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

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 with the following goal:

All physically possible digital computation can be directly implemented using ActorScript.

ActorScript is differentiated from previous programming 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.

---

i 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.

ii 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\(^1\)
  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 the 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\(^2\) enable secure communication with Actors

Computer software should not only work; it should also appear to work.\(^1\)

\(^1\) 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.

\(^2\) 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.:

- 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 words 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.1

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:

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:

---

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, \( x \leftarrow 3 \) 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 "Define", followed by a procedure name, a list of formal arguments, return type, "→" and body of the procedure. For example the procedure can be defined as follows:

\[
\text{Define } \text{Double} [v: \text{Integer}]: \text{Integer} \rightarrow v+ v
\]

The formal syntax of a procedure definition is in the end note: 8.

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 "\(\sqsubseteq\)" followed by a message with arguments delimited by "[" and "]". For example, \(\text{Double} \sqsubseteq [2+1]\) means send \(\text{Double}\) the message \([3]\). Thus \(\text{Double} \sqsubseteq [2+1]\) is equivalent to 61.

The formal syntactic definition of procedural message sending is in the end note: 10.

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 \(x+2\).

---

1 Anonymous procedures are also allowed as in the following:

\[
\lambda [v: \text{Integer}]: \text{Integer} \rightarrow v+ v
\]

2 e.g., _ matches 7
Identifiers\(^1\) can be bound using patterns as in the following examples:

- x is a pattern that matches "abc" and binds x to "abc"

**Cases, i.e., \(\$ , \_ \)\(^2\)**

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 "\(]".\(^3\) 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: "\(else\)" 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\(^iii\) 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:

```plaintext
Random [ ]
  0 $ // Random [ ] returned 0\(^v\)
    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: \(12\).
```

\(^1\) An identifier is a name that is used in a program to designate an Actor
\(^2\) "\(\)$" is fancy typography for "?"
\(^3\) *including patterns in previous else cases*
\(^iv\) As is standard, ActorScript uses the token "//" to begin a one-line comment.
\(^v\) Reserved words are shown in bold black.
Binding identifiers, i.e., ←
Identifiers can be bound using an identifier, followed by "←" and an expression. For example, aProcedure_a["G", "F", "F"]邺 is equivalent to the following:

\[
(x ← "F", // x is "F"
   aProcedure_a["G", x, x])邺
\]

Dependent bindings (in which each can depend on previous ones) can be accomplished as follows:

\[
(x ← "F", // x is "F"
y ← aProcedure_a["G", x, x], // y is aProcedure_a["G", "F", "F"]
anotherProcedure_a[x, y])邺
\]

The above is equivalent to
anotherProcedure_a["F", aProcedure_a["G", "F", "F"]]邺

The formal syntax of bindings is in the following end note: 13.

The formal syntactic definition of named-message sending is in the following end note: 14.

Lists, i.e., [ ] using Spread, i.e., [ ⩛ ]
The prefix operator "⩛" can be used to spread the elements of a list. For example

- \([1, ⩛[2, 3], 4]^{15}\) is equivalent to \([1, 2, 3, 4]\).
- \([[[1, 2], ⩛[3, 4]]]^{15}\) is equivalent to \([[1, 2], 3, 4]\)
- If y is \([5, 6]\), then \([1, 2, y, ⩛y]^{15}\) is equivalent \([1, 2, [5, 6], 5, 6]^{15}\)
- \([⩛[2, 3.0]]^{15}:[:Integer, Float]]^{15}\) is equivalent to \([2, 3.0]:[:Integer, Float]]^{15}\).

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

---

1: [:Integer, Float] is the type of a two element list, the first of which is of type Integer and the second of type Float
Within a list, "\"\" is used to match the pattern that follows with the list zero or more elements. For example:

- \[[x, 2], w\ y\] 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 \[[1, 2], w\ y\] matches \[[1, 2], 3, 4\]
- \[w x, w y\] is an illegal pattern because it can match ambiguously

Below is the definition of a procedure that computes the reverse of a list.

\[
\text{Define } \text{Reverse}\langle aType\rangle \cdot [aList: [aType\circ]] : [aType\circ] = \\
aList \spread \\
[ ] \spred [ ] \\
[\text{first}, rest] \spred [rest, first] \equiv 17
\]

The formal syntax of patterns is in the following end note: 18.

The following procedure returns every other element of a list beginning with the first:

\[
\text{Define } \text{AlternateElements}\langle aType\rangle \cdot [aList: [aType\circ]] : [aType\circ] = \\
aList \spread \\
[ ] \spred [ ] \\
[anElement] \spred [anElement]. \\
[\text{firstElement, secondElement}] \spred [\text{firstElement}]. \\
\text{else} \spred \\
[\text{firstElement, secondElement, wremainingElements}] \spred [\text{firstElement, wAlternateElements}, [\text{remainingElements}]] \equiv 1
\]

Consequently,

- \(\text{AlternateElements}\langle \text{Integer}\rangle \cdot [[[\ ]]]\) is equivalent to \([ ]: \text{Integer}\)
- \(\text{AlternateElements}\langle \text{Integer}\rangle \cdot [[3]]\) is equivalent to \([3]: \text{Integer}\)
- \(\text{AlternateElements}\langle \text{Integer}\rangle \cdot [[3, 4]]\) is equivalent to \([3]: \text{Integer}\)
- \(\text{AlternateElements}\langle \text{Integer}\rangle \cdot [[3, 4, 5]]\) is equivalent to \([3, 6]: \text{Integer}\)
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 "]", "\[\]", and a return type. For example, the interface type can be defined as follows:

```
Interface Account with
  availableBalance[ ]\[\]->Euro,
  deposit[Euro]->Void,
  withdraw[Euro]->Void
```

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.\textsuperscript{19}

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

```plaintext
Void; 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:

```plaintext
Actor SimpleAccount[startingBalance: Euro]
locals myBalance ::= startingBalance
// myBalance is an assignable variable initialized with startingBalance
implements Account using
availableBalance[ ]:Euro → myBalance
deposit[anAmount: Euro]: Void →
  Void myBalance ::= myBalance + anAmount
  // return Void; afterward the next message is
  // processed with myBalance reflecting the deposit
withdraw[anAmount: Euro]: Void →
  (amount > myBalance) True
  False Void myBalance ::= myBalance - anAmount
  // return Void; afterward 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: 21.
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:

```
(anAccount ← SimpleAccount[€6],
   // € is a reserved prefix operator
   anAccount.withdraw[€1] ||
   anAccount.withdraw[€2]●  // proceed only after both of the
   // withdrawals have been acknowledged
   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.**

The formal syntax of compound expressions is in the following end note: 22

An expression can be annotated for concurrent execution by preceding it with "♦" indicating that the following expression *must* be considered for parallel execution if resources are available. For example ♦Factorial[1000]+♦Fibonacci[2000]● 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.23

The formal syntax of explicit concurrency is in the following end note: 24.

---

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`

Actor `AccountSupervisor`[initialBalance: `Euro`]
uses `SimpleAccount`[initialBalance]
  // uses `SimpleAccount` implementation
locals `withdrawableIsRevoked` := False,
  `depositableIsRevoked` := False,
  `[revoker]: AccountRevoker → AccountRevoker`  // this Actor as `AccountRevoker`
  `[account]: Account → Account`   // this Actor as `Account`

withdrawFee[anAmount: `Euro`]: `Void` →
  Void ⇓ myBalance := myBalance−anAmount
  // withdraw fee even if balance goes negative
  // myBalance is myBalance⨀SimpleAccount

partially reimplements `Account` using
  // (availableBalance[ ]→`Euro`) from `SimpleAccount`
withdraw[anAmount: `Euro`]: `Void` →
  `withdrawableIsRevoked` ⇑
  True ⇓ `Throw Revoked[ ]`.  
  False ⇓ `SimpleAccount`→withdraw[anAmount]
  // use withdraw of `SimpleAccount`
deposit[anAmount: `Euro`]: `Void` →
  `depositableIsRevoked` ⇑
  True ⇓ `Throw Revoked[ ]`.  
  False ⇓ `SimpleAccount`→deposit[anAmount]
also implements `AccountRevoker` using
revokeDepositable[ ]: `Void` →
  Void ⇓ depositableIsRevoked := True
revokeWithdrawable[ ]: `Void` →
  Void ⇓ withdrawableIsRevoked := True
As a result of the above definition:

Implementation AccountSupervisor has

[revoker] \mapsto \text{AccountRevoker},

[account] \mapsto \text{Account},

withdrawFee[Euro] \mapsto \text{Void}

For example, the following expression returns negative €3:

(anAccountSupervisor ← AccountSupervisor,[€3],

anAccount ← anAccountSupervisor.[account],

aRevoker ← anAccountSupervisor.[revoker],

anAccount,withdraw[€2] \bullet // the balance is €1

aRevoker,\text{revokeWithdrawable}[] \bullet

Try anAccount.withdraw[€5] // try another withdraw

catch\_ Void \bullet // ignore the thrown exception

anAccountSupervisor,withdrawFee[€4] \bullet // €4 is withdrawn even though withdrawableIsRevoked

// myBalance is negative €3

anAccount,availableBalance[]

The formal syntax of the programs below is in the following end note: 30

**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

⟦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}]:\text{Integer} \mapsto \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:
  \[(\text{Leaf}[^{a}\text{STRING}]) \approx \text{Leaf}(\text{"The"}, ^{a}\text{STRING})\].
- \text{Leaf}(\text{"The"}, ^{\text{hash}}\text{]} \approx \text{Hash}[^{\text{hash}}\text{]}.

For example,
\[((\text{Leaf}(\text{"The"}))\text{: Tree})^{{\text{hash}}\text{]} \approx \text{Hash}[^{\text{hash}}\text{]}\.

\text{Fork} can be defined as an extension of \text{Tree} using:
\begin{align*}
\text{Structure Fork}\left[\text{left: Tree}, \text{right: Tree}\right] \\
\text{extends Tree using} \\
\quad [\text{hash}]: \text{Integer} \rightarrow \text{Hash}. \left[\text{left}^{{\text{hash}}\text{]}, \text{right}^{{\text{hash}}\text{]}\right]\).
\end{align*}

As a result of the above definition:
\begin{align*}
\text{Implementation structure Fork\text{[Tree, Tree] extends Tree}}\]
\end{align*}

For example, \text{Hash}[^{\text{hash}}\text{]}[^{\text{hash}}\text{]} \approx \text{Fork}[^{\text{hash}}\text{]}[^{\text{hash}}\text{]}.

Testing for convertibility from of a type to an extension of the type is done using an expression of the extension can followed by "↓?" and the type. For example,
- \(((\text{Leaf}(\text{"The"}))\text{: Tree})↓?\text{Fork}\approx \text{False}\]
- \(((\text{Leaf}(\text{"The"}))\text{: Tree})↓?\text{Leaf}\approx \text{True}\]

Conversion from of a type to an extension of the type is done using an expression of the extension can followed by "↓" and the type. For example,
- \(((\text{Leaf}(\text{"The"}))\text{: Tree})↓\text{Leaf}\approx \text{Leaf}[^{\text{hash}}\text{]}\]
- \(((\text{Leaf}(\text{"The"}))\text{: Tree})↓\text{Fork}\approx \text{Fork}[^{\text{hash}}\text{]}\]

"⊙⊙↓!" 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,\text{31}
\begin{align*}
\text{Define Fringe\text{[aTree: Tree] : String}} \\
\text{aTree} \equiv \text{aTree} \diamond \text{Leaf}[^{a}\text{STRING}] \diamond \text{Fork}[^{\text{left}}, ^{\text{right}}] \diamond \text{Fringe}_{\text{left}}, \text{Fringe}_{\text{right}}\]
\end{align*}
For example, ["The", "boy"]:String is equivalent to the following:
Fringe,[Fork,[Leaf,"The"],Leaf,"boy"]\]132

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

\[\text{Define SameFringe?:\[aTree:Tree, anotherTree:Tree\]:Boolean} = \]
// test if two trees have the same fringe
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
   c. an Actor to a restricted interface of the Actor
3. Conditional downcasting of an Actor of an interface type to an
   extension of the interface type.\footnote{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.}
   Downcasting of an interface type I 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

\textbf{Swiss cheese}

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

- **count** initially 0
- **continue** initially True

and concurrently sending it both a stop[ ] message and a go[ ] message such that:

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

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

A diagram is shown below for an implementation of 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.
Define CreateUnbounded[:,]:Integer =
(aCounter ← Counter[ ],) // let aCounter be a new Counter
  aCounter.go[ []] // send aCounter a go message and concurrently
  aCounter.stop[ ] // return the result of sending aCounter 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 Counter:

Actor Counter[]
  locals count := 0, // the variable count is initially 0
  continue := True |
  stop[ ]:Integer → count[] continue := False
  // return count; afterward continue is updated to
  // False for the next message received
  go[ ]:Void →
    continue |
    True $ (count := count+1), // increment count
    hole ..go[ ]). // send go[] to this counter
  False $ Void $ // if continue is False, return Void

As a result of the above definition
Implementation Counter has go[ ]→ Void,
  stop[ ]→ Integer

The above example illustrates how nondeterministic branching (in Turing
Machines) is not a good model for message reception in IoT.

The formal syntax of the programs above is in the following end note: 34
**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 operate concurrently with a reader.

Below are two implementations of readers/writer guardians for a shared resource that implement different policies:\(^{35}\)

1. *ReadingPriority*: The policy is to permit maximum concurrency among readers without starving writers.\(^{36}\)
   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.\(^{37}\)
   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:

\[
\text{Interface } \text{ReadersWriter with } \text{read[Query]} \rightarrow \text{QueryAnswer},
\]

\[
\text{write[Update]} \rightarrow \text{Void}
\]
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.\(^{\text{i}}\)

When an exception is thrown exogenously by an activity that is in a queue (e.g., **readersQ, writersQ**), 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: 38

---

\(^{\text{i}}\) A variable is orange in the diagram
\(^{\text{ii}}\) 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\(\geq 0\), writing\(\Rightarrow\) numberReading\(=0\)

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

locals writing \(\Rightarrow\) False

numberReading: \(\Rightarrow\) 0

implements ReadersWriter using

read[aQuery: Query]: QueryAnswer \(\rightarrow\)

\((\text{writing} \lor \neg \text{IsEmpty writersQ}) \Rightarrow \)

True \(\Rightarrow\) Enqueue readersQ // release cheese while in readersQ

backout (\(\neg\text{writing} \land \text{numberReading}=0 \land \text{IsEmpty readersQ}\) \(\Rightarrow\)

True \(\&\) Void permit writersQ

False \(\&\) Void

\(\implies\) Void

\(\neg\text{writing} \Rightarrow\) // commentary for error checking

\((\text{numberReading}++ \Rightarrow\) // increment numberReading

permit readersQ

hole theResource.read[aQuery] // release cheese for reading

\((\text{IsEmpty writersQ}) \Rightarrow\) // after releasing if writersQ is empty

True \(\Rightarrow\) Permit readersQ

False \(\Rightarrow\) numberReading=1

True \(\&\) Void permit writersQ also numberReading--

False \(\&\) numberReading=1

write[anUpdate: Update]. Void \(\rightarrow\)

\((\text{numberReading} \geq 0 \lor \neg \text{IsEmpty readersQ} \lor \text{writing} \lor \neg \text{IsEmpty writersQ}) \Rightarrow\)

True \(\Rightarrow\) Enqueue writersQ // release cheese while in writersQ

backout (\(\text{IsEmpty writersQ} \land \neg\text{writing}\) \(\Rightarrow\)

True \(\&\) Void permit readersQ

False \(\&\) Void

\(\implies\) Void

\(\neg\text{numberReading}=0 \lor \neg\text{writing} \Rightarrow\) precondition

// commentary for error checking

\((\text{writing} = \text{True} \Rightarrow\) // record that writing is happening

hole theResource.write[anUpdate] // release cheese for writing

\((\text{IsEmpty writersQ}) \Rightarrow\) // after writing if writersQ is empty

True \(\Rightarrow\) Permit writersQ also writing \(\Rightarrow\) False.

False \(\&\) Permit readersQ also writing \(\Rightarrow\) False
Illustration of writing-priority:

**Actor** WritingPriority [theResource: ReadersWriter]

- **Invariants**
  - numberReading ≧ 0, writing ⇒ numberReading ≦ 0

- **Locals**
  - numberReading := False,
  - numberReading := 0

- **Implements** ReadersWriter using

  `read[aQuery; Query]; QueryAnswer` →

  ```
  | ((writing → ¬Empty writersQ) ⬗
  |  True ⬗ Enqueue readersQ  // release cheese while in readersQ
  | backout ¬writing ∧ numberReading = 0 ∧ isEmpty readersQ ⬗
  |  True ⬗ permit writersQ,
  | False ⬗ Void ⬗
  |
  |
  |
  |
  |
  False ⬗ Void ⬗
  ¬writing precondition ⬗
  (numberReading++) ⬗
  permit isEmpty writersQ ⬗ True ⬗ readersQ, False ⬗ Void ⬗
  hole theResource, read[aQuery] // release cheese for reading
  (isEmpty writersQ) ⬗ // after reading if writersQ is empty

  | True ⬗ Permit readersQ,
  | False ⬗ numberReading = 1 ⬗
  | True ⬗ Permit writersQ also numberReading ̷ ⬗
  | False ⬗ numberReading ̷ ⬗

  `write[anUpdate; Update]; Void` →

  ```
  | ((numberReading > 0 ∨ ¬isEmpty readersQ ∨ writing ∨ ¬isEmpty writersQ) ⬗
  | True ⬗ Enqueue writersQ  // release cheese while in writersQ
  | backout (isEmpty writersQ ∨ ¬writing) ⬗
  | True ⬗ Void permit writersQ,
  | False ⬗ Void ⬗
  |
  |
  |
  |
  False ⬗ Void ⬗
  numberReading = 0 ∧ ¬writing precondition ⬗ // commentary for error checking

  (writing := True) ⬗
  hole theResource, write[anUpdate]  // release cheese for writing
  (isEmpty readersQ) ⬗ // after writing if readersQ is empty

  | True ⬗ Permit writersQ also writing := False,
  | False ⬗ Permit readersQ also writing := False ⬗
  ```
Conclusion

By the time the Software Engineering of a language gets in good shape, the language has become obsolete in "needed expressiveness"!
Alan Kay

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.

ActorScript has been designed for use with a TIDE (Team Integrated Development Environment). Implementation is the next task before us!

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 Decker, Ken Kahn, Henry Lieberman, Carl Manning, Mark S. Miller, Tom Reinhardt, Chuck Seitz, Dale Schumacher, Richard Steiger, Dan Theriault, 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.

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/

1 In the sense that the implementation holds a hardware lock.


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.


Carl Hewitt (2009b) *A historical perspective on developing foundations for client-cloud computing: iConsult™ & iEntertain™ Apps using iInfo™ Information Integration for iOrgs™ Information Systems* (Revised version of "Development of Logic Programming: What went wrong, What was done about it, and What it might mean for the future" AAAI Workshop on What Went Wrong. AAAI-08.) ArXiv 0901.4934.


Carl Hewitt (2010b) *iTooling™: Infrastructure for iAdaptive™ Concurrency*


Carl Hewitt, Erik Meijer, and Clemens Szyperski "The Actor Model (everything you wanted to know, but were afraid to ask)" http://channel9.msdn.com/Shows/Going-Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were afraid-to-ask Microsoft Channel 9, April 9, 2012.


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


Leslie Lamport How to make a multiprocessor computer that correctly executes multiprocess programs IEEE Transactions on Computers. 1979.


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


Henry Lieberman. An Object-Oriented Simulator for the Apiary Conference of the American Association for Artificial Intelligence, Washington, D. C., August 1983


Alexandre Miquel. *A strongly normalising Curry-Howard correspondence for IZF set theory* in Computer science Logic Springer. 2003


John Reppy, Claudio Russo, and Yingqi Xiao *Parallel Concurrent ML*. ICFP ’09.


David Taenzer, Murthy Ganti, and Sunil Podar, *Problems in Object-Oriented Software Reuse*. ECOOP’89.


Appendix 1. Extreme ActorScript

Parameterized Types, \(\text{i.e., } \langle \cdot \rangle \), \(\rangle\)

Parameterized Types are specialized using other types delimited by "\(\langle\)" and "\(\rangle\):\n
\[
\text{Actor Double}\langle\text{aType}\rangle \text{\par Arithmetic}\langle\text{aType}\rangle
\]

\[
[x:\text{aType}:\text{aType} \rightarrow \text{aType}|x+x|\$]
\]

// addition for \text{aType} that is \text{Arithmetic}

Parameterized Types have become increasingly important. For example, the following is adapted from [Greenman, Muehlboeck, and Tate 2014]:

\[
\begin{align*}
\text{Interface } & \text{Graph}\langle\text{Graph, Edge, Vertex}\rangle \\
& \text{with } [\text{vertices}] \mapsto [\text{Vertex}^{\odot}]\]
\end{align*}
\]

\[
\begin{align*}
\text{Interface } & \text{Edge}\langle\text{Graph, Edge, Vertex}\rangle \\
& \text{with } [\text{graph}] \mapsto \text{Graph}, \\
& \text{[source]} \mapsto \text{Vertex}, \\
& \text{[target]} \mapsto \text{Vertex}\]
\end{align*}
\]

\[
\begin{align*}
\text{Interface } & \text{Vertex}\langle\text{Graph, Edge, Vertex}\rangle \\
& \text{with } [\text{graph}] \mapsto \text{Graph}, \\
& \text{[incoming]} \mapsto [\text{Edge}^{\odot}], \\
& \text{[outgoing]} \mapsto [\text{Edge}^{\odot}]\]
\end{align*}
\]

\[
\text{Actor GeoMap}\langle\rangle \\
\text{implements } \text{Graph}\langle\text{GeoMap, Road, Intersection}\rangle \text{ using } \ldots\]
\]

\[
\text{Actor Road}\langle\rangle \text{ implements } \text{Edge}\langle\text{GeoMap, Road, Intersection}\rangle \text{ using } \ldots\]
\]

\[
\text{Actor Interaction}\langle\rangle \text{ implements } \text{Vertex}\langle\text{GeoMap, Road, Intersection}\rangle \text{ using } \ldots\]
\]

The formal syntax of parameterized types is in the following end note: \[41\].
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 discrimination can constructed using an expression followed by "↑" and the discrimination type. 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. An expression for a discrimination followed by "↓" and a type is the discriminate of that type.

For example, consider the following definition:

Discrimination IntegerOrString between Integer, String↓

Consequently,

- (3↑IntegerOrString)↓Integer↓ is equivalent to 3↓.
- ("a"↑IntegerOrString)↓Integer↓ throws an exception because String is not the same as the discriminant Integer.
- (3↑IntegerOrString)↓?Integer↓ is equivalent to True↓.
- (3↑IntegerOrString)↓?String↓ is equivalent to False↓.
- [3↑IntegerOrString, "a"↑IntegerOrString]↓IntegerOrString is of type [IntegerOrString*]

A pattern followed by "◯↓" and the type to be projected matches an Actor if the pattern matches the projection.

- The pattern x◯↓String matches "a"↑IntegerOrString and binds x to "a".
- The pattern x◯↓String does not match 3↑IntegerOrString
- The expression below is equivalent to 2↓:
  3↑IntegerOrString ◯ x↓Integer ≡ y-1,
  x↓String ≡ x  □↓

Discriminations can also be used in crypto as in the following definition:

Discrimination EmployeeNumberOrEncrypted between
   EmployeeNumber,
   Encrypted↓

with the result that having an address x of type EmployeeNumberOrEncrypted does not by itself provide access to an encrypted employee number from x without also having the type EmployeeNumber using Decrypt<EmployeeNumber> ↓[x↓Encrypted]

The formal syntax of type discrimination is in following end note: 42.
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>>[anActor:aType]`$

For example,

- The expression `(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:

Structure `TrieFork<<aType>>[left:Trie<<aType>>, right:Trie<<aType>>]$
  flip[]: TrieFork<<aType>> → // flip the branches
  TrieFork<<aType>>[right, left]$)

For example,

- The expression `(x ← 3, `TrieFork[Terminal[x], Terminal[x+1]]`)` is equivalent to the following:

  `TrieFork[Terminal[5], Trie[Terminal[6]]]$
- The pattern `TrieFork<<Integer>>[x, y]` matches

  `TrieFork[Terminal[5], Trie[Terminal[6]]]$
  and binds x to `Terminal[5]` and y to `Terminal[6].`

"☐☐↓" followed by a structure pattern an Actor if the pattern matches the projection.

---

$x$ is of type `Integer`
Below is the definition of a procedure that computes a list that is the "fringe" of the terminals of a Trie.¹

\[
\text{Define \ TrieFringe} \langle aType \rangle \cdot [\text{aTrie: Trie} \langle aType \rangle] : [\text{aType}^\flat] =
\]
\[
\text{Trie F} \quad \diamond \downarrow \text{Terminal} \langle aType \rangle [x] \uparrow
\]
\[
[\text{x}].
\]
\[
\diamond \downarrow \text{TrieFork} \langle aType \rangle [\text{left, right}] \uparrow
\]
\[
[\text{\forall TrieFringe} [\text{left}], \forall TrieFringe \langle aType \rangle [\text{right}]] \uparrow
\]

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

\[
\text{Define TrieSameFringe?} \langle aType \rangle \cdot [\text{left: Trie} \langle aType \rangle, \\
\text{right: Trie} \langle aType \rangle \rangle: \text{Boolean} =
\]
\[
\text{// test if two Tries have the same fringe}
\]
\[
\text{TrieFringe} \langle aType \rangle \cdot [\text{left}] = \text{TrieFringe} \langle aType \rangle [\text{right}] \uparrow
\]

The formal syntax of structures is in the following end note: ⁴⁴

**Nullable**

Distinguishing a special case to indicate the absence of an Actor is a long-time issue [Hoare 2009].

In an expression,

- "\(\text{Null} \)" followed by an expression of type Nullable\( \langle \text{aType} \rangle \) is the Actor (of type \(\text{aType} \)) in the nullable or throws an exception if there is no Actor.
- "Nullable" followed by an expression of type \(\text{aType} \) is the nullable (of type Nullable\( \langle \text{aType} \rangle \)) containing the value of the expression.
- "Null" followed by a type is the null for that type.

For example,

- Nullable 3 is of type Nullable\( \langle \text{Integer} \rangle \)
- Nullable 3 \(\text{Null} \) is equivalent to 3 \(\text{Null} \)
- Nullable 3 \(\text{Null} \) Integer throws an exception

¹ See definition of Trie above in this article.
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.
- **TheNull** only matches the null.

For example,
- The pattern ◇@x matches **Nullable** 3, binding x to 3
- The pattern ◇@x does not match **Null Integer**
- The pattern **TheNull** matches **Null Integer**

The formal syntax of nullables is in following end note: 45.

**Processing Exceptions, i.e., Try catchθ $, $ □ and Try cleanup**

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

```
Try Throw Exception["This is a test."] catchθ
   Exception[aString] $ aString □
```

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

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

The formal syntax of processing exceptions is in the following end note: 46.

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

A runtime requirement throws exception an exception if does not hold. For example, the following expression throws an exception that the requirement $x\geq0$ doesn't hold:

```
(x ← −1, 
$\ x\geq0$ precondition 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 λ[y:Float]:Boolean → y<1
```

The formal syntax requirements is in the following end note: 47.
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` →
    (theta ← Arcsine.[myImaginary/.[magnitude]],
    myReal>0 ◆
    True = theta,
    False $ myImaginary >0 ◆
    True =180°−theta,48
    False =180°+theta $§

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

For example, cartesisans can be used in the following procedure definitions:

Define `Times`<Complex>: [u: `Complex`, v: `Complex`]: `Complex` =
    Cartesian[u.[real] * v.[real] − u.[imaginary] * v.[imaginary],
        u.[imaginary] * v.[real] + u.[real] * v.[imaginary]] ¶50

Define `Equivalent`<Complex>: [u: `Complex`, v: `Complex`]: `Boolean` =
    u.[real] = v.[real] ∧ u.[imaginary] = v.[imaginary] ¶
Arguments with named fields, i.e.,  and :  
Polar Actors that implement Complex with named arguments angle and magnitude can be defined as follows:

```
Structure Polar[angle : Degrees default 0°, // angle of type Degrees is a named argument of Polar with
            // default 0°
magnitude : Length default 1.0] implements Complex using
            [real]: Float → magnitude * Sine.[angle]¶
            [imaginary]: Float → magnitude * Cosine.[angle]§
```

Consequently,
- Polar[].[real] is equivalent to 1

For example, the procedure Times for polars can be defined as follows:

```
Define Times <Polar>.
    [Polar[angle: Angle, magnitude: Length],
     Polar[angle: anotherAngle, magnitude: anotherMagnitude]]
        : Complex ≡
    Polar[angle: angle + anotherAngle,
          magnitude: magnitude * anotherMagnitude]§
```

The formal syntax of named arguments is in the following end note: 52.
Sets, i.e., \{ \} using spreading, i.e., \{ \forall \}
A set is unordered with duplicates removed.

The formal syntax of sets is in the following end note: 53.

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

The formal syntax of multisets is in the following end note: 54.

Maps,
A map is composed of pairs. For example, the following is a map:
\[
\text{Map}\langle \text{Integer, String}\rangle[[3]\mapsto \text{"a"}, [4]\mapsto \text{"b"}]\]

Pairs in maps are unordered, e.g.,
\[
\text{Map}\langle \text{Integer, String}\rangle[[3]\mapsto \text{"a"}, [4]\mapsto \text{"b"}]\] is equivalent to
\[
\text{Map}\langle \text{Integer, String}\rangle[[4]\mapsto \text{"a"}, [3]\mapsto \text{"b"}]\]

However, the expression \[
\text{Map}\langle \text{Integer, String}\rangle[[4]\mapsto \text{"a"}, [4]\mapsto \text{"b"}]\] throws an exception because a map is univalent.

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

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

\[\text{[ContactRecord\text{○}}\text{] has}
\text{filter[[ContactRecord] \mapsto \text{Boolean}]}
\text{\mapsto \{ContactRecord\text{○}}\text{),}
\text{collect [[ContactRecord] \mapsto [Age, Integer]]}
\text{\mapsto \text{Map}\langle \text{Age, Integer\text{○}}\rangle\text{\text{□}}\}]
\]

\[\text{Map\langle \text{Age, Integer\text{○}}\rangle\text{ has}
\text{reduceRange[[[Integer\text{○}}\text{] \mapsto \text{Float}]}
\text{\mapsto \text{Map}\langle \text{Age, Float\text{○}}\rangle\text{\text{□}}\}]
\]
\{\text{Number}\} \text{ has } \text{average[ ] |→ Float}

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

The program is as follows:

\text{Define AgeWithAverageOfNumberOfContactsSortedByAge,}
\text{[records:{ContactRecord}]: Sorted<Age> ⇐ records.filter [[aRecord: ContactRecord]}
\text{→ aRecord, [citizenship] ⃝ "US" , True, else ⃝ False]}
\text{⋅ collect [[aRecord: ContactRecord]}
\text{→ [aRecord, [yearsOld]], aRecord, [numberOfContacts]]}
\text{⋅ reduceRange}
\text{[[aSetOfNumberOfContacts:{Integer}]]}
\text{→ aSetOfNumberOfContacts, average[ ]}}
\text{⋅ sort[LessThanOrEqual<Age>]}\]

\text{Encryption}
Actor addresses can be type-encrypted using \text{Encrypt}. Using the above definition, the following is a contact record with fields \text{yearsOld}, \text{numberOfContacts}, and \text{citizenship} type encrypted:

\text{Encrypt ContactRecord\{yearsOld ⭕ 5, numberOfContacts ⭕ 7, citizenship ⭕ "UK"\}}

The above encrypted contact record can be decrypted only by using the type \text{ContactRecord}. For example, the encrypted record above matches the following pattern:

\text{Decrypted<\text{ContactRecord}> aRecord}
with aRecord bound to the decrypted record.
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]

\[(\text{aFuture} ← \text{Future Factorial} \[9999\], ◦ \text{aFuture}) \text{i} // do not proceed until Factorial \[9999\] has \text{been reduced} \]

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

\[(\text{aFuture} ← \text{Future Factorial} \[9999\]),
\text{g} ← (λ \text{[afuture:Future< Integer>]}: \text{Integer} → 5), \]

// g returns 5 regardless of its argument
\[\text{g, [aFuture]} \text{i} // \text{return 5 regardless of whether Factorial} \[9999\] \text{ has completed} \]

Note that the following are all equivalent:

- ◦ Future (4+Factorial \[9999\])
- 4+ ◦ Future Factorial \[9999\]
- 4+ ◦ Factorial \[9999\]
- ◦ (4+Factorial \[9999\])

Also ◦ Factorial \[9999\] + ◦ Fibonacci \[9000\] is equivalent to the following:

\[(n ← ◦ Factorial \[9999\]),
m ← ◦ Fibonacci \[9000\],
n+m) \text{i} // return Factorial \[9999\] + Fibonacci \[9000\]

---

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.
In the following example, the Factorial function might never be executed if `readCharacter` returns the character 'x':

```plaintext
(aFuture ← Future Factorial[9999],
readCharacter.[] ◊

'x' § 1,       // `readCharacter.[]` returned 'x'
else § 1+ ⓃaFuture Ⓝ[]

// `readCharacter.[]` returned something other than 'x'
```

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

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

```plaintext
Define TrieSameFringe? <aType>.
    [aTrie: Trie<aType>, anotherTrie: Trie<aType>]: Boolean =
    // test if two Tries have the same fringe
    TrieFringe <aType>, [aTrie] ≡ TrieFringe <aType>, [anotherTrie] // = reduces futures in the fringes
```

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

```plaintext
Define ListOfReducedElements <aType>.
    [aListOfFutures: [Future<aType>]]: [aType] =
    aListOfFutures ◊
      [] § []
      [aFirst, ∀aRest] §
        ⓃaFirst,
        ∀ListOfReducedElements <aType>, [aRest] ◊[ ]
```

The formal syntax of futures is in the following end note: 59.
Language extension, *i.e.*, \((\quad)\)

The following is an illustration of language extension that illustrates postponed execution:

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

\[\text{Postpone} \quad \text{aType}\]

\[\text{implement} \quad \text{Expression} <\text{Future}<\text{aType}> > \quad \text{using} \quad \text{eval}[e: \text{Environment}]: \text{Future}<\text{aType}> \rightarrow \]

\[\text{Future Actor implements aType using} \]

\[\text{aMessage} → \quad \text{// aMessage received} \]

\[(\text{postponed} ← \text{anExpression}, \text{eval}[e], \text{postponed}, \text{aMessage})|| \]

\[\text{// return result of sending aMessage to postponed} \quad \text{become postponed} \quad \text{// become the Actor postponed for} \quad \text{// the next message received}^1\]

The formal syntax of language extension is in the following end note: \(61\).
In-line Recursion (e.g., looping), i.e. Loop \[ \left. \vdots \right| \left. \vdots \right] \] is

Inline recursion (often called looping) is accomplished using "Loop", an initial invocation with identifiers initialized using \"\vdots\" followed by \"Is\" 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.

\[
\text{Loop Factorial,}[n \leftarrow 9, \text{accumulation} \leftarrow 1] \text{ is}
\]
\[
\begin{align*}
\text{n} &= 1 \quad \text{True} \quad \text{\text{\vdots accumulation,}} \\
\text{False} &\quad \text{Factorial,}[n-1, n*\text{accumulation}] \quad [\vdots]\text{I}\n\end{align*}
\]

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 \(P\) (of type \([\vdots]\rightarrow \text{Integer}\)):

\[
\text{Loop FirstTenSequentially,}[n \leftarrow 10] \text{ is}
\]
\[
\begin{align*}
\text{n} &= 1 \quad \text{True} \quad \text{\vdots [P,\vdots]} \\
\text{False} &\quad \text{FirstTenSequentially,}[n-1] \quad [\vdots]\text{I}\n\end{align*}
\]

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

\[
\text{Loop OneOfTen,}[n \leftarrow 10] \text{ is}
\]
\[
\begin{align*}
\text{n} &= 1 \quad \text{True} \quad \text{\vdots P,\vdots } \\
\text{False} &\quad \text{either P,\vdots OneOfTen,}[n-1] \quad [\vdots]\text{I}\n\end{align*}
\]

The formal syntax of looping is in the following end note: \(63\).

\(^i\) This construct is used instead of \text{while, for, etc.} loops used in other programming languages.

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

\[
\text{Loop Factorial,}[n:\text{Integer} \leftarrow 9, \text{accumulation} :\text{Integer} \leftarrow 1] :\text{Integer} \text{ is}
\]
\[
\begin{align*}
\text{n} &= 1 \quad \text{True} \quad \text{\vdots accumulation,} \\
\text{False} &\quad \text{Factorial,}[n-1, n*\text{accumulation}] \quad [\vdots]\text{I}\n\end{align*}
\]

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

\(^\text{iv}\) The procedure \(P\) may be indeterminate, \text{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".
- "", "2", "34"" is equivalent to "1234".
- "" is equivalent to "".

String patterns are delimited by "" and "". Within a string pattern, "$" is used to match the pattern that follows with the list zero or more characters. For example:
- "x", "2", "$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:

```
Define Reverse, [aString: String]: String ≡
aString "", "first", "rest", "$", "rest", first"
```

The formal syntax of string expressions is in the following end note: 65.

General Messaging, i.e., . and ⋈
The syntax for general messaging is to use an expression for the recipient followed by "" and an expression for the message.

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

The formal syntax of general messaging is in the following end note: 66.
Atomic Operations, *i.e.* Atomic compare update updated notUpdated

For example, the following example implements a lockable that spins to lock:

```
Actor SpinLock[]
locals locked := False // initially unlocked
implements Lockable\footnote{Interface Lockable with lock[]↦Void, unLock[]↦Void} using
  lock[]:Void ↦
    Loop Attempt[,] is // perform the loop Attempt as follows
      Atomic locked compare False update True ◀
        // attempt to atomically update locked from False to True
        updated $ locked := True precondition
        // commentary for error checking:
        // locked must have contents True
      Void // if updated return Void
    notUpdated $ Attempt[,] ◐ // if not updated, try again
  unLock[]:Void ↦
    locked := True precondition // commentary for error checking:
    // locked must have contents True
    Void ◐ locked := False $ // reset locked to False
```

The formal syntax of atomic operations is in the following end note: 68.
Enumerations, *i.e.*, Enumeration of using Qualifiers, *i.e.*, `Enumeration`

An enumeration definition provides symbolic names for alternatives using "Enumeration" followed by the name of the enumeration, "of", a list of distinct identifiers terminated by "▮".

For example,

\[
\text{Enumeration DayName of Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} \]

From the above definition, an enumerated day is available using a qualifier, *e.g.*, Monday\_\_\_DayName. Qualifiers provide for namespaces.

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

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

\[
\text{UsingNamespace DayName} \\
\text{Define FollowingDay[aDay: DayName]: DayName} = \\
\text{aDay \_ Monday }\_\_\_ \text{Tuesday,} \\
\text{Tuesday }\_\_\_ \text{Wednesday,} \\
\text{Wednesday }\_\_\_ \text{Thursday,} \\
\text{Thursday }\_\_\_ \text{Friday,} \\
\text{Friday }\_\_\_ \text{Saturday,} \\
\text{Saturday }\_\_\_ \text{Sunday,} \\
\text{Sunday }\_\_\_ \text{Monday} \]

The formal syntax of enumerations is in the following end note: 70.

Native types, *e.g.*, JavaScript, JSON, Java, HTML (HTTP), and XML

Because Actor addresses are typed, almost any kind of addressed can be accommodated.

**Object** can be used to create JavaScript Objects. Also, **Function** can be used to bind the reserved identifier **This**. For example, consider the following ActorScript for creating a JavaScript object aRectangle (with length 3 and width 4) and then computing its area 12:

\[
\text{(aRectangle}^\prime \leftarrow \text{Object ["length": 3, "width": 4]}, \\
\text{aFunction} \leftarrow \text{Function [\_\_\_ This["length"] * This["width",].} \\
\text{Rectangle["area"] := aFunction \_\_\_} \\
\text{aRectangle["area",[\_\_\_]\_\_])} \\
\]

\[\text{aRectangle is of type Object}^\prime \text{JavaScript}\]

---

\[\text{\_\_\_}_{1}\text{aRectangle is of type Object}^\prime \text{JavaScript}\]
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);
```

HTML strings can be used to create Actor addresses. For example, the Wikipedia English homepage can be retrieved as follows:

```javascript
(HTTPS://en.wikipedia.org).get()
```

**JSON** is a restricted version of **Object** that allows only Booleans, numbers, strings in objects and arrays.¹

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

Java types can be referenced using **Refer**, e.g.:

```java
Refer java.math.BigInteger
Refer java.lang.Number
```

The following notation is used for **XML**:²

```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>
```

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** type in Java

³ **Refer** is called **Import** in Java
One-way messaging, e.g., $\emptyset$, and $\odot$

One-way messaging is often used in hardware implementations.

Each one-way named-message send consists of an expression followed by "$\emptyset$", a message name, and arguments delimited by "$[~]$" and "$]$".

Each one-way message handler implementation consists of a named-message declaration pattern, ":", "$\emptyset$", "$\rightarrow$" and a body for the response which must ultimately be "$\emptyset$" which denotes no response.
The following is an implementation of an arithmetic logic unit that implements jumpGreater and addJumpPositive one-way messages:

Actor ArithmeticLogicUnit[aType] []
  implements ALU[aType] using
    jumpGreater[x:aType, y:aType, firstGreaterAddress:Address, elseAddress:Address]:Ο →
      InstructionUnit[Execute](x>y) $[
        True ↭ firstGreaterAddress, False ↭ elseAddress]
    addJumpPositive[x:aType, y:aType, sumLocation:Location[aType], positiveAddress:Address, elseAddress:Address]:Ο →
      (z ← (x+y), sumLocation $[
        aVariableLocation:VariableLocation[aType]: $[
          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] $[

The formal syntactic definition of one-way named-message and receiving is in the following end note: 73

---

1 VariableLocation[aType] has store[aType] → Void
2 TemporaryLocation[aType] has write[aType] → ⊥
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 → (⍠Male), [length]¶
   // using this Actor with Male interface
   [length]: Year → (⍠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]¶

Meta

Meta provides ability to provide extraordinary access to an Actor. For example, history of an Actor can be queried.

Interface Meta<aType> has
   [history] ‒→ [Request<aType> ⌼]
   reset anActor: aType] ‒→ Void¶

Interface Request<aType> has
   [message] ‒→ Message<aType>,
   [customer] ‒→ Customer<aType>,
   [response] ‒→ Future<Response<anotherType>>¶

Discrimination Response<anotherType> between
   Returned<anotherType>,
   Threw¶
Inconsistency Robust Logic Programs

Logic Programs\textsuperscript{75} can logically infer computational steps.

Forward Chaining
Forward chaining is performed using $\vdash$

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

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

Backward Chaining
Backward chaining is performed using $\models$

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

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

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

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

*When* \[ \vdash \text{Mortal}[x] \rightarrow (\vdash \text{Human} [\varnothing x] \rightarrow \vdash \text{Mortal} [x]) \]

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

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

**SubArguments**

This section explains how subarguments can be implemented in natural deduction.

*When* \[ \vdash_s (\psi \vdash_t \phi) \rightarrow (t' \leftarrow \text{Extension} [t], \vdash_{t'} \psi \parallel \vdash_{t'} \phi \rightarrow \vdash_t (\psi \vdash_t \phi)) \]

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 Link\(aNode, anotherNode, aCost\) that is true exactly when there is a path from \(aNode\) to \(anotherNode\) with \(aCost\).

**When** \(\models Path[aNode, aNode, aCost] \rightarrow\)

// when a goal is set for a cost between \(aNode\) and itself

\(\models aCost = 0\) \hspace{1cm} // assert that the cost from a node to itself is 0

The following goal-driven Logic Program works forward from \(start\) to find the cost to \(finish\):??

**When** \(\models Path[start, finish, aCost] \rightarrow\)

\(\models aCost = Minimum\{nextCost + remainingCost \mid \models Link[start, next\neq start, nextCost],\)

\(\models Path[next, finish, remainingCost]\}\)

// a cost from \(start\) to \(finish\) is the minimum of the set of the sum of the

// cost for the next node after \(start\) and

// the cost from that node to \(finish\)

![Diagram of a graph](image)

The following goal-driven Logic Program works backward from \(finish\) to find the cost from \(start\):

**When** \(\models Path[start, finish, aCost] \rightarrow\)

\(\models aCost = Minimum\{remainingCost + previousCost \mid \models Link[previous\neq finish, finish, previousCost],\)

\(\models Path[start, previous, remainingCost]\}\)

// the cost from \(start\) to \(finish\) is the minimum of the set of the sum of the

// cost for the previous node before \(finish\) and

// the cost from \(start\) to that Node

![Diagram of a reversed graph](image)

Note that all of the above Logic Programs work together concurrently providing information to each other.

55
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.

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 aType 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 aType extends Construct with
  eval[Environment] → aType,
  perform[Environment, CheeseQ] → aType

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

Actor BasicExpression aType []
  perform[e: Environment, c: CheeseQ] →
  Try (anActor ← [] Expression aType.eval[e] •
    c.release[] ||
    anActor)
  cleanup c.release[] §

The tokens ( and ) are used to delimit program syntax.

Actor (anIdentifier: Identifier aType); Expression aType:Type
  uses BasicExpression aType:[] |
  partially implements Expression aType using
    eval[e:Environment]:aType → e.lookup[anIdentifier]
The interface Type

Interface thisType:: with
  extension?[[]] |→ Boolean,
  has?[MethodSignature] |→ Boolean,
  sendOneWay[thisType, Message ↦ ⊥] ↦ ⊥,
  sendRequest[thisType, Message ↦ aReturnType] ↦ aReturnType,
  encrypt[.] ↦ Encrypted,
  encrypterType[Type ] ↦ EncrypterType\(^1\),
  decrypt[Encrypted] ↦ thisType,
  decryptType[thisType, EncrypterType] ↦ DecryptType\(^1\),
  decrypt?[Encrypted] ↦ Boolean,
  return[Customer ↦ aReturnType>, aReturnType] ↦ Void,
  throw[Customer, Exception] ↦ Void\(^1\)

**CommunicationType** is a restriction that can be used only for communication:

Interface thisType:CommunicationType restricts Type with
  sendOneWay[thisType, Message ↦ ⊥] ↦ ⊥,
  sendRequest[thisType, Message ↦ aReturnType] ↦ aReturnType,
  return[Customer ↦ aReturnType>, aReturnType] ↦ Void.
  throw[Customer, Exception] ↦ Void\(^1\)

**SendingType** is a restriction of **CommunicationType** that can be used only for sending:

Interface thisType:SendingType restricts CommunicationType with
  sendOneWay[thisType, Message ↦ ⊥] ↦ ⊥,
  sendRequest[thisType, Message ↦ aReturnType] ↦ aReturnType\(^79\)

---

\(^1\) Implementation **EncrypterType** has encrypt[thisType] ↦ Encrypted

\(^2\) Implementation **DecrypterType** has decrypt[Encrypted] ↦ thisType,
  decrypt?[Encrypted] ↦ Boolean
Suppose there is a type `Account` that needs have accounts that can be shared selectively among some IoT devices so that:

- some of the devices can operate using the address of an account
- some can only pass on an inoperable opaque address of the account
- some can convert an inoperable opaque address to an operable address of the account
- and some can convert an operable account address to an opaque inoperable address.

Construct type `et1` (that has the operations on accounts) using the constructor `EncrypterType` as follows:

```
et1 ← EncrypterType[Account] // et1:EncrypterType
```

Also, construct `dt1` (that has the operations on accounts) using the constructor `DecrypterType` as follows:

```
dt1 ← DecrypterType[Account, et1] // dt1:DecrypterType
```

An IoT device is given `x` (an address that can be used to perform operations on `anAccount`) and type `et1` (which can perform encryption) where:

```
u ← et1.encrypt[x] // u:Encrypted
```

An IoT device is given the encrypted address `u` and type `dt1` (which can perform decryption for addresses encrypted using `et1`) where:

```
y ← dt1.decrypt[u] // y:dt1
```

Then `x` and `y` are addresses for the same account and both can be used to operate on the account.

The same technique can be used for an individual account by creating encryption and decryption types for that account:

```
et2 ← EncrypterType[Account].
dt2 ← DecrypterType[Account, et2]
```
Only an Actor that possesses \texttt{dt2} can decrypt an Actor address encrypted using \texttt{et2}. 

\begin{verbatim}
Actor \{anotherType:Type\langle anotherType\rangle
    "?" aType:Type\langle aType\rangle;Expression\langle Boolean\rangle
uses BasicExpression\langle aType\rangle\[\] |
partially implements Expression\langle Boolean\rangle using
  eval[e:Environment]:Boolean \rightarrow
  \{anotherType.eval[e], extension?[aType.eval[e]]\}
\end{verbatim}

\textbf{Type Discrimination}

\textbf{Interface} \texttt{DiscriminationType} extends \texttt{Type} with
\begin{verbatim}
  up[Type] \mapsto \text{Discrimination},
  down[Discrimination] \mapsto \text{Type},
  down?[Discrimination] \mapsto \text{Boolean}
\end{verbatim}

\begin{verbatim}
Actor \{anExpression:Expression\langle aType\\rangle
    "↑↑" castExpression:Type\langle aDiscriminationType:Discrimination\rangle\}
uses BasicExpression\langle aType\rangle\[\] |
partially implements Expression\langle aType\rangle using
  eval[e:Environment]:aType \rightarrow
  castExpression.eval[e].up[anExpression.eval[e]]
\end{verbatim}
Actor
{aPattern: Pattern <aDiscriminationType>
  "↑" castExpression: Type <aDiscriminationType>
  : DiscriminationPatternUp <aType, aDiscriminationType>
uses BasicPattern <aDiscriminationType> [] |
partially implements Pattern <aDiscriminationType> using
  match [anActor: DiscriminationInstance <aType, aDiscriminationType>,
    e: Environment]: aType →
  aPattern.match[aDiscriminationType, up[anActor], e]}

Actor
{anExpression: Expression
  "↓" castExpression: Type <aType>
  : DiscriminationDown <aType, aDiscriminationType>
uses BasicExpression <aType> [] |
partially implements Expression <aType> using
  eval [e: Environment]: aType →
  castExpression, eval[e], down[anExpression, eval[e]]}

Actor
{aPattern: Pattern <aType>
  "↓" castExpression: Type <aType>
  : DiscriminationPatternDown <aType, aDiscriminationType>
uses BasicPattern <aType> [] |
partially implements Pattern <aType> using
  match [anActor: DiscriminationInstance <aType, aDiscriminationType>,
    e: Environment]: Nullable <Environment> →
  aPattern.match[aDiscriminationType, down[anActor], e]}
Actor
{ "\\[">" aStructurePattern:Pattern \(<aStructureType>\) \\
  :DownPattern\(<aStructureType, aDiscriminationType>\)
uses BasicPattern\(<aStructureType>[[ ]]
partially implements Pattern\(<aStructureType>\) using
  match\(\)anActor: DiscriminationStructureInstance\(<aType, \\
    aDiscriminationType>\),
    e: Nullable\(<Environment>\) \(\to\)
  structurePattern .match\(\)aDiscriminationStructureInstanceType \\
    .down?\(\)anActor, aStructureType, 
    e\) ◆
  True \(\$\)structurePattern \(\cdot\)match\(\)aDiscriminationStructureInstanceType \\n    .down\(\)anActor, aStructureType 
    e\)
  False \(\$\)Null \(<\Environment>\)\) }
Actor ("Discrimination" aDiscriminationType "between" typeExpressions: Types "I"): Definition

Actor implements Definition using
eval[e:Environment]:Environment →
(types ← typeExpressions.eval[e],
e.bind[aDiscriminationType, type <- DiscriminationType, to =>
Actor partially implements DiscriminationType with
up[anInstance:aType|types]:aDiscriminationType →
SimpleDiscriminationInstance
<aType, aDiscriminationType>[anInstance]¶
down[anUpped]
: DiscriminationInstance
<aType, aDiscriminationType>:aType|types →
anUpped ◆
≡≡SimpleDiscriminationInstance<aType>[anInstance]:
anInstance [2]
else := ThrowCastException[].
down?[anUpped: DiscriminationInstance
<aType, aDiscriminationType>:Boolean →
anUpped ◆
≡≡SimpleDiscriminationInstance<aType>[..] :=
True.
else := False [2]) □

Structure SimpleDiscriminationInstance<aType, aDiscriminationType>
[anInstance:aType]
extends DiscriminationInstance<aType, aDiscriminationType> □
Type restriction

**Interface** `RestrictionType<\text{aType}>`

extends `Type` with `up[\text{aType}] \mapsto RestrictionType<\text{aType}>`

**Actor** `{\text{anExpression}}: \text{Expression<\text{aType}>}`

"↑" `castExpression: Type < RestrictionType<\text{aType}>`:

`: RestrictionUp<\text{aType}>`

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

partially implements `Expression<RestrictionType<\text{aType}>` using

`eval[e: \text{Environment}]: RestrictionType<\text{aType}>`:

`castExpression.eval[e].\text{up[anExpression.eval[e]]}`

**Actor** `{\text{"Interface" \text{aRestrictionType}}`

rerestricts `\text{typeExpression}: Type < \text{aType}>`

"with" `signatureExpressions: Signatures"↑"`:

`Definition`

Actor implements `Definition` using

`eval[e: \text{Environment}]: \text{Environment} \mapsto`

`(\text{signatures} \leftarrow \text{signatureExpressions}, \text{eval[e]})`

`\text{typeExpression.eval[e].has[signatures].precondition e}`

`\text{bind[\text{aRestrictionType},}

\text{type \mapsto RestrictionType<\text{aType}>},

\text{to \mapsto}

\text{Actor implements RestrictionType<\text{aType}> using}

\text{\text{up[anInstance: \text{aType}]: aRestrictionType \mapsto RestrictionInstance[anInstance]]}}`

**Structure** `RestrictionInstance[anInstance: \text{aType}]` uses `BasicType[\text{]}`[]

partially reimplements

`RestrictionType<\text{aType}}` having

`(\text{aMessage} \mapsto \text{aReturnType})$signature$)`:

`using sendRequest[a\text{Recipient}: \text{aRestrictionType},

\text{aMessage: aMessage}: \text{aReturnType} \mapsto a\text{Recipient} \mapsto

\text{RestrictionInstance[anInstance]}$\leftarrow$ sendRequest[anInstance, aMessage].

\text{else$\mapsto$ Throw CastException[\text{]}$\mapsto$ sendOneWay[a\text{Recipient}: a\text{RestrictionType},

\text{aMessage: aMessage}]: \Theta \mapsto a\text{Recipient} \mapsto

\text{RestrictionInstance[anInstance]}$\leftarrow$ sendOneWay[anInstance, aMessage].

\text{else$\mapsto$ Throw CastException[\text{]}$\mapsto$]`
Type extension

Interface `Extension<>Type>`
    extends `Type` with
        up[`ExtensionInstance<Type>`] → `Type`,
        down[`Type`] → `Extension<Type>`,
        down?[`Type`] → `Boolean`

Actor `{anExpression: `Expression`<`anExtensionType`>}`
    up[`
        "↑" castExpression: `Type`<`aBaseType`>
    `>`]
        : `ExpressionUp`<`anExtensionType`, `aBaseType`>
    uses `BasicExpression`<`aBaseType`> []
    partially implements `Expression`<`aBaseType`> using
        eval[e:`Environment`]: `aBaseType` →
        castExpression, eval[e].up[anExpression, eval[e]]`

Actor `{aPattern: `Pattern`<`anExtensionType`>}`
    up[`
        "↑" castExpression: `Type`<`anExtensionType`>
    `>`]
        : `PatternUp`<`anExtensionType`, `aBaseType`>
    uses `BasicPattern`<`anExtensionType`> []
    partially implements `Pattern`<`anExtensionType`> using
        eval[e:`Environment`]: `aBaseType` →
        aPattern, match[anExtensionType, up[anActor], e]`

Actor `{anExpression: `Expression`<`aBaseType`>}`
    down[`
        "↓" castExpression: `Type`<`anExtensionType`>
    `>`]
        : `ExpressionDown`<`anExtensionType`, `aBaseType`>
    uses `BasicExpression`<`anExtensionType`> []
    partially implements `Expression`<`anExtensionType`> using
        eval[e:`Environment`]: `aBaseType` →
        castExpression, eval[e].down[anExpression, eval[e]]`
Actor

\[
\text{\{aPattern:Pattern} \left\langle \text{anExtensionType} \right\rangle
\]

\[
\text{"\text{\#1}" castExpression:Type} \left\langle \text{anExtensionType} \right\rangle
\]

\[
:\text{ExtensionPatternDown} \left\langle \text{aBaseType, anExtensionType} \right\rangle
\]

\text{uses BasicPattern} \left\langle \text{aBaseType} \right\rangle \mid |

\text{partially implements Pattern} \left\langle \text{aBaseType} \right\rangle \text{ using}

\text{match}[\text{anActor:ExtensionInstance} \left\langle \text{aBaseType, anExtensionType} \right\rangle, \]

\text{e:Environment}: \text{Nullable} \left\langle \text{Environment} \right\rangle \rightarrow |

\text{aPattern, match}[\text{castExpression, eval[e], down[anActor]}, \ e] \]

\]

Actor

\[
\text{\{anExpression:Expression} \left\langle \text{aBaseType} \right\rangle
\]

\[
\text{"\text{\#1}" castExpression:Type} \left\langle \text{anExtensionType} \right\rangle
\]

\[
:\text{ExpressionDownQuery} \left\langle \text{anExtensionType, aBaseType} \right\rangle
\]

\text{uses BasicExpression} \left\langle \text{Boolean} \right\rangle \mid |

\text{partially implements Expression} \left\langle \text{Boolean} \right\rangle \text{ using}

\text{eval[e:Environment] : aType} \rightarrow |

\text{castExpression, eval[e], down? [anExpression, eval[e]]} \]

\]
Actor ("Actor" anExtensionType "extends" Type <aType> "1")

Actor implements Definition using
  eval[e:Environment]:Environment →
  e, bind[anExtensionType, type ⦅ RestrictionType <aType> ⦆]
  to ⦅ Actor uses BasicType [ ] ⦆
  partially implements Extension <aType> using
  up[anInstance: anExtensionType]: aType →
  ExtensionInstance <aType> [anInstance] ⦆
  down[anUpped: aType]: anExtensionType →
  anUpped ⦆
  anInstance, Else Throw CastException [ ] ⦆
  down?[anUpped: aType]: Boolean →
  anUpped ⦆
  anInstance, Else False[ ] ⦆

True, Else False[ ] ⦆

Structure ExtensionInstance <anExtension, aType> [anInstance: anExtension]
exends aType[ ] ⦆
Nullable, e.g. ☐

The type Nullable is used for nullables:

- **Implementation Nullable<aType> has**
  - `reduce?[]` → Boolean,
  - `reduce[]` → `aType`

**Actor** ("Nullable" anExpression: Expression<aType>)

: Nullable <aType>

uses BasicExpression<Nullable<aType>> [] |

partially implements Expression<Nullable<aType>> using

  eval[e:Environment]: Nullable<aType> ➞

  (anActor ← anExpression.eval[e]) •

  Actor implements Nullable<aType> using
  reduce?[]: Boolean → True

  reduce[]: aType → anActor

**Actor** (Null aType: Type<aType>)

: Nullable <aType>

uses BasicExpression<Nullable<aType>> [] |

partially implements Expression<Nullable<aType>> using

  eval[e:Environment]: Nullable<aType> ➞

  Actor implements Nullable<aType> using
  reduce?[]: Boolean → False

  reduce[]: aType → Throw IsNullException[]

**Actor** (TheNull): Nullable <aType>

implements Pattern<Nullable<aType>> using

  match[anActor: Nullable<aType>, e: Environment]

  : Nullable <Environment> ➞

  anActor

  TheNull $ Nullable e,

  else $ Null Environment

**Actor** ("⦾" anExpression: Expression<Nullable<aType>>)

: Nullable <aType>

uses BasicExpression<aType> [] |

partially implements Expression<aType> using

  eval[e:Environment]: aType ➞

  (anExpression.eval[e].reduce[])

67
Future, e.g., ☐, and ☎

The type Future is used for futures:

**Implementation Future**<aType> has
reduce[ ]→ aType

----

**Actor** ("Future" anExpression: Expression <aType>)

uses BasicExpression <Future><aType> []]
partially implements Expression <Future><aType> using
eval[e: Environment]: Future <aType> →
(aFuture ←
Future Try anExpression .eval[e]
catch
anException :
Actor
implements Future <aType> using
reduce[ ]: aType →
Throw anException[])

Actor implements Future <aType> using
reduce[ ]: aType → (aFuture $)

----

**Actor** ("⊙" anExpression: Expression <Future><aType>)

uses BasicExpression <aType> []]
partially implements Expression <aType> using
eval[e: Environment]: aType →
anExpression .eval[e].reduce[ ]$
The message match
Patterns are analogous to expressions, except that they have receive match messages:

```plaintext
Actor (" inout " aPattern: Pattern<Future<aType>>)

  :Pattern<aType>

  implements Pattern<Future<aType>> using

  match[anActor: Future<aType>, e: Environment]

    : Nullable<Environment> ➔

    aPattern, match[anActor, reduce[ ], e] ⨠

    TheNull ⨠ Nullable e,

    else ⨠ Null Environment [] § 1
```

```plaintext
Actor (" ⊝ " anExpression: Expression <aType>)

  :Mandatory <aType>

  uses BasicExpression<aType>[] |

  implements Expression<aType> using

  eval[e: Environment]: aType ➔

  ⊝ Future anExpression, eval[e] § 1
```

The message match
Patterns are analogous to expressions, except that they have receive match messages:

```plaintext
Interface Pattern<aType> with

    match[aType, Environment]: Nullable<Environment> []
```

```plaintext
Actor (anIdentifier: Identifier <aType>): Pattern<aType>

  implements Pattern<aType> using

  match[anActor: aType, e: Environment]: Nullable<Environment> ➔

    e.bind[anIdentifier, type ⬇ aType, to ⬇ anActor] § 1
```

```plaintext
Actor (" _ "): UniversalPattern<aType>

  implements Pattern<aType> using

  match[anActor: aType, e: Environment]: Nullable<Environment> ➔

    Nullable e § 1
```
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]).(V(expressions, 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 →
(aRecipient ← recipient, eval[e].
  aRecipient.SimpleMessage[QualifiedName[name, recipientType],
  [V(arguments, 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<\mathrm{aType}> ","
  second:Expression<\mathrm{aType}>":") :Expression<\mathrm{aType}^\circ> 
uses BasicExpression<\mathrm{aType}^\circ>[ ] | 
partially implements Expression<\mathrm{aType}^\circ> using 
  eval[e:Environment]:[\mathrm{aType}^\circ] \rightarrow 
  [first,eval[e],second,eval[e]]§I

Actor ("" first:Expression<\mathrm{aType}> ","
  "\Psi" rest:Expression<\mathrm{aType}>"]") :Expression<\mathrm{aType}^\circ> 
uses BasicExpression<\mathrm{aType}^\circ>[ ] | 
partially implements Expression<\mathrm{aType}^\circ> using 
  eval[e:Environment]:[\mathrm{aType}^\circ] \rightarrow 
  [first,eval[e],\Psi,rest,eval[e]]§I

Actor ("" first:Pattern<\mathrm{aType}> ","
  "\Psi" rest:Pattern<\mathrm{aType}^\circ> "]") :Pattern<\mathrm{aType}^\circ> 
implements Pattern<\mathrm{aType}^\circ> using 
  match[anActor:\mathrm{aType}^\circ,]
  e:Environment]:\texttt{Nullable}<\texttt{Environment}> \rightarrow 
anActor \n  [first,\Psi,rest] §I 
  first,match[first,e] 
  TheNull $\bowtie$ \texttt{Null Environment}, 
  $\bowtie\bowtie\bowtie\mathrel{\bowtie}$ @\texttt{aNewEnvironment} $\bowtie$ 
  rest,match[restValue,\texttt{aNewEnvironment}] \n  else $\bowtie$ \texttt{Null Environment}[\texttt{?}§I]

71
Exceptions

Actor ("Try" anExpression:Expression → aType)
"catch" exceptions:ExpressionCases → Exception → aType
:TryExpression → aType

uses BasicExpression → [] | 
partially implements Expression → aType using

eval[e:Environment]:aType →
Try anExpression.eval[e] catch
anException:Exception
CasesEval.[anException, exceptions, e]  исполнило

Actor ("Try" anExpression:Expression → aType)
"cleanup" aCleanup:Expression → aType
:TryExpression → aType

uses BasicExpression → [] | 
partially implements Expression → aType using

eval[e:Environment]:aType →
Try anExpression.eval[e]
catch
_ ⧆ (aCleanup.eval[e])
  Rethrow ⧆

Continuations using perform
A continuations is a generalization of expression for executing in cheese, which receives perform messages:

Interface Continuation→aType extends Construct with
perform[Environment, CheeseQ]→ aType

Actor Execute→aType
[aConstruct:Construct,
  e:Environment,
  c:CheeseQ]:aType →
aConstruct ❖ aContinuation Continuation → aType
  aContinuation.perform[e, c],
anExpression Expression → aType
  anExpression.eval[e]  исполнило
Atomic compare and update

Actor {"Atomic" location: Expression <Location <anotherType>>,
 "compare" comparison: Expression <anotherType>
 "update" update: Expression <anotherType> "\n "updated" "\n compareIdentical: ContinuationList <aType> "," 
 "notUpdated" "\n compareNotIdentical: ContinuationList <aType> )
 :Atomic <aType>}

implements Continuation <aType> using
 perform[e:Environment, c:CheeseQ]:aType →
 (location, eval[e])
 .compareAndConditionallyUpdate[comparison, eval[e],
 update, eval[e]] "\n True $ compareIdentical, perform[e, c].
 False $ compareNotIdentical, perform[e, c] \n
Actor SimpleLocation <anotherType> [initialContents]
 locals contents := initialContents |
 implements Location <anotherType> using
 compareAndConditionallyUpdate[comparison, update]:Boolean →
 (contents = comparison) "\n True $ True U contents := update,
 False $ False \n
Cases

**Actor** (anExpression:Expression<aType> "\bable"
   cases:ExpressionCases<aType> aType> "[\b"])
   :CasesExpression aType>

uses BasicExpression:aType> [ ] |
partially implements Expression aType> using
eval[e:Environment]:aType >
   CasesEval,[anExpression,eval[e], cases, e]§ |

**Actor** CasesEval
   [anActor:aType>]
cases:[ExpressionCase<aType> aType> "\b"]],
e:e:Environment]:aType >
cases |
   [ ] § Throw NoApplicableCase[],
   [first, \brest] §
   first "\b" (aPattern:Pattern<aType> "\b"
      anExpression:Expression aType>)
   :ExpressionCase aType> §
   aPattern,\bmatch[anActor, e] §
   TheNull §
      CasesEval,[anActor, rest, e],
      \be\bnewEnvironment §
      anExpression,eval[newEnvironment] §,
   ("\b else" \belsePattern:Pattern <anotherType> "\b"
      elseExpression:Expression aType>)
   :ExpressionElseCase aType> §
   elsePattern,\bmatch[anActor, e] §
   TheNull §
      Throw ElsePatternMustMatch[],
      \be\bnewEnvironment §
      elseExpression,eval[newEnvironment] §,
   ("\b else" "\b"
      elseExpression:Expression aType>)
   :ExpressionElseCase aType> §
   elseExpression,eval[e],
   else § Throw NoApplicableCase[] §
Actor \{ anExpression: Expression <> anotherType >"\r"
    cases: ContinuationCases <> anotherType, aType >"\rr"
\}

implements Continuation <> aType > using
    perform[c:Environment, c:CheeseQ]:aType ->
        CasesPerform <> [anExpression, eval[e], cases, e, c] §

Actor CasesPerform
    [anActor: anotherType, cases:[ContinuationCase <> aType ], c:CheeseQ]:aType ->
cases
    [ ] => Throw NoApplicableCase [ ], [first, \rest] =>
    first (aPattern: Pattern <> anotherType >"\r"
        aContinuation: Continuation <> aType >)
    : ContinuationCase <> aType >:
    aPattern.match[anActor, e] \r
    TheNull :
        CasesPerform[anActor, rest, e, c],
    @newEnvironment @
        aContinuation.perform[newEnvironment, c] \r.
    ("else"
        elsePattern: Pattern <> anotherType >"\r"
        elseContinuation: Continuation <> aType >)
    : ContinuationElseCase <> aType >:
    elsePattern.match[anActor, e] \r
    TheNull :
        Throw ElsePatternMustMatch [ ].
    @newEnvironment @
        elseContinuation.eval[newEnvironment] \r,
    ("else" "\r"
        elseContinuation: Continuation <> aType >)
    : ContinuationElseCase <> aType >:
    elseContinuation.perform[e, c],
else => Throw NoApplicableCase [ ] \r}
Holes in the cheese

```plaintext
Actor (anExpression:Expression:aType)
"↺" someAssignments:Assignments)
:Afterward:aType

implements Continuation:aType using
perform[e:Environment, c:CheeseQ]:aType →
(anActor ← anExpression.eval[e] ●
someAssignments.carryOut[e, c] ●
c.release[] ●
anActor)s1

Actor (aVariable:Variable:aType)
"≔" anExpression:Expression:aType):Assignment

implements Assignment using
carryOut[e:Environment]:Void →
e.assign[aVariable, to □ anExpression.eval[e]]s1

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

implements Continuation:aType using
perform[e:Environment, c:CheeseQ]:aType →
(frozenEnvironment ← e.freeze[] ●
// create frozen environment so that subsequent assignments
// subsequent assignments do not affect evaluating anExpression

  c.release[] ●
anExpression.eval[frozenEnvironment])s1
```
Actor ("" aPreparations: Preparations
anExpression: Expression <\text{aType}>"")
: CompoundExpression <\text{aType}>

implements Continuation <\text{aType}> using
perform [e: Environment, c: CheeseQ]: aType →
(frozenEnvironment ← e.freeze[[]] ●
// create frozen environment so that
// preparation does not affect evaluating anExpression
aPreparation.carryOut[e, c] ●
c.release[] ●
anExpression.eval[frozenEnvironment]) §

Actor ("" Hole" anExpression: Expression <\text{anotherType}>
"↺" anAfterward: AfterwardContinuation <\text{aType}> "↺")
:Hole <\text{aType}>

implements Continuation <\text{aType}> using
perform [e: Environment, c: CheeseQ]: aType →
(frozenEnvironment ← e.freeze[] ●
c.release[] ●
Try (anActor ← anExpression.eval[frozenEnvironment] ●
Holding c in anAfterward.perform[e, c] ●
anActor)
catch ũ (Holding c in anAfterward.perform[e, c] ●
Rethrow) ũ §
Actor ("Holding" resourceExpression:Expression⦅Resource⦆ "in"
   anExpression Expression⦅aType⦆ "["]
   :HoldingExpression⦅aType⦆)

uses BasicExpression⦅aType⦆ |
partially implements Expression⦅aType⦆ using
   eval[e:Environment]:aType →
      (resource ← resourceExpression.eval[e],
       resource.acquire[] ⦆
       Try (anActor ← anExpression.eval[e],
            resource.release[],
            anActor)
      catchœ
         œ retiredCases:ContinuationCases⦅anotherType, aType⦆ "["
      :Hole⦅anotherType, aType⦆
   return[œ]

⦅"Hole" anExpression:Expression⦅anotherType⦆
   "returnedœ"
      returnedCases:ContinuationCases⦅anotherType, aType⦆ "["
   "threwœ"
      threwCases:ContinuationCases⦅anotherType, aType⦆ "["
      :Hole⦅anotherType, aType⦆
   implements Continuation⦅aType⦆ using
      perform[e:Environment, c:CheeseQ]:aType →
      (frozenEnvironment ← e.freeze[] ⦆
      c.release[] ⦆
      Try (anActor ← anExpression.eval[frozenEnvironment] ⦆
           c.acquire[] ⦆
           CasesPerform⦅anActor, returnedCases, e, c⦆)
      cleanup
         ⦆
      CasesPerform⦅anException, threwCases, e, c⦆"
   :Hole⦆

Actor ("Enqueue" anExpression:QueueExpression "["):Enqueue
   implements Continuation using
      perform[e:Environment, c:CheeseQ]:Void →
      anExpression.eval[e].enqueueAndLeave[] ⦆
Simple Implementation of Actor

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

```
Actor ("Enqueue" anExpression:QueueExpression "●"
      aContinuation;Continuation<aType>;Enqueue<aType>)
  implements Continuation<aType> using
  perform[e:Environment, c:CheeseQ]:aType →
  (anInternalQ ← anExpression.eval[e],
   anInternalQ.enqueueAndLeave[●]
   aContinuation.perform[e, c]) §

Actor ("Actor" declarations:ActorDeclarations
  "implements" Identifier<aType>
  "using" handlers:Handlers<anInterface> "§":Definition
  implements Expression<anInterface> using
  eval[e:Environment]:aType →
  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:Type<Message> →
  // receivedMessage received for anInterface
  (c.acquire[●]
   aReturned ← Try Select.[receivedMessage, handlers, e, c]
   cleanup c.release[●])
  // release cheese and rethrow exception
  c.release[●]
  aReturned)§
```
Actor Select
[receivedMessage: Message,
  handlers: [Handler ◦],
  e: Environment,
  c: CheeseQ]: aType →
  handlers ᶝ
  [ ] ⊸ Throw MessageRejected[ ],
  [ [aMessageDeclaration: MessageDeclaration <aType>
       "." ReturnDeclaration <aType> "" →
       body: Continuation <aType>]
    ContinuationHandler <aType>],
  ⬅️ restHandlers] ᶝ
  aMessageDeclaration.match[receivedMessage, e] ᶝ
TheNull ᶝ
  Select, [receivedMessage, restHandlers, e, c],
  // process next handler
  ⏩ newEnvironment ᶝ
  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:

```plaintext
Actor CheeseQ[
  invariants aTail = Null Activity ⇔ previousToTail = Null Activity |
  locals aHeadHint := Null Activity,   // aHeadHint: Nullable<Activity>
  aTail := Null Activity|   // aTail: Nullable<Activity>
acquire ():Void nonexclusive in myActivity ➞
  myActivity▪[previous] = Null Activity ∧
  myActivity▪[nextHint] = Null Activity
precondition   // commentary for error checking
  Loop attempt,[]:Void is
    (myActivity▪[previous = aTail] ◦)   // set provisional tail of queue
    Atomic aTail compare aTail update myActivity ◊
      updated ⇧ // inserted myActivity in cheese queue with previous
      myActivity▪[previous] ◊
      TheNull: Void // successfully entered cheese
      else ⇧ Suspend [], // current activity is suspended
      notUpdated ⇧ attempt,[] [?] ¶  // make another attempt
release ():Void nonexclusive in myActivity ➞
  // release message received running myActivity
  aTail ≠ Null Activity   precondition   // commentary for error checking
  (ahead ← SubCheeseQ▪[head] ◦)
  ahead = myActivity
precondition   // commentary for error checking
  Atomic aTail compare ahead update Null Activity ◊
    updated ⇧ // last activity has left this cheese queue
    Void ◦ aHeadHint := Null Activity.
    notUpdated ⇧ // another activity is in this cheese queue
    MakeRunnable @ahead▪[nextHint]◆
    @aHeadHint := ahead▪[nextHint][B]
Internal SubCheeseQ using   // internal interface
  [head]: Activity nonexclusive ➞
  aTail ≠ Null Activity  precondition   // commentary for error checking
  Loop findHead, [backIterator▪Activity ←
    aHeadHint ◊
    TheNull: @aTail.
    ◦@anActivity ◦ anActivity [?] : Activity is
  backIterator▪[previous] ◊
    TheNull: // backIterator is head of this cheese queue
    (aHeadHint = Nullable backIterator ◦
     backIterator)
    ◦@previousBackIterator ⇧
    // backIterator is not the head of this cheese queue
    (previousBackIterator▪[nextHint = Nullable backIterator] ◦
     // set nextHint of previous to backIterator
    findHead▪[previousBackIterator]) [B]
```

81
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
- `acquire[]` \(\mapsto\) Void
- `release[]` \(\mapsto\) Void

The implementation CheeseQ uses activities to implement its queue where

**Implementation Activity** has
- `[previous]` \(\mapsto\) Nullable<Activity>
  
  // if null then head of queue else, pointer to backwards list to head
- `[previous = Nullable<Activity>]` \(\mapsto\) Nullable<Activity>
  
  // returns self so that updates can be chained
- `[nextHint]` \(\mapsto\) Nullable<Activity>
  
  // if non-null then pointer to next activity to get cheese after this one
- `[nextHint = Nullable<Activity>]` \(\mapsto\) Nullable<Activity>
  
  // returns self so that updates can be chained

Implementation type InternalQ is defined on the next page

where:

**Implementation InternalQ** has
- `enqueueAndLeave[ ]` \(\mapsto\) Void,
- `enqueueAndDequeue[InternalQ]` \(\mapsto\) Activity
- `dequeue[]` \(\mapsto\) Activity
- `empty?[ ]` \(\mapsto\) Boolean
Actor $\text{InternalQ}[\text{CheeseQ}]$

locals $a\text{Queue} \leftarrow \text{SimpleFIFO}\langle \text{Activity}\rangle[]$

$\text{enqueueAndLeave}[]:\text{Void} \rightarrow \text{myActivity}$

  // $\text{enqueueAndLeave}$ message received in $\text{myActivity}$
  $(a\text{Queue}\cdot\text{add}\text{[myActivity]}\bullet$

  $\text{c}\cdot\text{release}\text{[\]}$  // $\text{myActivity}$ is the head of $\text{aCheeseQ}$

  $\text{Suspend}\text{[\]}$  // $\text{myActivity}$ is suspended and when resumed returns $\text{Void}$

$\text{enqueueAndDequeue}[]:\text{anI}\langle\text{nternalQ}\rangle:\text{InternalQ} \rightarrow \text{Activity} \rightarrow$

  $\rightarrow \text{anInternalQ}\cdot\text{emptyfff}[\] \text{precondition}$

  $(a\text{Queue}\cdot\text{add}\text{[myActivity]}\bullet$

  $..\text{dequeue}\text{[\]}$  // commentary for error checking

  $\text{Suspend}\text{[\]}$  // $\text{myActivity}$ is the head of $\text{aCheeseQ}$

$\text{dequeue}[]:\text{Activity} \rightarrow \rightarrow ..\text{emptyfff}[\] \text{precondition}$  // commentary for error checking

  $(\text{c}\cdot\text{release}\text{[\]}\bullet$

  $\text{MakeRunnable} \rightarrow a\text{Queue}\cdot\text{remove}[]$  // make runnable the removed activity

$\text{emptyfff}[\]:\text{Boolean} \rightarrow a\text{Queue}\cdot\text{emptyfff}[\]$

where

Interface $\text{FIFO}\langle\text{aType}\rangle$ has

$\text{add}\text{[anActivity}:\text{aType}] \mapsto \text{Void},$

$\text{remove}\text{[anActivity}:\text{aType}] \mapsto \text{aType},$

$\text{emptyfff}[\] \mapsto \text{Boolean}$
### Appendix 3. ActorScript Symbols with IDE ASCII, and Unicode codes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>IDE ASCII</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></td>
<td>25AE</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>is a type</td>
<td>postfix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>☰</td>
<td>[:]</td>
<td>this Actor with interface (aspect)</td>
<td>prefix</td>
<td></td>
<td>2360</td>
</tr>
<tr>
<td>☳</td>
<td>\O86</td>
<td>reduce (nullables, futures)</td>
<td>prefix</td>
<td></td>
<td>29BE</td>
</tr>
<tr>
<td>☲</td>
<td>~\O87</td>
<td>match reduced (nullables, futures)</td>
<td>prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td>\v/</td>
<td>down</td>
<td>infix</td>
<td></td>
<td>2193</td>
</tr>
<tr>
<td>↓?</td>
<td>\v/?</td>
<td>down query</td>
<td>infix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>☳↓</td>
<td>~\v/</td>
<td>match downed</td>
<td>infix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td>^^</td>
<td>up</td>
<td>infix</td>
<td></td>
<td>2191</td>
</tr>
<tr>
<td>☳↑</td>
<td>~^</td>
<td>match upped</td>
<td>prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✰</td>
<td>( . )</td>
<td>qualified by</td>
<td>infix</td>
<td></td>
<td>22A1</td>
</tr>
<tr>
<td></td>
<td>\ /</td>
<td>procedure</td>
<td>prefix</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>==</td>
<td>defined as</td>
<td>infix</td>
<td></td>
<td>2261</td>
</tr>
<tr>
<td></td>
<td>. .</td>
<td>is sent</td>
<td>infix</td>
<td></td>
<td>2025</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>\p88</td>
<td>necessarily concurrent</td>
<td>prefix</td>
<td></td>
<td>29B7</td>
</tr>
<tr>
<td>→</td>
<td>l -&gt;</td>
<td>message type returns type</td>
<td>infix</td>
<td></td>
<td>21A6</td>
</tr>
<tr>
<td>→&gt;</td>
<td>l ..&gt;</td>
<td>cacheable message received</td>
<td>infix</td>
<td></td>
<td>2192</td>
</tr>
<tr>
<td>↔</td>
<td>- - &gt;</td>
<td>message received</td>
<td>infix</td>
<td></td>
<td>21A0</td>
</tr>
<tr>
<td>←</td>
<td>&lt; - -</td>
<td>be³⁰³</td>
<td>infix</td>
<td></td>
<td>2190</td>
</tr>
<tr>
<td>☳</td>
<td>??</td>
<td>cases</td>
<td>separator</td>
<td></td>
<td>FFFFD</td>
</tr>
<tr>
<td>☳</td>
<td>?? ??</td>
<td>end cases</td>
<td>terminator</td>
<td></td>
<td>2370</td>
</tr>
<tr>
<td>☳</td>
<td>\P92</td>
<td>another message handler separator for handlers</td>
<td>→</td>
<td></td>
<td>00B6</td>
</tr>
<tr>
<td>§</td>
<td>\s</td>
<td>end handlers terminator</td>
<td>implements and extension</td>
<td></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>binding, preparation and Enqueue</td>
<td>2BC3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>binding, preparation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>and Enqueue</td>
<td></td>
</tr>
</tbody>
</table>

¹ These are only examples. They can be redefined using keyboard macros according to personal preference.
\[ \text{:=} \quad \text{\(\text{=}\)} \quad \text{is assigned} \quad \text{infix} \quad 2254 \]

\[ \text{\(\cup\)} \quad \text{\(^{\text{U^\circ}}\)} \quad \text{afterward} \quad \text{infix} \quad 21BA \]

\[ \text{\(\cup\)} \quad \text{\(\setminus\)} \quad \text{matches value of} \quad \text{prefix} \quad 2315 \]

\[ \text{\(=\)} \quad \text{\(=\)} \quad \text{same as?} \quad \text{infix} \quad 2260 \]

\[ \text{\(\neq\)} \quad \text{\(\neq\)} \quad \text{Different from?} \quad \text{infix} \quad 2260 \]

\[ \text{\(\lbrack\)} \quad \text{\{\}} \quad \text{keyword or field} \quad \text{infix} \quad 2338 \]

\[ \text{\(\triangleright\)} \quad \text{\(<\)} \quad \text{begin type} \quad \text{left delimiter} \quad \text{prefix} \quad (\text{Unicode hex: 0076}) \quad 0076 \]

\[ \text{\(\triangledown\)} \quad \text{\(\\setminus\)} \quad \text{spread}^{95} \quad \text{prefix} \quad 2A5B \]

\[ \text{\{\} \quad \text{begin set} \quad \text{left delimiter} \quad \} \quad 2983 \]

\[ \text{\{\} \quad \text{begin multi-set} \quad \text{left delimiter} \quad \} \quad 2983 \]

\[ \text{\(\text{\lbrack\}} \quad \text{\(\lbrack\)} \quad \text{begin list} \quad \text{left delimiter} \quad \} \quad 2983 \]

\[ \text{\(\text{\lbrack\} \quad \text{\{\} \quad \text{formatted} \quad \text{left delimiter} \quad \} \quad 27E6 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\}} \quad \text{Left string structure} \quad \text{left delimiter} \quad \} \quad 201C \]

\[ \text{\(\text{\lbrack\} \quad \text{\lbrack\} \quad \text{begin grouping} \quad \text{left delimiter} \quad \} \quad 2983 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{begin syntax} \quad \text{left delimiter} \quad \} \quad 2983 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{zero or more} \quad \text{postfix} \quad 229B \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{uniformly of a type} \quad \text{infix} \quad 22EE \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{nothing}^{96} \quad \text{expression} \quad 229D \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{one-way send} \quad \text{infix} \quad 219E \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{join} \quad \text{infix} \quad 2294 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{constrained by} \quad \text{infix} \quad 2291 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{extends} \quad \text{infix} \quad 2292 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{logical implication} \quad \text{infix} \quad 21E8 \]

\[ \text{\(\text{\lbrack\} \quad \text{\text{\lbrack\} \quad \text{logical equivalence} \quad \text{infix} \quad 21D4 \]

\[ \text{\(\wedge\)} \quad \text{\(\wedge\)} \quad \text{logical conjunction} \quad \text{infix} \quad 00D9 \]

\[ \text{\(\vee\)} \quad \text{\(\vee\)} \quad \text{logical disjunction} \quad \text{infix} \quad 00DA \]

\[ \text{\(\neg\)} \quad \text{\(\neg\)} \quad \text{logical negation} \quad \text{prefix} \quad 00D8 \]

\[ \text{\(\text{\lbrack\)} \quad \text{\text{\lbrack\)} \quad \text{assert} \quad \text{prefix and infix} \quad 22A2 \]

\[ \text{\(\llbracket\)} \quad \text{\(\text{\lbrack\)} \quad \text{goal} \quad \text{prefix and infix} \quad 22A9 \]

\[ \text{//} \quad \text{\(\text{//\)} \quad \text{begin 1-line} \quad \text{prefix} \quad \text{comment} \quad \text{EndOfLine} \]

\[ /* \quad */ \quad \text{begin comment} \quad \text{prefix} \quad */ \]
### Appendix 4. ActorScript Reserved Words

#### Prefix

<table>
<thead>
<tr>
<th>Token</th>
<th>Separators</th>
<th>Terminator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Try</td>
<td>catch $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cleanup</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>extends with</td>
<td></td>
</tr>
<tr>
<td></td>
<td>restricts with</td>
<td></td>
</tr>
<tr>
<td>Discrimination</td>
<td>between</td>
<td></td>
</tr>
<tr>
<td>Actor</td>
<td>Structure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>invariants</td>
<td></td>
</tr>
<tr>
<td></td>
<td>uses</td>
<td></td>
</tr>
<tr>
<td></td>
<td>queues</td>
<td></td>
</tr>
<tr>
<td></td>
<td>implements using $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>internal using $</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>has</td>
<td></td>
</tr>
<tr>
<td>Holding</td>
<td>in</td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td>is</td>
<td></td>
</tr>
<tr>
<td>Hole</td>
<td>returned $itim $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>threw $itim $</td>
<td></td>
</tr>
<tr>
<td>Enqueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nullable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MakeRunnable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suspend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic</td>
<td>compare update updated notUpdated</td>
<td></td>
</tr>
</tbody>
</table>

#### Infix

<table>
<thead>
<tr>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>thats</td>
</tr>
<tr>
<td>postcondition</td>
</tr>
<tr>
<td>precondition</td>
</tr>
<tr>
<td>permit</td>
</tr>
</tbody>
</table>

#### Unary

<table>
<thead>
<tr>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
</tr>
<tr>
<td>False</td>
</tr>
<tr>
<td>TheNull</td>
</tr>
<tr>
<td>Void</td>
</tr>
</tbody>
</table>
Index

—, 21
\, 7, 75, 84
\ ... \?, 74
\, 85
[. 41, 48, 81, 85
[customer], 52
[history], 52
[message], 52
[response], 52
○, 12, 33, 84, 85
\, 40, 85
\, 85, See Expressions
*/, 85
//, 85
; 84
\:=, 85
\, 6, 10, 85
\, 69
\, 85
\, 11, 86
\••\••\••, 38, 40, 57, 84
\¬\, 61
", 46
", 46
++, 21
⊗, 50, 57, 85
⊙, 85
⊗, 36, 42, 43, 67, 68, 81, 84
=, 47, 49, 81, 85
≠, 55, 81, 85
||, 12, 18, 44, 51, 54, 56, 84
■, 6, 46, 70, 84
||, 18, 38, 83, 84
Ψ, 8, 9, 36, 40, 43, 46, 70, 85
expression, 71
pattern, 71
⇒, 11, 76, 81, 83, 85
\, 85
\, 33
\?, 59
\, 85
\, 53, 55, 85
\, 53, 55, 85
\, 46, 48, 49, 84
\, 37, 68, 81, 84
\, 64, 84
\, 15, 36, 64, 84
\, 34
\, 8
\, 54, 85
\, 39, 85
\, 52, 56, 81, 83, 84
\, 7, 84
\, 50, 85
\, 5, 84, 86, See Expressions
\, 34, 59, 60, 63, 64, 84
\, 11, 50, 84
\, 10, 84
\, 85
\, 15, 34, 60, 64, 84
\, 15, 34, 61, 65, 84
\, 6, 8, 45, 84, See definition
[$\Leftrightarrow$, 85]
[$\otimes$, 1, 11, 13, 18, 21, 22, 47, 76, 81, 85]
[$\|$., 11]
[$\|$, 84]
[$\odot$, 12, 18, 69, 84]
[$\ast$, 7, 84]

**Activity**, 82

**Actor**, 11, 13, 18, 21, 52, 79, 86

  - CheeseQ, 81
  - dequeue, 83
  - enqueueAndDequeue, 83
  - enqueueAndLeave, 83
  - InternalQ, 83
  - Swiss cheese, 16

Actor Model

  - Message passing, 2
  - types, 2

Agha, G., 23

ASCII, 84

Athas, W., 23

Atkinson, R., 23

**Atomic**, 47, 81, 86

**Atomic** ... **compare** ... **update** ...

  - updated ... **notUpdated** ..., 73

Attardi, G., 23

**backout**, 21, 22

Baker, H., 23

Barber, G., 23

Beard, P., 23

**become**, 44

**between**, 86

Bishop, P., 23

Boden, N., 23

Briot, J., 23

Cartesian, 38

cases, 7

cast

  - downcast, 16
  - self to interface of this Actor, 16
  - upcast, 16

catch, 37, 86

cheese, 20

  - dequeue, 82
  - enqueueAndDequeue, 82
  - enqueueAndLeave, 82

CheeseQ, 79, 81, 82

  - release, 56, 81, 82, 83
  - SubCheeseQ, 81
  - take, 81, 82

cleanup, 37, 86

Clinger, W., 23

**compare**, 86

Complex, 38, 39

**Construct**, 56, 72

**Continuation**, 72

**Customer**, 52

Dahl, O., 1

Dally, W., 23

de Jong, P., 23

Decrypt, 34

**Decrypted**, 41

Dedecker, J., 23

**default**, 38, 39

Define, 6, 9

definition

  - identifier, 6

**Discrimination**, 34, 62, 86

either, 45

Encrypt, 41

**Encrypted**, 34

Encryption, 41

**Enqueue**, 21, 22, 78, 86

**Enumeration**, 48

eval, 56

exception, 37

Expressions, 5

**extends**, 66

**extension?**, 57

**ExtensionType**, 59, 64

False, 86

**Fork**, 15

FriAM, 23

Fringe, 15
Seitz, C., 23
sendOneWay, 57
sendRequest, 57, 63
Simi, M., 23
Smith, S., 23
Steiger, R., 23
Structure, 14, 15, 38, 39
Suspend, 81, 83, 86
Swiss cheese, 16
Symbols, 84
Talcott, C., 23
Terminal, 35
Thati, P., 23
thatIs, 7, 86
TheNull, 37, 67, 86
Theriault, D., 23
This (JavaScript), 48
threw, 86
Throw, 52
throw, 57
Throw, 11, 37
Tokoro, M., 23
Tree, 14, 15
Trie, 35
TrieFork, 35
True, 86
Try, 37, 86
Try ... catch, 72
Try ... cleanup, 72
type, 48
paramaterized, 33
Type, 57
CommunicationType, 57
types, 5
Unicode, 84
update, 86
updated, 86
uses, 13, 52, 86
using, 86
UsingNamespace, 48
Varela, C., 23
variable
Actor, 20
ActorScript, 10
variables, 10, 20
Void, 11, 86
When, 53, 54, 55
with, 86
Woelk, D., 23
XML, 49
Yonezawa, A., 23
\( \lambda \), 37, 42, 84
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,
   
   $\{x: \text{Numerical} \ "+" \ y: \text{Numerical}\}; \text{Numerical}$
   $\text{Expression}$

   Also,
   
   $\{\text{Numerical} \ "-" \ \text{Numerical}\}; \text{Numerical}$
   $\text{Expression}$

   $\{\text{Numerical} \ "*" \ \text{Numerical}\}; \text{Numerical}$
   $\text{Expression}$

   $\{\text{Numerical} \ "/" \ \text{Numerical}\}; \text{Numerical}$
   $\text{Expression}$

   $\{\text{"Remainder"} \ \text{Numerical} \ "/" \ \text{Numerical}\}; \text{remainder}: \text{Numerical}$
   $\text{Expression}$

   $\{\text{"QuotientRemainder"} \ \text{Numerical} \ "/" \ \text{Numerical}\}$
   $\{\text{Numerical}, \ \text{Numerical}\}$
   $\text{Expression}$

4. 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.

   $\text{Identifier} \in \text{Word} \in \text{Expression}$
   // an Identifier is a Word, which is a subcategory of Expression
   $\{([\text{Expression} \cup \text{Definition} \cup \text{Judgment}]) \ "\}; \text{Top}$

5. $\{\text{Identifier} \ "\}; \text{Declaration}$
   // Identifier is declared to be of Type
   $\{\text{Identifier} \ "\}; \text{Declaration}$
   // Identifier is declared to be a type
   $\{\text{Type}\Rightarrow \text{Type}\}; \text{Signature}$
   $\{\text{Type}\Rightarrow \ "\}; \text{Signature}$
The symbol ∎ is fancy typography for an ordinary period when it is used to denote message sending.

The symbol ⊑ is used to separate declarations.

The symbol ◊ solves the infamous "dangling else" problem [Abrahams 1966].
⦅ExpressionElseCase ⊔ (ExpressionElseCase
  "," MoreExpressionElseCases))):ExpressionElseCases⦆
⦅ExpressionElseCase
  ⊔ (ExpressionElseCase
    "," MoreExpressionElseCases)):MoreExpressionElseCases⦆
⦅ExpressionElseCase
  ⊔ (else "i" Preparations)
⦅("else" Pattern:"i" Preparations)):ExpressionElseCase⦆
⦅("else" Pattern:"i" Preparations)):ExpressionElseCase⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
⦆⦆:ExpressionElseCases⦆
All Equivalent to the following:

\[
\begin{align*}
\text{Define } & \quad \text{Reverse} \langle \text{aType} \rangle \cdot \text{[aList: [aType]}]: \text{[aType]} = \\
& \quad \text{[first, rest]} \cdot \text{[rest, first]} \\
\end{align*}
\]

Dijkstra[1968] famously blamed the use of the goto as a cause and symptom of poorly structured 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 has 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.

\[
\begin{align*}
\text{"Actor" ConstructorDeclaration ActorBody}: \text{Expression} = \\
& \quad \text{// The above expression creates an Actor with} \\
& \quad \text{// declarations for variables and message handlers} \\
& \quad \text{( [ "uses" ConstructorList ] )} \\
& \quad \text{( [ "management" Expression ] )} \\
& \quad \text{MessageHandlers} \\
& \quad \text{InterfaceImplementations}: \text{ActorBody} \\
& \quad \text{(Identifier "<" ParametersDeclarations "$"}) \\
& \quad \text{( [ "ArgumentDeclarations"] )}: \text{ConstructorDeclaration} \\
\end{align*}
\]
{(Constructor”,” MoreConstructors”|”}):ConstructurList
{(Constructor
   \{ (Constructor”,” MoreConstructors }}):MoreConstructors
{( \{ ”locals” LocalsDeclarations ”|”}:LocalsDeclaration
{(QueueDeclarations LocalsDeclaration}:Declaration

{(LocalDeclaration
   ”|”, MoreLocalDeclarations}}):MoreLocalDeclarations
{(Identifier”→” Expression}):IdentifierDeclaration
IdentifierDeclaration\{LocalDeclaration
{(Variable”=” Expression InstanceVariableAQualifications}
:VariableDeclaration
VariableDeclaration\{LocalDeclaration
Variable\{Word
InstanceVariableQualifications \{ InstanceQualifications
{( InstanceVariableQualification
   \{ ( InstanceVariableQualification
      InstanceVariableQualifications
   \}
   ” nonpersistent” \{ InstanceVariableQualification
   // A nonpersistent variable must be Nullable,
   // and can be nulled out before a message is received
(" queues” QueueName):QueueDeclarations
QueueName \{Word
QueueName \{Expression
(" Void”):Expression
{(InterfaceImplementation
   \{ ( MoreInterfaceImplementations
   \}
   :InterfaceImplementations
{(" also” InterfaceImplementation
   \{ ( MoreInterfaceImplementations
   \}
   :MoreInterfaceImplementations
{( (" partially")
   (”implements” \{ "reimplements”
      ( \{ ”exportable”} Type “using”
         (MessageHandlers’|”)\{ UniversalMessageHandler
   \}:InterfaceImplementation
{(MessagePattern “|” Type ( \{ ”sponsor” Identifier
   ”→” ExpressionsContinuation}:UniversalMessageHandler
{( MoreMessageHandlers}:MessageHandlers}

95
(MessageHandler
  \{MessageHandler "§" MoreMessageHandlers\})
  ;MoreMessageHandlers

  // The message handler separator is ¶.
  (MessageName "[
  \"ArgumentDeclarations \"] \:" Type
  (\{\"sponsor\" Identifier\})
  "\" ExpressionsContinuation\}:MessageHandler

  // For a message with MessageName with arguments,
  // the response is ExpressionsContinuation
  (Expression "\" Afterward\}:Continuation

  // Return Expression and afterward perform
  // MoreVariableAssignments
  VariableAssignments\{Afterward

  (VariableAssignment
   ",," MoreVariableAssignments\}:VariableAssignments

  (VariableAssignment
   \{VariableAssignment
    ",," MoreVariableAssignments\})
    ;MoreVariableAssignments

  \{Variable "=" Expression\}:VariableAssignment

  22 (\" MoreAntecedents Continuation \")\}:CompoundContinuation

  (Antecedent \{MoreAntecedents

  (Antecedent \"\" \" MoreAntecedents\}:MoreAntecedents

  (Binding ",," MoreAntecedents\}:MoreAntecedents

  Expression\{Antecedent

  StructureAssignment \{Antecedent

  ArrayAssignment \{Antecedent


96
For example, consider the following:

```
Actor NeedTwo[
  queues waiting]
  locals hasOne := False
  go[]:Void → hasOne ◦ True \ Void permit waiting.
  False \ (hasOne := True\)
          \ enqueue waiting\)
          \ Void)♀§
```

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

```
(aNeedTwo ← NeedTwo,]
  ℗aNeedTwo, go[)[\)
  aNeedTwo, go[])♀
```

However following expression might never return because of optional concurrency:

```
(aNeedTwo ← NeedTwo,]
  aNeedTwo, go[)[\)
  aNeedTwo, go[)♀
```

23 For example, consider the following:

```
Actor NeedTwo[]
  queues waiting]
  locals hasOne := False]
  go[]:Void → hasOne ◦ True \ Void permit waiting.
  False \ (hasOne := True\)
          \ enqueue waiting\)
          \ Void)♀§
```

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

```
(aNeedTwo ← NeedTwo,[]],
  ℗aNeedTwo, go[)♀
  aNeedTwo, go[]♀
```

However following expression might never return because of optional concurrency:

```
(aNeedTwo ← NeedTwo,[]],
  aNeedTwo, go[]♀
  aNeedTwo, go[]♀
```

24 \(\bigcirc\) anExpression:Expression( U {"sponsor" Expression})

```
// Execute anExpression in parallel and respond with the outcome.
// In every case, anExpression must complete before execution leaves
// the lexical scope in which it appears.
```


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

27 note the absence of "." in the implementation subexpression

28 equivalent to the following:

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

29 ignoring exceptions in this way is not a good practice
Enqueue QueueExpression "●" Continuation; Continuation

(*
1. Enqueue activity in QueueExpression
2. Leave the cheese
3. When the cheese is re-entered perform Continuation.
*)

("(" Antecedents
    "enqueue" QueueExpression "●" Continuation")")
    ; Continuation

(*
1. Perform the Antecedents
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 "?["): Continuation
    (ContinuationCase
        | [{ContinuationCase ""," MoreContinuationCases"}]
        ContinuationElseCases: ContinuationCases
        (ContinuationCase
            | [{ContinuationCase ""," MoreContinuationCases"}]
            : MoreContinuationCases
    )
    {Pattern "s" ExpressionsContinuation()): ContinuationCase
    (| MoreContinuationElseCases): ContinuationElseCases
    (ContinuationElseCase
        | [{ContinuationElseCase ""," MoreContinuationElseCases"}]
        : MoreContinuationElseCases
    )
    ("else" "s" ExpressionsContinuation)
    (| ("else" Pattern "s" ExpressionsContinuation))
    (Continuation): ExpressionsContinuation
    (preparation ("," "●") MoreExpressionsContinuation))
    : MoreExpressionsContinuation
    (Continuation)
    (| {Expression "", MoreExpressionsContinuation})
    : MoreExpressionsContinuation

Equivalent to the following:

Define Fringe[aTree:Tree]: [String ⨿]
aTree
    Leaf[aString] $ [aString],
    Fork[left, right] $ [∀Fringe.[left], ∀Fringe.[right]] $

Equivalent to the following:

Fringe.[Fork[Leaf["The"]↑Tree&Leaf["boy"]↑Tree]↑Tree]
Swiss cheese was called "serializers" in the literature.

Delegation:

```
⦅ "Message": Expression ⦆
  // Delegate message to this Actor.
  ("Antecedents "hole" Expression\"\") : Continuation ⦆*
```

1. Carry out Antecedents
2. Leave the cheese
3. The result is the result of evaluating Expression */

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:

```
ReadingPriority[ReadersWriterConstraintMonitor[\( r \)].
```

**Actor** ReadersWriterConstraintMonitor[theResource:ReadersWriter]

locals

```plaintext
writing \(:=\) False,
numberReading \(:=\) 0 |
```

implements ReadersWriter using

```plaintext
read[aQuery:Query]: QueryAnswer

→writing precondition // commentary for error checking
  (numberReading++ ;)
  hole theResource.read[aQuery]
  ↝ numberReading--)
write[anUpdate:Update]: Void →

numberReading\(=0\) ←writing precondition
  (writing \(:=\) True ;)
  hole theResource.write[anUpdate]
  ↝ writing \(:=\) False)
```

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.
100

38 ("Antecedents
    "enqueue" QueueExpression( U "backout" Preparations)
    Continuation"));Continuation

I
1. Perform Antecedents
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 Preparations, and release the cheese.
5. If no exception is generated by the activity while in the queue, then when allowed to continue, re-acquire the cheese to perform Continuation. */

Cases can be continuations:
(test:Expression(" ContinuationCases" []));Continuation
(ContinuationCase U MoreContinuationCases);ContinuationCases
(ContinuationCase
   (ContinuationCase "," MoreContinuationCases)
   U ContinuationElseCases);MoreContinuationCases
( U ContinuationElseCase
   (ContinuationElseCase "," MoreContinuationElseCases))
    ;ContinuationElseCases
(ContinuationElseCase
   U {ContinuationElseCase "," MoreContinuationElseCases})
     ;MoreContinuationElseCases
(("else" "," ContinuationList)
   U {("else" Pattern:" ExpressionsContinuation))
    ;ContinuationElseCase
    // The else case is executed only if the patterns before the else case do not match the value of test.
    {Pattern "," ExpressionsContinuation};ContinuationCase)

The following are allowed in the cheese for a response to message affecting the next message:
(Expression
   ( U ("permit" aQueue:Expression)))
   ( U ("↺" Afterward)));Continuation
   /* If there are activities in aQueue, then the one of them gets the cheese next and also perform Afterward, then release the cheese and return the value of Expression. */
VariableAssignments:Afterward
("Permit" aQueue:Expression
   ( U ("also" VariableAssignments)));PermitAlso

100
The following can be used temporarily release the cheese:

```plaintext
(Hole Expression):Continuation
/*
  1. Leave the cheese
  2. The response is the result of evaluating Expression */

(C Antecedents
  hole Expression (  U ("↺ Afterward "))):Continuation
/*
  1. Carry out Antecedents
  2. Leave the cheese
  3. Evaluate Expression

4. When a response is received, reacquire the cheese, carry out
   Afterward and the result is the result of evaluating Expression */

(C Antecedents
  hole Expression
    (  U ("returned assume ContinuationCases" "" ))
    (  U ("threw assume ContinuationCases" "" )) ):Continuation
/*
  1. Carry out Preparation
  2. Leave the cheese
  3. Evaluate Expression
  4. When a response is received, reacquire the cheese
     • If Expression returns, continue using the returned
       Actor with normal.
     • If Expression throws an exception, continue using the
       exception with exceptional. */
```

---

39 -- is postfix decrement

40 Joe Armstrong interviews Alan Kay Erlang Conference on YouTube.
   November 22, 2016.

41 (Identifier "<" ParametersDeclarations ">
   Preparation )
   :ParameterizedDefinition

   ParameterizedDefinition:Definition
   // Parameterize definition with ParametersDeclarations
   (  U MoreParameterDeclarations ):ParametersDeclarations

   ParameterDeclaration
   (  "" MoreParameterDeclarations )
   :MoreParameterDeclarations

   (Identifier (  U Qualifier )):ParameterDeclaration

   ( Identifier ("extends" Type )):TypeQualifier

   (Identifier "<" Parameters ">" ):TypeExpression

   (Identifier (  U (Identifier," Parameters )):Parameters

   ("Discrimination" Identifier MoreTypeDiscriminations "1")
   :Definition

42
\begin{verbatim}
(Identifier U (Identifier","MoreTypeDiscriminations))
   :MoreTypeDiscriminations

(Expr Type):Expression
   // Discriminate to be of Type if possible.
   // Otherwise, an exception is thrown.
(Expr Type):Expression
   // If Expr discriminates to be of Type,
   // then True, else False.
(Pattern TypePattern):Pattern
   // If matching Actor is a discrimination that can be discriminated
   // then Pattern must match the discriminate.
("\text{null}" StructurePattern):Pattern
   // Matching Actor must be discrimination that
   // can downed as StructurePattern which matches

43 Equivalent to the following:
(\x \leftarrow 3,
   TrieFork<\text{Integer}>(Terminal<\text{Integer}>[\x]Trie<\text{Integer}>,
   Terminal<\text{Integer}>[\x+1]Trie<\text{Integer}>))

44 (Identifier[" Arguments"]) :Expression
   (Identifier[" Patterns"]) :Pattern

45 ("\text{null}" Expression):Expression
   // reduce Expression if not null.
   // Otherwise, an exception is thrown.
("\text{null}" Pattern):Pattern
   // If matching Actor is a non-null nullable
   // then Pattern must match the Actor in the nullable.
("TheNull") :Pattern
   // matches only the null

46 ("Try" anExpression:Expression"catch" ExpressionCases )
   :Expression

/*
 * If anExpression throws an exception that matches the pattern
 * of a case, then the value of TryExpression is the value
 * computed by ExpressionCases
 * If anExpression doesn’t throw an exception, then the value of
 * TryExpression is the value computed by anExpression. */
\end{verbatim}
"Try" anExpression: Expression "catch" ContinuationCases "[?]": Continuation

/*
  • If anExpression throws an exception that matches the pattern of
    a case, then the response of TryContinuation is the
    response computed by the expression of the case.
  • If anExpression doesn’t throw an exception, then the response
    of TryExpression is the response computed by anExpression.
*/

"Try" anExpression: Expression "cleanup" cleanup: Expression

/*
  • 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.
*/

{ test: Expression "precondition" Preparations } : Expression

// test must evaluate to True or an exception is thrown
{ test: Expression "precondition" ExpressionsContinuation } :

// test must evaluate to True or an exception is thrown
{ value: Expression "postcondition" pre: Expression } :
  // The expression pre must evaluate to True when sent value
  // or an exception is thrown

"o" is a reserved postfix operator for degrees of angle
Using parameterized procedures like the ones below can improve the
simplicity and effectiveness of types by comparison with other approaches
Equivalent to the following:
Define Times [u: Complex, v: Complex] : Complex →
  Cartesian [u[real] * v[real] - u[imaginary] * v[imaginary],
             u[imaginary] * v[real] + u[real] * v[imaginary]] ↑ Complex

Equivalent to the following:
Define Times [Polar [angle: anAngle, magnitude: aMagnitude],
  Polar [angle: anotherAngle, magnitude: anotherMagnitude]] : Complex →
  Polar [angle: anAngle + anotherAngle, magnitude: aMagnitude * anotherMagnitude] ↑ Complex
Structure Identifier [ FieldDeclarations ]
  { U ( "uses" ConstructorList "|" ) }
NamedDeclaration
MessageHandlers
  MoreInterfaceImplementations):Definition
  // Structure definition with StructureImplementation
  { anExpression:Expression "↓" Type }):Expression
  { anExpression:Expression "↑" Type }):Expression
  // If anExpression is an extension of Type, then True else False
  { aPattern:Pattern "↓" Type }):Pattern
  // Matching Actor must be an extension of Type which
  // matches aPattern
  { U MoreFieldDeclarations):FieldDeclarations
  (SimpleFieldDeclaration
  (U ("," MoreNamedFieldDeclarations)))
  U { SimpleFieldDeclaration
    "," MoreFieldDeclarations }):MoreFieldDeclarations
  (Identifier { U "default" Expression }):SimpleFieldDeclaration
  (NamedFieldDeclaration
    U ( NamedFieldDeclaration
      "," MoreNamedFieldDeclarations))
  :MoreNamedFieldDeclarations
  { SimpleName
    (""," SimpleFieldDeclaration))
  :NamedFieldDeclaration
FieldName ⊑ QualifiedName
  // ": qualifiedName is used for assignable fields.
  (U Identifier) ActorBody):StructureImplementation
  { Expression "(" FieldName ") " }):FieldSelector
  // FieldName of Expression which must be a structure
FieldSelector ⊑ Expression
  (StructureName "[" FieldExpressions "] "):StructureExpression
  StructureExpression ⊑ Expression
  { U MoreFieldExpressions):FieldExpressions
  (SimpleFieldExpression
    "," MoreFieldExpressions }):MoreFieldExpressions
  (NamedFieldExpression
    U ( NamedFieldExpression
      "," MoreNamedFieldExpressions))
  :MoreNamedFieldExpressions
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 "|$->$"

It is possible to define a procedure that will produce a "bottomless" future. For example,

```plaintext
Actor f,[]:Future<BottomLessFuture> -> Future f,
Define BottomLessFuture ([] => BottomLessFuture)
```
// A future for aValue.
("Future" aValue:Expression(⊔("sponsor" Expression)))

// Reduce a future

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

Define IntegersBeginningWith[n:Integer][Integer]
[n, ∀IntegersBeginningWith,[n+1]]

Note: A Postpone expression can limit performance by preventing concurrency

The implementation below requires careful optimization.

The implementation below can be highly inefficient.
Atomic aLocation: Expression
  "compare" comparison: Expression
  "update" update: Expression "♀"
  "updated" "♂"
    compareIdentical: ExpressionsContinuation "♂"
  "notUpdated" "♂"
    compareNotIdentical: ExpressionsContinuation "♀")
  :Continuation

/* 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. */

Identifier ""Qualifier": QualifiedName
QualifiedName □ Expression □ Identifier □ QualifiedName □

(Identifier ⊓ (Identifier ""Qualifier)): QualifiedName □

("Enumeration" Identifier
  MoreEnumerationNames | ""); Definition □
(EnumerationName
  "", MoreEnumerationNames); MoreEnumerationNames □

EnumerationName □ Word □

equivalently (HTTPS[ "en.wikipedia.org" ][ ])

Declarations provide version number, encoding, schemas, etc.

(recipient: Expression
  "♂" MessageName "" Arguments "")]; Expression □

/* recipient is sent one-way message with MessageName and Arguments. Note that Expression cannot be used to produce a value. */

(MessageName "" ArgumentDeclarations "") "♂" "♀"
  ( □ ("sponsor" Identifier))
  "→" ExpressionsContinuation]; MessageHandler □

/* one-way message handler implementation with ArgumentDeclarations that has a one-way continuation that returns nothing */

("♂") ( □ ("permit" aQueue: Expression))
  ( □ ("♂" Afterward))); Continuation]

note the absence of "♀" in the implementation subexpressions.
Male[aMagnitude] is invoked concurrently with Human[aLength].

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 Filar 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:

```
⦅ "if" test:Expression then:Expression else:Expression "}"
```

which returns the value of then if test evaluates to True and otherwise returns the value of else.

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

```lisp
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 Rest of expression else
        (Eval (First (Rest (Rest (Rest expression)) environment))
         // return Eval of First of Rest of Rest of Rest of expression
        ...
    )
```

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

For example, the message could be of type

```
Message<<DepositOnlyAccount, deposit[Euro] → Void>
```

where

```lisp
Interface DepositOnlyAccount restricts Account with deposit[Euro] → Void
```

the device may have no access to anAccount or Account

the device may have no access to anAccount, x, etC, or Account

If non-null points to head with current holder of cheese

If non-null, pointer to backwards list ending with head that holds cheese

/* acquire message received running myActivity

/* this cheese queue is not empty because myActivity is at the head of the queue */

Not to be confused with \0 which is the null character or with \0 which is $\emptyset$. 

108
Not to be confused with \0 which is the null character or with \0 which is 0.

Not to be confused with \p which is ¶.

Used in type specifications for interfaces.

Used in message handlers.

Used to bind identifiers.

Not to be confused with \P which is ℘.

Not to be confused with \0 which is the null character or with \0 which is 0.

Used in patterns.

Used in structures.

Used in one-way message passing.