language constructs (e.g., Actor)
- logical variables (e.g., x)
- comments in programs (e.g. /* this is a comment */)

There is a diagram of the syntax categories of ActorScript in an appendix of this paper in addition to an appendix with an index of symbols and names along with an explanation of the notation used to express the syntax of ActorScript.³

Actors
ActorScript is based on the Actor Model of Computation [Hewitt, Bishop, and Steiger 1973; Hewitt 2010a] in which all computational entities are Actors and all interaction is accomplished using message passing.

The Actor model is a mathematical theory that treats “Actors” as the universal conceptual primitive of digital computation. The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. Unlike previous models of computation, the Actor model was inspired by

¹ The choice of typography in terms of font and color has no semantic significance. The typography in this paper was chosen for pedagogical motivations and is in no way fundamental. Also, only the abstract syntax of ActorScript is fundamental as opposed to the surface syntax with its many symbols, e.g., →, etc.
physical laws. The advent of massive concurrency through client-cloud computing and many-core computer architectures has galvanized interest in the Actor model.

An Actor is a computational entity that, in response to a message it receives, can concurrently:
- send messages to addresses of Actors that it has
- create new Actors
- designate how to handle the next message it receives.

There is no assumed order to the above actions and they could be carried out concurrently. In addition two messages sent concurrently can be received in either order. Decoupling the sender from communication it sends was a fundamental advance of the Actor model enabling asynchronous communication and control structures as patterns of passing messages.

The Actor model can be used as a framework for modeling, understanding, and reasoning about, a wide range of concurrent systems. For example:
- Electronic mail (e-mail) can be modeled as an Actor system. Mail accounts are modeled as Actors and email addresses as Actor addresses.
- Web Services can be modeled with endpoints modeled as Actor addresses.
- Object-oriented programing objects with locks (e.g. as in Java and C#) can be modeled as Actors.

Actor technology will see significant application for coordinating all kinds of digital information for individuals, groups, and organizations so their information usefully links together. Information coordination needs to make use of the following information system principles:
- **Persistence**: Information is collected and indexed.
- **Concurrency**: Work proceeds interactively and concurrently, overlapping in time.
- **Quasi-commutativity**: Information can be used regardless of whether it initiates new work or becomes relevant to ongoing work.
- **Sponsorship**: Sponsors provide resources for computation, i.e., processing, storage, and communications.
- **Pluralism**: Information is heterogeneous, overlapping and often inconsistent. There is no central arbiter of truth.
- **Provenance**: The provenance of information is carefully tracked and recorded.

The Actor Model is designed to provide a foundation for inconsistency robust information coordination.
Notation
To ease interoperability, ActorScript uses an intersection of the orthographic conventions of Java, JavaScript, and C++ for words and numbers.

Expressions
ActorScript makes use of a great many symbols to improve readability and remove ambiguity. For example the symbol “▮” is used as the top level terminator to designate the end of input in a read-eval-print loop. An Integrated Development Environment (IDE) can provide a table of these symbols for ease of input as explained below:

Expressions evaluate to Actors. For example, 1+3▮ is equivalent to 4▮.

Parentheses “(” and “)”) can be used for precedence. For example using the usual precedence for operators, 3*(4+2▮ is equivalent to 18▮, while 3*4+2▮ is equivalent to 14▮.

Identifiers, e.g., x, are expressions that can be used in other expressions. For example if x is 1 then x+3▮ is equivalent to 4▮. The formal syntax of identifiers is in the following end note: 4.

Types
Types are Actors. Type names are shown as follows:
- blue for types in general (e.g., Account)
- green for the special case of implementation types (e.g., SimpleAccount)

The formal syntax for types is in the following end note: 5.

---

1 sometimes called “names”
2 Furthermore, all special symbols have ASCII equivalents for input with a keyboard. An IDE can convert ASCII for a symbol equivalent into the symbol. See table in an appendix to this article.
3 An IDE can provide a box with symbols for easy input in program development. The grey callout bubble is a hover tip that appears when the cursor hovers above a symbol to explain its use.
4 in the sense of having the same value and the same effects
Identifier Definitions, i.e., ←
An identifier definition has an identifier to be defined followed by “←” followed by the definition. For example, \(x ← 3\) defines the identifier \(x\) to be the Actor 3.

The formal syntax of an identifier definition is in the end note: 6.

Procedure Definitions, i.e., →
A procedure is an Actor that can receive a list of Actors in a message and return an Actor as its value, which can be defined using “Actor”, followed by a procedure name, a list of formal arguments, return type, “→” and body of the procedure. For example,

\[
\text{Actor Double } [v: \text{Integer}]: \text{Integer} → v + v
\]

The formal syntax of a procedure definition is in the end note: 7.

Sending messages to procedures, i.e., ⋄[ ]
Sending a message to a procedure (i.e. “calling” a procedure with arguments) is expressed by an expression that evaluates to a procedure followed by “⋄” followed by a message with arguments delimited by “[" and “"]”. For example, \(\text{Double}, [2+1] ⋄\) means send Double the message \([3]\). Thus \(\text{Double}, [2+1] ⋄\) is equivalent to \(61\).

The formal syntactic definition of procedural message sending is in the end note: 9.
Patterns
Patterns are fundamental to ActorScript. For example,

- 3 is a pattern that matches 3
- “abc” is a pattern that matches “abc”.
- _ is a pattern that matches anything
- $\varnothing_x$ is a pattern that matches the value of x.
- $\varnothing(x+2)$ is a pattern that matches the value of the expression $x+2$.

Identifiers can be bound using patterns as in the following examples:

- x is a pattern that matches “abc” and binds x to “abc”

---

i e.g., _ matches 7
ii An identifier is a name that is used in a program to designate an Actor
Cases, i.e.,  ⎕  ⎕  ⎕

Cases are used to perform conditional testing. In a Cases Expression, an expression for the value on which to perform case analysis is specified first followed by “⎕” and then followed by a number of cases separated by “⎕” terminated by “⎕”。A case consists of

- a pattern followed by “⎕” and an expression to compute the value for the case. All of the patterns before an else case must be disjoint; i.e., it must not be possible for more than one to match.
- optionally (at the end of the cases) one or more of the following cases: “⎕” followed by an optional pattern, “⎕”, and an expression to compute the value for the case. An else case applies only if none of the patterns in the preceding cases match the value on which to perform case analysis.

As an arbitrary example purely to illustrate the above, suppose that the procedure Random, which has no argument and returns Integer, in the following example:

```
Random[ | ]  ⎕
0  ⎕ // Random[ | ] returned 0
  Throw’RandomNumberException[ | ] ⎕  // throw an exception
  // because Fibonacci[0] is undefined
1  ⎕ // Random[ | ] returned 1
6  ⎕ // the value of the cases expression is 6
  else y thatis < 5  ⎕
    // Random[ | ] returned y that is not 0 or 1 and is less than 5
    Fibonacci[y] ⎕  // return Fibonacci of the value returned by Random[ | ]
    else z  ⎕
      // Random[ | ] returned z that is not 0 or 1 and is not less than 5
      Factorial[z] ⎕  // return Factorial of the value returned by Random[ | ]
```

The formal syntax of cases is in the following end note: 11.

---

1 “⎕” is fancy typography for “?”
2 including patterns in previous else cases
3 As is standard, ActorScript uses the token “//” to begin a one-line comment.
4 Reserved words are shown in bold black.
**Binding locals, i.e., Let ← .**

Local identifiers can be bound using “Let” followed by a list of bindings separated by commas and terminated with “;.” Each binding consists of a pattern, “←”, and an expression for the Actor to be matched. For example, \( \text{aProcedure}_c["G", "F", "F"]_! \) is equivalent to the following:

\[
\begin{align*}
\text{Let } x & \leftarrow "F"_a \quad // \text{ x is } "F"_a \\
\text{aProcedure}_c["G", x, x]_! \\
\text{Let } y & \leftarrow \text{aProcedure}_c["G", x, x]_a \\
& \quad // \text{ y is } \text{aProcedure}_c["G", "F", "F"]_a \\
& \quad \\n& \text{anotherProcedure}_c[x, y]_! \\
\end{align*}
\]

Dependent bindings (in which each can depend on previous ones) can be accomplished by nesting Let. For example:

\[
\begin{align*}
\text{Let } x & \leftarrow "F"_a \\
& \quad // \text{ x is } "F"_a \\
\text{Let } y & \leftarrow \text{aProcedure}_c["G", x, x]_a \\
& \quad // \text{ y is } \text{aProcedure}_c["G", "F", "F"]_a \\
& \quad \\n& \text{anotherProcedure}_c[x, y]_! \\
\end{align*}
\]

The above is equivalent to

\[
\begin{align*}
& \text{anotherProcedure}_c["F", \text{aProcedure}_c["G", "F", "F"]_! ]_!
\end{align*}
\]

The formal syntax of bindings is in the following end note: 12.

The formal syntactic definition of named-message sending is in the following end note: 13.

**General Message-passing interfaces**

An interface can be defined using “ Interface ” followed by an interface name, “with”, and a list of message handler signatures, where message handler signature consists of a message name followed by argument types delimited by “[ ]”, “→”, and a return type. For example, the interface type can be defined as follows:

\[
\begin{align*}
\text{Interface } \text{Account with} \text{ availableBalance}[ ] \rightarrow \text{Euro}, \\
& \quad \text{deposit[Euro]} \rightarrow \text{Void}, \\
& \quad \text{withdraw[Euro]} \rightarrow \text{Void}_! \\
\end{align*}
\]
**Actors that change**, *i.e.*, **Actor** using $=$

Using the expressions introduced so far, actors do not change. However, some Actors change behaviors over time.

Message handlers in an Actor execute mutually exclusively while in a region of mutual exclusion which is called “cheese.” In this paper assignable variables are colored orange, which by itself has no semantic significance, *i.e.*, printing this article in black and white does not change any meaning. The use of assignments is strictly controlled in order to achieve better structured programs.  

Below is a diagram for the implementation **SimpleAccount** of **Account**: 

---

10
Variable races are impossible in ActorScript
An Actor can be created using "Actor" optionally followed by the following:
- constructor name with formal arguments delimited using brackets
- declarations of variables terminated by ‘;’
- implementations of interface(s).

ActorScript is referentially transparent in the sense that a variable never changes while in a continuous part of the cheese. For example, in the deposit message handler change is accomplished using the following:

Void afterward myBalance := myBalance + anAmount
which returns Void and updates myBalance for the next message received.

An implementation that of the Account interface can be expressed as follows:

Actor SimpleAccount[startingBalance: Euro]
  myBalance := startingBalance:
  // myBalance is an assignable variable initialized with startingBalance
  implements Account using
  availableBalance[ ]: Euro → myBalance
  deposit[anAmount: Euro]: Void →
    Void afterward myBalance := myBalance + anAmount
    // the next message is processed with
    // myBalance reflecting the deposit
  withdraw[anAmount: Euro]: Void →
    (amount > myBalance) →
    True → Throw Overdrawn[ ]
    False → Void afterward myBalance := myBalance – anAmount
    // the next message is processed with updated myBalance

As a result of the above definition,
Implementation SimpleAccount extends Account

The formal syntax of Actor expressions is in the following end note: 16.

---

1 variable declarations separated by commas
Antecedents, Preparations, and Necessary Concurrency, i.e., □

Concurrency can be controlled using preparation that is expressed in a continuation using preparatory expressions, “●” and an expression that proceeds only after the preparations have been completed.

The following expression creates an account anAccount with initial balance €6 and then concurrently withdraws €1 and €2 in preparation for reading the balance:

\[
\text{Let } \text{anAccount} \leftarrow \text{SimpleAccount}[\€6]. \quad \text{// is a reserved prefix operator}
\text{Prep anAccount, withdraw[€1]},
\text{anAccount, withdraw[€2] ●.}
\quad \text{// proceed only after both of the}
\quad \text{// withdrawals have been acknowledged}
\text{anAccount, availableBalance[[]].}
\]

The above expression returns €3.

Operations are quasi-commutative to the extent that it doesn’t matter in which order they occur.

Quasi-commutativity can be used to tame indeterminacy while at the same time facilitating implementations that run exponentially faster than those in the parallel lambda calculus.\(^1\)

The formal syntax of compound expressions is in the following end note: 17

An expression can be annotated for concurrent execution by preceding it with “□” indicating that the following expression must be considered for concurrent execution if resources are available. For example □Factorial[1000]+□Fibonacci[2000][1] is annotated for concurrent execution of Factorial[1000] and Fibonacci[2000] both of which must complete execution. This does not require that the executions of Factorial[1000] and Fibonacci[2000] actually overlap in time.\(^1\)

The formal syntax of explicit concurrency is in the following end note: 19.

\(^{1}\) For example, implementations using Actors of Direct Logic can be exponentially faster than implementations in the parallel lambda calculus.
Implementing multiple interfaces, i.e., also implements

The above implementation of `Account` can be extended as follows to provide the ability to revoke some abilities to change an account. For example, the `AccountSupervisor` implementation below implements both the `Account` and `AccountRevoker` interfaces as an extension of the implementation `SimpleAccount` where:

Interface `AccountRevoker` with `revokeDepositable[] ⟷ Void, revokeWithdrawable[] ⟷ Void`


```plaintext
withdrawableIsRevoked := False,
depositableIsRevoked := False,
[revoker]: AccountRevoker → AccountRevoker
    // this Actor as AccountRevoker

[account]: Account → Account
    // this Actor as Account
withdrawFee[anAmount: Euro]: Void →
    Void afterward myBalance := myBalance– anAmount
    // withdraw fee even if balance goes negative
```

partially reimplements `Account` using

```plaintext
withdraw[anAmount: Euro]: Void →
    withdrawableIsRevoked

Deposit[anAmount: Euro]: Void →
    depositableIsRevoked
```

also implements `AccountRevoker` using

```plaintext
revokeDepositable[]: Void →
    Void afterward depositableIsRevoked := True

revokeWithdrawable[]: Void →
    Void afterward withdrawableIsRevoked := True
```
As a result of the above definition:

Implementation AccountSupervisor has

\[
\begin{align*}
&[\text{revoker}] \mapsto \text{AccountRevoker}, \\
&[\text{account}] \mapsto \text{Account}, \\
&\text{withdrawFee}[\text{Euro}] \mapsto \text{Void}
\end{align*}
\]

For example, the following expression returns negative €3:

Let anAccountSupervisor ← AccountSupervisor[€3].
Let anAccount ← anAccountSupervisor.\([\text{account}]\).
Let aRevoker ← anAccountSupervisor.\([\text{revoker}]\).

Prep

\[
\begin{align*}
&\text{anAccount}.\text{withdraw}[\text{€2}] \bullet \quad \text{// the balance is €1} \\
&\text{aRevoker}.\text{revokeWithdrawable[]} \bullet \\
&\text{// withdrawableIsRevoked is True} \\
&\text{Try anAccount}.\text{withdraw}[\text{€5}] \quad \text{// try another withdraw} \\
&\text{catch Name}_\circ = \text{Void} \bullet \\
&\text{// ignore the thrown exception} \\
&\text{// myBalance remains €1} \\
&\text{anAccountSupervisor}.\text{withdrawFee}[\text{€4}] \bullet \\
&\text{// €4 is withdrawn even though withdrawableIsRevoked} \\
&\text{// myBalance is negative €3}
\end{align*}
\]

anAccount.\text{availableBalance[]} \bullet

The formal syntax of the programs below is in the following end note: 25

**Type Extension**

Subtyping of an implementation is not allowed so that an implementation can be securely branded.1

The following interface expresses that each Tree has an integer identifier:

Interface Tree with \([\text{hash}]\mapsto \text{Integer}\)

An implementation of Leaf can be defined as an extension of Tree as follows:

Structure Leaf[aString: String]

implements Tree using

\([\text{hash}]\mapsto \text{Integer} \rightarrow \text{Hash}, [\text{aString}]\)

As a result of the above definition:

Implementation structure Leaf[String] extends Tree

---

1 As shown elsewhere in this article, multiple implementations can be used in another implementation. Of course, interface types can be extended
For example,

- "The" is equivalent to the following:
  \[ \text{Let Leaf}[\text{aString}] \leftarrow \text{Leaf["The"].aString}. \]
- \[ \text{Leaf["The"]}.[\text{hash}] \text{ is equivalent to Hash, } [\text{"The"}]. \]

Conversion from of a type to an extension of a type is done using an expression of the extension can followed by “:” and the type. For example, \((\text{Leaf["The"]}: \text{Tree}).[\text{hash}]\) is equivalent to Hash, ["The"].

\textbf{Fork} can be defined as an extension of \textbf{Tree} using:

\begin{verbatim}
Structure Fork[aLeft: Tree, aRight: Tree]
  extends Tree using
    [hash]: Integer \rightarrow Hash, aLeft.[hash], aRight.[hash]
\end{verbatim}

As a result of the above definition:

\begin{verbatim}
Implementation structure Fork[Tree, Tree] extends Tree
\end{verbatim}

For example, Hash. [Hash, ["The"], Hash,"boy"] is equivalent to the following:

\[ (\text{Fork[Leaf["The"], Leaf["boy"]]}).[\text{hash}] \]

Testing for convertibility from of a type to an extension of the type is done using an expression of the extension can followed by “\(\downarrow\)?” and the type. For example,

- \((\text{Leaf["The"]}): \text{Tree}\downarrow\text{Fork} \) is equivalent to \text{False}.
- \((\text{Leaf["The"]}): \text{Tree}\downarrow\text{Leaf} \) is equivalent to \text{True}.

Conversion from of a type to an extension of the type is done using an expression of the extension can followed by “\(\downarrow\)” and the type. For example,

- \((\text{Leaf["The"]}): \text{Tree}\downarrow\text{Leaf} \) is equivalent to \text{Leaf,"The"}.
- \((\text{Leaf["The"]}): \text{Tree}\downarrow\text{Fork} \) throws an exception.

"\(\downarrow\)" followed by a pattern can be used to match the pattern with something which has been extended from the type of that pattern. For example,

\begin{verbatim}
Actor Fringe
  [aTree: Tree]: [<!String> *] \rightarrow
  aTree \downarrow\downarrow Leaf[aString] \# [aString] \downarrow\downarrow Fork[aLeft, aRight] \# [\forall Fringe, aLeft], [\forall Fringe, aRight]] \|^{26}
\end{verbatim}
For example, ["The", "boy"]:::<String>*]▮ is equivalent to the following:
Fringe.[Fork[Leaf["The"], Leaf["boy"]]]▮

The procedure Fringe can be used to define SameFringe? that determines if two trees have the same fringe [Hewitt 1972]:

```
Actor SameFringe?
  [aTree:Tree, anotherTree:Tree]:Boolean →
  Fringe,[aTree] = Fringe,[anotherTree]▮
```

Casting is as allowed only as follows:
1. Casting self to an interface implemented by this Actor
2. Upcasting
   a. an Actor of an implementation type to the interface type of the implementation
   b. an Actor of an interface type to the interface type that was extended
3. Conditional downcasting of an Actor of an interface type to an extension of the interface type. Conditional downcasting of an Actor of an interface type to an extension of the interface type is allowed only to an extension of I. For example, if x is of interface type I, then either
   i. E is an extension of I and there is some y of type E such that x=y:I and therefore x↓E=y
   ii. x↓E throws an exception because E is not an extension of I or there is no y of type E such that x=y:I

Swiss cheese
Swiss cheese [Hewitt and Atkinson 1977, 1979; Atkinson 1980] is a generalization of mutual exclusion with the following goals:
- **Generality:** Ability to conveniently program any scheduling policy
- **Performance:** Support maximum performance in implementation, e.g., the ability to minimize locking and to avoid repeatedly recalculating a condition for proceeding.
- **Understandability:** Invariants for the variables of a mutable Actor should hold whenever entering or leaving the cheese.
- **Modularity:** Resources requiring scheduling should be encapsulated so that it is impossible to use them incorrectly.

---

1 An implementation type cannot be downcast because there is nothing to which to downcast. Note that this means that an implementation type cannot be subtyped although an implementation can use other implementations for modularity. Of course, for interface types there is no semantic guarantee of what an implementation of the interface might do as long as it obeys the signatures.
By contrast with the nondeterministic lambda calculus, there is an always-halting Actor Unbounded that when sent a \([\_]\) message can compute an integer of unbounded size. This is accomplished by creating a \textbf{Counter} with the following variables:

- \texttt{count} initially 0
- \texttt{continue} initially \texttt{True}

and concurrently sending it both a \texttt{stop[\_]} message and a \texttt{go[\_]} message such that:

- When a \texttt{go[\_]} message is received:
  1. if \texttt{continue} is \texttt{True}, increment \texttt{count} by 1 and return the result of sending this counter a \texttt{go[\_]} message.
  2. if \texttt{continue} is \texttt{False}, return \texttt{Void}
- When a \texttt{stop[\_]} message is received, return \texttt{count} and set \texttt{continue} to \texttt{False} for the next message received.

By the Actor Model of Computation [Clinger 1981, Hewitt 2006], the above Actor will eventually receive the \texttt{stop[\_]} message and return an unbounded number.

A diagram is shown below for an implementation of \textbf{Counter}. In the diagram, a hole in the cheese is highlighted in grey and variables are shown in orange. The color has no semantic significance.
Actor CreateUnbounded

\[
\text{Unbounded: Integer } \rightarrow \\
\text{Let } a\text{Counter } \gets \text{Counter[ ]}. \quad \text{// let } a\text{Counter } \text{be a new Counter} \\
\text{Prep } a\text{Counter, go[ ]}. \quad \text{// send } a\text{Counter } \text{a go message and concurrently} \\
\text{aCounter, stop}[ ] \quad \text{// return the result of sending } a\text{Counter } \text{stop[ ]}
\]

As a notational convenience, when an Actor receives message then it can send an arbitrary message to itself by prefixing it with “∎∎” as in the following example for the Actor implementation SimpleCounter:

Actor Counter[ ]
\[
\text{count } \coloneqq 0, \quad \text{// the variable } \text{count } \text{is initially 0} \\
\text{continue } \coloneqq \text{True}. \\
\text{stop[ ]: Integer } \rightarrow \text{count} \quad \text{// return } \text{count} \\
\text{afterward } \text{continue } \coloneqq \text{False}. \\
\text{// continue is updated to False for the next message received} \\
\text{go[ ]: Void } \rightarrow \\
\text{continue } \diamond \\
\text{True } \circ \text{Prep } \text{count } \coloneqq \text{count}+1 \bullet \quad \text{// increment } \text{count} \\
\text{hole . . . go[ ] } \mathbin{\diamond} \quad \text{// send go[ ] to this counter} \\
\text{False } \circ \text{Void } \mathbin{\$} \mathbin{\$} \mathbin{\$} \\
\text{// if } \text{continue } \text{is False, return Void}
\]

As a result of the above definition
Implementation Counter has go[ ]\(\rightarrow\) Void,
stop[ ]\(\rightarrow\) Integer

The formal syntax of the programs above is in the following end note: 29
Coordinating Activities

Coordinating activities of readers and writers in a shared resource is a classic problem. The fundamental constraint is that multiple writers are not allowed to operate concurrently and a writer is not allowed to operate concurrently with a reader.

Below are two implementations of readers/writer guardians for a shared resource that implement different policies:

1. ReadingPriority: The policy is to permit maximum concurrency among readers without starving writers.
   a. When no writer is waiting, all readers start as they are received.
   b. When a writer has been received, no more readers can start.
   c. When a writer completes, all waiting readers start even if there are writers waiting.

2. WritingPriority: The policy is that readers get the most recent information available without starving writers.
   a. When no writer is waiting, all readers start as they are received.
   b. When a writer has been received, no more readers can start.
   c. When a writer completes, just one waiting reader is permitted to complete if there are waiting writers.

The interface for the readers/writer guardian is the same as the interface for the shared resource:

\[
\text{Interface } \text{ReadersWriter} \text{ with } \text{read}[\text{Query}] \rightarrow \text{QueryAnswer}, \quad \text{write}[\text{Update}] \rightarrow \text{Void}\]

Cheese diagram for ReadersWriter implementations:

Note:
1. At most one activity is allowed to execute in the cheese.
2. The value of a variable changes only when leaving the cheese.

When an exception is thrown exogenously by an activity that is in a queue (e.g., readersQ, writerQ), a backout handler can be used to clean up cheese variables before rethrowing the exception.

The formal syntax of the programs below is in the following end note: 33

---

1 A variable is orange in the diagram
2 Of course, other external Actors can change.
In the implementations below, preconditions present are commentary for error checking. An exception is thrown if a precondition is not met at runtime. A precondition has no operational effect.

**Actor ReadingPriority**[theResource: ReadersWriter]

invariants numberReading≥0, writing⇒ numberReading=0.

queues readersQ, writersQ. // readersQ and writersQ are initially empty

writing := False.

numberReading := 0.

implements ReadersWriter using

read[aQuery: Query]: QueryAnswer →

Prep (writing ∨ ¬isEmpty writersQ) ◐ // leave cheese while in readersQ

backout (~writing ∧ numberReading=0 ∧ isEmpty readersQ) ◐

True  Void permit writersQ

False Void ⌞)

Preconditions ¬writing. // commentary for error checking

Prep numberReading++. // increment numberReading

permit readersQ

hole theResource, read[aQuery] // leave cheese while reading afterward

(isEmpty writersQ) ◐

True  Permit readersQ also numberReading-–

False  numberReading=1 ◐

Preconditions 34

Prep numberReading≥0 ∨ ¬isEmpty readersQ ∨ writing ∨ ¬isEmpty writersQ ◐

write[anUpdate: Update] Void →

Prep (numberReading>0 ∨ ¬isEmpty readersQ ∨ writing ∨ ¬isEmpty writersQ) ◐

True  Enqueue writersQ ◐ // leave cheese while in writersQ

backout (isEmpty writersQ ∧ ¬writing) ◐

True  Void permit readersQ

False  Void ⌞)

Preconditions 35

Prep writing := True ◐. // record that writing is happening

hole theResource, write[anUpdate] // leave cheese while writing afterward (isEmpty readersQ) ◐

True  Permit writersQ also writing := False

False  Permit readersQ also writing := FalseASHBOARD
Illustration of writing-priority:

**Actor** WritingPriority [theResource: ReadersWriter]

- **invariants**
  - numberReading $\geq$ 0, writing $\Rightarrow$ numberReading $= 0$.
  - queues readersQ, writersQ.

- **writing** := False.
- numberReading := 0.

**implements** ReadersWriter using

- read[aQuery: Query]: QueryAnswer $\rightarrow$
  - Prep
    - (writing $\lor$ $\neg$ Empty writersQ)$\Rightarrow$
      - True $\Rightarrow$ Enqueue readersQ$\bullet$
        // leave cheese while in readersQ
      - backout $\Rightarrow$ writing $\land$ numberReading$= 0$ $\land$ isEmpty readersQ$\Rightarrow$
        - True $\Rightarrow$ Void permit writersQ$
      - False $\Rightarrow$ Void $
    - Void$
  - False $\Rightarrow$ Void $\otimes$.

- **Preconditions**
  - $\neg$writing. // commentary for error checking

- Prep
  - numberReading++
  - permit isEmpty writersQ $\Rightarrow$
    - True $\Rightarrow$ readersQ$\otimes$
    - False $\Rightarrow$ Void $\otimes$
  - hole theResource, read[aQuery] // leave cheese while reading
    - afterward
      - (isEmpty writersQ)$\Rightarrow$
        - True $\Rightarrow$ Permit readersQ also numberReading$\Rightarrow$
        - False $\Rightarrow$ numberReading$= 1$
          - True $\Rightarrow$ Permit writersQ also numberReading$\Rightarrow$
          - False $\Rightarrow$ numberReading$\Rightarrow$
  - write[anUpdate: Update]: Void $\rightarrow$
    - Prep
      - (numberReading $> 0$ $\lor$ $\neg$ isEmpty readersQ $\lor$ writing $\lor$ $\neg$ isEmpty writersQ)$\Rightarrow$
        - True $\Rightarrow$ Enqueue writersQ$\bullet$
          // leave cheese while in writersQ
        - backout (isEmpty writersQ $\land$ $\neg$ writing)$\Rightarrow$
          - True $\Rightarrow$ Void permit readersQ$
        - False $\Rightarrow$ Void $
      - Void
    - False $\Rightarrow$ Void $\otimes$.

- **Preconditions**
  - numberReading$= 0$, $\neg$writing. // commentary for error checking

- Prep
  - writing := True.
  - hole theResource, write[anUpdate] // leave cheese while writing
    - afterward
      - (isEmpty readersQ)$\Rightarrow$
        - True $\Rightarrow$ Permit writersQ also writing$\Rightarrow$ False$
        - False $\Rightarrow$ Permit readersQ also writing$\Rightarrow$ False$\otimes$

22
Conclusion

Before long, we will have billions of chips, each with hundreds of hyper-threaded cores executing hundreds of thousands of threads. Consequently, GOFIP (Good Old-Fashioned Imperative Programming) paradigm must be fundamentally extended. ActorScript is intended to be a contribution to this extension.

Acknowledgements

Important contributions to the semantics of Actors have been made by: Gul Agha, Beppe Attardi, Henry Baker, Will Clinger, Irene Greif, Carl Manning, Ian Mason, Ugo Montanari, Maria Simi, Scott Smith, Carolyn Talcott, Prasanna Thati, and Aki Yonezawa.

Important contributions to the implementation of Actors have been made by: Bill Athas, Russ Atkinson, Beppe Attardi, Henry Baker, Gerry Barber, Peter Bishop, Nanette Boden, Jean-Pierre Briot, Bill Dally, Peter de Jong, Jessie Dedecker, Ken Kahn, Henry Lieberman, Carl Manning, Mark S. Miller, Tom Reinhardt, Chuck Seitz, Dale Schumacher, Richard Steiger, Dan Therault, Mario Tokoro, Darrell Woelk, and Carlos Varela.

Research on the Actor model has been carried out at Caltech Computer Science, Kyoto University Tokoro Laboratory, MCC, MIT Artificial Intelligence Laboratory, SRI, Stanford University, University of Illinois at Urbana-Champaign Open Systems Laboratory, Pierre and Marie Curie University (University of Paris 6), University of Pisa, University of Tokyo Yonezawa Laboratory and elsewhere.

The members of the Silicon Valley Friday AM group made valuable suggestions for improving this paper. Discussions with Blaine Garst were helpful in the development of the implementation of Swiss cheese that doesn't hold a lock as well providing background on the historical development of interfaces. Patrick Beard found bugs and suggested improvements in presentation. Fanya S. Montalvo and Ike Nassi suggested simplifying the syntax. Dale Schumacher found many typos, suggested including a syntax diagram, and suggested improvements to the syntax of collections, binding and assignment. In particular, Dale contributed greatly to the development of the lock-free implementation of cheese in the appendix. Chip Morningstar provided an excellent critique with many useful comments and suggestions. Many important comments and suggestions were provided by Stu Bailey and members of the Silicon Valley FriAM group.

---

1 In the sense that the implementation holds a hardware lock.
ActorScript is intended to provide a foundation for information coordination in client-cloud computing that protects citizens sensitive information [Hewitt 2009b].

Bibliography
Sarita Adve and Hans-J. Boehm Memory Models: A Case for Rethinking Parallel Languages and Hardware CACM. August 2010.
Joe Armstrong Erlang. CACM. September 2010/
Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007a) Interactive small-step algorithms I: Axiomatization Logical Methods in Computer Science. 2007.
Per Brinch Hansen *Monitors and Concurrent Pascal: A Personal History* CACM 1996.


Luca Cardelli and Andrew Gordon *Mobile Ambients* FoSSaCS’98.


Alonzo Church *The Calculi of Lambda-Conversion* Princeton University Press. 1941.


Tyler Close *Web-key: Mashing with Permission* WWW’08.


William Dally and Wills, D. *Universal mechanisms for concurrency* PARLE ’89.


Christopher Fuchs *Quantum mechanics as quantum information (and only a little more)* in A. Khrenikov (ed.) Quantum Theory: Reconstruction of Foundations (Växjo: Växjo University Press, 2002).

Blaine Garst. *Origin of Interfaces* Email to Carl Hewitt on October 2, 2009.


Cordell Green. *Application of Theorem Proving to Problem Solving* IJCAI’69.


Pat Hayes *Some Problems and Non-Problems in Representation Theory* AISB. Sussex. July, 1974


Carl Hewitt, Peter Bishop and Richard Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence* IJCAI’73.


Frederick Knabe *A Distributed Protocol for Channel-Based Communication with Choice* PARLE ’92.


Peter Landin. *A correspondence between ALGOL 60 and Church’s lambda notation*. CACM. August 1965.


Paul Philips. *We’re Doing It all Wrong* Pacific Northwest Scala 2013.

John Reppy, Claudio Russo, and Yingqi Xiao *Parallel Concurrent ML* ICFP ’09.
Guy Steele, Jr. *Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO*. MIT AI Lab Memo 443. October 1977.
David Taenzer, Murthy Ganti, and Sunil Podar, *Problems in Object-Oriented Software Reuse* ECOOP’89.
Appendix 1. Extreme ActorScript

Parameterized Types, i.e., $\langle >$ , $\rangle$

Parameterized Types are specialized using other types delimited by “$\langle$” and “$\rangle$”:

\[
\text{Actor Double} \langle \text{aType} \rightarrow \text{Arithmetic} \rangle
\]
\[
[x: \text{aType}]: \text{aType} \rightarrow \text{aType}[x+x]$
\]

// addition for aType that is Arithmetic

The formal syntax of parameterized types is in the following end note: 36.

Type Discrimination, i.e., Discrimination and ↓

A discrimination definition is a type of alternatives differentiated by type using “Discrimination” followed by a type name, “between”, types separated using “,”, terminated by “↓”.

A discriminate can constructed using the discrimination followed by “[”, an expression for the discriminant and “]”,

A discriminate can be projected as follows:

- In an expression, by using an expression for a discrimination followed by “↓” and the type to be projected. Also, a discrimination can be tested if it holds a discrimination of a certain type with an expression for the discrimination followed by “↓?” and the type to be tested.
- In a pattern, by using a pattern followed by “↓” and the type to be projected.

For example, consider the following definition:

\[
\text{Discrimination IntegerOrString between Integer, String}$
\]

Consequently,

- \((\text{IntegerOrString}[3])↓\text{Integer}\) is equivalent to 3↓.
- \((\text{IntegerOrString}[^a])↓\text{Integer}\) throws an exception because String is not the same as the discriminant Integer.
- \((\text{IntegerOrString}[3])↓?\text{Integer}\) is equivalent to True↓.
- The pattern \(x↓\text{String}\) matches \(\text{IntegerOrString}["a"]\) and binds \(x\) to "a".
- The expression below is equivalent to 2↓:
  \[
  \text{IntegerOrString}[3] \diamond y↓\text{Integer} \& y-1\overline{x}
  \]
  \[
  x↓\text{String} \& x \overline{y↓}$
The formal syntax of type discrimination is in following end note: 37.

**Structures**

A structure is an Actor used in pattern matching that can be defined using an identifier by “|”, parts separated by “,” and “|”.

Discrimination can be used with structures. For example, a `Trie[aType]` is a discrimination of `Terminal[aType]` and `TrieFork[aType]`:

**Discrimination Trie[aType] between**

- `Terminal[aType]`,
- `TrieFork[aType]`

where the structure `Terminal` can be defined as follows:

**Structure Terminal[aType][aType]**

For example,

- The expression `let x = 3. Terminal[Integer][x]` is equivalent to `Terminal[Integer][3]`
- The pattern `Terminal[Integer][x]` matches `Terminal[Integer][3]` and binds `x` to 3.

The structure `TrieFork` can be defined as follows:

```plaintext
Structure TrieFork[aType][left:Trie[aType], right:Trie[aType]]
  flip[]:TrieFork[aType] -> // flip the branches
  TrieFork[aType][right,left]
```

\[x\] is of type `Integer`
For example,

- The expression
  
  \[ \text{Let } x \leftarrow 3, \]
  
  \[ \text{TrieFork}[\text{Terminal}[x], \text{Terminal}[x+1]] \]
  
  is equivalent to the following:
  
  \[ \text{TrieFork}[\text{Trie}[\text{Terminal}[5], \text{Trie}[\text{Terminal}[6]]] \]

- The pattern \( \text{TrieFork}<\text{Integer}>[x, y] \) matches
  
  \[ \text{TrieFork}[\text{Trie}[\text{Terminal}[5], \text{Trie}[\text{Terminal}[6]]] \]
  
  and binds \( x \) to \( \text{Terminal}[5] \) and \( y \) to \( \text{Terminal}[6] \).

Below is the definition of a procedure that computes a list that is the “fringe” of the terminals of a Trie.\(^1\)

```
Actor TrieFringe<\text{aType}>
[aTrie: Trie<\text{aType}>]:[\text{aType}*] \rightarrow
aTrie \( \downarrow \)
  \text{Terminal}<\text{aType}>[x] \( \uparrow \)[x] \( \rightarrow \)
  \text{TrieFork}<\text{aType}>[\text{left}, \text{right}] \( \rightarrow \)
  [\mathbf{v}\text{TrieFringe}, \text{left}], \mathbf{v}\text{TrieFringe}<\text{aType}>,[\text{right}] \( \rightarrow \) \( \rightarrow \)
```

The above procedure can be used to define \( \text{TrieSameFringe?} \) that determines if two lists have the same fringe [Hewitt 1972]:

```
Actor TrieSameFringe?<\text{aType}>
[left: Trie<\text{aType}>, right: Trie<\text{aType}>]:\text{Boolean} \rightarrow
// test if two Tries have the same fringe
  \text{TrieFringe}<\text{aType}>,[\text{left}] = \text{TrieFringe}<\text{aType}>,[\text{right}] \( \rightarrow \)
```

The formal syntax of structures is in the following end note: 41

---

\(^1\) See definition of \text{Trie} above in this article.
Nullable
Distinguishing a special case to indicate the absence of an Actor of a type is a long-time issue [Hoare 2009].

In an expression,
- “Nullable” followed by an Expression is a non-null nullable.
- “Null” followed by a type is the nullable that is the null of that type.
- “⦾” followed by an expression for a nullable is the Actor in the nullable or throws an exception if an only if the nullable is null.

For example,
- Nullable 3 is of type Nullable Integer
- 3⦾ is equivalent to⦾ Nullable 3⦾
- ⦾ Null Integer⦾ throws an exception

In a pattern,
- “⦾” followed by a pattern matches a nullable if and only if it is non-null and the pattern matches the Actor in the nullable.
- “Null” followed by a type only matches the null of the type.

For example,
- The pattern⦾ x matches Nullable 3, binding x to 3

The formal syntax of nullables is in following end note: 42.
**Processing Exceptions, i.e., Try catch and Try cleanup**

It is useful to be able to catch exceptions. The following illustration returns the string “This is a test.”:

```plain
Try Throw Exception["This is a test."] catch Exception[aString] $ aString
```

The following illustration performs Reset and then rethrows Exception["This is another test."]:

```plain
Try Throw Exception["This is another test."] cleanup Reset
```

The formal syntax of processing exceptions is in the following end note: 43.

**Runtime Requirements, i.e., Preconditions and postcondition**

A runtime requirement throws exception an exception if does not hold. For example, the following expression throws an exception that the requirement \( x \geq 0 \) doesn't hold:

```plaintext
Let \( x \leftarrow \neg 1. \)
    Preconditions \( x \geq 0. \)  // commentary for error checking
    SquareRoot[x]
```

Post conditions can be tested using a procedure. For example, the following expression throws an exception that postcondition failed because square root of 2 is not less than 1:

```
SquareRoot[2] postcondition [y:Float]:Boolean $ y < 1
```

The formal syntax requirements is in the following end note: 44.
Multiple implementations of a type

The interface type `Complex` is defined as follows:

```
Interface Complex with [Real] |→ Float,
  [Imaginary] |→ Float,
  [Magnitude] |→ Float,
  [Angle] |→ Degrees
```

Cartesian Actors that implement `Complex` can be defined as follows:

```
Structure Cartesian [myReal: Float default 0, myImaginary: Float default 0]
  implements Complex using
  [real]: Float → myReal¶
  [imaginary]: Float → myImaginary¶
  [magnitude]: Float →
    SquareRoot.[myReal*myReal + myImaginary*myImaginary]¶
  [angle]: Degrees →
    Let theta ← Arcsine.[myImaginary/ .*[magnitude]].
    myReal>0 ◆
      True $\theta$
    False $\theta > 0 ◆$
      True $180^\circ - \theta$ 45
      False $180^\circ + \theta$ 45

Consequently,
- `Cartesian[1, 2] [real]` is equivalent to 1
- `Cartesian[3, 4] [magnitude]` is equivalent to 5.0

For example:

**Actor** Times
```
[u: Complex, v: Complex]: Complex →
  Cartesian[u [real] *v [real] – u [imaginary] *v [imaginary],
    u [imaginary] *v [real] + u [real] *v [imaginary]]
```

**Actor** Equivalent
```
[u: Complex, v: Complex]: Boolean →
  myReal = u [real] = v [real] $\land$ u [imaginary] = v [imaginary]
```
Arguments with named fields, i.e., \( \) and \( : \)

Polar Actors that implement \texttt{Complex} with named arguments \texttt{angle} and \texttt{magnitude} can be defined as follows:

\[
\text{Structure} \quad \text{Polar}[\text{angle} : \text{Degrees } \text{default} \ 0^\circ, \\
\quad \quad \quad \quad \text{// angle of type Degrees is a named argument of Polar with} \\
\quad \quad \quad \quad \text{// default } 0^\circ \\
\quad \quad \quad \quad \text{magnitude} : \text{Length } \text{default} \ 1]
\]

\text{implements Complex using}

\[
\begin{align*}
\text{[real]} : \text{Float } \rightarrow & \text{ magnitude} \ast \text{Sine.}[\text{angle}]^	op \\
\text{[imaginary]} : \text{Float } \rightarrow & \text{ magnitude} \ast \text{Cosine.}[\text{angle}]_\perp
\end{align*}
\]

Consequently,

- \texttt{Polar[]} \texttt{[real]} \texttt{]} \texttt{is equivalent to } \texttt{1} \texttt{]} 

For example:

\texttt{Actor Times}

\[
\begin{align*}
\text{[Polar[angle anAngle, magnitude aMagnitude]}, \\
\quad \text{Polar[angle anotherAngle, magnitude anotherMagnitude]}] \\
\quad : \text{Complex } \rightarrow \\
\quad \text{Polar[angle anAngle+anotherAngle,} \\
\quad \quad \text{magnitude aMagnitude*anotherMagnitude]} \quad 47
\end{align*}
\]

The formal syntax of named arguments is in the following end note: \( 48. \)
Lists, *i.e.*, `[ ]` using Spread, *i.e.*, `[ ⩛ ]

The prefix operator "⩛" can be used to spread the elements of a list. For example

- `[1, ⩛[2, 3], 4]` is equivalent to `[1, 2, 3, 4].
- `[[1, 2], ⩛[3, 4]]` is equivalent to `[[1, 2], 3, 4]`
- If `y` is `[5, 6]`, then `[1, 2, y, ⩛y]` is equivalent to `[1, 2, [5, 6], 5, 6]`

The formal syntax of list expressions is in the following end note: 49.

Within a list, "⩛" is used to match the pattern that follows with the list zero or more elements. For example:

- `[[x, 2], ⩛y]` is a pattern that matches `[[1, 2], 3, 4]` and binds `x` to `1` and `y` to `[3, 4]`
- if `y` is `[3, 4]` then `[[1, 2], ⩛y]` matches `[[1, 2], 3, 4]`
- `⩛x, ⩛y]` is an illegal pattern because it can match ambiguously

The formal syntax of patterns is in the following end note: 50.
As an example of the use of spread, the following procedure returns every other element of a list beginning with the first:

```
Actor AlternateElements <aType>
[aList:[aType*]]:[aType*] →
  aList ◁
    [] $ [] □
    [anElement] $ [anElement] □
    [firstElement, secondElement] $ [firstElement] □
  else $
    [firstElement, secondElement, ∀remainingElements] $
    [firstElement, ∀AlternateElements, [remainingElements]] □
```

Consequently,

- AlternateElements<Integer>·[[ ]]1 is equivalent to []:[Integer*]1
- AlternateElements<Integer>·[[3]]1 is equivalent to [3]:[Integer*]1
- AlternateElements<Integer>·[[3, 4]]1 is equivalent to [3]:[Integer*]1
- AlternateElements<Integer>·[[3, 4, 5]]1 is equivalent to [3, 6]:[Integer*]1

Sets, i.e., { } using spreading, i.e., { ∀ } A set is unordered with duplicates removed.

The formal syntax of sets is in the following end note: 52.

Multisets, i.e., {} using spreading, i.e., { ∀ } A set is unordered with duplicates allowed.

The formal syntax of multisets is in the following end note: 53.
Maps, *i.e.*, \( \text{Map\{ } \}\)

A map is composed of pairs. For example \( \text{Map\{3, "a"}, ["x", "b"]\} \)

Pairs in maps are unordered, *e.g.*, \( \text{Map\{[3, "a"}, ["x", "b"]\} \) is equivalent to \( \text{Map\{["x", "b"}, [3, "a"]\} \).

However, the expression \( \text{Map\{["y", "b"}, ["y", "a"]\} \) throws an exception because a map is univalent.

As another example, for the contact records of 1.1 billion people, the following can compute a list of pairs from age to average number of social contacts of US citizens sorted by increasing age making use of the following:

\[
\text{Structure ContactRecord\{yearsOld} \equiv \text{: Age,}
\quad \text{numberOfContacts} \equiv \text{: Integer,}
\quad \text{citizenship} \equiv \text{: String} \}
\]

\[
[\text{ContactRecord*}] \] has
\quad \text{filter}[[\text{ContactRecord}] |\rightarrow \text{Boolean}]
\quad |\rightarrow \{\text{ContactRecord*}\},
\quad \text{collect}[[\text{ContactRecord}] |\rightarrow \{\text{Age, Integer}\}]
\quad |\rightarrow \text{Map<Age, } \{\text{Integer*}\} >\]\n
\[
\text{Map<Age, } \{\text{Integer*}\} > \] has
\quad \text{reduceRange}[[\{\text{Integer*}\}] |\rightarrow \text{Float}]
\quad |\rightarrow \text{Map<Age, Float>}\]

\[
\{\text{Number*}\} \] has: \text{average[]} |\rightarrow \text{Float}\]

\[
\text{Map<Age, Float> has}
\quad \text{sort}[[\text{Age, Age}] |\rightarrow \text{Boolean}]
\quad |\rightarrow \{\text{Age, Float}\}\]

42
The program is as follows:

```
Actor AgeWithAverageOfNumberOfContactsSortedByAge
  [records:{(ContactRecord*):Sorted <Age>}] →
  records.filter([aRecord:ContactRecord] → (aRecord.[citizenship] "US" ? True [else ? False])
  .collect([aRecord:ContactRecord] → [aRecord.[yearsOld], aRecord.[numberOfContacts]])
  .reduceRange([aSetOfNumberOfContacts:{Integer*}] → aSetOfNumberOfContacts.average[])
  .sort(LessThanOrEqual <Age>)]
```

The formal syntax of maps is in the following end note: 55.

**Futures, i.e., Future and ☉**

A future [Baker and Hewitt 1977] for an expression can be created in ActorScript by using “Future” preceding the expression. The operator “☉” can be used to “reduce” a future by returning an Actor computed by the future or throwing an exception. For example, the following expression is equivalent to Factorial\([9999]\)↓

```
Let aFuture ← Future Factorial\([9999]\].
☉aFuture // do not proceed until Factorial\([9999]\) has been reduced
```

Futures allow execution of expressions to be adaptively executed indefinitely into the future.56 For example, the following returns a future

```
Let aFuture ← Future Factorial\([9999]\],
g ← ([afuture: Future<Integer>]: Integer → 5).
// g returns 5 regardless of its argument

g.[aFuture]↓
// return 5 regardless of whether Factorial\([9999]\) has completed
```

---

1 f is of type Future<Integer>
2 i.e. returned or threw an exception
3 i.e. Factorial\([1000]\) might not have returned or thrown an exception when 5 is returned. The future f will be garbage collected.
Note that the following are all equivalent:

- \( \text{Future}(4+\text{Factorial}[9999]) \)
- \( 4+\text{Future}(\text{Factorial}[9999]) \)
- \( 4+\text{Factorial}[9999] \)
- \( (4+\text{Factorial}[9999]) \)

Also \( \text{Factorial}[9999]+\text{Fibonacci}[9000] \) is equivalent to the following:

\[
\text{Let } n \leftarrow \text{Factorial}[9999],
\text{m} \leftarrow \text{Fibonacci}[9000],
n+m \text{ } \text{return Factorial}[9999]+\text{Fibonacci}[9000]
\]

In the following example, Factorial[9999] might never be executed if readCharacter[] returns the character 'x':

\[
\text{Let aFuture } \leftarrow \text{Future(9999)}.
\text{readCharacter}[] \leftarrow \text{'}x'\text{ else } 1+\text{aFuture} \text{ } \text{\text{returned something other than 'x'}}
\]

In the above, program resolution of aFuture is highlighted in yellow.

The procedure Size below can compute the size of a FutureList concurrently with its being created:

\[
\text{Actor Size}
[aFutureList:FutureList<String>]:Integer \rightarrow
\text{aFutureList } \leftarrow
[ ] \leftarrow 0\{\{first, \text{rest} \} \leftarrow first, [\text{length} + \text{Size}[]] \leftarrow \text{rest} \}
\]

// reducing a FutureList reduces only the head

Below is the definition of a procedure that postpones computation of a FutureList that is the “fringe” of a Trie:

\[
\text{Actor TrieFringe } \langle aType \rangle
[aTrie:Trie<aType>]:FutureList<aType> \rightarrow
\text{aTrie } \leftarrow
\text{Terminal } \langle aType \rangle \leftarrow [x] \leftarrow \text{left, right } \leftarrow
\text{[TrieFringe, left, Postpone] } \rightarrow \text{TrieFringe } \langle aType \rangle \rightarrow \text{[right]}\]
\]

The above procedure can be used to define SameFringe? that determines if two lists have the same fringe [Hewitt 1972]:

---

1 See definition of Tree above in this article.
Actor TrieSameFringe? <aType>
[aTrie: Trie <aType>, anotherTrie: Trie <aType>]: Boolean →
   // test if two Tries have the same fringe
TrieFringe <aType>, anotherTrie ↦ TrieFringe <aType>, anotherTrie
   // = reduces futures in the fringes

The procedure below given a list of futures returns a FutureList with the same elements reduced:

Actor FutureListOffReducedElements <aType>
[aListOfFutures: [Future <aType>]*]: FutureList <aType> →
aListOfFutures ⦅
   [] ⦅ [] ⦅
[aFirst, ∀aRest] ⦅
   [aFirst, ∀Future] FutureListOfReducedElements <aType>, [aRest] ⦅

The formal syntax of futures is in the following end note: 62.

Language extension, i.e., ( )
The following is an illustration of language extension that illustrates postponed execution: 63
Actor ("Postpone" anExpression: Expression <aType>)
   :Postpone <aType>
   implements Expression <Future <aType>> using
      eval[e: Environment]: Future <aType> →
      Future Actor implements aType using
      aMessage →   // aMessage received
         Let postponed ← anExpression, eval[e].
         postponed, aMessage
            // return result of sending aMessage to postponed
               become postponed
               // become the Actor postponed for
               // the next message received

The formal syntax of language extension is in the following end note: 64.

____________________________________
* this is allowed because postponed is of type aType

45
**In-line Recursion (e.g., looping), i.e.** $\ast \leftarrow , \leftarrow \ast$  

In-line recursion (often called looping) is accomplished using an initial invocation with identifiers initialized using “$\leftarrow$” followed by “$\ast$” and the body.\(^i\)

Below is an illustration of a loop Factorial with two loop identifiers $n$ and accumulation. The loop starts with $n$ equals 9 and value equal 1. The loop is iterated by a call to Factorial with the loop identifiers as arguments.

$\text{Factorial}_{[n \leftarrow 9, \text{accumulation} \leftarrow 1]} \triangleq$  

$$n = 1 \diamond \text{True} \hspace{1em} \triangledown \text{accumulation} \hspace{1em} \Box \text{Factorial}_{[n-1, n \times \text{accumulation}]} \Box \Box \Box$$

The above compiles as a loop because the call to Factorial in the body is a “tail call” [Hewitt 1970, 1976; Steele 1977].

The following expression returns a list of ten times successively calling the parameterless procedure $P^{ii}$ (of type $[] \rightarrow \text{Integer}$):

$\text{FirstTenSequentially}_{[n \leftarrow 10]} \triangleq$  

$$n = 1 \diamond \text{True} \hspace{1em} [P_{[]}][] \hspace{1em} \Box \text{Let} \hspace{1em} x \leftarrow P_{[]}. \hspace{1em} [x, \Box \text{FirstTenSequentially}_{[n-1]}] \Box \Box \Box \Box$$

The following returns one of the results of concurrently calling the procedure $P^{iv}$ (which has no arguments and returns $\text{Integer}$) ten times with no arguments:

$\text{OneOfTen}_{[n \leftarrow 10]} \triangleq$  

$$n = 1 \diamond \text{True} \hspace{1em} [P_{[]}][[]] \hspace{1em} \Box \text{Fail} \hspace{1em} \Box P_{[]}[] \hspace{1em} \Box \text{Either} \hspace{1em} \Box \text{OneOfTen}_{[n-1]}] \Box \Box \Box \Box$$

The formal syntax of looping is in the following end note: 67.

---

\(^i\) This construct takes the place of $\text{while}$, $\text{for}$, $\text{etc.}$ loops used in other programming languages.

\(^{ii}\) equivalent to the following:

$\text{Factorial}_{[n: \text{Integer} \leftarrow 9, \text{accumulation}: \text{Integer} \leftarrow 1]: \text{Integer} \triangleq}$  

$$n = 1 \diamond \text{True} \hspace{1em} \triangledown \text{accumulation} \hspace{1em} \Box \text{Factorial}_{[n-1, n \times \text{accumulation}]} \Box \Box \Box$$

\(^{iii}\) The procedure $P$ may be indeterminate, i.e., return different results on successive calls.

\(^{iv}\) The procedure $P$ may be indeterminate, i.e., return different results on different calls.
Strings
Strings are Actors that can be expressed using """, string arguments, and """". For example,

- """"1", "23", "4""""] is equivalent to "1234".
- """"1", "2", "34", "56""""] is equivalent to "123456".
- " """"1", "2", "34", """""" is equivalent to "1234".
- " """" is equivalent to """".

String patterns are delimited by """"" and """"". Within a string pattern, ""y"" is used to match the pattern that follows with the list zero or more characters. For example:

- "x, ""2", "y""" is a pattern that matches "1234" and binds x to "1" and y to "34".
- """"1", "2", "y""" is a pattern that only matches "1234" if y is "34".
- ""x, "y"" is an illegal pattern because it can match ambiguously.
As an example of the use of spread, the following procedure reverses a string:

\[
\begin{align*}
\textbf{Actor} & \quad \text{Reverse} \\
[aString: \text{String}]: \text{String} & \\
\text{aString} & \quad \textbf{⪞} \\
\text{"first", \text{rest} \quad \textbf{⪞} \quad \text{rest, first} } & \quad \textbf{¶}
\end{align*}
\]

The formal syntax of string expressions is in the following end note: 69.

**General Messaging, \textit{i.e.,} ⪞ and ⨇**

The syntax for general messaging is to use an expression for the recipient followed by "\textbf{⪞}" and an expression for the message.

For example, if anExpression is of type `Expression < Integer >` then, anExpression.eval[anEnvironment] is equivalent to the following:

\[
\begin{align*}
\text{Let} \quad \text{aMessage} & \quad \textbf{← eval[Expression < Integer >][anEnvironment].} \\
\text{anExpression, aMessage} & \quad \textbf{¶}
\end{align*}
\]

The formal syntax of general messaging is in the following end note: 70.
Atomic Operations, i.e. Atomic compare update updated notUpdated
For example, the following example implements a lockable that spins to lock:

**Actor SpinLock[]**
locked := False. // initially unlocked
implements Lockable¹ using
lock[][Void →
Attemptt[]≜ // perform the loop Attempt as follows
Atomic locked compare False update True ◁
// attempt to atomically update locked from False to True
updated ≜ Preconditions locked=True.
// commentary for error checking:
// locked must have contents True
Void ≜ // if updated return Void
notUpdated≜ Attemptt[] // if not updated, try again
unLock[][Void →
Preconditions locked=True. // commentary for error checking:
// locked must have contents True
Void afterward locked= False $ // reset locked to False

The formal syntax of atomic operations is in the following end note: 72.

¹ Interface Lockable with lock[]⇒ Void,
    unLock[]⇒ Void
Enumerations, \emph{i.e., Enumeration of} using Qualifiers, \emph{i.e.,}`

An enumeration definition provides symbolic names for alternatives using "\textit{Enumeration}" followed by the name of the enumeration, "\textit{of}" , a list of distinct identifiers terminated by "\textit{I}".

For example,

\textbf{Enumeration DayName of} Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday\\

From the above definition, an enumerated day is available using a qualifier, \emph{e.g.}, Monday\text{\textsubscript{DayName}}. Qualifiers provide for namespaces.

The formal syntax of qualifiers is in the following end note: 73.

The procedure below computes the name of following day of the week given the name of any day of the week:

\textbf{UsingNamespace DayName}
\textbf{Actor} FollowingDay
\textbf{[aDay:\text{DayName}:DayName Actor}
\textbf{aDay} \text{\textsubscript{Monday}} \Rightarrow \text{Tuesday},
Tuesday \Rightarrow \text{Wednesday},
Wednesday \Rightarrow \text{Thursday},
Thursday \Rightarrow \text{Friday},
Friday \Rightarrow \text{Saturday},
Saturday \Rightarrow \text{Sunday},
Sunday \Rightarrow \text{Monday} \text{\textsubscript{I}}

The formal syntax of enumerations is in the following end note: 74.

Native types, \emph{e.g.,} JavaScript, JSON, Java, and XML
\textbf{Object} can be used to create JavaScript Objects. Also, \textbf{Function} can be used to bind the reserved identifier \textit{This}. For example, consider the following ActorScript for creating a JavaScript object \textit{aRectangle} (with length 3 and width 4) and then computing its area 12:

\textbf{Let} \textit{aRectangle} \leftarrow \textbf{Object} ["length": 3, "width": 4],
\textbf{aFunction} \leftarrow \textbf{Function} [\rightarrow \textit{This}["length"] \ast \textit{This}["width"]].
\textbf{Prep} \textit{Rectangle}["area"] \leftarrow \textit{aFunction} \text{\textsubscript{aRectangle}}[\text{["area"]}]\text{\textsubscript{I}}

\textit{aRectangle} is of type \textit{Object}\text{\textsubscript{JavaScript}}
The setTimeout JavaScript object can be invoked with a callback as follows that logs the string "later" after a time out of 1000:

```javascript
setTimeout(function(){
  console.log("later")
}, 1000)
```

**JSON** is a restricted version of **Object** that allows only Booleans, numbers, strings in objects and arrays.

Native types can also be used from Java. For example, if `s: String` Java, then `s.substring[3, 5]` is the substring of `s` from the 3rd to the 5th characters inclusive.

Java types can be imported using **Import**, *e.g.*:

```java
Namespace mynamespace
Import java.math.BigInteger
Import java.lang.Number
```

After the above, `BigInteger.new("123") instanceof Number` is equivalent to `true`.

The following notation is used for XML:

```xml
<"PersonName"> <"First">"Ole-Johan" <"First">
<"Last"> "Dahl"<"Last"> <"PersonName">
```

and could print as:

```xml
<PersonName> <First> Ole-Johan </First>
<Last> Dahl </Last> </PersonName>
```

**XML Attributes** are allowed so that the expression

```xml
<"Country" capital="Paris"> "France" </"Country">
```

and could print as:

```xml
<Country capital="Paris"> France </Country>
```

XML construction can be performed in the following ways using the append operator:

- **XML** `"<doc">1, 2, 3</doc>"` is equivalent to **XML** `"<doc">1, 2, 3</doc>"
- **XML** `"<doc">1, 2, 3, 4</doc>"` is equivalent to **XML** `"<doc">1, 2, 3, 4</doc>"

---

1. *i.e.* the following JavaScript types are not included in JSON: Date, Error, Regular Expression, and Function.
2. *substring* is a method of the **String** class in Java.
One-way messaging, e.g., ⊞, ←, and ↞

One-way messaging is often used in hardware implementations.

Each one-way named-message send consists of an expression followed by “↞”, a message name, and arguments delimited by “[” and “]”.

The following is a interface for a customer that is used in request/response message passing for return type \texttt{aType}:\footnote{76}

\begin{verbatim}
Interface Customer<\texttt{aType}> with
       return [\texttt{aType}] ↞ ⊞,
       throw[Exception] ↞ ⊞
\end{verbatim}

For example, if \texttt{aCustomer} is of type \texttt{Customer<Integer>}, then 3 can be returned to \texttt{aCustomer} using \texttt{aCustomer↞return[3]}.

The formal syntactic definition of one-way named-message sending is in the end note: \footnote{77}

Each one-way message handler implementation consists of a named-message declaration pattern followed by “↠” and a body for the response which must ultimately be “⊖” which denotes no response.

The formal syntactic definition of one-way named-message implementation is in the following end note: \footnote{78}
The following is an implementation of an arithmetic logic unit that implements `jumpGreater` and `addJumpPositive` one-way messages:

```plaintext
Actor ArithmeticLogicUnit<aType>:: []
  implements ALU<aType> using
  jumpGreater[x:aType, y:aType, firstGreaterAddress: Address, elseAddress: Address] ↞
    InstructionUnit ← Execute[(x>y) \ True \firstGreaterAddress \ False \elseAddress]
  addJumpPositive[x:aType, y:aType, sumLocation: Location<aType>, positiveAddress: Address, elseAddress: Address] ↞
    Let z ← (x+y).
    sumLocation ⧫
      aVariableLocation: VariableLocation<aType> †
        Prep VariableLocation.store[z] ⬶
        // continue after acknowledgement of store
        (z > 0) \ True \ InstructionUnit ← execute[positiveAddress] □
        False \ InstructionUnit ← execute[elseAddress] □□
        aTemporaryLocation: TemporaryLocation<aType> ‡
        aTemporaryLocation ← write[z],
        // continue concurrently with processing write
        (z > 0) \ True \ InstructionUnit ← execute[positiveAddress] □
        False \ InstructionUnit ← execute[elseAddress] □□§ △

† VariableLocation<aType> has store[aType] ↦ Void △
‡ TemporaryLocation<aType> has write[aType] ↦ ⊝ △
```

53
Using multiple other implementations, i.e., ⍠

This section presents an example of using multiple other implementations such as the ones below:

Actor Male[aLength:Meter]
     [length]:Meter → aLength$↓

Actor Human[aMagnitude:Year]
     [magnitude]:Year → aMagnitude$↓

Boy below makes use of both the Male and Human implementations:

Actor Boy[aMagnitude:Meter, aLength:Year]
     uses Male[aMagnitude], Human[aLength].
     // uses implementations Male and Human
     [magnitude]:Meter → (言えば)Male).[length]$↓
     // using this Actor with Male interface
     [length]:Year → (言えば)Human).[magnitude]$↓
     // using this Actor with Human interface

For example,
- Boy[Meter[3], Year[4]].[magnitude]$↓ is equivalent to Meter[3]$↓
- Boy[Meter[3], Year[4]].[length]$↓ is equivalent to Year[4]$↓
Inconsistency Robust Logic Programs

Logic Programs\(^8\) can logically infer computational steps.

**Forward Chaining**
Forward chaining is performed using \(\vdash\)

\[
\{ \text{""} \vdash \text{Theory PropositionExpression} \text{""} \}
\]
Assert PropositionExpression for Theory.

\[
\{ \text{""} \text{When} \vdash \text{Theory aProposition:Pattern \rightarrow Expression} \text{""} \}
\]
When aProposition holds for Theory, evaluate Expression;

Illustration of forward chaining:
\(\vdash\) Human[Socrates]\(\text{I}\)
\(\text{When } \vdash\) Human\([x]\) \(\rightarrow\) \(\vdash\) Mortal\([x]\]\(\text{I}\)
will result in asserting Mortal[Socrates] for theory \(t\)

**Backward Chaining**
Backward chaining is performed using \(\models\)

\[
\{ \text{""} \models \text{Theory aGoal:Pattern} \rightarrow Expression \text{""} \}
\]
Set aGoal for Theory and when established evaluate Expression.

\[
\{ \text{""} \models \text{Theory aGoal:Pattern } \text{:Expression} \text{""} \}
\]
Set aGoal for Theory and return a list of assertions that satisfy the goal.

\[
\{ \text{""} \text{When } \models \text{Theory aGoal:Pattern } \text{\rightarrow Expression} \text{""} \}
\]
When there is a goal that matches aGoal for Theory, evaluate Expression.
Illustration of backward chaining:

\[ \vdash \text{Human}[\text{Socrates}] \]

*When* \[ \vdash \text{Mortal}[x] \rightarrow (\vdash \text{Human}[^\text{}\emptyset, x] \rightarrow \vdash \text{Mortal}[^\text{\{x\}}]) \]

\[ \vdash \text{Mortal}[\text{Socrates}] \]

will result in asserting \text{Mortal}[\text{Socrates}] for theory t.

**SubArguments**

This section explains how subarguments\(^1\) can be implemented in natural deduction.

*When* \[ \vdash_s (\psi \vdash_t \phi) \rightarrow \]

*Let* \[ t' \leftarrow \text{Extension}[^t]. \]

\[ \vdash_t \psi, \vdash_{t'} \phi \rightarrow \vdash_t (\psi \vdash_t \phi) \]

Note that the following hold for \( t' \) because it is an extension of \( t \):

- *when* \[ \vdash_t \theta \rightarrow \vdash_{t'} \theta \]
- *when* \[ \vdash_{t'} \theta \rightarrow \vdash_t \theta \]

\(^1\) See appendix on Inconsistency Robust Natural Deduction.
Aggregation using Ground-Complete Predicates

Logic Programs in ActorScript are a further development of Planner. For example, suppose there is a ground-complete predicate \( \text{Link}[aNode, anotherNode, aCost] \) that is true exactly when there is a path from \( aNode \) to \( anotherNode \) with \( aCost \).

When \( \models \text{Path}[aNode, aNode, aCost] \rightarrow \)
// when a goal is set for a cost between \( aNode \) and itself
\( \models aCost=0 \) // assert that the cost from a node to itself is 0

The following goal-driven Logic Program works forward from \textit{start} to find the cost to \textit{finish}:

When \( \models \text{Path}[\textit{start}, \textit{finish}, aCost] \rightarrow \)
\( \models aCost=\text{Minimum} \{ \text{nextCost} + \text{remainingCost} \)
\( | \models \text{Link}[\textit{start}, \textit{next} \neq \textit{start}, \text{nextCost}], \)
\( \text{Path}[\textit{next}, \textit{finish}, \text{remainingCost}] \} \)
// a cost from \textit{start} to \textit{finish} is the minimum of the set of the sum of the
// cost for the next node after \textit{start} and
// the cost from that node to \textit{finish}

The following goal-driven Logic Program works backward from \textit{finish} to find the cost from \textit{start}:

When \( \models \text{Path}[\textit{start}, \textit{finish}, aCost] \rightarrow \)
\( \models aCost=\text{Minimum} \{ \text{remainingCost} + \text{previousCost} \)
\( | \models \text{Link}[\textit{previous} \neq \textit{finish}, \textit{finish}, \text{previousCost}], \)
\( \text{Path}[\textit{start}, \textit{previous}, \text{remainingCost}] \} \)
// the cost from \textit{start} to \textit{finish} is the minimum of the set of the sum of the
// cost for the previous node before \textit{finish} and
// the cost from \textit{start} to that Node

Note that all of the above Logic Programs work together concurrently providing information to each other.
Appendix 2: Meta-circular definition of ActorScript

It might seem that a meta-circular definition is a strange way to define a programming language. However, as shown in the references, concurrent programming languages are not reducible to logic. Consequently, an augmented meta-circular definition may be one of the best alternatives available.

The message eval

John McCarthy is justly famous for Lisp. One of the more remarkable aspects of Lisp was the definition of its interpreter (called Eval) in Lisp itself. The exact meaning of Eval defined in terms of itself has been somewhat mysterious since, on the face of it, the definition is circular.83

The basic idea is to send an expression an eval message with an environment to instead of the Lisp approach of sending the procedure Eval the expression and environment as arguments.

Construct is the fundamental type for ActorScript programming language constructs. Expression<atType> is an extension of Construct with an eval message that has an environment with the bindings of program identifiers and a message with an environment and cheese:

Interface Expression<atType> extends Construct with eval[Environment]↦atType,
perform[Environment, CheeseQ]↦aType

BasicExpression<atType> is an implementation that performs the functionality of leaving the cheese for expression being used as the continuation:

Actor BasicExpression<atType> []
perform[e:Environment, c:CheeseQ] →
Try Let anActor ← ⍠Expression<atType> ..eval[e] ▮.
Prep c . leave[].
anActor
cleanup c . leave[] §

The tokens ⟨ and ⟩ are used to delimit program syntax.

Actor (anIdentifier: Identifier<atType>); Expression<atType>
uses BasicExpression<atType> []
partially implements Expression<atType> using
  eval[e:Environment] → e .lookup[anIdentifier] ▮

83 Interface Construct
The interface **Type**

The interface **Type** is defined as follows:

\[
\text{Interface } \text{Type} <\text{recipientType} > (\text{Message} \mapsto \text{returnType}) \rightarrow \text{Boolean},
\]

\[
\text{has?[MethodSignature]} \mapsto \text{Boolean},
\]

\[
\text{send[recipientType, Message]} \mapsto \text{returnType},
\]

\[
\text{return[returnType]} \mapsto \text{Void},
\]

\[
\text{throw[Exception]} \mapsto \text{Void},
\]

\[
[\text{constructor}] \mapsto \text{Procedure},
\]

\[
[\text{sending}] \mapsto \text{SendingType}<\text{recipientType}>,
\]

\[
[\text{receiving}] \mapsto \text{ReceivingType}<\text{returnType}>
\]

\[
\text{SendingType} \text{ is a restriction of Type that can be used only for sending:}
\]

\[
\text{Interface } \text{SendingType} <\text{recipientType} > (\text{Message} \mapsto \text{returnType}) \rightarrow \text{Void},
\]

\[
\text{send[recipientType, Message]} \mapsto \text{returnType},
\]

\[
\text{has?[MethodSignature]} \mapsto \text{Boolean},
\]

\[
[\text{constructor}] \mapsto \text{Procedure},
\]

\[
[\text{receiving}] \mapsto \text{ReceivingType}<\text{returnType}>,
\]

\[
\text{return[returnType]} \mapsto \text{Void},
\]

\[
\text{throw[Exception]} \mapsto \text{Void}
\]

\[
\text{ReceivingType} \text{ is a restriction of Type that can be used only for receiving:}
\]

\[
\text{Interface } \text{ReceivingType} <\text{recipientType} > (\text{Message} \mapsto \text{returnType}) \rightarrow \text{Void},
\]

\[
\text{send[recipientType, Message]} \mapsto \text{returnType},
\]

\[
\text{has?[MethodSignature]} \mapsto \text{Boolean},
\]

\[
[\text{constructor}] \mapsto \text{Procedure},
\]

\[
[\text{sending}] \mapsto \text{SendingType}<\text{recipientType}>,
\]

\[
[\text{receiving}] \mapsto \text{ReceivingType}<\text{returnType}>
\]

\[
\text{Actor} \text{ (anotherType: } \text{Type} <\text{anotherType} > \rightarrow \text{Boolean})
\]

\[
\text{uses BasicExpression } <\text{aType} > [\text{ ]}
\]

\[
\text{partially implements Expression } <\text{aType} > [\text{ ]}
\]

\[
\text{eval[e: Environment]} : \text{Boolean} \rightarrow
\]

\[
(\text{anotherType } . \text{eval[e]}), \text{extension?[aType } . \text{eval[e]}]
\]

\[
\text{Actor} \text{ (aType: } \text{Type}
\]

\[
\text{“has?” aSignature: Signature} : \text{Expression } <\text{Boolean}>
\]

\[
\text{uses BasicExpression } <\text{aType} > [\text{ ]}
\]

\[
\text{partially implements Expression } <\text{aType} > [\text{ ]}
\]

\[
\text{eval[e: Environment]} : \text{Boolean} \rightarrow
\]

\[
(aType . \text{eval[e]}). \text{has?[aSignature .eval[e]]}
\]
Interface `CastableType[fromType, toType]` extends `Type` with

- `up[fromType]` \(\rightarrow\) `toType`,
- `down[fromType]` \(\rightarrow\) `toType`,
- `down?[fromType]` \(\rightarrow\) `Boolean`

Actor `SimpleCastableType[fromType, toType]`

uses `FundamentalType[]`

partially reimplements `CastableType[fromType, toType]` using

- `up[anActor[fromType]:toType]` \(\rightarrow\) `Throw IllegalUpcast[ ]`
- `down[anActor[fromType]:toType]` \(\rightarrow\) `Throw IllegalDowncast[ ]`
- `down?[anActor[fromType]:toType]` \(\rightarrow\) `Throw IllegalDowncastQuery[]`

Interface `RestrictionType[aType]` extends `Type`

Actor `{anExpression:Expression[fromType]}`

- "↑" `castExpression:Type[fromType]`: `Up[toType]`

uses `BasicExpression[toType]`

partially implements `Expression[toType]` using

- `eval[e:Environment]:toType` \(\rightarrow\)
  - `castExpression.eval[e]` \(\rightarrow\)
  - `aRestrictionType:RestrictionType`
  - `aRestrictionType.up:anExpression.eval[e]` \(\rightarrow\)
  - `else` \(\rightarrow\)
    - `(fromType)CastableType[fromType, toType]` \(\rightarrow\)
    - `up:anExpression.eval[e]` \(\rightarrow\)

Actor `{anExpression:Expression[fromType]}`

- "↓" `castExpression:Type[toType]`: `Down[toType]`

uses `BasicExpression[toType]`

partially implements `Expression[toType]` using

- `eval[e:Environment]:toType` \(\rightarrow\)
  - `(castExpression.eval[e])\downarrow CastableType[fromType, toType]` \(\rightarrow\)
  - `down:anExpression.eval[e]` \(\rightarrow\)
Type Discrimination

Actor ("Discrimination" $\text{aDiscriminationType}$ "between"
  types: $\text{Types}$ $\downarrow$): $\text{Definition}$

Actor implements $\text{Definition}$ using
  $\text{eval}[\text{e:$\text{Environment}$}]:\text{Environment}$ $\mapsto$
    $\text{SimpleDiscrimination}[[\text{typeExpressions},\text{eval}[\text{e}]]]$ $\uparrow$

Actor $\text{SimpleDiscrimination}[\text{types:}\{\text{Type}\}]$
  [aDiscriminant: $\text{aType}$] $\text{extends} \text{InstanceDiscriminationType}$
  uses $\text{SimpleCastableType}<\text{InstanceDiscriminationType}<\text{aType}>$, $\text{aType}$ $\downarrow$

  partially reimplements
    $\text{CastableType}<\text{InstanceDiscriminationType}<\text{aType}>$, $\text{aType}$ $\downarrow$
    $\text{down}[\text{anActor}:\text{InstanceDiscriminationType}<\text{aType}>]:\text{aType}$ $\mapsto$
    anActor $\text{CastableType}<\text{InstanceDiscriminationType}<\text{aType}>$, $\text{aType}$ $\uparrow$
    aDiscriminant $\uparrow$
    else $\text{Throw} \text{IllegalDowncast}[\uparrow]$

  $\text{down}[\text{anActor}:\text{InstanceDiscriminationType}<\text{aType}>]:\text{Boolean}$ $\mapsto$

  anActor $\text{CastableType}<\text{InstanceDiscriminationType}<\text{aType}>$, $\text{aType}$ $\uparrow$

  True $\uparrow$
  else $\text{False}$ $\uparrow$
Type extends

Actor {"Actor" anExtensionType
  "extends" typeExpression:Type aType} "I":Definition
Actor implements Definition using
eval[e:Environment]:Environment →
e bind[anExtensionType,
  SimpleExtensionType(anExtensionType,
    typeExpression,eval[e])]

Actor SimpleExtensionType aType, extendedFrom
  extends ExtensionType
  uses SimpleCastableType aType, extendedFrom]
  partially reimplements CastableType aType,
  extendedFrom using
    up[anInstance:aType]:extendedFrom →
    SimpleUppedType aType, extendedFrom[anInstance]

Actor SimpleUppedType aType, extendedFrom
  [anInstance:aType]
  uses SimpleCastableType aType, extendedFrom
  partially reimplements CastableType aType,
  extendedFrom using
    down[anActor:CastableType aType,
        extendedFrom]:aType →
    anActor
      ⬤CastableType aType, extendedFrom $ anInstance $*
      else $ Throw IllegalDownCast [] $]
    down?[anActor:CastableType aType,
        extendedFrom]:Boolean →
    anActor
      ⬤CastableType aType, extendedFrom $ True $*
      else $ False $ $}
Nullable, e.g., \(\Diamond\)
The type Nullable is used for nullables:

```
Interface Nullable\(\langle aType\rangle\)
extends Type with reduce[\] \(\rightarrow\) aType
```

```
Actor ("Nullable" anExpression:Expression\(\langle aType\rangle\))
uses BasicExpression\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\) [ ]
partially implements Expression\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\) using
\(\text{eval[e:Environment]}:\text{Nullable\(\langle aType\rangle\)}\rightarrow\)
Let anActor \(\leftarrow\) anExpression.eval[e] \(\bullet\).
Actor implements Nullable\(\langle aType\rangle\) using
\(\text{reduce[\]} \rightarrow\) anActor\(\$\)
```

```
Actor {Null aType:Type\(\langle aType\rangle\)}:NullExpression\(\langle aType\rangle\)
uses BasicExpression\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\) [ ]
partially implements Expression\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\) using
\(\text{eval[e:Environment]}:\text{Nullable\(\langle aType\rangle\)} \rightarrow\)
Actor implements Nullable\(\langle aType\rangle\) using
\(\text{reduce[\]} \rightarrow\) Throw IsNullException[ ] \(\$\)
```

```
Actor {Null aType:Type\(\langle aType\rangle\)}:NullPattern\(\langle aType\rangle\)
implements Pattern\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\) using
\(\text{match[anActor:Nullable\(\langle aType\rangle\)}, e:Environment]}\)
\(\rightarrow\)
anActor \(\diamond\)
Null aType,eval[e] \(\$\) Nullable e \(\Box\)
else \(\$\) Null Environment\(\$\)
```

```
Actor ("\(\Diamond\)" anExpression:Expression\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\))
uses BasicExpression\(\langle aType\rangle\) [ ]
partially implements Expression\(\langle aType\rangle\) using
\(\text{eval[e:Environment]}:\text{aType} \rightarrow\)
\(\langle\text{anExpression.eval[e]}\rangle\)\(\langle\text{Nullable\(\langle aType\rangle\)}\rangle\),\(\text{reduce[\]}\)
```

63
Future, e.g., ⦾, and □
The type Future is used for futures:

\[
\text{Interface } \text{Future}\langle\text{aType}\rangle \\
\text{extends Type with reduce} [\ ] \rightarrow \text{aType}
\]

\[
\text{Actor} \left(\text{"Future" anExpression: Expression } \langle\text{aType}\rangle\right) \\
\text{: } \text{Future } \langle\text{aType}\rangle
\]

\[
\text{uses BasicExpression } \langle\text{Future } \langle\text{aType}\rangle\rangle \Downarrow [ ] \\
\text{partially implements Expression } \langle\text{Future } \langle\text{aType}\rangle\rangle \text{ using} \\
\text{eval}[e: \text{Environment}]: \text{Future } \langle\text{aType}\rangle \rightarrow \\
\text{Let aFuture } \leftarrow \\
\text{Future Try anExpression } \text{eval}[e] \\
\text{catch} \\
\text{anException } \varepsilon
\]

\[
\text{Actor} \left(\text{"Future" anExpression: Expression } \langle\text{aType}\rangle\right) \\
\text{implements Future } \langle\text{aType}\rangle \\
\text{reduce} [\ ] \rightarrow \text{Throw anException} \varepsilon. \\
\text{Actor implements Future } \langle\text{aType}\rangle \text{ using} \\
\text{reduce} [\ ] \rightarrow @\text{aFuture } \varepsilon
\]

\[
\text{Actor} \left(\text{"⦾" anExpression: Expression } \langle\text{Future } \langle\text{aType}\rangle\rangle\right) \\
\text{: Reduction } \langle\text{aType}\rangle
\]

\[
\text{uses BasicExpression } \langle\text{aType}\rangle [ ] \\
\text{partially implements Expression } \langle\text{aType}\rangle \text{ using} \\
\text{eval}[e: \text{Environment}]: \text{aType} \rightarrow \\
\left(\text{anExpression } \text{eval}[e] \Downarrow \text{Future } \langle\text{aType}\rangle\right) . \text{reduce} [ ] \varepsilon
\]

\[
\text{Actor} \left(\text{"□" anExpression: Expression } \langle\text{aType}\rangle\right) \\
\text{: Mandatory } \langle\text{aType}\rangle
\]

\[
\text{uses BasicExpression } \langle\text{aType}\rangle [ ] \\
\text{implements Expression } \langle\text{aType}\rangle \text{ using} \\
\text{eval}[e: \text{Environment}]: \text{aType} \rightarrow \\
\text{}@\text{Future anExpression } \text{eval}[e] \varepsilon
\]
The message **match**

Patterns are analogous to expressions, except that they take receive match messages:

```
Interface Pattern:<aType> with
     match [aType, Environment] → Nullable<Environment>
```

```
Actor {anIdentifier: Identifier:<aType>}: Pattern:<aType>
    implements Pattern:<aType> using
        match[anActor:<aType>, e:Environment]: Nullable<Environment> →
            e.bind[anIdentifier, to ≡ anActor]
```

```
Actor ("_"): UniversalPattern:<aType>
    implements Pattern:<aType> using
        match[anActor:<aType>, e:Environment]: Nullable<Environment> →
            Nullable e
```

```
Actor ("ρ" anExpression:Expression:<aType>)
    implements Pattern:<aType> using
        ValuePattern:<aType>
        match[anActor, e:Environment]: Nullable<Environment> →
            anActor bind[anExpression, eva[e] : Nullable e]
        else : Null Environment
```

65
Message sending, e.g.,

Actor (procedure:Expression <argumentsType→returnType>  
  "→" "[" arguments:Arguments <argumentsType> "]") 
  :ProcedureSend <returnType> 
uses BasicExpression <returnType> [] 
partially implements Expression <returnType> using 
  eval[e:Environment]:returnType →  
  (procedure.eval[e]).[\(\forall \text{expressions}.\text{eval}[e]\)]§

Actor (recipient:Expression <recipientType>  
  "→" name:MessageName  
  "[" arguments:Arguments <argumentsType> "]") 
  :NamedMessageSend <returnType> 
uses BasicExpression <returnType> [] 
partially implements Expression <returnType> using 
  eval[e:Environment]:returnType →  
  Let aRecipient ← recipient.eval[e],  
  aRecipient  
  .SimpleMessage[QualifiedName[name, recipientType],  
  [\(\forall \text{arguments}.\text{eval}[e]\)]§

Actor (recipient:Expression <recipientType>  
  "→" aMessage:Message <messageType>) 
  :UnnamedMessageSend <returnType> 
uses BasicExpression <returnType> [] 
partially implements Expression <returnType> using 
  eval[e:Environment]:returnType →  
  recipientType.send[recipient.eval[e], aMessage.eval[e]]§
List Expressions and Patterns

Actor ("" first:Expression<<aType>> ",
  second:Expression<<aType>> ]:Expression<<[aType]*>>)

uses BasicExpression<<[aType]*>> []

partially implements Expression<<[aType]*>> using

eval[e:Environment]:[aType]* →
  [first.eval[e], second.eval[e]]:[aType]* §

Actor ("" first:Expression<<aType>> ",
  "\" rest:Expression<<aType>> ]:Expression<<[aType]*>>)

uses BasicExpression<<[aType]*>> []

partially implements Expression<<[aType]*>> using

eval[e:Environment]:[aType]* →
  [first.eval[e], rest.eval[e]]:[aType]* §

Actor ("" first:Pattern<<aType>> ",
  "\" rest:Pattern<<aType>> ]:Pattern<<[aType]*>>)

implements Pattern<<[aType]*>> using

match[anActor:aType, e:Environment]:Nullable<<Environment>> →
anActor
  [first, rest:[aType]*] §
  first.match[first, e] ◆
    Null Environment § Null Environment □
    @aNewEnvironment §
      rest.match[restValue, aNewEnvironment] □
  else § Null Environment □ §
Exceptions

Actor ("Try" anExpression: Expression Type
   "catch" exceptions: ExpressionCases Exception, Type
   : TryExpression Type)
uses BasicExpression Type []
partially implements Expression Type using
   eval[e: Environment]: Type →
   Try anExpression, eval[e] catch
   anException: Exception
   CasesEval, [anException, exceptions, e] Type

Actor ("Try" anExpression: Expression Type
   "cleanup" aCleanup: Expression Type)
uses BasicExpression Type []
partially implements Expression Type using
   eval[e: Environment]: Type →
   Try anExpression, eval[e] catch
   _։ Prep aCleanup, eval[e] ●.
   Rethrow Type

Continuations using perform
A continuations is a generalization of expression for executing in cheese, which receives perform messages:
   Interface Continuation Type extends Construct with
   perform[Environment, CheeseQ] Type aType

Actor Execute Type
   [aConstruct: Construct,
    e: Environment,
    c: CheeseQ]: Type →
   aConstruct ◦ aContinuation Continuation Type
   aContinuation, perform[e, c] ●
   anExpression Expression Type
   anExpression, eval[e] Type
Atomic compare and update

Actor ("Atomic" location:Expression\<Location<anotherType\>>,
  "compare" comparison:Expression<anotherType>
  "update" update:Expression<anotherType> "✓"
  "updated" "✓"
  compareIdentical:ContinuationList<anotherType> "□"
  "notUpdated" "✓"
  compareNotIdentical:ContinuationList<anotherType>)

implements Continuation<anotherType> using
    perform[e::Environment, c::CheeseQ]:anotherType →
        (location.eval[e])
        .compareAndConditionallyUpdate[comparison.eval[e], update.eval[e]] "✓"
            True $ compareIdentical, perform[e, c] □
            False $ compareNotIdentical, perform[e, c] □

Actor SimpleLocation<anotherType> [initialContents]
    contents := initialContents.

implements Location<anotherType> using
    compareAndConditionallyUpdate[comparison, update]:Boolean →
        (contents = comparison) "✓"
            True $ True afterward contents := update □
            False $ False □
Cases

Actor (anExpression: Expression <anotherType> "\n"
cases: ExpressionCases <anotherType, aType> "(\n")
): CasesExpression <aType>

uses BasicExpression <aType> []
partially implements Expression <aType> using
eval[e: Environment]: aType →
CasesEval * [anExpression, eval[e], cases, e]
Actor CasesEval
[anActor: anotherType, cases: [ExpressionCase <anotherType, aType> *], e: Environment]: aType →
cases
[] $ Throw NoApplicableCase [ ]™
[first, $rest] $±
first $ (aPattern: Pattern <anotherType> "\n"
anExpression: Expression <aType>)
: ExpressionCase <aType> $±
aPattern.match [anActor, e] $±
@Null $±
CasesEval * [anActor, rest, e] $™
@newEnvironment $±
anExpression.eval [newEnvironment] $|\n
(" else" elsePattern: Pattern <anotherType> "\n"
elseExpression: Expression <aType>)
: ExpressionElseCase <aType> $±
elsePattern.match [anActor, e] $±
@Null $±
Throw ElsePatternMustMatch [ ]™
@newEnvironment $±
elseExpression.eval [newEnvironment] $|\n
(" else" "\n"
elseExpression: Expression <aType>)
: ExpressionElseCase <aType> $±
elseExpression.eval[e] $™
else $ Throw NoApplicableCase [ ] $™\n
70
Actor \{ anExpression: Expression \langle anotherType \rangle \} "\●" 
  cases: ContinuationCases \langle anotherType, aType \rangle \} "\□"
: CasesContinuation \langle aType \rangle

implements Continuation \langle aType \rangle using
  perform \langle c: Environment, c: CheeseQ: aType \rightarrow CasesPerform \rangle [anExpression, eval[e], cases, e, c] § I

Actor CasesPerform
  \{ anActor: anotherType, 
  cases: (ContinuationCase \langle aType \rangle) *, 
  e: Environment, 
  c: CheeseQ: aType \rightarrow cases \●
  \} \+ Throw NoApplicableCase[].
  \{ first, \$ rest \} \+$
  first \● \langle aPattern: Pattern \langle anotherType \rangle "\●" 
    aContinuation: Continuation \langle aType \rangle \} 
    : ContinuationCase \langle aType \rangle \●
  aPattern, match[anActor, e] \●
    @ Null \+$
    CasesPerform[anActor, rest, e, c] \□
  @ newEnvironment \+$
    aContinuation, perform[newEnvironment, c] \□ \□\□
(" else")
  elsePattern: Pattern \langle anotherType \rangle "\●" 
  elseContinuation: Continuation \langle aType \rangle \} \+$
  elsePattern, match[anActor, e] \●
    @ Null \+$
    Throw ElsePatternMustMatch[] \□
  @ newEnvironment \+$
    elseContinuation, eval[newEnvironment] \□ \□\□
(" else" "\●")
  elseContinuation: Continuation \langle aType \rangle \} 
  : ContinuationElseCase \langle aType \rangle \+$
  elseContinuation, perform[e, c] \⫟
  else \+ Throw NoApplicableCase[] \□ \□ I
Holes in the cheese

Actor (anExpression:Expression <aType>
   "afterward" someAssignments:Assignments ")
   :Afterward <aType>

  implements Continuation <aType> using
  perform[e:Environment, c:CheeseQ]:aType →
  Let anActor ← anExpression .eval[e] ●
  Prep someAssignments .carryOut[e, c] ●
  c .leave[ ] ●
  anActor §

Actor (aVariable:Variable <aType>
   "=" anExpression:Expression <aType>):Assignment

  implements Assignment using
carryOut[e:Environment]:Void →
e .assign[aVariable, to anExpression .eval[e]]§

Actor ("Hole" anExpression:Expression <aType>): Hole <aType>

  implements Continuation <aType> using
  perform[e:Environment, c:CheeseQ]:aType →
  Let frozenEnvironment ← e .freeze[] ●
  // create frozen environment so that subsequent assignments
  // subsequent assignments do not affect evaluating anExpression
  Prep c .leave[ ] ●
  anExpression .eval[frozenEnvironment]$1

Actor ("Prep" aPreparations:Preparations ")
  anExpression:Expression <aType>):Prep <aType>

  implements Continuation <aType> using
  perform[e:Environment, c:CheeseQ]:aType →
  Let frozenEnvironment ← e .freeze[] ●
  // create frozen environment so that
  // preparation does not affect evaluating anExpression
  Prep aPreparation .carryOut[e, c] ●
  c .leave[ ] ●
  anExpression .eval[frozenEnvironment]$1
Actor ("Hole" anExpression: Expression <anotherType>  
"afterward"
  anAfterward: AfterwardContinuation <aType> "[?]"
  :Hole <aType> 

implies Continuation <aType> using
  perform[e: Environment, c: CheeseQ]: aType →
  Let frozenEnvironment ← e.freeze[[]].
  Prep c.leave[[]].
  Try Let anActor ← anExpression.eval[frozenEnvironment].
    Prep c.enter[[]]
      anAfterward.perform[e, c]
      c.leave[[]]
      anActor
    catch
      Prep c.enter[[]]
      anAfterward.perform[e, c]
      c.leave[[]]
    Rethrow ⍰ §

("Hole" anExpression: Expression <anotherType>  
"returned"
  returnedCases: ContinuationCases <anotherType, aType> "[?]"

"threw"
  threwCases: ContinuationCases <anotherType, aType> "[?]"
  :Hole <anotherType, aType> 

implies Continuation <aType> using
  perform[e: Environment, c: CheeseQ]: aType →
  Let frozenEnvironment ← e.freeze[[]].
  Prep c.leave[[]].
  Try Let anActor ← anExpression.eval[frozenEnvironment].
    Prep c.enter[[]].
    CasesPerform[anActor, returnedCases, e, c]
  cleanup
  Prep c.enter[[]].
  CasesPerform[anException, threwCases, e, c] ⍰ §
Actor ("Enqueue" anExpression: QueueExpression "●")\]
Enqueue implements Continuation using
  perform[e:Environment, c:CheeseQ]→ anExpression, eval[e].enqueueAndLeave[] ⦅

Actor ("Enqueue" anExpression: QueueExpression "●"
  aContinuation: Continuation<aType>)\]
Enqueue<aType> implements Continuation<aType> using
  perform[e:Environment, c:CheeseQ]:aType →
  Let anInternalQ ← anExpression, eval[e].
  Prep anInternalQ.enqueueAndLeave[] ●
  aContinuation .perform[e, c] ⦅
Type Discrimination, i.e., Discrimination, ↓

Actor ("Discrimination" aDiscrimination "between"
typeExpressions:Expressions <!Type> "T"):Definition
  implements Definition using
  eval[e:Environment]:Void →
  Let types ← typeExpressions,*eval[e].
  Actor aDiscrimination
      [aType:Type] →
      aType ∈ types
      True $ DiscriminationInstance*[x, aType]$ $\|$ False $ Throw NotADiscriminant[ ] \|$.

Actor DiscriminationInstance[x,aType, aType:Type]
  partially reimplements CastableType<!DiscriminationInstance, aType> using
    down[anotherType]:aType →
    anotherType
    aType $ x$ $\|$ else $ Throw WrongDisciminant[ ] $\|$
    down?[anotherType]:Boolean →
    anotherType
    aType $ True$ $\|$ else $ False$ $\|$

Actor ("↓ " discriminant:Pattern <!aType>)
  :Pattern<!aDiscrimination>
  implements Pattern<!aDiscrimination> using
  match[anActor:aDiscrimination, e:Environment]
    :Nullable<!Environment> →
    anActor↓?aType $\|$ True $ apattern*match[anActor↓aType, e]$ $\|$ False $ Null Environment$ $\|$. 

75
A Simple Implementation of Actor

The implementation below does not implement queues, holes, and relaying.

```plaintext
Actor ("Actor" declarations:ActorDeclarations

  "implements" Identifier<aType>

  "using" handlers:Handlers <anInterface> "§"; Definition

  implements Expression <anInterface> using

  eval[e:Environment] →

    Initialized <aType> <anInterface> eval[e],
    handlers,
    declarations, initialize[e],
    CheeseQ[]]§

Actor Initialized <aType>

  [anInterface:aType,
   handlers: [Handler*],
   e: Environment,
   c: CheeseQ]: aType →

  Actor implements anInterface using

    receivedMessage →  // receivedMessage received for anInterface

    Prep c.enter[] ●.

    Let aReturned ←

    Try Select, [receivedMessage, handlers, e, c]

    cleanup c.leave[] ●.  // leave cheese and rethrow exception

    Prep c.leave[] ●.

  aReturned§
```

76
**Actor** Select
[receivedMessage: `Message`,
 handlers: `[Handler*]`,
 e: `Environment`,
 c: `CheeseQ`: `aType →` handlers

  [ ] ⮚ Throw `MessageRejected` [ ] ⮜
  [[aMessageDeclaration: `MessageDeclaration` <`aType`> "→"
    body: `Continuation` <`aType`>]
    : `ContinuationHandler` <`aType`>]

  ∀restHandlers] ⮚
  aMessageDeclaration.match[receivedMessage, e] ⮚

  Null `Environment` [ ⮚
    Select*[receivedMessage, restHandlers, e, c] ⮜
    // process next handler
  ∀newEnvironment] ⮚
  Execute <`aType`>[body, newEnvironment, c] ⮜
An implementation of cheese that never holds a lock

The following is an implementation of cheese that does not hold a lock:

`Actor CheeseQ []`

```plaintext
invariants aTail = Null Activity ⇒ previousToTail = Null Activity,
aHeadHint ≔ Null Activity,                           // aHeadHint Nullable<Activity>
aTail ≔ Null Activity,                               // aTail Nullable<Activity>

enter[]: Void nonexclusive in myActivity →

Preconditions myActivity ≧ [previous] ≔ Null Activity,
                         myActivity ≧ [nextHint] ≔ Null Activity.

attempt[]: Void ≜

Prep myActivity ≧ [previous] ≔ aTail
    // set provisional tail of queue

Atomic aTail compare aTail update myActivity
updated ‡ // inserted myActivity in cheese queue with previous
    myActivity ≧ [previous]
    Null Activity ∠ Void  // successfully entered cheese
else ‡ Suspend  // current activity is suspended
    notUpdated ‡ attempt[,] ‡† // make another attempt

leave[]: Void nonexclusive in myActivity →
    // leave message received running myActivity

Preconditions aTail ≦ Null Activity

let ahead ← SubCheeseQ[head]
    // commentary for error checking

Atomic aTail compare ahead update Null Activity
updated ‡ // last activity has left this cheese queue
    Void afterward aHeadHint ≔ Null Activity  // the activity has left
notUpdated ‡ // another activity is in this cheese queue
    MakeRunnable ahead ≧ [nextHint]
    afterward aHeadHint ≔ ahead ≧ [nextHint]  // make another attempt

internal SubCheeseQ using
    // internal interface

[head]: Activity nonexclusive →

Preconditions aTail ≧ Null Activity

findHead.[[backIterator]: Activity ←
    aHeadHint
    Null Activity ∠ @aTail
    @anActivity ≦ anActivity [\]: Activity
backIterator ≧ [previous]
    // backIterator is head of this cheese queue
    Prep aHeadHint ≔ Nullable backIterator
    backIterator
    @previousBackIterator ≦
        // backIterator is not the head of this cheese queue
    Prep previousBackIterator ≦ [nextHint] ≔ Nullable backIterator
        // set nextHint of previous to backIterator

findHead.[[previousBackIterator] ⧦§]
```

78
The algorithm used in the implementation of CheeseQ above is due to Blaine Garst [private communication] cf. [Ladan-Mozes and Shavit 2004].

There is a state diagram for the implementation below:

As a consequence of the definition of CheeseQ:
Implementation CheeseQ has enter[ ]  Void
leave[ ]  Void

The implementation CheeseQ uses activities to implement its queue where
Implementation Activity has
[previous]  Nullable<Activity>
// if null then head of queue else, pointer to backwards list to head
[previous == Nullable<Activity> ]  Nullable<Activity>
// returns self so that updates can be chained
[nextHint]  Nullable<Activity>
// if non-null then pointer to next activity to get cheese after this one
[nextHint == Nullable<Activity> ]  Nullable<Activity>
// returns self so that updates can be chained

Implementation type InternalQ is defined on the next page where:
Implementation InternalQ has
enqueueAndLeave[ ]  Void,
enqueueAndDequeue[InternalQ]  Activity
dequeue[ ]  Activity
eempty?[ ]  Boolean
Actor InternalQ[c::CheeseQ]
  aQueue ← SimpleFIFO<Activity>[].
  enqueueAndLeave[]:Void in myActivity →
    // enqueueAndLeave message received in myActivity
    Prep aQueue • add[myActivity] •
    c • leave[]. // myActivity is the head of aCheeseQ
  Suspend[]
    // myActivity is suspended and when resumed returns Void[]
  enqueueAndDequeue[anInternalQ]:InternalQ[]:Activity in myActivity →
    Preconditions ¬ anInternalQ • empty?[].
    // commentary for error checking
    Prep aQueue • add[myActivity] •
    . • deQueue[] •
    Suspend[]
    deQueue[]:Activity in myActivity →
    Preconditions ¬ . • empty?[].
    // commentary for error checking
    Prep c • leave[] •
    // myActivity is the head of aCheeseQ
    MakeRunnable aQueue • remove[][]
    // make runnable the removed activity
    empty?[]:Boolean → aQueue • empty?[][§]

where
Interface FIFO<>Type> has
  add[anActivity::Type] ↦ Void,
  remove[anActivity::Type] ↦ aType,
  empty?[] ↦ Boolean[¶]
Appendix 3. ActorScript Symbols with IDE ASCII, and Unicode codes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>IDE ASCII(^1)</th>
<th>Read as</th>
<th>Category</th>
<th>Matching Delimiters</th>
<th>Unicode (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>; ;</td>
<td>end</td>
<td>top level terminator</td>
<td>25AE</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>of specified type</td>
<td>infix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{ }</td>
<td>[: ]</td>
<td>this Actor with interface (aspect)</td>
<td>prefix</td>
<td>2360</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>( i )</td>
<td>reduce (nullables, futures)</td>
<td>prefix</td>
<td>29BE</td>
<td></td>
</tr>
<tr>
<td>( v / )</td>
<td>( v / )</td>
<td>down</td>
<td>infix</td>
<td>2193</td>
<td></td>
</tr>
<tr>
<td>( v / ? )</td>
<td>( v / ? )</td>
<td>down query</td>
<td>infix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( v / v / )</td>
<td>( v / v / )</td>
<td>match downed</td>
<td>prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ^ )</td>
<td>( ^ )</td>
<td>up</td>
<td>infix</td>
<td>2191</td>
<td></td>
</tr>
<tr>
<td>( . . )</td>
<td>( . . )</td>
<td>qualified by</td>
<td>infix</td>
<td>22A1</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>is sent</td>
<td>infix</td>
<td>2025</td>
<td></td>
</tr>
<tr>
<td>( . )</td>
<td>.</td>
<td>send to this Actor</td>
<td>prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>necessarily concurrent</td>
<td>prefix</td>
<td>29B7</td>
<td></td>
</tr>
<tr>
<td>( - &gt; )</td>
<td>( - &gt; )</td>
<td>message type returns type(^5)</td>
<td>infix</td>
<td>21A6</td>
<td></td>
</tr>
<tr>
<td>( - &gt; )</td>
<td>( - &gt; )</td>
<td>cacheable ( - &gt; )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( = &gt; )</td>
<td>( = &gt; )</td>
<td>message received(^5)</td>
<td>( \downarrow )</td>
<td>2192</td>
<td></td>
</tr>
<tr>
<td>( \leftrightarrow )</td>
<td>( \leftrightarrow )</td>
<td>be(^\text{tt} )</td>
<td>infix</td>
<td>2190</td>
<td></td>
</tr>
<tr>
<td>( ? )</td>
<td>( ? )</td>
<td>cases</td>
<td>separator</td>
<td>( ? ) and ( ? )</td>
<td>FFFD</td>
</tr>
<tr>
<td>( ? )</td>
<td>( ? )</td>
<td>alternative case</td>
<td>separator</td>
<td>29B6</td>
<td></td>
</tr>
<tr>
<td>( ? )</td>
<td>( ? )</td>
<td>end cases</td>
<td>terminator</td>
<td>( ? )</td>
<td>2370</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
<td>another message handler</td>
<td>separator for handlers</td>
<td>( )</td>
<td>00B6</td>
</tr>
<tr>
<td>( $ )</td>
<td>( $ )</td>
<td>end handlers</td>
<td>terminator</td>
<td>implements and extension</td>
<td>00A7</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
<td>case</td>
<td>separator for case</td>
<td>|</td>
<td>2982</td>
</tr>
<tr>
<td>( _ / )</td>
<td>( _ / )</td>
<td>before</td>
<td>separator</td>
<td>Let binding, preparation, and Enqueue</td>
<td>2BC3</td>
</tr>
<tr>
<td>( _ / )</td>
<td>( _ / )</td>
<td>end</td>
<td>terminator</td>
<td>preparations, Preconditions, extends, and</td>
<td>FF61</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>( _ / )</td>
<td>( _ / )</td>
<td>to be</td>
<td>infix</td>
<td>( )</td>
<td>225C</td>
</tr>
<tr>
<td>( _ / )</td>
<td>( _ / )</td>
<td>is assigned</td>
<td>infix</td>
<td>( )</td>
<td>2254</td>
</tr>
<tr>
<td>( _ / )</td>
<td>( _ / )</td>
<td>matches value of(^3)</td>
<td>prefix</td>
<td>( )</td>
<td>2315</td>
</tr>
<tr>
<td>( = )</td>
<td>( = )</td>
<td>same as?</td>
<td>infix</td>
<td>( )</td>
<td>2260</td>
</tr>
</tbody>
</table>

\(^1\) These are only examples. They can be redefined using keyboard macros according to personal preference.
<table>
<thead>
<tr>
<th></th>
<th>=</th>
<th>keyword or field</th>
<th>infix</th>
<th>2338</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>:=</td>
<td>assignable field</td>
<td>infix</td>
<td>0076</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>begin type parameters</td>
<td>left delimiter</td>
<td>2A5B</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>begin set</td>
<td>left delimiter</td>
<td>2983</td>
</tr>
<tr>
<td></td>
<td></td>
<td>begin list</td>
<td>left delimiter</td>
<td>27E6</td>
</tr>
<tr>
<td>[</td>
<td>[</td>
<td>begin multi-set</td>
<td>left delimiter</td>
<td>201C</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>begin grouping</td>
<td>left delimiter</td>
<td>2985</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
<td>nothing&lt;sup&gt;2&lt;/sup&gt;</td>
<td>expression</td>
<td>229D</td>
</tr>
<tr>
<td>←</td>
<td>←</td>
<td>one-way send</td>
<td>infix</td>
<td>219E</td>
</tr>
<tr>
<td>➞</td>
<td>➞</td>
<td>one-way receive</td>
<td>infix</td>
<td>21A0</td>
</tr>
<tr>
<td>⟦</td>
<td>⟦</td>
<td>join</td>
<td>infix</td>
<td>2294</td>
</tr>
<tr>
<td>[</td>
<td>[</td>
<td>constrained by</td>
<td>infix</td>
<td>2291</td>
</tr>
<tr>
<td>]</td>
<td>]</td>
<td>extends</td>
<td>infix</td>
<td>2292</td>
</tr>
<tr>
<td>⇒</td>
<td>⇒</td>
<td>logical implication</td>
<td>infix</td>
<td>21E8</td>
</tr>
<tr>
<td>≡</td>
<td>≡</td>
<td>logical equivalence</td>
<td>infix</td>
<td>21D4</td>
</tr>
<tr>
<td>∧</td>
<td>∧</td>
<td>logical conjunction</td>
<td>infix</td>
<td>00D9</td>
</tr>
<tr>
<td>∨</td>
<td>∨</td>
<td>logical disjunction</td>
<td>infix</td>
<td>00DA</td>
</tr>
<tr>
<td>¬</td>
<td>¬</td>
<td>logical negation</td>
<td>prefix</td>
<td>00D8</td>
</tr>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>assert</td>
<td>prefix and</td>
<td>22A2</td>
</tr>
<tr>
<td>⊢</td>
<td>⊢</td>
<td>goal</td>
<td>prefix and</td>
<td>22A9</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>begin 1-line comment</td>
<td>prefix</td>
<td>EndOfLine</td>
</tr>
<tr>
<td>/*</td>
<td>/*</td>
<td>begin comment</td>
<td>prefix</td>
<td>*/</td>
</tr>
</tbody>
</table>
Index

→, 21
⁄, 8, 71, 81
⋯, 70
_, 82
√, 43, 50, 78, 82
∂, 12, 33, 81, 82
Æ, 41, 82
(, 82, See Expressions
*, 46
/, 81
\/, 82
→, 15
\, 81
∈, 82
[,] 6, 9, 33, 46, 82
list, 67
\, 65
\{, 82
\[\]>, 81
+++, 21
⊙, 36, 43, 44, 63, 64, 78, 81
=, 49, 51, 78, 81
≠, 57, 78, 81
□, 6, 48, 66, 81
⊙⊙, 18, 38, 80, 81
\:\, 35, 40, 41, 44, 45, 47, 66, 82
expression, 67
pattern, 67
\=, 11, 72, 78, 80, 81
≠, 46, 49, 78, 81
□, 82
≒, 33, 59
∪, 82
∩, 55, 57, 82
\∩, 55, 57, 82
⋄, 13, 48, 50, 51, 81
△, 56, 81
□, 39, 82
\, 81
□, 54, 58, 78, 80, 81
\□, 8, 81
c, 12, 18, 64, 81
[,] 5, 81, See Expressions
\\, 60, 81
\→, 11, 21, 22, 81
\→, 52, 82
\→, 9, 81
⇒, 82
\↓, 15, 33, 60, 81
\↓?, 15, 60, 81
\↓\downarrow, 15, 75, 81
→, 6, 46, 81, See Binding locals, See definition
⇐, 52, 82
⇔, 82
$, 11, 81
¶, 11, 81
*, 8, 81

Activity, 79
Actor, 11, 13, 18, 21, 54, 76
CheeseQ, 78
dequeue, 80
enqueueAndDequeue, 80
enqueueAndLeave, 80
InternalQ, 80
Swiss cheese, 16
Actor Model
Message passing, 2
types, 2
afterward, 11, 18, 49
Agha, G., 23
ASCII, 81
Athas, W., 23
Atkinson, R., 23
Atomic, 49, 78
Atomic... compare... update... updated... notUpdated... 69
Attardi, G., 23
backout, 21, 22
Baker, H., 23
Barber, G., 23
Beard, P., 23
become, 45
Bishop, P., 23
Boden, N., 23
Briot, J., 23
Cartesian, 38
cases, 8
cast
downcast, 16
self to interface of this Actor, 16
upcast, 16
catch\, 37
cheese, 20
dequeue, 79
enqueueAndDequeue, 79
enqueueAndLeave, 79
send, 59
Simi, M., 23
Smith, S., 23
Steiger, R., 23
String, 47
Structure, 14, 15
Suspend, 78, 80
Swiss cheese, 16
Symbols, 81
Talcott, C., 23
Terminal, 34, 44
Thati, P., 23
thatIs, 8
Theriault, D., 23
This (JavaScript), 50
throw, 52, 59
Throw, 11, 37
Tokoro, M., 23
Tree, 14, 15
Trie, 34, 44
TrieFork, 34
Try, 37
Try ... catch, 68

Try ... cleanup, 68
type
Discrimination, 62
paramaterized, 33
Type, 59
ReceivingType, 59
Restriction, 60
SendingType, 59
types, 5
Unicode, 81
uses, 13, 54
UsingNamespace, 50
Varela, C., 23
variable
Actor, 20
ActorScript, 10
variables, 10, 20
Void, 11
When, 55, 56, 57
Woelk, D., 23
XML, 51
Yonezawa, A., 23
End Notes

1 Quotation by the author from late 1960s.
2 to use a reserved word as an identifier it could prefixed, e.g., _actor
3 The delimiters { and } are used to delimit program syntax with the character “ and the character ” to delimit tokens. For example, {3 “+” 4} is an expression that can be evaluated to 7. A special font is used for syntactic categories.

For example,

$$\{x:\text{Numerical} \, ^{+} \, y:\text{Numerical}\};\text{Numerical} \, \langle \text{Expression} \rangle$$

Also,

$$\{\text{Numerical} \, ^{-} \, \text{Numerical}\};\text{Numerical} \, \langle \text{Expression} \rangle$$
$$\{\text{Numerical} \, ^{*} \, \text{Numerical}\};\text{Numerical} \, \langle \text{Expression} \rangle$$
$$\{\text{Numerical} \, ^{/} \, \text{Numerical}\};\text{Numerical} \, \langle \text{Expression} \rangle$$
$$\{\text{"Remainder" Numerical} \, ^{/} \, \text{Numerical}\};\text{remainder:}\text{Numerical} \, \langle \text{Expression} \rangle$$
$$\{\text{"QuotientRemainder" Numerical} \, ^{/} \, \text{Numerical}\} : \{\text{Numerical, Numerical}\} \langle \text{Expression} \rangle$$
$$\{\"True\" \or \"False\" \} : \{\text{Expression} \langle \text{Boolean}\rangle \}$$
$$\{\text{Expression} \langle \text{Boolean}\rangle \ \"\&\" \ \text{Expression} \langle \text{Boolean}\rangle \} : \{\text{Expression} \langle \text{Boolean}\rangle \}$$
$$\{\text{Expression} \langle \text{Boolean}\rangle \ \"\mid\" \ \text{Expression} \langle \text{Boolean}\rangle \} : \{\text{Expression} \langle \text{Boolean}\rangle \}$$
$$\{\text{Expression} \langle \text{Boolean}\rangle \ \"\neg\" \ \text{Expression} \langle \text{Boolean}\rangle \} : \{\text{Expression} \langle \text{Boolean}\rangle \}$$
$$\{\text{Expression} \langle \text{Boolean}\rangle \ \"\text{Throw}\" \ \text{Expression} \langle \text{Boolean}\rangle \} : \{\text{Expression} \langle \text{Boolean}\rangle \}$$
See explanation of syntactic categories above. A word must begin with an alphabetic character and may be followed by one or more numbers and alphabetic characters.

Identifier ⊑ Word ⊑ Expression

// an Identifier is a Word, which is a subcategory of Expression

{[Expression ⊔ Definition ⊔ Judgment ] } "": Top

(Type ← Expression < Type > ) : Definition

{ messageType : Type ( "→" Type ) ; returnType : Type ; Type
  ("[ " Types "]" : Type
  { MoreTypes } : Types
  { Type ⊔ Type , MoreTypes } : MoreTypes

6 (Identifier < aType > "←" Expressions < aType > ) : Definition

{[Expression < aType > ( "" , ) ]
  U { Expression ( "" , "" , ) MoreExpressions < aType > )
    : MoreExpressions < aType >

  { [ Type , MoreTypes ] : MoreTypes

7 ("Actor" ProcedureName

  [" ArgumentDeclarations "] (" ; Type < returnType > ) →
  Expression < returnType > ) : Definition

ProcedureName < Expression

{ MoreDeclarations } : ArgumentDeclarations

{ SimpleDeclaration ( "" , MoreKeywordDeclarations )
  U { SimpleDeclaration , MoreDeclarations )

  : MoreDeclarations

  // Comma is used to separate declarations.

  { Identifier
    U { Identifier "" , [ Type ]
      ( "default" Expression ) : SimpleDeclaration
    } (KeywordArgumentDeclaration
      U {KeywordDeclaration , MoreKeywordDeclarations )
        : MoreKeywordDeclarations
    ) {Keyword ""] SimpleDeclaration} : KeywordDeclaration

Keyword < Word

8 The symbol ∎ is fancy typography for an ordinary period when it is used to denote message sending.

9 [Recipient: Expression "<" [" Arguments "]" ) : ProcedureSend

ProcedureSend < Expression

// Recipient is sent a message with Arguments

{ MoreArguments } : Arguments
((Expression {⊔ "", MoreKeywordArguments }))
 {⊔ Expression "", MoreArguments }):MoreArguments

 {KeywordArgument
   {⊔ KeywordArgument
     "", MoreKeywordArguments }):MoreKeywordArguments
 {Keyword "expression" Expression):KeywordArgument
 {Identifier<Procedure>
   "[" ArgumentDeclarations "] ") ": returnType:Type <aType> \text{→ Expressions}<\text{aType}>."
): Definition <Procedure>

 {test:Expression <patternType> "\text{Φ}
   ExpressionCases <patternType, aType> "\text{T}"):Expression <aType>
 (ExpressionCase <patternType, aType>
   {⊔ MoreExpressionCases <patternType, aType>)
 ;ExpressionCases <patternType, aType>
 (ExpressionCase <patternType, aType>
   {⊔ ExpressionElseCases <patternType, aType>)
 ;ExpressionElseCases <patternType, aType>
 (ExpressionElseCase <patternType, aType>
   {⊔ ExpressionElseCase <patternType, aType>
   "\text{T}":MoreExpressionElseCases <patternType, aType>)
 ;ExpressionElseCases <patternType, aType>
 (ExpressionElseCase <patternType, aType>
   {⊔ ExpressionElseCase <patternType, aType>
   "\text{T}":MoreExpressionElseCases <patternType, aType>)
 ;ExpressionElseCases <patternType, aType>
 ("else" "i" Expressions <aType>)
 {⊔ ("else" Pattern <patternType> "i" Expressions <aType>)
 ;ExpressionElseCase <patternType, aType>
 // The else case is executed only if the patterns before
 // the else case do not match the value of test.
 (Pattern <patternType> "i" Expressions <aType>)
 ;ExpressionCase <aType>

 ("let" MoreLetBindings "return:
 result:Expressions <aType>):Expression <aType>
 // Bindings are independent of each other
 {LetBinding {⊔ (LetBinding "", MoreBindings ))}:MoreLetBindings

 10 \text{Φ} takes care of the infamous "dangling else" problem [Abrahams 1966].

 11 \text{Φ}
Dijkstra[1968] famously blamed the use of the goto as a cause and symptom of poorly structure programs. However, assignments are the source of much more serious problems.

Continuations in ActorScript are related to continuations introduced in [Reynolds 1972] in that they represent a continuation of a computation. The difference is that a continuation of Reynolds is a procedure that takes as an argument the result of the preceding computation. Consequently, a continuation of Reynolds is closer to a customer in the Actor Model of computation.
The above expression creates an Actor with declarations for variables and message handlers:

```plaintext
( U ( "uses" ConstructorList )))
( U "management" Expression <Management> )
NamedDeclaration
MessageHandlers
InterfaceImplementations):ActorBody
(Identifier " ParametersDeclarations"
  ( U (" ArgumentDeclarations")) )
:ConstructorDeclaration

( Constructor ( U ".")
  ( U (Constructor "," MoreConstructors ".") ) ; ConstructorList
( Constructor
  ( Constructor "," MoreConstructors ) ; MoreConstructors
( ActorQueues NamesDeclarations ) : NamedDeclaration
( U ( MoreNameDeclarations ".") ) ; NamesDeclarations
( NameDeclaration
  ( Identifier " ", MoreNamesDeclarations ) ) ; MoreNameDeclarations
( Identifier
  ( U (" Type <aType>" ))
  "=" Expression <aType> ) ; IdentifierDeclaration
IdentifierDeclaration < NameDeclaration
( Variable ( U (" Type <aType>" ) )
  "=" Expression <aType> InstanceVariableQualifications )
  : VariableDeclaration
VariableDeclaration < NameDeclaration
Variable < Word
InstanceVariableQualifications < InstanceQualifications
( U InstanceVariableQualification
  ( InstanceVariableQualification
    InstanceVariableQualifications )
  : InstanceVariableQualifications

"nonpersistent" < InstanceVariableQualification
// A nonpersistent variable must be Nullable,
// and can be nulled out before a message is received
( "queues" QueueNames "."
  : ActorQueues
QueueName ( U (QueueName "," QueueNames ) ) : QueueNames
QueueName < Word
QueueName < Expression < Queue >
("Void") : Expression
```
{{InterfaceImplementation
  ( U MoreInterfaceImplementations))
  :InterfaceImplementations

{{"also" InterfaceImplementation
  ( U MoreInterfaceImplementations))
  :MoreInterfaceImplementations

{{ ( U "partially")
  ("implements" U "reimplements")
    ( U "exportable") Type "using"
      (MessageHandlers"§")U UniversalMessageHandler
    :InterfaceImplementation <aType>

(MessagePattern
  ( U (:" Type))
    ( U ("sponsor" Identifier< Sponsor>))
    " →" ExpressionsContinuation <aType>)
  :UniversalMessageHandler <aType>

( U MoreMessageHandlers);MessageHandlers

(MessageHandler
  U (MessageHandler "§" MoreMessageHandlers))
  :MoreMessageHandlers

  // The message handler separator is ¶.
  (MessageName "[" ArgumentDeclarations "]"
    ( U (":" returnType: Type <aType>)
      ( U ("sponsor" Identifier< Sponsor>)))
    " →" ExpressionsContinuation <aType>);MessageHandler

  // For a message with MessageName with arguments,
  // the response is Continuation
  (Expression <aType>
    " afterward" VariableAssignments);Continuation <aType>

  // Return Expression and afterward perform
  // MoreVariableAssignments
  (VariableAssignment
    ( U (VariableAssignment
       "," MoreVariableAssignments ","));VariableAssignments
  (VariableAssignment
    ( U (VariableAssignment
       "," MoreVariableAssignments)
      :MoreVariableAssignments
  (Variable "=" Expression <aType>);VariableAssignment <aType>

91
For example, consider the following:

```plaintext
Actor NeedTwo[]
  queues waiting.
  hasOne := False
  go[]: Void → hasOne ◁ True △ Void permit waiting △
  False □ Prep hasOne := True ◁,
    enqueue waiting ◁
  Void
```

The following expression must return `Void` because of mandatory concurrency:

```plaintext
Let aNeedTwo ← NeedTwo[].
  Prep □aNeedTwo.go[] ◁,
  aNeedTwo.go[]
```

However following expression might never return because of optional concurrency:

```plaintext
Let aNeedTwo ← NeedTwo[].
  Prep aNeedTwo.go[] ◁,
  aNeedTwo.go[]
```

The ability to extend implementation is important because it helps to avoid code duplication.

The following expression is equivalent to the following:

```plaintext
myBalance@SimpleAccount :=
  myBalance@SimpleAccount – anAmount
```

Ignoring exceptions in this way is not a good practice.
25 {"Enqueue" QueueExpression "●"
    Continuation<dataType>;Continuation<dataType>}
/*
1. Enqueue activity in QueueExpression
2. Leave the cheese
3. When the cheese is re-entered perform Continuation. */
("Prep" Preparation;"
    "enqueue" QueueExpression "●"
    Continuation<dataType>;Continuation<dataType>}
/*
1. Perform Preparation
2. Enqueue activity in QueueExpression
3. Leave the cheese
4. When the cheese is re-entered perform Continuation. */
Cases can be continuations:
{test;Expression " ❄"
    ContinuationCases<dataType, dataType> "["})
    :Continuation<dataType>}
{ContinuationCase<dataType, dataType>
    | {ContinuationCase<dataType, dataType>}
        "❄" MoreContinuationCases<dataType, dataType>})
    ContinuationElseCases
    :ContinuationCases<dataType, dataType>}
{ContinuationCase<dataType, dataType>
    | {ContinuationCase<dataType, dataType>}
        "❄" MoreContinuationCases<dataType, dataType>})
    :MoreContinuationCases<dataType, dataType>}
{Pattern<dataType> "¨"
    ExpressionsContinuation<dataType, dataType>}
    :ContinuationCase<dataType, dataType>}
| {MoreContinuationElseCases<dataType, dataType>}
    :ContinuationElseCases<dataType, dataType>}
{ContinuationElseCase<dataType, dataType>
| {ContinuationElseCase<dataType, dataType>}
        "❄" MoreContinuationElseCases<dataType, dataType>})
    :MoreContinuationElseCases<dataType, dataType>}
|{("else" "¨" ExpressionsContinuation<dataType>}
    | {("else" Pattern<dataType> "¨"
        ExpressionsContinuation<dataType, dataType>})
    :ContinuationElseCase<dataType, dataType>}
73
26 Equivalent to the following:

\[
\text{Actor Fringe} \quad \text{[aTree: } \text{Tree}[\text{aTree}] \text{]} \quad \rightarrow \\
\quad \text{aTree} \quad \text{[Leaf:aString] } \rightarrow \text{[String]*} \\
\quad \text{Fork[aLeft, aRight]} \quad \rightarrow \text{[String]*} \\
\quad \text{[Fringe, aLeft], [Fringe, aRight]} \quad \rightarrow \text{[String]*} \\
\]

27 Equivalent to the following:

\[
\text{Fringe} \quad \text{[Fork[Leaf]"The", Leaf]"boy" Tree] Tree} \\
\]

28 Swiss cheese was called “serializers” in the literature.

29(“” Message <aType>; Expression <aType> I

29(“” Message <aType>; Expression <aType> I

30 ReadersWriterConstraintMonitor defined below monitors a resource and throws an exception if it detects that ReadersWriter constraint is violated, e.g., for a resource r using the above scheduler:

\[
\text{Actor ReadersWriterConstraintMonitor[theResource: ReadersWriter]} \\
\quad \text{writing := False,} \\
\quad \text{numberReading := 0,} \\
\quad \text{implements ReadersWriter using} \\
\quad \text{read[aQuery: Query]: QueryAnswer} \\
\quad \text{Preconditions } \neg \text{writing.} \quad \text{// commentary for error checking} \\
\quad \text{Prep numberReading++ } \bullet \\
\quad \text{hole theResource, read[aQuery]} \\
\quad \text{afterward numberReading--} \bullet \\
\quad \text{write[anUpdate: Update]: Void } \rightarrow \\
\quad \text{Preconditions } \text{numberReading=0, } \neg \text{writing.} \\
\quad \text{Prep writing := True } \bullet \\
\quad \text{hole theResource, write[anUpdate]} \\
\quad \text{afterward writing := False } \bullet \\
\]

94
A downside of this policy is that readers may not get the most recent information.

A downside of this policy is that writing and reading may be delayed because of lack of concurrency among readers.

```plaintext
("Prep" Preparation,  
enqueue" QueueExpression  
  (⊔ "backout" Expressions))\ Continuation\{aType\});Continuation\{aType\}

/*
1. Perform Preparation
2. Enqueue activity in QueueExpression.
3. Leave the cheese
4. If an exception is generated by the activity while in the queue, then reenter the cheese, perform Expressions, and leave the cheese.
5. If no exception is generated by the activity while in the queue, then when allowed to continue, re-enter the cheese to perform Continuation. */

Cases can be continuations:
(test:Expression\{patternType\}  "<"  
  ContinuationCases\{patternType, aType\}  "["  
  :Continuation\{aType\}

(ContinuationCase\{patternType, aType\}  
  | MoreContinuationCases\{patternType, aType\})  
  :ContinuationCases\{patternType, aType\}

(ContinuationCase\{patternType, aType\}  
  { ContinuationCase\{patternType, aType\}  
    "[" MoreContinuationCases\{patternType, aType\}  
    :ContinuationCase\{patternType, aType\}  
  }  
  | MoreContinuationCases\{patternType, aType\})  
  :MoreContinuationCases\{patternType, aType\}

( | ContinuationElseCase\{patternType, aType\}  
  | (ContinuationElseCase\{patternType, aType\}  
    "[" MoreContinuationElseCases\{patternType, aType\}  
    :ContinuationElseCases\{patternType, aType\})  
  )  
  | (ContinuationElseCase\{patternType, aType\}  
    | (ContinuationElseCase\{patternType, aType\}  
      "[" MoreContinuationElseCases\{patternType, aType\}  
      :ContinuationElseCases\{patternType, aType\})  
    )  
  )  
  | ("else" "i" ContinuationList\{aType\})  
  | ("else" Pattern\{patternType\}  
    "[" ExpressionsContinuation\{aType\}  
    :ContinuationElseCase\{patternType, aType\})

// The else case is executed only if the patterns before
// the else case do not match the value of test.
```
The following are allowed in the cheese for a response to message affecting the next message:

```plaintext
(patternType "ExpressionsContinuation" aType) :ContinuationCase(patternType aType)  
```

The following can be used temporarily leave the cheese:

```plaintext
("Hole" Expression aType):Continuation aType  
```

```plaintext
("Prep" Preparation).  
```

```plaintext
("Prep" Preparation).  
```

```plaintext
("Prep" Preparation).  
```
-- is postfix decrement
Preconditions present for error checking

(Identifier<Type>
 "< ParametersDeclarations">"
 Expressions):ParameterizedDefinition
 ParameterizedDefinition<<Definition>
 // Parameterize definition with ParametersDeclarations
 ( U MoreParameterDeclarations):ParametersDeclarations
 (ParameterDeclaration
  U (ParameterDeclaration
    ""," MoreParameterDeclarations))
 :MoreParameterDeclarations
 (Identifier<Type> ([ U Qualifier]):ParameterDeclaration
 ( U ("extends" Type)):TypeQualifier
 (Identifier<Type> "< Parameters">):TypeExpression
 (Identifier<Type>
  U ( U (Identifier<Type> "," Parameters)):Parameters
 (" Discrimination" Identifier<Type>
  MoreTypeDiscriminations"I"):Definition
 (Identifier<Type>
  U (Identifier<Type> ",","MoreTypeDiscriminations))
 :MoreTypeDiscriminations
 (Expression:<DiscriminationType> "↓ Type <aType>
  :Expression:<aType>
 // Discriminate to have the type Type <aType> if possible.
 // Otherwise, an exception is thrown.
 (Expression:<aDiscriminationType> "? Type <aType>
  :Expression:<Boolean>
 // If Expression discriminates to have the type Type <aType>,
 // then True, else False.
 (Pattern:<DiscriminationType> "↓ Type <aType>
  :Pattern:<aType>
 // If matching Actor is a discrimination that can be discriminated
 // then Pattern must match the discriminate.
 ("↓" Type <aType>):Pattern:<aType>
 // Matching Actor must be discrimination that can be
 // discriminated as aType

Equivalent to the following:

Let x ← 3.

TrieFork<Integer>[Terminal<Integer>[x]]Trie<Integer>,
Terminal<Integer>[x+1]]Trie<Integer>]
39 Equivalent to the following:

\[
\text{Actor TrieFringe} : \text{aType} \\
[\text{aTrie}: \text{Trie} : \text{aType}] : \text{aType*} \rightarrow \\
\text{aTrie} \downarrow \text{Terminal} : \text{aType} \downarrow [x] : [\text{aType*}] \downarrow \\
\text{aTrieFork} : \text{aType} \downarrow [\text{left, right}] : \\
[\text{TrieFringe} : \text{left}] \downarrow \text{TrieFringe} : \text{aType} : [\text{right}] : [\text{aType*}] \downarrow \]

40 Equivalent to the following:

\[
\text{Actor TrieSameFringe} : \text{aType} \\
[\text{left}: \text{Trie} : \text{aType}, \text{right}: \text{Trie} : \text{aType}] : \text{Boolean} \rightarrow \\
\text{TrieFringe} : \text{left} = \text{TrieFringe} : \text{right} \downarrow \\
(\text{Identifier} : \text{aType} = "[" \text{Arguments} "]") : \text{Expression} : \text{aType} \downarrow \\
(\text{Identifier} : \text{aType} = "[" \text{Patterns} "]") : \text{Pattern} : \text{aType} \downarrow \\
("\text{Nullable}\) \text{Expression} : \text{aType} = \text{Expression} : \text{Nullable} : \text{aType} \downarrow \\
("\text{Nullable}\) \text{Expression} : \text{Nullable} : \text{aType} = \text{Expression} : \text{Nullable} : \text{aType} \downarrow \\
// reduce \text{Expression} if not null. \\
// Otherwise, an exception is thrown. \\
("\text{Nullable}\) \text{Pattern} : \text{Pattern} : \text{Nullable} : \text{aType} \downarrow \\
// If matching Actor is a non-null nullable \\
// then \text{Pattern} must match the Actor in the nullable. \\
43 ("\text{Try}\) \text{anExpression}: \text{Expression} : \text{aType} \\
("\text{Catch}\) \text{ExpressionCases} : \text{Exception}, \text{aType} = "["") : \text{Expression} : \text{aType} \downarrow \\
/* \\
- If \text{anExpression} throws an exception that matches the pattern \\
of a case, then the value of \text{TryExpression} is the value 
computed by \text{ExpressionCases} \\
- If \text{anExpression} doesn't throw an exception, then then the value 
of \text{TryExpression} is the value computed by \text{anExpression}. */ \\
("\text{Try}\) \text{anExpression}: \text{Expression} : \text{aType} \\
("\text{Catch}\) \text{ContinuationCases} : \text{Exception}, \text{aType} = "["") : \text{Continuation} : \text{aType} \downarrow \\
/* \\
- If \text{anExpression} throws an exception that matches the pattern of 
a case, then the response of \text{TryContinuation} is the response 
computed by the expression of the case. \\
- If \text{aContinuation} doesn't throw an exception, then then the response of \text{TryExpression} is the response computed by \text{anExpression}. */
(`Try` anExpression:Expression:<aType>)
  
  `cleanup` cleanup:Expression:<aType>;Expression:<aType>`

`/*`

- If anExpression throws an exception, then the value of TryExpression is the value computed by cleanup.
- If anExpression doesn't throw an exception, then the value of TryExpression is the value computed by anExpression. */

44 {(`/Preconditions` test:Expressions <Boolean>` "" `Expressions:<aType>;Expression:<aType>`
  // Each of expressions in test must evaluate to True or
  // an exception is thrown

(`/Preconditions` Expressions:<Boolean>` "" `ExpressionsContinuation:<aType>;Continuation:<aType>`
  // Each of expressions in Expressions must evaluate to True or
  // an exception is thrown

{(`/value:Expression:<aType>`
  `postcondition` pre:Expression:[[aType]->Boolean]`)
  // The expression pre must evaluate to True when sent value
  // or an exception is thrown

45 `o` is a reserved postfix operator for degrees of angle

46 Equivalent to the following:

.Actor Times
  [u:Complex, v:Complex]:Complex ->
  Cartesian[u.[real]*v.[real] − u.[imaginary]*v.[imaginary],
  u.[imaginary]*v.[real] + u.[real]*v.[imaginary]] `Complex`

47 Equivalent to the following:

.Actor Times
  [Polar[angleกอง, magnitudeจะ]:Complex]` Polar[angleกอง, magnitudeจะ]:Complex ->
  Polar[angleกอง+anotherAngle, magnitudeจะ*anotherMagnitude]:Complex

48 (`Structure` Identifier<Type>` "" `FieldDeclarations` `]
  ( `uses` ConstructorList `))
NamedDeclaration
MessageHandlers
  MoreInterfaceImplementations):Definition`
  // Structure definition with StructureImplementation
  {anExpression:Expression:anotherType; "" Type:<aType>`
  :Expression:<aType>`
anExpression:Expression anotherType ★

"if" Type aType:Expression Boolean ★
// If anExpression is an extension of aType, then True else False
Pattern:Pattern anotherType ★

"if" Type anotherType;Pattern aType ★
// Matching Actor must be an extension of aType which
// matches aPattern

("if" Type Extension anotherType ★);Pattern aType ★
// Matching Actor must be an extension of aType

MoreFieldDeclarations:FieldDeclarations ★

SimpleFieldDeclaration ★

Identifier ★

default:Expression ★

NamedFieldDeclaration ★

FieldName ★

 QualifiedName ★
// "" is used for assignable fields.
ActorBody:StructureImplementation ★
Expression "" FieldName ★ FieldSelector ★
// FieldName of Expression which must be a structure
FieldSelector ★ Expression ★

StructureName "" FieldExpressions ""] ★ StructureExpression ★
StructureExpression ★ Expression ★

MoreFieldExpressions:FieldExpressions ★

SimpleFieldExpression ★

"", MoreFieldExpressions ★

NamedFieldExpression ★

"", MoreNamedFieldExpressions ★

FieldName ★

SimpleFieldExpression ★

"", MoreNamedFieldExpressions ★

NamedFieldExpression ★

SimpleFieldExpression ★

NamedFieldExpression ★

NamedFieldExpression ★
(StructureName "[ FieldPatterns ]" : StructurePattern
  StructurePattern ⊑ Pattern
  ( MoreFieldPatterns : FieldPatterns
  ( [ SimpleFieldPattern { U ("," MoreNamedFieldPatterns) }
    U { SimpleFieldPattern "\", MoreFieldPatterns }
  ) : MoreFieldPatterns
(NamedFieldPattern
  U { NamedFieldPattern "," MoreNamedFieldPatterns }
  ( ("\") U ":" ) SimpleFieldExpression)
  : NamedFieldPattern

49 ("[" ComponentExpressions<atType> "]")
  : Expression<[atType]>:
    // An ordered list with elements ComponentExpressions
    ( U MoreComponentExpressions<atType> )
    : ComponentExpressions<atType>:
    ( 
      ( U "\" ) Expression<atType>
      U { U "\" ) Expression<atType>
        "," MoreComponentExpressions<atType> )
    )
    : MoreComponentExpressions<atType>:
    ("[" TypeExpressions<atType> "]") TypeExpression<atType>:
    ( U MoreTypeExpressions<atType> ) TypeExpressions<atType>
      ( TypeExpression<atType> 
      U ( TypeExpression<atType> "," MoreTypeExpressions<atType> )
      )
      : MoreTypeExpressions<atType>:

50 ("\_\_\_\_\_": UnderscorePattern
  UnderscorePattern ⊑ Pattern
  Identifier ⊑ Pattern
    ( Pattern "that\_s" Expression ) : ThatIs
      ThatIs ⊑ Pattern
    ( "\_\" Expression<atType> ) : Pattern<atType>
      ("[" ComponentPatterns<atType> "]") : Pattern<atType>:
      // A pattern that matches a list whose elements match
      // ComponentPatterns
    ( U MoreComponentPatterns<atType> )
      : ComponentPatterns<atType>:
    ( Pattern<atType>
      U { "\" Pattern<atType> }
      U { Pattern<atType> "," MoreComponentPatterns<atType> }
    )
    : MoreComponentPatterns<atType>:

101
51 Equivalent to the following:

```
Actor AlternateElements<[aType]>
[aList:[aType*]]:[aType*] \to
aList \(\\square\)
[ ]:[aType*] \& [ ]:[aType*] \(\square\)
[anElement]:[aType*] \& [anElement]:[aType*] \(\square\)
[firstElement, secondElement]:[aType*] \(\square\)
[firstElement]:[aType*] \(\square\)
```

else \(\\square\)
[firstElement, secondElement, \(\forall\)remainingElements]:[aType*] \(\\uparrow\)
[firstElement, \(\forall\)AlternateElements,remainingElements]:[aType*] \(\square\)

52 \{
"ComponentExpressions":
"Expression":(aType*)\}\l
// A set of Actors without duplicates

53 \{
"ComponentPatterns":
"Pattern":(aType*)\}\l
// A multiset of the Actors with possible duplicates

54 Optimization of this program is facilitated because:
- The records are cacheable because their type is \{ContactRecord*\}
- All of the operators are cacheable
- The operators are annotated as cacheable using "|\-->

55 \{
"Map":
"ComponentExpressions":
"Expression":Map\}\l

56 It is possible to define a procedure that will produce a “bottomless” future.

For example, \texttt{Actor f [ ]:Future<(aType)> \to Future f [[]]}

57 An Actor of \texttt{FutureList<(aType)>} is either

1. the empty list of type \texttt{FutureList<(aType)>}
2. a list whose first element is of \texttt{aType} and whose rest is of \texttt{FutureList<(aType)>}

58 Equivalent to the following:

```
Actor Size
[aFutureList:FutureList<String>]:Integer \to
aFutureList \(\square\)
[]:FutureList<String> \(\square\) 0 \(\square\)
[first, \(\forall\)rest]:FutureList<String> \(\square\)
first, [length]+Size, [\(\forall\)rest] \(\square\)
```

59 \{Postpone Expression:(aType)>;Expression:Future<(aType)>\}\l
// postpone execution of the expression until the value is needed.
60 Equivalent to the following:

\[
\begin{align*}
\text{Actor} &\quad \text{TrieFringe} \langle \text{aType}\rangle \\
&\quad [\text{aTrie} : \text{Trie} \langle \text{aType}\rangle : \text{FutureList} \langle \text{aType}\rangle] \rightarrow \\
&\quad \text{aTrie} \\
&\quad \quad \quad \text{Terminal} \langle \text{aType}\rangle \langle x \rangle [\text{x}] : [\text{aType}^*] \\
&\quad \quad \quad \text{ForkTrie} \langle \text{aType}\rangle \langle \text{left, right}\rangle \\
&\quad \quad \quad [\text{TrieFringe}, \langle \text{left}\rangle] , \\
&\quad \quad \quad \forall \text{Postpone} \text{TrieFringe} \langle \text{aType}\rangle , [\text{right}] : [\text{aType}^*] \\
\end{align*}
\]

61 Equivalent to the following:

\[
\begin{align*}
\text{Actor} &\quad \text{FutureListOfReducedElements} \langle \text{aType}\rangle \\
&\quad [\text{aListOfFutures} : [\text{Future} \langle \text{aType}^*\rangle] : \text{FutureList} \langle \text{aType}\rangle] \rightarrow \\
&\quad \text{aListOfFutures} \\
&\quad \quad \quad \langle [] : [\text{Future} \langle \text{aType}^*\rangle] \rangle : [\text{FutureList} \langle \text{aType}\rangle] \\
&\quad \quad \quad \langle \text{aFirst, \text{aRest}} : [\text{Future} \langle \text{aType}^*\rangle] \rangle \\
&\quad \quad \quad \langle \text{@aFirst,} \\
&\quad \quad \quad \quad \forall \text{Future} \text{FutureListOfReducedElements} \langle \text{aType}\rangle , [\text{aRest}] ] :	ext{FutureList} \langle \text{aType}\rangle \\
\end{align*}
\]

62 {"Future" aValue : Expression < aType >}:

\[
\begin{align*}
&\quad \{ \text{U ("sponsor" Expression < Sponsor >))} \\
&\quad : \text{Expression} < \text{Future} \langle \text{aType}\rangle \rangle \\
\end{align*}
\]

// A future for aValue.

{"@" Expression < Future < aType > >} : Expression < aType >

// Reduce a future

63 A Postpone expression does not begin execution of Expression_1 until a request is received as in the following example:

\[
\begin{align*}
\text{Actor} &\quad \text{IntegersBeginningWith} \\
&\quad [n : \text{Integer} : [\text{FutureList} \langle \text{Integer} \rangle] \rightarrow \\
&\quad [n, \text{aPostpone} \text{IntegersBeginningWith} [n+1]] ] \\
\end{align*}
\]

Note: A Postpone expression can limit performance by preventing concurrency

64 {"(" MoreGrammars ")"} : Grammar 1

\[
\begin{align*}
&\quad \{ \text{("Grammar" U"Grammar")"} \} : \text{Grammar} 1 \\
&\quad \{ \text{ReservedWord ( U StartsWithIdentifier)}) : \text{StartsWithReserved} 1 \\
&\quad \text{StartsWithReserved MoreGrammars} 1 \\
&\quad \{ \text{Identifier} ( U StartsWithReserved) ) : \text{StartsWithIdentifier} 1 \\
&\quad \text{StartsWithIdentifier MoreGrammars} 1 \\
&\quad (\" \" \text{Word} \"\") : \text{ReservedWord} 1 \\
&\quad \langle \text{Grammar} : : \text{GrammarIdentifier} "1" \rangle : \text{Judgment} 1 \\
&\quad \{ \text{Identifier} \langle \text{Grammar} : \"\"\text{Identifier} \langle \text{Grammar} : "1" \rangle : \text{Judgment} 1 \\
\end{align*}
\]
FirstTenSequentially [n ← 10]: [Integer]\* 

\[ n = 1 \rightarrow \text{True} \; \&\& \; [P.]\* [\text{Integer}] \&\& \]

False \&\& \text{Let} \; x \leftarrow P.\* \bullet \;

\[ x, \text{FirstTenSequentially} [n-1] : [\text{Integer}] \&\& \]

OneOfTen [n: Integer ← 10]: [Integer] 

\[ n = 1 \rightarrow \text{True} \; \&\& \; P.\* \&\& \]

False \&\& \text{let} \; x \leftarrow P.\* \bullet \;

\[ x, \text{OneOfTen} [n-1] : \text{Integer} \&\& \]

The implementation below requires careful optimization.

The implementation below can be highly inefficient.

/* Atomically compare the contents of aLocation with the value of comparison. If identical, update the contents of aLocation with the value of update and execute compareIdentical. */

QualifiedNames Continuing

"Enumeration" Identifier [aType]: Definition
Declarations provide version number, encoding, schemas, etc.

If a customer is sent more than one response (i.e., return or throw message) then it will throw an exception to the sender of the response.

recipient: Expression

"↞" MessageName"["Arguments"]":Expression<Void> ∧
/* recipient is sent one-way message with MessageName and Arguments. Note that Expression<Θ> cannot be used to produce a value. */

Disallow ArgumentDeclarations that has a one-way continuation that returns nothing */

("∎" ( "permit" aQueue:Expression ))
("afterward" Afterward)))):Continuation<"Θ"> ∧

note the absence of "∎" in the implementation subexpressions

[Church 1932; McCarthy 1963; Hewitt 1969, 1971, 2010; Milner 1972, Hayes 1973; Kowalski 1973]. Note that this definition of Logic Programs does not follow the proposal in [Kowalski 1973, 2011] that Logic Programs be restricted only to clause-syntax programs.

A ground-complete predicate is one for which all instances in which the predicate holds are explicitly manifest, i.e., instances can be generated using patterns. See [Ross and Sagiv 1992, Eisner and Filardo 2011].

Execution can proceed differently depending on how sets fit into computer storage units.
Consider a dialect of Lisp which has a simple conditional expression of the following form:

\[
\text{⦅}(\text{\"if\"} \text{test:Expression then:Expression else:Expression})\text{⦆}
\]

which returns the value of then if test evaluates to \textbf{True} and otherwise returns the value of else.

The definition of \texttt{Eval} in terms of itself might include something like the following [McCarthy, Abrahams, Edwards, Hart, and Levin 1962]:

\begin{verbatim}
Define (Eval expression environment)
  // Eval of expression using environment defined to be
  (if (Numberp expression)                  // if expression is a number then
      expression)                            // return expression else
    (if ((Equal (First expression) (Quote if)))
        // if First of expression is "if" then
      (if (Eval (First (Rest expression) environment)   // if Eval of First of Rest of expression is True then
          (Eval (First (Rest (Rest expression)) environment))
        // return Eval of First of Rest of expression else
        (Eval (First (Rest (Rest (Rest expression)) environment)))
      // return Eval of First of Rest of Rest of expression
...
  ))
\end{verbatim}

The above definition of \texttt{Eval} is notable in that the definition makes use of the conditional expressions using \texttt{if} expressions in defining how to evaluate an \texttt{if} expression! */

If non-null points to head with current holder of cheese
If non-null, pointer to backwards list ending with head that holds cheese
// \texttt{enter} message received running \texttt{myActivity}
/* this cheese queue is not empty because \texttt{myActivity} is at the head of the queue */
Not to be confused with \texttt{\0} which is the null character or with \texttt{\0} which is \texttt{\0}
Used in type specifications for interfaces.
Used in message handlers.
Used to bind identifiers in \texttt{Let}.
Not to be confused with \texttt{\0} which is the null character or with \texttt{\0} which is \texttt{\0}.
Used in patterns.
Used in structures.
Used in one-way message passing.