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

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[ii] 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
  - Programs are *extension invariant*, *i.e.*, extending a program does not change the meaning of the program that is extended.
  - Applications cannot directly harm each other.
  - Variable races are eliminated while allowing flexible concurrency.
  - Lexical singleness of purpose. Each syntactic token is used for exactly one purpose.
- Performance[i]
  - 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.
  - Execution dynamically adjusted for system load and capacity (*e.g.* cores)
  - Locality because execution is not bound by a sequential global memory model
  - Inherent concurrency because execution is not limited by being restricted to communicating *sequential* processes
  - 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[ii] enable secure communication with Actors

Computer software should not only work; it should also appear to work. [1]

---

[i] Performance can be tricky as illustrated by the following:

- "Those who would forever give up correctness for a little temporary performance deserve neither correctness nor performance." [Philips 2013]
- "The key to performance is elegance, not battalions of special cases" [Jon Bentley and Doug McIlroy]
- "If you want to achieve performance, start with comprehensible." [Philips 2013]
- Those who would forever give up performance for a feature that slows everything down deserve neither the feature nor performance.

[ii] Each type is an Actor. However, it may be the case that a type will work some places and not others. For example, to be used in message passing, the type of an address may require access to particular hardware.

Introduction

ActorScript is based on the Actor mathematical model of computation that treats "*Actors*" as the universal conceptual primitive of digital computation [Hewitt, Bishop, and Steiger 1973; Hewitt 1977; Hewitt 2010a]. Actors have been used as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems.

ActorScript

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

This paper makes use of the following typographical conventions that arise from underlying namespaces for types, messages, language constructs, syntax categories, *etc.*[i]

- 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[2] for language constructs (*e.g.,* **Actor**)
- logical variables (e.g., *x*)
- comments in programs (e.g. /* this is a comment */)

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

Actors

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

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

[i] 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.*

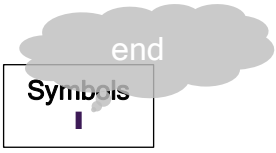
The Actor Model is designed to provide a foundation for inconsistency robust information coordination.

Notation

To ease interoperability, ActorScript uses an intersection of the orthographic conventions of Java, JavaScript, and C++ for words[i] 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:[ii]



Expressions evaluate to Actors. For example, 1+3❙[iii] is equivalent[iv] to 4❙.

Parentheses "(" and ")" can be used for precedence. For example using the usual precedence for operators, 3∗(4+2)❙ is equivalent to 18❙, while 3∗4+2❙ is equivalent to 14❙,

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

Types

Types are Actors. Type names are shown as follows:

- blue for types in general (*e.g.,* Account)
- green for the special case of implementation types (*e.g.,* SimpleAccount)

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

---

[i] sometimes called "names"

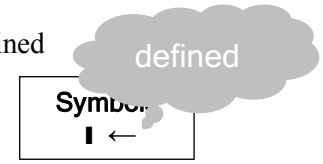
[ii] Furthermore, all special symbols have ASCII equivalents for input with a keyboard. An IDE can convert ASCII for a symbol equivalent into the symbol. See table in an appendix to this article.

[iii] 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.

[iv] in the sense of having the same value and the same effects

Identifier Definitions, *i.e.*, ←

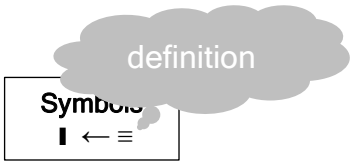
An identifier definition has an identifier to be defined followed by "←" followed by the definition. For example, x←3▮ defines the identifier x to be the Actor 3.



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

Procedure Definitions, i.e., **→**

A procedure is an Actor that can receive a list of Actors in a message and return an Actor as its value, which can be defined using "**Define**", followed by a procedure name, a list of formal arguments, return type, "→" and body of the procedure.[7] For example the procure can be defined as follows:[i]



Define Double▪[v:Integer]:Integer ≡ 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 "▪"[9] followed by a message with arguments delimited by "[" and "]". For example, Double▪[2+1]▮ means send Double the message [3]. Thus Double▪[2+1]▮ is equivalent to 6▮.

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[ii]
- ρx is a pattern that matches the value of x.
- ρ(x+2) is a pattern that matches the value of the expression x+2.

---

[i] Anonymous procedures are also allowed as in the following:

λ [v:Integer]:Integer → v+v▮

[ii] e.g., _ matches 7

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

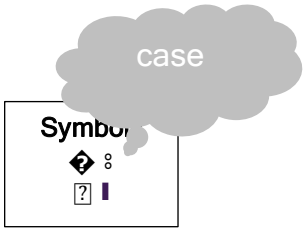
- x is a pattern that matches "abc" and binds x to "abc"

Cases, *i.e.,* **? : , : ?**

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 "?"[ii] and then followed by a number of cases separated by "," terminated by "?".[11] 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:




```
Random.[ ] ?
    0 :        // Random.[ ] returned 0[iv]
      Throw[v] 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.**

---

[i] An identifier is a name that is used in a program to designate an Actor

[ii] "?" is fancy typography for "**?**"

[iii] *including* patterns in previous else cases

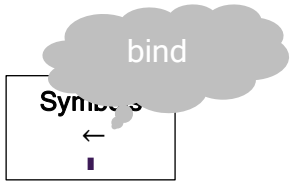
[iv] As is standard, ActorScript uses the token "//" to begin a one-line comment.

[v] Reserved words are shown in bold black.

Binding identifiers, *i.e.*, ←

Identifiers can be bound using an ideentifier, followed by "←" and an expression. For example, aProcedure.["G", "F", "F"]! is equivalent to the following:

```
(x ← "F",                    // x is "F"
 aProcedure.["G", x, x] )!
```



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

```
(x ← "F",                                  // x is "F"
 y ← aProcedure.["G", x, x]],              // y is aProcedure.["G", "F", "F"]
  anotherProcedure.[x, y] )!
```

The above is equivalent to
anotherProcedure.["F", aProcedure.["G", "F", "F"]]!

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]] is equivalent to [[1, 2], 3, 4]
- If y is [5, 6], then [1, 2, y, ⩛y]! is equivalent [1, 2, [5, 6], 5, 6]!
- [⩛[2, 3.0]]::[Integer, Float] is equivalent to
  [2, 3.0]::[Integer, Float].[i]

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

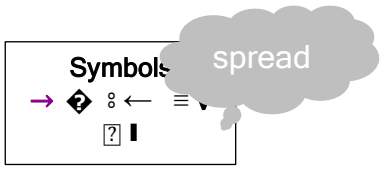
[i] ::[Integer, Float] is the type of a two element list, the first of which is of type Integer and the second of type Float

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

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

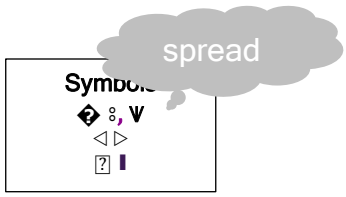
Below is the definition of a procedure that computes the reverse of a list.

Define Reverse◁aType▷**.** **[**aList:[aType⊛]**]**:[aType⊛] ≡
  aList **⦁**
    [ ] ⁝ **[ ]**,
    [first, **⩛**rest**]** ⁝ **[⩛**rest, first**]** ▯**❙**[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:



Define AlternateElements◁aType▷**.** **[**aList:[aType⊛]**]**:[aType⊛] ≡
  aList **⦁**
    [ ] ⁝ **[ ]**,
    [anElement**]** ⁝ **[**anElement**]**,
    [firstElement, secondElement**]** ⁝ **[**firstElement**]**,
    else ⁝
      [firstElement, secondElement, **⩛**remainingElements**]** ⁝
        [firstElement, **⩛**AlternateElements**.[**remainingElements**]]** ▯**❙**

Consequently,

- AlternateElements◁Integer▷**.[[ ]]❙** is equivalent to **[ ]⁝Integer❙**
- AlternateElements◁Integer▷**.[**[3]**]❙** is equivalent to **[**3**]⁝Integer❙**
- AlternateElements◁Integer▷**.[**[3, 4]**]❙** is equivalent to **[**3**]⁝Integer❙**
- AlternateElements◁Integer▷**.[**[3, 4, 5]**]❙** is equivalent to **[**3, 6**]⁝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:

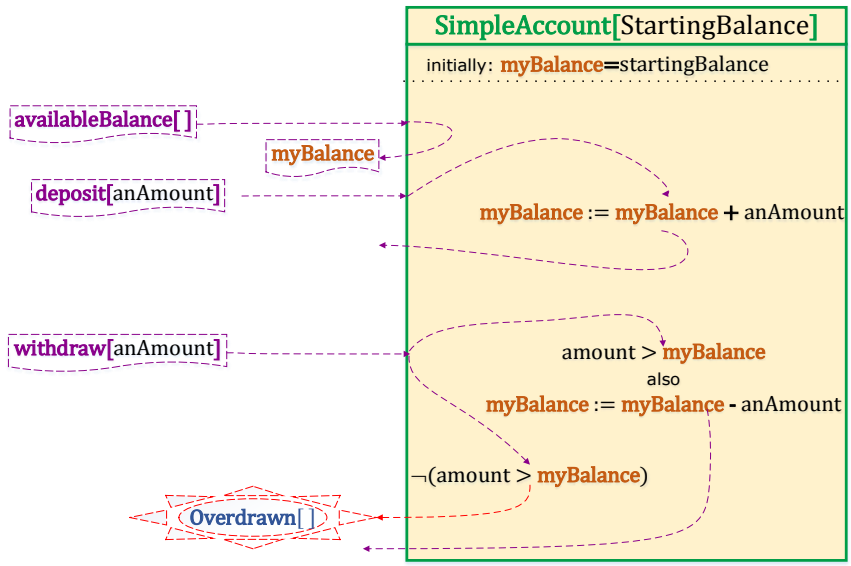
```
Interface Account with availableBalance[ ]↦Euro,
                       deposit[Euro]↦Void,
                       withdraw[Euro]↦Void
```

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.[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:

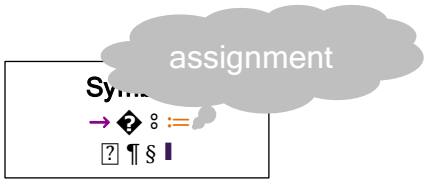
- constructor name with formal arguments delimited using brackets
- declarations of variables[i] terminated by " | "
- implementations of interface(s).

ActorScript is referentially transparent in the sense that a variable never changes while in a continuous part of the cheese.[20] For example, in the **deposit** message handler change is accomplished using the following:

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:




```
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 ⁞ Throw Overdrawn[ ],
          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❙

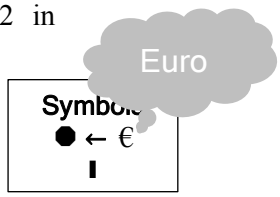
The formal syntax of **Actor** expressions is in the following end note: **21.**

[i] variable declarations separated by commas

Antecedents, Preparations, and Necessary Concurrency, *i.e.*, Ⓟ

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

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




```
(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.[i]

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.**

[i] 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[25] some abilities to change an account.[26] 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 [27]
    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 [28]
                    // 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]] ↦ AccountRevoker,
    [[account]] ↦ Account,
    withdrawFee[Euro] ↦ Void▮
```

For example, the following expression returns *negative* €3:

```
(anAccountSupervisor ← AccountSupervisor.[€3],
 anAccount ← anAccountSupervisor.[[account]],
 aRevoker ← anAccountSupervisor.[[revoker]],
 anAccount.withdraw[€2]●                    // the balance is €1
 aRevoker.revokeWithdrawable[ ]●
                                  // withdrawableIsRevoked is True
 Try anAccount.withdraw[€5]          // try another withdraw
       catch□ _ ⁚ Void ?●            // ignore the thrown exception[29]
                                            // myBalance remains €1
 anAccountSupervisor.withdrawFee[€4]●
                 // €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.[i]

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

```
Interface Tree with [[hash]]↦Integer▮
```

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

```
Structure Leaf[aString:String]
  implements Tree using
    [[hash]]:Integer → Hash.[aString]▮
```

As a result of the above definition:

```
Implementation structure Leaf[String] extends Tree▮
```

[i] 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:
  (Leaf[aString] ← Leaf["The"], aString)❙.
- Leaf["The"].[[hash]]❙ is equivalent to Hash.["The"]❙.

For example,
((Leaf["The"]):Tree).[[hash]]❙ is equivalent to Hash.["The"]❙.

Fork can be defined as an extension of Tree using:

```
Structure Fork[left:Tree, right:Tree]
  extends Tree using
    [[hash]]:Integer → Hash.[left.[[hash]], right.[[hash]]]§❙
```

As a result of the above definition:

```
Implementation structure Fork[Tree, Tree] extends Tree❙
```

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

```
(Fork[Leaf["The"], Leaf["boy"]]).[[hash]]❙
```

Testing for convertibility from of a type to an extension of the type is done using an expression of the extension can followed by "↓?" and the type. For example,

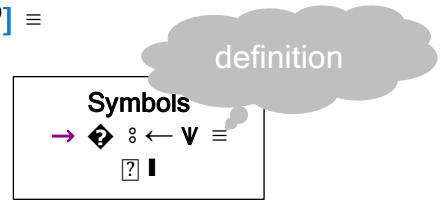
- ((Leaf["The"]):Tree)↓?Fork❙ is equivalent to False❙.
- ((Leaf["The"]):Tree)↓?Leaf❙ is equivalent to True❙.

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,

- ((Leaf["The"]):Tree)↓Leaf❙ is equivalent to Leaf.["The"]❙.
- ((Leaf["The"]):Tree)↓Fork❙ throws an exception.

"◇◇↓" 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,[31]

```
Define Fringe.[aTree:Tree]:[String⊛] ≡
    aTree ⊡
      ◇◇↓Leaf[aString] ⫶
          [aString],
      ◇◇↓Fork[left, right] ⫶
          [∨Fringe.[left],
           ∨Fringe.[right]] ⍰❙
```



For example, ["The", "boy"]:String▮ is equivalent to the following:

```
Fringe.[Fork[Leaf["The"], Leaf["boy"]]]▮
```
[32]

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

```
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.[i] 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

Swiss cheese

Swiss cheese [Hewitt and Atkinson 1977, 1979; Atkinson 1980][33] is a generalization of mutual exclusion with the following goals:

- *Generality:* Ability to conveniently program any scheduling policy
- *Performance:* Support maximum performance in implementation, e.g., the ability to minimize locking and to avoid repeatedly recalculating a condition for proceeding.
- *Understandability:* Invariants for the variables of a mutable Actor should hold whenever entering or leaving the cheese.
- *Modularity:* Resources requiring scheduling should be encapsulated so that it is impossible to use them incorrectly.

[i] An implementation type *cannot* be downcast because there is nothing to which to downcast. Note that this means that an implementation type *cannot* be subtyped although an implementation can use other implementations for modularity. Of course, for interface types there is *no* semantic guarantee of what an implementation of the interface might do as long as it obeys the signatures.

By contrast with the nondeterministic lambda calculus, there is an always-halting Actor Unbounded that when sent a [ ] message can compute an integer of unbounded size. This is accomplished by creating a Counter with the following variables:

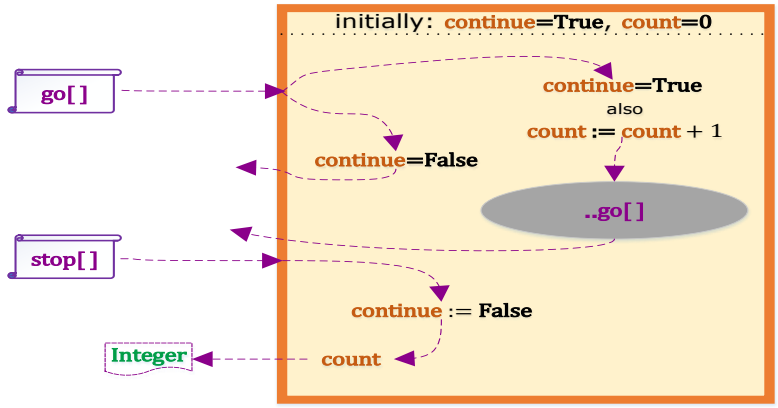
- count initially **0**
- continue initially **True**

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:

Symbols
→ ◆ ⁞
? ¶ § ▮

```
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.

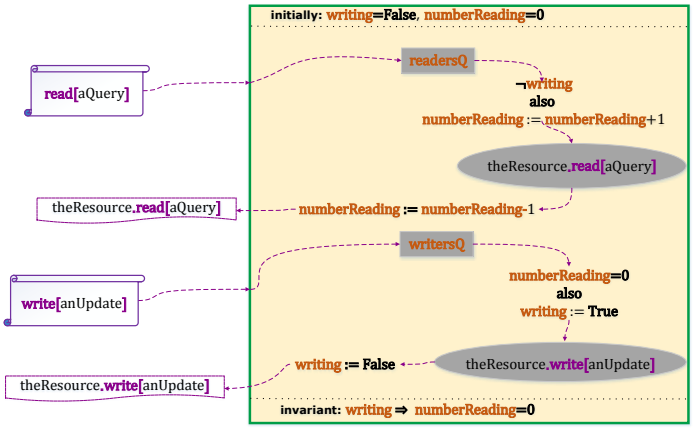
Below are two implementations of readers/writer guardians for a shared resource that implement different policies:[35]

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:

```
Interface ReadersWriter with read[Query]↦ QueryAnswer,
                             write[Update]↦ 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[i] changes only when leaving the cheese.[ii]

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**

[i] A variable is orange in the diagram

[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≧0 ,writing⇨ numberReading=0|
  queues readersQ, writersQ|      // readersQ and writersQ are initially empty
  locals writing := False,
         numberReading: := 0|
  implements ReadersWriter using
    read[aQuery:Query]:QueryAnswer →
      ((writing ∨ ¬IsEmpty writersQ) ⊘
            True ⦂ Enqueue readersQ   // release cheese while in readersQ
                    backout (¬writing ∧ numberReading=0 ∧ IsEmpty readersQ) ⊘
                              True ⦂ Void permit writersQ,
                              False ⦂ Void ?
                    Void,
             False ⦂ Void ? ●
       ¬writing          // commentary for error checking
          (numberReading++●        // increment numberReading
           permit readersQ
              hole theResource.read[aQuery] // release cheese for reading
                ↺ (IsEmpty writersQ) ⊘      // after releasing if writersQ is empty
                        True ⦂ Permit readersQ, 39
                         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 readersQ,
                               False ⦂ Void ?
                    Void,
             False ⦂ Void ? ●
       numberReading=0 ∧ ¬writing precondition
                    // commentary for error checking
          (writing := True●                    // record that writing is happening
            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?)§▮
```

Symbols
→ ⊘ ⦂ ∧ ∨ ¬
? ¶ § ▮

Illustration of writing-priority:

```
Actor WritingPriority[theResource:ReadersWriter]
  invariants numberReading≧0,  writing⇨numberReading=0|
  queues readersQ, writersQ|
  locals writing := 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 ⁞ Void permit writersQ,
                              False ⁞ Void ?
                 Void,
           False ⁞ Void ? ●
      ¬writing  precondition              // commentary for error checking
        (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 readersQ,
                              False ⁞ Void ?
                 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 ?)§▮
```

Symbols
→ ◆ ⁞ ∧ ∨ ¬
? ¶ § ▮

Conclusion

> *By the time the Software Engineering of a language gets in good shape, the language has become obsolete in "needed expressiveness"!*
> Alan Kay[40]

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[i] 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

Hal Abelson and Gerry *Sussman Structure and Interpretation of Computer Programs* 1984.

Paul Abrahams. *A final solution to the Dangling else of ALGOL 60 and related languages* CACM. September 1966.

Sarita Adve and Hans-J. Boehm *Memory Models: A Case for Rethinking Parallel Languages and Hardware* CACM. August 2010.

Mikael Amborn. *Facet-Oriented Program Design*. LiTH-IDA-EX–04/047–SE Linkőpings Universitet. 2004.

Joe Armstrong *History of Erlang* HOPL III. 2007.

Joe Armstrong. *Erlang*. CACM. September 2010/

William Athas and Charles Seitz *Multicomputers: message-passing concurrent computers* IEEE Computer August 1988.

William Athas and Nanette Boden *Cantor: An Actor Programming System for Scientific Computing* in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.

Russ Atkinson. *Automatic Verification of Serializers* MIT Doctoral Dissertation. June, 1980.

Henry Baker. *Actor Systems for Real-Time Computation* MIT EECS Doctoral Dissertation. January 1978.

Henry Baker and Carl Hewitt *The Incremental Garbage Collection of Processes* Proceeding of the Symposium on Artificial Intelligence Programming Languages. SIGPLAN Notices 12, August 1977.

Paul Baran. *On Distributed Communications Networks* IEEE Transactions on Communications Systems. March 1964.

Gerry Barber. *Reasoning about Change in Knowledgeable Office Systems* MIT EECS Doctoral Dissertation. August 1981.

Philippe Besnard and Anthony Hunter. *Quasi-classical Logic: Non-trivializable classical reasoning from inconsistent information* Symbolic and Quantitative Approaches to Reasoning and Uncertainty. Springer LNCS. 1995.

Peter Bishop *Very Large Address Space Modularly Extensible Computer Systems* MIT EECS Doctoral Dissertation. June 1977.

[i] In the sense that the implementation holds a hardware lock.

Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007a) *Interactive small-step algorithms I: Axiomatization* Logical Methods in Computer Science. 2007.

Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007b*) Interactive small-step algorithms II: Abstract state machines and the characterization theorem.* Logical Methods in Computer Science. 2007.

Per Brinch Hansen *Monitors and Concurrent Pascal: A Personal History* CACM 1996.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, Dave Winer. *Simple Object Access Protocol (SOAP) 1.1* W3C Note. May 2000.

Jean-Pierre Briot. *Acttalk: A framework for object-oriented concurrent programming-design and experience* 2nd France-Japan workshop. 1999.

Jean-Pierre Briot. *From objects to Actors: Study of a limited symbiosis in Smalltalk-80* Rapport de Recherche 88-58, RXF-LITP. Paris, France. September 1988.

Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. *Modula-3 report (revised)* DEC Systems Research Center Research Report 52. November 1989.

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

Arnaud Carayol, Daniel Hirschkoff, and Davide Sangiorgi. *On the representation of McCarthy's amb in the π-calculus "*Theoretical Computer Science" February 2005.

Alonzo Church "A Set of postulates for the foundation of logic (1&2)" Annals of Mathematics. Vol. 33, 1932. Vol. 34, 1933.

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

Will Clinger. *Foundations of Actor Semantics* MIT Mathematics Doctoral Dissertation. June 1981.

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

Eric Crahen. Facet: A pattern for dynamic interfaces. CSE Dept. SUNY at Buffalo. July 22, 2002.

Haskell Curry and Robert Feys. *Combinatory Logic*. North-Holland. 1958.

Ole-Johan Dahl and Kristen Nygaard. "Class and subclass declarations" *IFIP TC2 Conference on Simulation Programming Languages.* 1967.

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

William Dally, et al. *The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms* IEEE Micro. April 1992.

Jack Dennis and Earl Van Horn. *Programming Semantics for Multiprogrammed Computations* CACM. March 1966.

Edsger Dijkstra. *Cooperating sequential processes* Technical Report EWD-123, Technological University, Eindhoven, The Netherlands. 1965.

Edsger Dijkstra. *Go To Statement Considered Harmful* Letter to Editor CACM. March 1968.

Jason Eisner and Nathaniel W. Filardo. *Dyna: Extending Datalog for modern AI.* Datalog Reloaded. Springer. 2011.

Arthur Fine. *The Shaky Game: Einstein Realism and the Quantum Theory* University of Chicago Press, Chicago, 1986.

Frederic Fitch. *Symbolic Logic: an Introduction.* Ronald Press. 1952.

Nissim Francez, Tony Hoare, Daniel Lehmann, and Willem-Paul de Roever. *Semantics of nondeterminism, concurrency, and communication* Journal of Computer and System Sciences. December 1979.

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.

Elihu M. Gerson. *Prematurity and Social Worlds* in Prematurity in Scientific Discovery. University of California Press. 2002.

Andreas Glausch and Wolfgang Reisig. *Distributed Abstract State Machines and Their Expressive Power* Informatik Berichete 196. Humboldt University of Berlin. January 2006.

Brian Goetz *State of the Lambda* Brian Goetz's Oracle Blog. July 6, 2010.

Adele Goldberg and Alan Kay (ed.) *Smalltalk-72 Instruction Manual* SSL 76-6. Xerox PARC. March 1976.

Dina Goldin and Peter Wegner. *The Interactive Nature of Computing: Refuting the Strong Church-Turing Thesis* Minds and Machines March 2008.

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

Ben Greenman, Fabian Muehlboeck, and Ross Tate. *Getting F-Bounded Polymorphism into Shape.* PLDI'14, June 09–11, 2014.

Irene Greif and Carl Hewitt. *Actor Semantics of PLANNER-73* Conference Record of ACM Symposium on Principles of Programming Languages. January 1975.

Irene Greif. *Semantics of Communicating Parallel Professes* MIT EECS Doctoral Dissertation. August 1975.

William Gropp, et. al. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press. 1998

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

Werner Heisenberg. *Physics and Beyond: Encounters and Conversations* translated by A. J. Pomerans (Harper & Row, New York, 1971), pp. 63 – 64.

Carl Hewitt. *More Comparative Schematology* MIT AI Memo 207. August 1970.

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

Carl Hewitt, *et al. Actor Induction and Meta-evaluation* Conference Record of ACM Symposium on Principles of Programming Languages, January 1974.

Carl Hewitt and Henry Lieberman. *Design Issues in Parallel Architecture for Artificial Intelligence* MIT AI memo 750. Nov. 1983.
Carl Hewitt, Tom Reinhardt, Gul Agha, and Giuseppe Attardi *Linguistic Support of Receptionists for Shared Resources* MIT AI Memo 781. Sept. 1984.
Carl Hewitt, *et al. Behavioral Semantics of Nonrecursive Control Structure* Proceedings of *Colloque sur la Programmation*, April 1974.
Carl Hewitt. *How to Use What You Know* IJCAI. September, 1975.
Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages* AI Memo 410. December 1976. Journal of Artificial Intelligence. June 1977.
Carl Hewitt and Henry Baker *Laws for Communicating Parallel Processes* IFIP-77, August 1977.
Carl Hewitt and Russ Atkinson. *Specification and Proof Techniques for Serializers* IEEE Journal on Software Engineering. January 1979.
Carl Hewitt, Beppe Attardi, and Henry Lieberman. *Delegation in Message Passing* Proceedings of First International Conference on Distributed Systems Huntsville, AL. October 1979.
Carl Hewitt and Gul Agha. *Guarded Horn clause languages: are they deductive and Logical?* in Artificial Intelligence at MIT, Vol. 2. MIT Press 1991.
Carl Hewitt and Jeff Inman. *DAI Betwixt and Between: From "Intelligent Agents" to Open Systems Science* IEEE Transactions on Systems, Man, and Cybernetics. Nov./Dec. 1991.
Carl Hewitt and Peter de Jong. *Analyzing the Roles of Descriptions and Actions in Open Systems* Proceedings of the National Conference on Artificial Intelligence. August 1983.
Carl Hewitt. (2006). "What is Commitment? Physical, Organizational, and Social" *COIN@AAMAS'06*. (Revised version to be published in Springer Verlag Lecture Notes in Artificial Intelligence. Edited by Javier Vázquez-Salceda and Pablo Noriega. 2007) April 2006.
Carl Hewitt (2007a). "Organizational Computing Requires Unstratified Paraconsistency and Reflection" *COIN@AAMAS*. 2007.
Carl Hewitt (2008a) *Norms and Commitment for iOrgs*™ Information Systems: *Direct Logic*™ *and Participatory Argument Checking* ArXiv 0906.2756.
Carl Hewitt (2008b) "Large-scale Organizational Computing requires Unstratified Reflection and Strong Paraconsistency" *Coordination, Organizations, Institutions, and Norms in Agent Systems III* Jaime Sichman, Pablo Noriega, Julian Padget and Sascha Ossowski (ed.). Springer-Verlag. http://organizational.carlhewitt.info/
Carl Hewitt (2008e). *ORGs for Scalable, Robust, Privacy-Friendly Client Cloud Computing* IEEE Internet Computing September/October 2008.
Carl Hewitt (2008f) *Common sense for concurrency and inconsistency robustness using Direct Logic*™ and the Actor Model in Inconsistency Robustness. College Publications. 2015.
Carl Hewitt (2009a) *Perfect Disruption: The Paradigm Shift from Mental Agents to ORGs* IEEE Internet Computing. Jan/Feb 2009.

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 (2013) *Inconsistency Robustness in Logic Programs* Inconsistency Robustness. College Publications. 2015.

Carl Hewitt (2010a) *Actor Model of Computation* Inconsistency Robustness. College Publications. 2015.

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

Carl Hewitt (editor). *Inconsistency Robustness 1011* Stanford University. 2011.

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.

Carl Hewitt. "Health Information Systems Technologies" http://ee380.stanford.edu/cgi-bin/videologger.php?target=120606-ee380-300.asx Slides for this video: http://HIST.carlhewitt.info Stanford CS Colloquium. June 6, 2012.

Carl Hewitt. *What is computation? Actor Model versus Turing's Model* in "A Computable Universe: Understanding Computation & Exploring Nature as Computation". edited by Hector Zenil. World Scientific Publishing Company. 2012.

Tony Hoare *Quick sort* Computer Journal 5 (1) 1962.

Tony Hoare *Monitors: An Operating System Structuring Concept* CACM. October 1974.

Tony Hoare. *Communicating sequential processes* CACM. August 1978.

Tony Hoare. *Communicating Sequential Processes* Prentice Hall. 1985.

Tony Hoare. *Null References: The Billion Dollar Mistake.* QCon. August 25, 2009.

W. Horwat, Andrew Chien, and William Dally. *Experience with CST: Programming and Implementation* PLDI. 1989.

Anthony Hunter. *Reasoning with Contradictory Information using Quasi-classical Logic* Journal of Logic and Computation. Vol. 10 No. 5. 2000.

M. Jammer *The EPR Problem in Its Historical Development* in Symposium on the Foundations of Modern Physics: 50 years of the Einstein-Podolsky-Rosen Gedankenexperiment, edited by P. Lahti and P. Mittelstaedt. World Scientific. Singapore. 1985.

Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*, POPL'96.

Ken Kahn. *A Computational Theory of Animation* MIT EECS Doctoral Dissertation. August 1979.

Alan Kay. "Personal Computing" in *Meeting on 20 Years of Computing Science* Instituto di Elaborazione della Informazione, Pisa, Italy. 1975. http://www.mprove.de/diplom/gui/Kay75.pdf

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

Bill Kornfeld and Carl Hewitt. *The Scientific Community Metaphor* IEEE Transactions on Systems, Man, and Cybernetics. January 1981.

Bill Kornfeld. *Parallelism in Problem Solving* MIT EECS Doctoral Dissertation. August 1981.

Robert Kowalski. *A proof procedure using connection graphs* JACM. October 1975.

Robert Kowalski *Algorithm = Logic + Control* CACM. July 1979.

Robert Kowalski. *Response to questionnaire* Special Issue on Knowledge Representation. SIGART Newsletter. February 1980.

Robert Kowalski (1988a) *The Early Years of Logic Programming* CACM. January 1988.

Robert Kowalski (1988b) *Logic-based Open Systems* Representation and Reasoning. Stuttgart Conference Workshop on Discourse Representation, Dialogue tableaux and Logic Programming. 1988.

Edya Ladan-Mozes and Nir Shavit. *An Optimistic Approach to Lock-Free FIFO Queues* Distributed Computing. Sprinter. 2004.

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

Peter Landin. *A Generalization of Jumps and Labels* UNIVAC Systems Programming Research Report. August 1965. (Reprinted in *Higher Order and Symbolic Computation.* 1998)

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

Edward Lee and Stephen Neuendorffer *Classes and Subclasses in Actor-Oriented Design*. Conference on Formal Methods and Models for Codesign (MEMOCODE). June 2004.

Steven Levy *Hackers: Heroes of the Computer Revolution* Doubleday. 1984.

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

Henry Lieberman. *Thinking About Lots of Things at Once without Getting Confused: Parallelism in Act 1* MIT AI memo 626. May 1981.

Henry Lieberman. *A Preview of Act 1* MIT AI memo 625. June 1981.

Henry Lieberman and Carl Hewitt. *A real Time Garbage Collector Based on the Lifetimes of Objects* CACM June 1983.

Barbara Liskov and Liuba Shrira *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls* SIGPLAN'88.

Barbara Liskov and Jeannette Wing . *A behavioral notion of subtyping*, TOPLAS, November 1994.

Carl Manning. *Traveler: the Actor observatory* ECOOP 1987. Also appears in Lecture Notes in Computer Science, vol. 276.

Carl Manning. *Acore: The Design of a Core Actor Language and its Compile* Master Thesis. MIT EECS. May 1987.

Satoshi Matsuoka and Aki Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages Research Directions in Concurrent Object-Oriented Programming* MIT Press. 1993.

John McCarthy *Programs with common sense* Symposium on Mechanization of Thought Processes. National Physical Laboratory, UK. Teddington, England. 1958.

John McCarthy. *A Basis for a Mathematical Theory of Computation* Western Joint Computer Conference. 1961.

John McCarthy, Paul Abrahams, Daniel Edwards, Timothy Hart, and Michael Levin. *Lisp 1.5 Programmer's Manual* MIT Computation Center and Research Laboratory of Electronics. 1962.

John McCarthy. *Situations, actions and causal laws* Technical Report Memo 2, Stanford University Artificial Intelligence Laboratory. 1963.

John McCarthy and Patrick Hayes. *Some Philosophical Problems from the Standpoint of Artificial Intelligence* Machine Intelligence 4. Edinburgh University Press. 1969.

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

Giuseppe Milicia and Vladimiro Sassone. *The Inheritance Anomaly: Ten Years After* SAC. Nicosia, Cyprus. March 2004.

Mark S. Miller *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control* Doctoral Dissertation. John Hopkins. 2006.

Mark S. Miller *et. al. Bringing Object-orientation to Security Programming*. YouTube. November 3, 2011.

George Milne and Robin Milner. "Concurrent processes and their syntax" *JACM*. April, 1979.

Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics* Chapman and Hall. 1976.

Robin Milner. *Logic for Computable Functions: description of a machine implementation.* Stanford AI Memo 169. May 1972

Robin Milner *Processes: A Mathematical Model of Computing Agents* Proceedings of Bristol Logic Colloquium. 1973.

Robin Milner *Elements of interaction: Turing award lecture* CACM. January 1993.

Marvin Minsky (ed.) *Semantic Information Processing* MIT Press. 1968.

Eugenio Moggi *Computational lambda-calculus and monads* IEEE Symposium on Logic in Computer Science. Asilomar, California, June 1989.

Allen Newell and Herbert Simon. *The Logic Theory Machine: A Complex Information Processing System.* Rand Technical Report P-868. June 15, 1956

Carl Petri. *Kommunikation mit Automate* Ph. D. Thesis. University of Bonn. 1962.
Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. *A semantics for imprecise exceptions* Conference on Programming Language Design and Implementation. 1999.
Paul Philips. *We're Doing It all Wrong* Pacific Northwest Scala 2013.
Gordon Plotkin. *A powerdomain construction* SIAM Journal of Computing. September 1976.
George Polya (1957) *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving Combined Edition* Wiley. 1981.
Karl Popper (1935, 1963) *Conjectures and Refutations: The Growth of Scientific Knowledge* Routledge. 2002.
John Reppy, Claudio Russo, and Yingqi Xiao *Parallel Concurrent ML* ICFP'09.
John Reynolds. *Definitional interpreters for higher order programming languages* ACM Conference Proceedings. 1972.
Bill Roscoe. *The Theory and Practice of Concurrency* Prentice-Hall. Revised 2005.
Kenneth Ross, Yehoshua Sagiv. *Monotonic aggregation in deductive databases*. Principles of Distributed Systems. June 1992Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971.
Charles Seitz. *The Cosmic Cube* CACM. Jan. 1985.
Peter Sewell, et. al. *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Microprocessors* CACM. July 2010.
Michael Smyth. *Power domains* Journal of Computer and System Sciences. 1978.
Guy Steele, Jr. *Lambda: The Ultimate Declarative* MIT AI Memo 379. November 1976.
Guy Steele, Jr. *Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO*. MIT AI Lab Memo 443. October 1977.
Gunther Stent. *Prematurity and Uniqueness in Scientific Discovery* Scientific American. December, 1972.
Bjarrne Stroustrup *Programming Languages — C++* ISO N2800. October 10, 2008.
Gerry Sussman and Guy Steele *Scheme: An Interpreter for Extended Lambda Calculus* AI Memo 349. December, 1975.
David Taenzer, Murthy Ganti, and Sunil Podar, *Problems in Object-Oriented Software Reuse* ECOOP'89.
Daniel Theriault. *A Primer for the Act-1 Language* MIT AI memo 672. April 1982.
Daniel Theriault. *Issues in the Design and Implementation of Act 2* MIT AI technical report 728. June 1983.
Hayo Thielecke *An Introduction to Landin's"A Generalization of Jumps and Labels"* Higher-Order and Symbolic Computation. 1998.

Dave Thomas and Brian Barry. *Using Active Objects for Structuring Service Oriented Architectures: Anthropomorphic Programming with Actors* Journal of Object Technology. July-August 2004.

Kazunori Ueda *A Pure Meta-Interpreter for Flat GHC, A Concurrent Constraint Language* Computational Logic: Logic Programming and Beyond. Springer. 2002.

Darrell Woelk. *Developing InfoSleuth Agents Using Rosette: An Actor Based Language* Proceedings of the CIKM '95 Workshop on Intelligent Information Agents. 1995.

Akinori Yonezawa, Ed. *ABCL: An Object-Oriented Concurrent System* MIT Press. 1990.

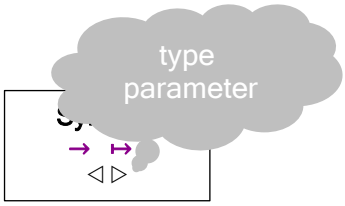
Aki Yonezawa *Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics* MIT EECS Doctoral Dissertation. December 1977.

Yizhou Zhang, Matthew C. Loring, Guido Salvaneschiy, Barbara Liskov, and Andrew C. Myers. *Lightweight, Flexible Object-Oriented Generics.* PLDI'15, June 13–17, 2015.

Hadasa Zuckerman and Joshua Lederberg. *Postmature Scientific Discovery?* Nature. December, 1986.

Appendix 1. Extreme ActorScript

Parameterized Types, *i.e.,* ◁ , ▷



Parameterized Types are specialized using other types delimited by "◁" and "▷":

```
Actor Double◁aType⊒Arithmetic▷
  [x:aType]:aType → aType[x+x]§
                          // addition for aType that is Arithmetic
```

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

```
Interface Graph◁Graph, Edge, Vertex▷
 with [[vertices]] ↦ [Vertex⊛]

Interface Edge◁Graph, Edge, Vertex▷
 with [[graph]] ↦ Graph,
      [[source]] ↦ Vertex,
      [[target]] ↦ Vertex

Interface Vertex◁Graph, Edge, Vertex▷
 with [[graph]] ↦ Graph,
      [[incoming]] ↦ [Edge⊛],
      [[outgoing]] ↦ [Edge⊛]

Actor GeoMap[ ]
        implements Graph◁GeoMap, Road, Intersection▷ using …

Actor Road[ ] implements Edge◁GeoMap, Road, Intersection▷ using …

Actor Intersection[ ] implements
                    Vertex◁GeoMap, Road, Intersection▷ using …
```

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 ⦁ y↓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[i] ← 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]])▮[43]
  is equivalent to the following:
  TrieFork[Trie[Terminal[5], Trie[Terminal[6]]▮
- The pattern TrieFork◁Integer▷[x, y] matches
  TrieFork[Trie[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.

[i] 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.[i]

```
Define TrieFringe◁aType▷.[aTrie:Trie◁aType▷]:[aType⊛] ≡
    aTrie ?
      ◈◈↓Terminal◁aType▷[x] ⁞
        [x],
      ◈◈↓TrieFork◁aType▷[left, right] ⁞
        [⩛TrieFringe.[left], ⩛TrieFringe◁aType▷.[right]] ⍰❙
```

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

```
Define TrieSameFringe?◁aType▷.[left:Trie◁aType▷,
                               right:Trie◁aType▷]:Boolean ≡
                          // test if two Tries have the same fringe
    TrieFringe◁aType▷.[left] = TrieFringe◁aType▷.[right]❙
```

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,

- "◎" followed by an expression of type Nullable◁aType▷ is the Actor (of type aType) in the nullable or throws an exception if there is no Actor.
- "**Nullable**" followed by an expression of type aType is the nullable (of type Nullable◁aType▷) 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◁Integer▷
- ◎**Nullable** 3❙ is equivalent to 3❙
- ◎**Null** Integer❙ throws an exception

[i] 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 catch⁞ , ⁞ ⍰** *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."] catch⁞
    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≥0 doesn't hold:

```
(x ← −1,
 x≥0  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**.

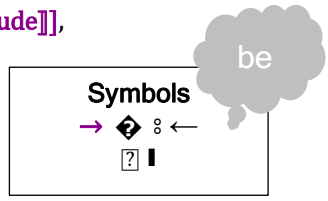
Multiple implementations of a type

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▮

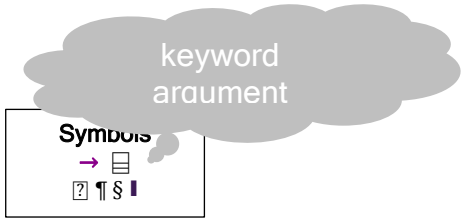
For example, cartesisans can be used in the following procedure definitions:[49]

```
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]▮[51]
```

The formal syntax of named arguments is in the following end note: **52.**

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

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

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

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

Maps,
A map is composed of pairs. For example, the following is a map:

```
Map◁Integer, String▷[[3]↠"a", [4]↠"b"]❙
```

Pairs in maps are unordered, *e.g.*,
`Map◁Integer, String▷[[3]↠"a", [4]↠"b"]❙` is equivalent to
`Map◁Integer, String▷[[4]↠"a", [3]↠"b"]❙`.

However, the expression `Map◁Integer, String▷[[4]↠"a", [4]↠"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:

```
  Structure ContactRecord[yearsOld:Age⨱,
                          numberOfContacts:Integer⨱,
                          citizenship:String⨱]❙

[ContactRecord⍟] has
   filter[[ContactRecord] |••> Boolean]
      |••> {ContactRecord⍟},
    collect [[ContactRecord] |••> [Age, Integer]]
      |••> Map◁Age, {Integer⍟}▷❙

Map◁Age, {Integer⍟}▷ has
    reduceRange[[{Integer⍟}] |••> Float]
      |••> Map◁Age, Float▷❙
```

```
{Number⊛} has average[ ] |••> Float▮

Map◁Age, Float▷ has
    sort[[Age, Age] |••> Boolean]
      |••> [Age, Float]▮
```

The program is a follows:[56]

```
Define AgeWithAverageOfNumberOfContactsSortedByAge▪
                        [records:{ContactRecord⊛}]:Sorted◁Age▷ ≡
    records▪filter [[aRecord:ContactRecord]
                    ••> aRecord▪[[citizenship]] ⍰
                          "US" ⁞ True,
                          else ⁞ False ⍰]
            ▪collect [[aRecord:ContactRecord]
                      ••> [aRecord▪[[yearsOld]],
                           aRecord▪[[numberOfContacts]]]
            ▪reduceRange
              [[aSetOfNumberOfContacts:{Integer⊛}]
                ••> aSetOfNumberOfContacts▪average[ ]]
            ▪sort[LessThanOrEqual◁Age▷]▮
```

Encryption

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

```
Encrypt ContactRecord[yearsOld ⍇ 5,
                      numberOfContacts ⍇ 7,
                      citizenship ⍇ "UK"]▮
```
[57]

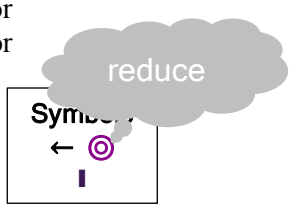
The above encrypted contact record can be decrypted only by using the type **ContactRecord**. For example, the encrypted record above matches the following pattern:

```
Decrypted◁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]❙




```
(aFuture[i] ←Future Factorial.[9999],
 ◎aFuture)❙     // do not proceed until Factorial.[9999] has
                //  been reduced[ii]
```

Futures allow execution of expressions to be adaptively executed indefinitely into the future.[58] For example, the following returns a future

```
(aFuture ← Future Factorial.[9999],
 g ← (λ [afuture:Future◁Integer▷]:Integer → 5),
                                  // g returns 5 regardless of its argument
 g.[aFuture])❙
      // return 5 regardless of whether Factorial.[9999] has completed[iii]
```

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)❙    // return Factorial.[9999]+Fibonacci.[9000]
```

[i] f is of type Future◁Integer▷

[ii] i.e. returned or threw an exception

[iii] *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, Factorial.[9999] might never be executed if readCharacter.[ ] returns the character 'x':

```
(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]:

```
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:

```
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.,* ⟪ ⟫

The following is an illustration of language extension that illustrates postponed execution:[60]

```
Actor ⟪"Postpone" anExpression:Expression◁aType▷⟫
                                                    :Postpone◁aType▷
  implements Expression◁Future◁aType▷▷ using
    eval[e:Environment]:Future◁aType▷ →
      Future Actor implements aType using
              aMessage →      // aMessage received
                (postponed ← anExpression.eval[e],
                 postponed.aMessage|||
                      // return result of sending aMessage to postponed
                 become postponed)§
                      // become the Actor postponed for
                        // the next message received[i]
```

The formal syntax of language extension is in the following end note: **61**.

[i] this is allowed because postponed is of type aType

In-line Recursion (*e.g.*, looping) , *i.e.* Loop .[← , ←] is

Inline recursion (often called looping) is accomplished using "**Loop**", an initial invocation with identifiers initialized using "←" 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.

```
Loop Factorial.[n ← 9, accumulation ← 1] is
   n=1 ? True : accumulation,
         False : Factorial.[n−1, n∗ accumulation] □▮[ii]
```

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[iii] (of type [ ]↦ Integer):

```
Loop FirstTenSequentially.[n ← 10] is
   n=1 ? True : [P.[ ]],
         False : (x ← P.[ ]●
                  [x, ⩛FirstTenSequentially.[n−1]] )□▮
```

The following returns one of the results of concurrently calling the procedure P[iv] (which has no arguments and returns Integer) ten times with no arguments:

```
Loop OneOfTen.[n ← 10] is
   n=1 ? True : P.[ ],
         False : ⓅP.[ ] either ⓅOneOfTen.[n−1]] □▮[62]
```

The formal syntax of looping is in the following end note: **63.**

---

[i] This construct is used instead of **while**, **for**, *etc*. loops used in other programming languages.

[ii] equivalent to the following:

```
Loop Factorial.[n:Integer ← 9, accumulation:Integer ← 1]:Integer is
   n=1 ? True : accumulation,
         False : Factorial.[n−1, n∗accumulation] □▮
```

[iii] The procedure P may be indeterminate, *i.e.*, return different results on successive calls.

[iv] The procedure P may be indeterminate, *i.e.*, return different results on different calls.

Strings

Strings are Actors that can be expressed using "“", string arguments, and "”". For example,

- “"1", "23", "4"”❙ is equivalent to "1234"❙.
- “"1", "2", "34", "56"”❙ is equivalent to "123456"❙.
- “ “"1", "2"”, "34"”❙ is equivalent to "1234"❙.
- “ ”❙ is equivalent to ""❙.

String patterns are delimited by "“" and "”". Within a string pattern, "⩛" 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:[64]

```
Define Reverse▪[aString:String]:String ≡
    aString ⦁
      “ ” ⨾ “ ”,
      “first, ⩛rest” ⨾ “⩛rest, first” ⍰❙
```

Symbols
⦁ ⨾ ⩛
⍰ ¶ § ❙

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,

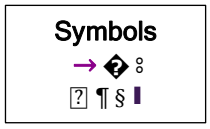
```
anExpression▪eval[anEnvironment]❙
```

is equivalent to the following:

```
(aMessage ← eval⊙Expression◁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:[67]




```
Actor SpinLock[ ]
  locals locked := False|          // initially unlocked
  implements Lockable[i] 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.**

[i] Interface Lockable with lock[ ]↦ Void,
unLock[ ]↦ Void|

Enumerations, *i.e.,* **Enumeration of** using **Qualifiers,** *i.e.,* `

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,

```
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:

```
UsingNamespace DayName❚
Define FollowingDay[aDay:DayName]:DayName ≡
   aDay ? Monday ⁞ Tuesday,
          Tuesday ⁞ Wednesday,
          Wednesday ⁞ Thursday,
          Thursday ⁞ Friday,
          Friday ⁞ Saturday,
          Saturday ⁞ Sunday,
          Sunday ⁞ 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:

```
(aRectangle[i] ← Object {"length": 3, "width": 4]},
 aFunction ← Function [ ]→ This[["length"]] * This[["width"]],
 Rectangle[["area"]] := aFunction●
 aRectangle[["area"]].[ ])❚
```

[i] aRectangle is of type Object`JavaScript

The setTimeout JavaScript object can be invoked with a callback as follows that logs the string "later" after a time out of 1000:

```
setTimeout⊙JavaScript.[1000,
                    Function [ ]→
                        console⊙JavaScript.["log"].["later"]]▮
```

HTML strings can be used to create Actor addresses. For example, the Wikipedia English homepage can be retrieved as follows:[71]

```
(HTTPS["en.wikpedia.org"]).get[ ]
```

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

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

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

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

The following notation is used for XML:[72]

```
XML <"PersonName"> <"First">"Ole-Johan" </"First">
                    <"Last"> "Dahl"</"Last"> </"PersonName">
```

and could print as:

```
<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:

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

[i] *i.e.* the following JavaScript types are not included in JSON: Date, Error, Regular Expression, and Function.

[ii] substring is a method of the String type in Java

[iii] **Refer** is called **Import** in Java

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

- **XML** <"doc"> 1, 2, **V**[3] </"doc">❙ is equivalent to **XML** <"doc">1, 2, 3</"doc">❙
- **XML** <"doc">1, 2, **V**[3], **V**[4] </"doc">❙ is equivalent to **XML** <"doc"> 1, 2, 3, 4 </"doc">❙

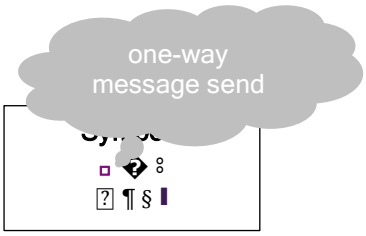
One-way messaging, *e.g.*, ⊖, and ▫

One-way messaging is often used in hardware implementations.

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

Each one-way message handler implementation consists of a named-message declaration pattern, ":", "⊖", "→" and a body for the response which must ultimately be "⊖" 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]:⊖ →
        InstructionUnit▫Execute[(x>y) ❖
                                  True ⁞ firstGreaterAddress,
                                 False ⁞ elseAddress ⍰]¶
    addJumpPositive[x:aType, y:aType, sumLocation:Location◁aType▷,
                    positiveAddress:Address, elseAddress:Address]:⊖ →
      (z ← (x+y),
       sumLocation ❖
         aVariableLocation:VariableLocation◁aType▷[i] ⁞
           (VariableLocation▪store[z]●
                          // continue after acknowledgement of store
            (z >0) ❖ True ⁞ InstructionUnit▫execute[positiveAddress],
                     False ⁞ InstructionUnit▫execute[elseAddress] ⍰),
         aTemporaryLocation:TemporaryLocation◁aType▷[ii] ⁞
           (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**

[i] VariableLocation◁aType▷ has store[aType]↦ Void▮
[ii] 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[74]
   [[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◁aType▷ between
   Returned◁anotherType▷,
   Threw▮
```

Inconsistency Robust Logic Programs

Logic Programs[75] can logically infer computational steps.

Forward Chaining

Forward chaining is performed using ⊢

(("⊢" *Theory* *PropositionExpression*))
Assert *PropositionExpression* for *Theory*.

(("When" "⊢" *Theory* aProposition:*Pattern* "→" *Expression*))
When aProposition holds for *Theory*, evaluate *Expression*.

Illustration of forward chaining:

$\vdash_t$ Human[Socrates]▮

When $\vdash_t$ Human[*x*] ⟶ $\vdash_t$ Mortal[*x*]▮
will result in asserting Mortal[Socrates] for theory t

Backward Chaining

Backward chaining is performed using ⊩

(("⊩" *Theory* aGoal:*Pattern* "→" *Expression*))
Set aGoal for *Theory* and when established evaluate *Expression*.

(("⊩" *Theory* aGoal:*Pattern*):*Expression*
Set aGoal for *Theory* and return a list of assertions that satisfy the goal.

(("When" "⊩" *Theory* aGoal:*Pattern* "→" *Expression*))
When there is a goal that matches aGoal for *Theory*, evaluate *Expression*.

Illustration of backward chaining:

$\vdash_t$ Human[Socrates]❙

When $\Vdash_t$ Mortal[*x*] → ($\Vdash_t$ Human[⌕*x*] → $\vdash_t$ Mortal[*x*])❙

$\Vdash_t$ Mortal[Socrates]❙

will result in asserting Mortal[Socrates] for theory t.

SubArguments

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

When $\Vdash_s$ (*psi* $\vdash_t$ *phi*) →

(t' ← Extension.[t],

$\vdash_{t'}$ *psi* |||

$\Vdash_{t'}$ *phi* → $\vdash_s$ (*psi* $\vdash_t$ *phi*))❙

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

- **when** $\vdash_t$ *theta* → $\vdash_{t'}$ *theta*❙
- **when** $\Vdash_{t'}$ *theta* → $\Vdash_t$ *theta*❙

[i] See appendix on Inconsistency Robust Natural Deduction.

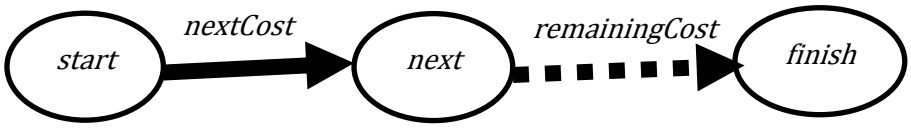
Aggregation using Ground-Complete Predicates

Logic Programs in ActorScript are a further development of Planner. For example, suppose there is a ground-complete predicate[76] Link[*aNode*, *anotherNode*, *aCost*] that is true exactly when there is a path from *aNode* to *anotherNode* with *aCost*.

```
When ⊩ Path[aNode, aNode, aCost]→
                    // when a goal is set for a cost between aNode and itself
   ⊢ aCost=0❙              // assert that the cost from a node to itself is 0
```

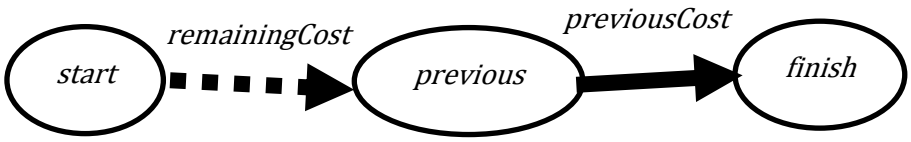
The following goal-driven Logic Program works forward from *start* to find the cost to *finish*:[77]

```
When ⊩ Path[start, finish, aCost]→
   ⊢ aCost=Minimum {nextCost + remainingCost
                        | ⊨ Link[start, next≠start, nextCost],
                            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
```



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

```
When ⊩ Path[start, finish, aCost]→
   ⊢ aCost = Minimum {remainingCost + previousCost
                        | ⊨ Link[previous≠finish, finish, previousCost],
                            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
```



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

Appendix 2: Meta-circular definition of ActorScript

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

The message eval

John McCarthy is justly famous for Lisp. One of the more remarkable aspects of Lisp was the definition of its interpreter (called Eval) in Lisp itself. The exact meaning of Eval defined in terms of itself has been somewhat mysterious since, on the face of it, the definition is circular.[78]

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[i] 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]❙
```

[i] Interface Construct❙

The interface Type

```
Interface thisType:: with
    extension?[_] ] |••> Boolean,
    has?[MethodSignature] |••> Boolean,
    sendOneWay[thisType, Message ↦⊖] ↦ ⊖,
    sendRequest[thisType, Message ↦aReturnType] ↦ aReturnType,
    encrypt[_] ↦ Encrypted,
    encrypterType[Type ] ↦ EncrypterType[i],
    decrypt[Encrypted] ↦ thisType,
    decrypterType[thisType, EncrypterType] ↦ DecryptType[ii],
    decrypt?[Encrypted] ↦ Boolean,
    return[Customer◁aReturnType▷, aReturnType] ↦ Void,
    throw[Customer, Exception] ↦ Void▮
```

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▮
```

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]

---

[i] **Implementation** EncrypterType has encrypt[thisType] ↦ Encrypted▮

[ii] **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[i]

- 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[ii]
```

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

```
dt1 ← DecrypterType[Account, et1]     // dt1:DecrypterType[iii]
anAccount ← Account[$5]       // anAccount is a new Account with $5
anAccount.deposit[$1]         // afterward anAccount has $6
x ← et1[anAccount]            // x:et1
anAccount.deposit[$2]         // afterward anAccount has $8
```

An IoT device[80] 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[81] 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
y.deposit[$3]       // afterward anAccount has $11 provided there
                    // were no withdrawals or deposits after
                    // the $2 deposit above
```

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]
```

---

[i] Possession of **Account** enables both encryption and decryption of individual accounts

[ii] **Implementation** EncrypterType **has** encrypt[_:EncrypterType] ↦ Encrypted▮

[iii] **Implementation** DecrypterType **has** decrypt[Encrypted] ↦ _:DecrypterType, decrypt?[Encrypted] ↦ Boolean▮

Only an Actor that possesses dt2 can decrypt an Actor address encrypted using et2.

```
Actor ⟪anotherType:Type◁anotherType▷
          "⩳?" aType:Type◁aType▷⟫:Expression◁Boolean▷
 uses BasicExpression◁aType▷[ ] |
 partially implements Expression◁Boolean▷ using
   eval[e:Environment]:Boolean →
    (anotherType.eval[e]).extension?[aType.eval[e]]▮
```

Type Discrimination

```
Interface DiscriminationType extends Type with
      up[Type] ↦ Discrimination,
      down[Discrimination] ↦ Type,
      down?[Discrimination] ↦ Boolean▮
```

```
Actor ⟪anExpression:Expression◁aType:Type▷
          "↑" castExpression:Type◁aDiscriminationType:Discrimination▷⟫
                    :DiscriminationUp◁aType, aDiscriminationType▷

 uses BasicExpression◁aType▷[ ] |
 partially implements Expression◁aType▷ using
  eval[e:Environment]:aType →
    castExpression.eval[e].up[anExpression.eval[e]]▮
```

```
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
                ◁DiscriminationInstance◁aType, aDiscriminationType▷▷
     "↓" 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[aDiscrminationType.down[anActor], e]❙
```

```
Actor
(("⋒⋒↓" aStructurePattern:Pattern◁aStructureType▷))
            :DownPattern◁aStructureType, aDiscriminationType▷
 uses BasicPattern◁aStructureType▷[ ] |
 partially implements Pattern◁aStructureType▷ using
   match[anActor:DiscriminationStructureInstance◁aType,
                                                  aDiscriminationType▷,
         e:Environment]:Nullable◁Environment▷ →
     structurePattern.match[aDiscrminationStructureInstanceType
                                          .down?[anActor, aStructureType],
                                  e] ❖
       True ⫶ structurePattern.match[aDiscrminationStructureInstanceType
                                          .down[anActor, aStructureType]
                                    e]
       False ⫶ Null Environment)❙
```

```
Actor
(( aDiscrminationStructureInstanceType
     "⋒↓" aStructurePattern:Pattern◁aStructureType▷))
            :TypeDownPattern◁aStructureType, aDiscriminationType▷
 uses BasicPattern◁aStructureType▷[ ] |
 partially implements Pattern◁aStructureType▷ using
   match[anActor:DiscriminationStructureInstance◁aType,
                                                  aDiscriminationType▷,
         e:Environment]:Nullable◁Environment▷ →
     structurePattern.match[aDiscrminationStructureInstanceType
                                          .down?[anActor, aStructureType],
                                  e] ❖
       True ⫶ structurePattern.match[aDiscrminationStructureInstanceType
                                          .down[anActor, aStructureType],
                                    e]
       False ⫶ Null Environment)❙
```

```
Actor
 (anExpression:Expression
                    ◁DiscriminationInstance◁aType,
                                          aDiscriminationType▷▷
        "↓?" castExpression:Type◁aType▷)
                 :DiscriminationDownQuery◁aType,
                                          aDiscriminationType▷
 uses BasicExpression◁Boolean▷[ ] |
 partially implements Expression◁Boolean▷ using
   eval[e:Environment]:Boolean →
     aDiscriminationType.down?[anExpression.eval[e]]▮
```

```
Actor ("Discrimination" aDiscriminationType "between"
          typeExpressions:Types "▮"):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 ?
                       else ⁞ Throw CastException[ ],
                 down?[anUpped:DiscriminationInstance
                                 ◁aType, aDiscriminationType▷]:Boolean →
                     anUpped ⍰
                       ⋒⋒↓SimpleDiscriminationInstance◁aType▷[_] ⁞
                           True,
                       else ⁞ False ?)▮

Structure SimpleDiscriminationInstance◁aType, aDiscriminationType▷
                                                    [anInstance:aType]
   extends DiscriminationInstance◁aType, aDiscriminationType▷▮
```

Type restriction

```
Interface RestrictionType◁aType▷
    extends Type with up[aType] ↦ RestrictionType◁aType▷▮
```

```
Actor ⟪anExpression:Expression◁aType▷
        "↑" castExpression:Type◁RestrictionType◁aType▷▷⟫
                                        :RestrictionUp◁aType▷
 uses BasicExpression◁aType▷[ ] |
 partially implements Expression◁RestrictionType◁aType▷▷ using
  eval[e:Environment]:RestrictionType◁aType▷→
   castExpression.eval[e].up[anExpression.eval[e]]▮
```

```
Actor ⟪"Interface" aRestrictionType
        "restricts" typeExpression:Type◁aType▷
        "with" signatureExpressions:Signatures"▮"⟫:Definition
 Actor implements Definition using
  eval[e:Environment]:Environment →
   (signatures ← signatureExpressions.eval[e]
    typeExpression.eval[e].has?[signatures] precondition
     e.bind[aRestrictionType,
            type ⌸ RestrictionType◁aType▷,
            to ⌸
              Actor implements RestrictionType◁aType▷ using
                up[anInstance:aType]:aRestrictionType →
                  RestrictionInstance[anInstance]])▮

Structure RestrictionInstance[anInstance:aType] uses BasicType[ ] |
 partially reimplements
     RestrictionType◁aType having
                        (aMessage↦aReturnType)∈signatures)▷ using
  sendRequest[aRecipient:aRestrictionType,
              aMessage:aMessage]:aReturnType →
    aRecipient ❖
      ⋒⋒↓RestrictionInstance[anInstance] ⁝
        sendRequest[anInstance, aMessage],
      else ⁝ Throw CastException[ ]⍰
  sendOneWay[aRecipient:aRestrictionType,
              aMessage:aMessage]:⊖ →
    aRecipient ❖
      ⋒⋒↓RestrictionInstance[anInstance] ⁝
        sendOneWay[anInstance, aMessage],
      else ⁝ Throw CastException[ ]⍰▮
```

Type extension

```
Interface Extension◁aType▷
    extends Type with
      up[ExtensionInstance◁aType▷] ↦ aType,
      down[aType] ↦ Boolean❙
```

```
Actor ⦇anExpression:Expression◁anExtensionType▷
        "↑" 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▷▷
        "⋒↑" castExpression:Type◁anExtensionType▷⦈
                    :PatternUp ◁anExtensionType, aBaseType▷
 uses BasicPattern◁anExtensionType▷[ ] |
 partially implements Pattern◁anExtensionType▷ using
   match[anActor:ExtensionInstance◁anExtensionType, aBaseType▷,
          e:Environment]:aType →
     aPattern.match[anExtensionType.up[anActor], e]❙
```

```
Actor
 ⦇anExpression:Expression◁aBaseType▷
     "↓" castExpression:Type◁anExtensionType▷⦈
                    :ExtensionDown ◁anExtensionType, aBaseType▷
  uses BasicExpression◁anExtensionType▷[ ] |
  partially implements Expression◁anExtensionType▷ using
    eval[e:Environment]:aType →
      castExpression.eval[e].down[anExpression.eval[e]]❙
```

```
Actor
 ⦇aPattern:Pattern◁anExtensionType▷
    "⋒↓" castExpression:Type◁anExtensionType▷⦈
          :ExtensionPatternDown◁aBaseType, anExtensionType▷
 uses BasicPattern◁aBaseType▷[ ] |
 partially implements Pattern◁aBaseType▷ using
   match[anActor:ExtensionInstance◁aBaseType, anExtensionType▷,
          e:Environment]:Nullable◁Environment▷ →
     aPattern.match[castExpression.eval[e].down[anActor], e]▮
```

```
Actor
 ⦇anExpression:Expression◁aBaseType▷
    "↓?" castExpression:Type◁anExtensionType▷⦈
                :ExpressionDownQuery◁anExtensionType, aBaseType▷
  uses BasicExpression◁Boolean▷[ ] |
  partially implements Expression◁Boolean▷ using
    eval[e:Environment]:aType →
     castExpression.eval[e].down?[anExpression.eval[e]]▮
```

```
Actor (("Actor" anExtensionType "extends" Type◁aType▷ "▮"))
                                                        :Definition
 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 ⍰
                          ⋒⋒↓ExtensionInstance◁anExtensionType, aType▷
                                                        [anInstance] ⁝
                             anInstance,
                          else Throw CastException[ ]⍰¶
                      down?[anUpped:aType]:Boolean →
                         anUpped ⍰
                          ⋒⋒↓ExtensionInstance◁anExtensionType, aType▷
                                                              [_] ⁝
                             True,
                          else False⍰§▮

Structure ExtensionInstance◁anExtension, aType▷
                                              [anInstance:anExtension]
    extends 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▷):NullExpression◁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):NullPattern◁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▷▷))
                                        :Expression◁aType▷
  uses BasicExpression◁aType▷[ ] |
  partially implements Expression◁aType▷ using
    eval[e:Environment]:aType →
      (anExpression.eval[e].reduce[ ]§▮
```

```
Actor ⟪"⋒◎" aPattern:Pattern◁Nullable◁aType▷▷⟫
                                        :Pattern◁aType▷
  implements Pattern◁Nullable◁aType▷▷ using
    match[anActor:Nullable◁aType▷, e:Environment]
                                        :Nullable◁Environment▷ →
      anActor.reduce?[ ] ⦻
          True ⁝ aPattern.match[anActor.reduce[ ], e] ⦻
                  TheNull ⁝ Nullable e,
                  else ⁝ Null Environment ?,
          False⁝ Null Environment ?§▮
```

Future, e.g., ◎, and Ⓟ

The type Future is used for futures:

```
Implementation Future◁aType▷ has
  reduce[ ]↦ aType▮
```

```
Actor ⟪"Future" anExpression:Expression◁aType▷⟫
                                        :Future◁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▷▷⟫
                                        :Reduction◁aType▷
  uses BasicExpression◁aType▷[ ] |
  partially implements Expression◁aType▷ using
    eval[e:Environment]:aType →
      anExpression.eval[e].reduce[ ]§▮
```

```
Actor ⟪"⋒◎" 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 ⍰,§▮
```

```
Actor ⟪"Ⓟ" anExpression:Expression◁aType▷⟫
                                        :Mandatory ◁aType▷
  uses BasicExpression◁aType▷[ ] |
  implements Expression◁aType▷ using
    eval[e:Environment]:aType →
      ◎Future anExpression▪eval[e] §▮
```

The message match

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

```
Interface Pattern◁aType▷ with
  match [aType, Environment]↦ Nullable◁Environment▷▮
```

```
Actor ⟪anIdentifier:Identifier ◁aType▷⟫:Pattern◁aType▷
  implements Pattern◁aType▷ using
    match[anActor:aType, e:Environment]:Nullable◁Environment▷ →
      e▪bind[anIdentifier, type ⌸ aType, to ⌸ anActor]▮
```

```
Actor ⟪"_"⟫:UniversalPattern◁aType▷
  implements Pattern◁aType▷ using
    match[anActor:aType, e:Environment]:Nullable◁Environment▷ →
      Nullable e▮
```

```
Actor ⟪"⌕" anExpression:Expression◁aType▷⟫
                                        :ValuePattern◁aType▷
 implements Pattern◁aType▷ using
  match[anActor, e:Environment]:Nullable◁Environment▷ →
   anActor ⍰
     anExpression.eval[e] ⁝ Nullable e,
     else ⁝ Null Environment ⍰❙
```

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]).[∀(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],
                                        [∀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◁aType▷ ","
            second:Expression◁aType▷"]"⦈:Expression ◁[aType⊛]▷
  uses BasicExpression◁[aType⊛]▷[ ]  |
  partially implements Expression◁[aType⊛]▷ using
    eval[e:Environment]:[aType⊛] →
      [first.eval[e], second.eval[e]] §▮
```

```
Actor ⦇"[" first:Expression◁aType▷ ","
        "⩛" rest:Expression◁aType▷ "]"⦈ :Expression◁[aType⊛]▷
  uses BasicExpression◁[aType⊛]▷[ ] |
  partially implements Expression◁[aType⊙]▷ using
    eval[e:Environment]:[aType⊛] →
      [first.eval[e], ⩛ rest.eval[e]] §▮
```

```
Actor ⦇"[" first:Pattern◁aType▷ ","
            "⩛" rest:Pattern◁[aType⊛]▷ "]"⦈:Pattern◁[aType⊛]▷
 implements Pattern◁[aType⊛]▷ using
   match[anActor:[aType⊛],
         e:Environment]:Nullable◁Environment▷ →
     anActor ⦶
       [first, ⩛rest] ⁝
         first.match[first, e] ⦶
           TheNull ⁝ Null Environment,
           ⋒◎aNewEnvironment ⁝
                rest.match[restValue, aNewEnvironment] ⍰,
       else ⁝ Null Environment⍰§▮
```

Exceptions

```
Actor (("Try" anExpression:Expression◁aType▷
  "catch" exceptions:ExpressionCases◁Exception, aType▷ "?"))
                                        :TryExpression◁aType▷
  uses BasicExpression◁aType▷[ ] |
  partially implements Expression◁aType▷ using
    eval[e:Environment]:aType →
      Try anExpression.eval[e] catch
        anException:Exception ⁝
          CasesEval.[anException, exceptions, e] ?§▮
```

```
Actor (("Try" anExpression:Expression◁aType▷
          "cleanup" aCleanup:Expression◁aType▷))
                                        :TryExpression◁aType▷
  uses BasicExpression◁aType▷[ ] |
  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 ↓Continuation◁aType▷ ⁝
                  aContinuaton.perform[e, c],
                anExpression↓Expression◁aType▷ ⁝
                  anExpression.eval[e] ?▮
```

Atomic compare and update

```
Actor (("Atomic" location:Expression◁Location◁anotherType▷▷,
              "compare"  comparison:Expression◁anotherType▷
              "update" update:Expression◁anotherType▷ "⊡"
              "updated" "⁞"
                  compareIdentical:ContinuationList◁aType▷ ","
               "notUpdated" "⁞"
                  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]] ⊡
          True ⁞ compareIdentical.perform[e, c],
          False ⁞
            compareNotIdentifical.perform[e, c] ?▮

  Actor SimpleLocation◁anotherType▷[initialContents]
    locals contents := initialContents
    implements Location◁anotherType▷ using
       compareAndConditionallyUpdate[comparison, update]:Boolean →
          (contents = comparison) ⊡
              True ⁞ True ↺ contents := update,
               False ⁞ False ?§▮
```

Cases

```
Actor ⟪anExpression:Expression◁anotherType▷ "⊡"
                    cases:ExpressionCases◁anotherType, aType▷ "?"⟫
                                                  :CasesExpression◁aType▷
  uses BasicExpression◁aType▷[ ] |
  partially implements Expression◁aType▷ using
    eval[e:Environment]:aType →
      CasesEval.[anExpression.eval[e], cases, e]§▮

Actor CasesEval
  [anActor:anotherType,
   cases:[ExpressionCase◁anotherType, aType▷⊛],
   e:Environment]:aType →
  cases ⊡
    [ ] ∶ Throw NoApplicableCase[ ],
    [first, ∀rest] ∶
      first ⊡ ⟪aPattern:Pattern◁anotherType▷ "∶"
                  anExpression:Expression◁aType▷⟫
                                              :ExpressionCase◁aType▷ ∶
                aPattern.match[anActor, e] ⊡
                  TheNull ∶
                    CasesEval.[anActor, rest, e],
                  ⋒◎newEnvironment ∶
                    anExpression.eval[newEnvironment] ?,
              ⟪"else" elsePattern:Pattern◁anotherType▷"∶"
                       elseExpression:Expression◁aType▷⟫
                                              :ExpressionElseCase◁aType▷ ∶
                elsePattern.match[anActor, e] ⊡
                  TheNull ∶
                    Throw ElsePatternMustMatch[ ],
                  ⋒◎newEnvironment ∶
                    elseExpression.eval[newEnvironment] ?,
              ⟪"else" "∶"
                    elseExpression:Expression◁aType▷⟫
                                              :ExpressionElseCase◁aType▷ ∶
                elseExpression.eval[e],
              else ∶ Throw NoApplicableCase[ ] ??▮
```

```
Actor ⟪anExpression:Expression◁anotherType▷ "⊡"
                cases:ContinuationCases◁anotherType, aType▷ "⍰")
                                        :CasesContinuation◁aType▷
 implements Continuation◁aType▷ using
     perform[e:Environment, c:CheeseQ]:aType →
         CasesPerform▪[anExpression▪eval[e], cases,  e, c]§▮

Actor CasesPerform
    [anActor:anotherType,
     cases:[ContinuationCase◁aType▷⍟],
     e:Environment,
     c:CheeseQ]:aType →
  cases ⊡
   [ ] ⁞ Throw NoApplicableCase[ ],
   [first, ∀rest] ⁞
     first ⊡ ⟪aPattern:Pattern◁anotherType▷"⁞"
                  aContinuation:Continuation◁aType▷⟫
                                        :ContinuationCase◁aType▷ ⁞
                aPattern▪match[anActor, e] ⊡
                   TheNull ⁞
                       CasesPerform▪[anActor, rest, e, c],
                    ⋒◎newEnvironment ⁞
                     aContinuation▪perform[newEnvironment, c] ⍰,
               ⟪"else"
                     elsePattern:Pattern◁anotherType▷ "⁞"
                         elseContinuation:Continuation◁aType▷⟫
                                    :ContinuationElseCase◁aType▷ ⁞
                       elsePattern▪match[anActor, e] ⊡
                          TheNull ⁞
                             Throw ElsePatternMustMatch[ ],
                           ⋒◎newEnvironment ⁞
                              elseContinuation▪eval[newEnvironment] ⍰,
               ⟪"else" "⁞"
                        elseContinuation:Continuation◁aType▷⟫
                                        :ContinuationElseCase◁aType▷ ⁞
                  elseContinuation▪perform[e, c],
               else ⁞ Throw NoApplicableCase[ ] ⍰⍰▮
```

Holes in the cheese

```
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)§▮
```

```
Actor ⦇aVariable:Variable◁aType▷
              ":=" anExpression:Expression◁aType▷⦈:Assignment
 implements Assignment using
   carryOut[e:Environment]:Void →
      e.assign[aVariable, to ⌸ anEpression.eval[e]]§▮
```

```
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)§▮
```

```
Actor ⟪"(" aPreparations:Preparations
          anExpression:Expression◁aType▷ ")"⟫
                                        :CompoundExpression◁aType▷
  implements Continuation◁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◁anotherType▷
          "↺" anAfterward:AfterwardContinuation◁aType▷ "□"⟫
                                                  :Hole◁aType▷
 implements Continuation◁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□
            _ : (Holding c in anAfterward.perform[e, c]●
                 Rethrow)□§▮
```

```
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)
      catch❓
         _ ⁈ (resource.release[ ]●
              Rethrow)?§❙
```

```
("Hole" anExpression:Expression◁anotherType▷
   "returned❓"
       returnedCases:ContinuationCases◁anotherType, aType▷ "?"
   "threw❓"
      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
         (c.acquire[ ]●
          CasesPerform.[anException, threwCases, e, c])?§❙
```

```
Actor ("Enqueue" anExpression:QueueExpression "●"):Enqueue
 implements Continuation using
   perform[e:Environment, c:CheeseQ]:Void →
      anExpression.eval[e].enqueueAndLeave[ ] §❙
```

```
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]) §▮
```

Simple Implementation of Actor

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

```
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 ⦁
     [ ] ⁞ Throw MessageRejected[ ],
     [((aMessageDeclaration:MessageDeclaration◁aType▷
          ":" ReturnDeclaration◁aType▷ "→"
          body:Continuation◁aType▷))
                                   :ContinuationHandler◁aType▷,
      restHandlers] ⁞
        aMessageDeclaration.match[receivedMessage, e] ⦁
          TheNull ⁞
            Select.[receivedMessage, restHandlers, e, c],
                        // process next handler
          ⋒◎newEnvironment ⁞
            Execute◁aType▷.[body, newEnvironment, c] ▯▯▮
```

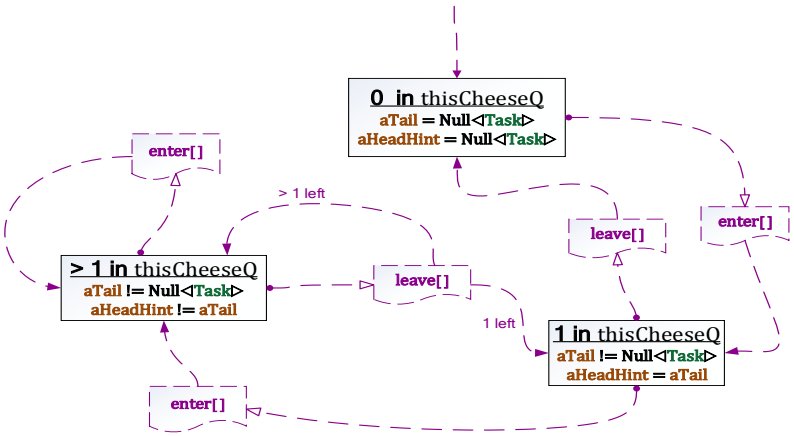
An implementation of cheese that never holds a lock

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

```
Actor CheeseQ[ ]
  invariants aTail=Null Activity ⇨ previousToTail=Null Activity |
  locals aHeadHint := Null Activity,          // aHeadHint:Nullable◁Activity▷[82]
         aTail := Null Activity|              // aTail:Nullable◁Activity▷[83]
  acquire[ ]:Void nonexclusive in myActivity →[84]
    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[85] 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]][?])§
  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)] [?]§▮
```

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[ ] ↦ Void
release[ ] ↦ Void▮

The implementation CheeseQ uses activities to implement its queue where

Implementation Activity **has**
⟦previous⟧ ↦ Nullable◁Activity▷
// if null then head of queue else, pointer to backwards list to head
⟦previous := Nullable◁Activity▷⟧ ↦ Nullable◁Activity▷
// returns self so that updates can be chained
⟦nextHint⟧ ↦ Nullable◁Activity▷
// if non-null then pointer to next activity to get cheese after this one
⟦nextHint := Nullable◁Activity▷⟧ ↦ Nullable◁Activity▷▮
// returns self so that updates can be chained

Implementation type InternalQ is defined on the next page where:

Implementation InternalQ **has**
enqueueAndLeave[ ] ↦ Void,
enqueueAndDequeue[InternalQ] ↦ Activity
dequeue[ ] ↦ Activity
empty?[ ] ↦ Boolean▮

```
Actor InternalQ[c:CheeseQ]
  locals aQueue ← SimpleFIFO◁Activity▷[ ]|
  enqueueAndLeave[ ]:Void  in myActivity →
                  // enqueueAndLeave message received in myActivity
            (aQueue▪add[myActivity]●

             c▪release[ ]●        // myActivity is the head of aCheeseQ
             Suspend)¶
                  // myActivity is suspended and when resumed returns Void ¶
  enqueueAndDequeue[anInternalQ:InternalQ]:Activity in myActivity →
      ¬anInternalQ▪empty?[ ] precondition
                                                  // commentary for error checking
            (aQueue▪add[myActivity]●

             ..dequeue[ ]●
             Suspend)¶
  dequeue[ ]:Activity  in myActivity →
      ¬ ..empty?[ ]  precondition          // commentary for error checking
            (c▪release[ ]●
                                      // myActivity is the head of aCheeseQ

             MakeRunnable aQueue▪remove[ ])¶
                                      // make runnable the removed activity
  empty?[ ]:Boolean  → aQueue▪empty?[ ]§▮

where
   Interface FIFO◁aType▷ has
     add[anActivity:aType]  ↦ Void,
     remove[anActivity:aType]  ↦ aType,
     empty?[ ] ↦ Boolean▮
```

Appendix 3. ActorScript Symbols with IDE ASCII, and Unicode codes

Symbol	IDE ASCII[i]	Read as	Category	Matching Delimiters	Unicode (hex)
❙	;;	end	top level terminator		25AE
:	:	of specified type	infix		
::	::	is a type	postfix		
⍰	[:]	this Actor with interface (aspect)	prefix		2360
◎	\0[86]	reduce (nullables, futures)	prefix		29BE
◈◎	~\0[87]	match reduced (nullables, futures)	prefix		
↓	\v/	down	infix		2193
↓?	\v/?	down query	infix		
◈↓	~\v/	match downed	infix		
↑	^^	up	infix		2191
◈↑	~^	match upped	prefix		
⊙	(.)	qualified by	infix		22A1
λ	/\	procedure	prefix	≡ and/or →	03BB
≡	===	defined as	infix	**Define**	2261
▪	.	is sent	infix		
▪▪	..	send to this Actor	prefix		2025
Ⓟ	\P[88]	necessarily concurrent	prefix		29B7
↦	|->	message type returns type[89]	infix		21A6
|••>	|..>	cacheable ↦			
→	-->	message received[90]		λ and/or ¶	2192
↠		pair	infix		21A0
←	<--	be[91]	infix		2190
⍰	??	cases	separator	⍰	FFFD
⍰	???	end cases	terminator	⍰ and **catch⍰**	2370
¶	\p[92]	another message handler	separator for handlers	→	00B6
§	\s	end handlers	terminator	**implements** and **extension**	00A7
⁞	(:)	case	separator for case		2982
●	;	before	separator	binding, preparation and **Enqueue**	2BC3
⫴	|||	concurrently	separator	binding, **preparation**	2225

[i] These are only examples. They can be redefined using keyboard macros according to personal preference.

				and **Enqueue**	
≔	:=	is assigned	infix		2254
↺	U^	afterward	infix		21BA
⌕	\o[93]	matches value of[94]	prefix		2315
=	=	same as?	infix		
≠	!=	Different from?	infix		2260
⌸	[=]	keyword **or** field	infix		2338
:⌸	:[=]	assignable field	infix		
◁	<|	begin type parameters	left delimiter	▷ (Unicode hex: 0077)	0076
Ѱ	|/	spread[95]	prefix		2A5B
{	{	begin set	left delimiter	}	
⌊	[	begin list	left delimiter	]	
⦃	{|	begin multi-set	left delimiter	⦄	2983
⟦	[|	formatted message	left delimiter	⟧	27E6
"	\"	Left string structure	left delimiter	"	201C
(	(	begin grouping	left delimiter	)	
⦇	(|	begin syntax	left delimiter	⦈	2985
⊛	(*)	zero or more	postfix		229B
⋮	:::	uniformly of a type	infix		22EE
⊖	(-)	nothing[96]	expression		229D
▫	\.	one-way send	infix		219E
⊔	_|	join	infix		2294
⊑	[<=]	constrained by	infix		2291
⊒	[>=]	extends	infix		2292
⇒	==>	logical implication	infix		21E8
⇔	<=>	logical equivalence	infix		21D4
∧	/\	logical conjunction	infix		00D9
∨	\/	logical disjunction	infix		00DA
¬	-|	logical negation	prefix		00D8
⊢	|-	assert	prefix and infix		22A2
⊩	||-	goal	prefix and infix		22A9
//	//	begin 1-line comment	prefix	EndOfLine	
/*	/*	begin comment	prefix	*/	

Appendix 4. ActorScript Reserved Words

Prefix

Token	Separators	Terminator
Try	catch§ cleanup	
Interface	extends with restricts with	❙
Discrimination	between	❙
Actor | Structure	invariants | uses | queues | implements using § internal using §	❙
Implementation	has	❙
Holding	in	
Loop	is	
Hole	returned ↺ threw ↺	
Enqueue		
Null		
Nullable		
MakeRunnable		
Suspend		
Atomic	compare update updated notUpdated	

Infix

Token
thatIs
postcondition
precondition
permit

Unary

Token
True
False
TheNull
Void

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
⩛, 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

⇔, 85
↺, 11, 13, 18, 21, 22, 47, 76, 81, 85
§, 11, 84
¶, 11
¶, 84
℗, 12, 18, 69, 84
ȣ, 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

Function (JavaScript), 48
Future, 42, 43, 52, 68
Garst, B., 23, 82
general messaging, 46
Greif, I., 23
has, 14, 40, 86
has?, 57
having, 57
Holding, 77, 86
hole, 18
Hole, 76, 86
Hole ... returned ... threw, 77
HTML, 49
HTTPS, 49
identifier, 6
Implementation, 11, 14, 82, 86
implements, 11, 13, 21, 22, 86
in, 81, 83
Integrated Development
  Environment, 5
Interface, 10, 13, 14, 38, 56, 69,
  72, 86
internal, 81
InternalQ, 82
invariants, 21
is, 45, 47, 81, 86
JavaScript, 48
JSON, 49
Kahn, K., 23
Leaf, 14
Lieberman, H., 23
locals, 11, 21
Logic Program
  Backward chaining, 53
  forward chaining, 53
  subarguments, 54
Loop, 45, 86
MakeRunnable, 81, 83, 86
Manning, C., 23
Map, 40
Mason, I., 23
match, 69
Message, 52
Meta, 52
Miller, M. S., 23
Montalvo, F. S., 23
Montanari, U., 23
Morningstar, C., 24
Nassi, I., 23
nextHint, 82
Null, 67, 81, 86
Nullable, 36, 67, 69, 71, 81, 82,
  86
Nygaard, K., 1
Object, 48
Object (JavaScript), 48
One-way messaging, 50
parameterized
  type, 33
partially, 13
patterns, 6
perform, 72
permit, 21, 22, 86
Polar, 39
postcondition, 37, 86
Postpone, 44
Prep, 76
Prep ... ⬢, 81
previous, 82
Qualifiers, 48
queues, 21, 22, 86
reimplements, 13
Reinhardt, T., 23
Request, 52
resolve future, 42
resource
  release, 77
  take, 77
Response, 52
RestrictionType, 63
restricts, 57, 63
Rethrow, 72, 77
return, 57
returned, 86
Returned, 52
Schumacher, D., 23

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
Threw, 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
λ, 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:*Numerical* "+" y:*Numerical*⟫:*Numerical*❙
Numerical⊑*Expression*❙

Also,

⟪*Numerical* "-" *Numerical*⟫:*Numerical*❙
⟪"-" *Numerical*⟫:*Numerical*❙
⟪*Numerical* "*" *Numerical*⟫:*Numerical*❙
⟪*Numerical* "/" *Numerical*⟫:*Numerical*❙
⟪"**Remainder**" *Numerical* "/" *Numerical*⟫:remainder:*Numerical*❙
⟪"**QuotientRemainder**" *Numerical* "/" *Numerical*⟫
:[*Numerical, Numerical*]❙
⟪"**True**" ⊔ "**False**"⟫:*Expression*❙
⟪*Expression* "∧" *Expression*⟫:*Expression*❙
⟪*Expression* "∨" *Expression*⟫:*Expression*❙
⟪"¬" *Expression*⟫:*Expression*❙
⟪"**Throw**" *Expression*⟫:*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.

Identifier⊑*Word*⊑*Expression*❙
// an *Identifier* is a *Word,* which is a subcategory of *Expression*
⟪⟪*Expression*⊔ *Definition*⊔ *Judgment*⟫⟫ "❙"):*Top*❙

[5] ⟪*Identifier* ":" *Type* ⟫:*Declaration*
// *Identifier* is declared to be of *Type*
⟪*Identifier* "::" ⟫:*Declaration* // *Identifier* is declared to be a type
⟪*Type* "↦" *Type* ⟫:*Signature*❙
⟪*Type*"↦" "⊖"⟫ :*Signature*❙

(("[" Types "]"):Type▮
(⊔ MoreTypes):Types▮
(Type ⊔(Type "," MoreTypes)):MoreTypes▮
[6] (Identifier "←" Expression):Definition▮
(Preparation (("," ⊔ "●") MorePreperations)):Preparations▮
(Expression):MorePreperations▮

[7] Generalization of the notation of [Church 1932].

[8] (("Define" ProcedureName"." "[" ArgumentDeclarations "]" ":" Type "≡" Expression):Definition▮
ProcedureName⊑Expression▮
(⊔ MoreDeclarations):ArgumentDeclarations▮
(SimpleDeclaration(⊔ (("," MoreKeywordDeclarations))
⊔ (SimpleDeclaration "," MoreDeclarations))
:MoreDeclarations▮
// Comma is used to separate declarations.
(((Identifier (⊔ "default" Expression)):SimpleDeclaration▮
(KeywordArgumentDeclaration
⊔ (KeywordDeclaration ","MoreKeywordDeclarations))
:MoreKeywordDeclarations▮
(Keyword "⌸"SimpleDeclaration)):KeywordDeclaration▮
Keyword⊑Word▮

[9] The symbol . is fancy typography for an ordinary period when it is used to denote message sending.

[10] (Recipient:Expression "." "[" Arguments "]"):ProcedureSend▮
ProcedureSend⊑Expression▮
// Recipient is sent a message with Arguments
(⊔ MoreArguments):Arguments▮
(((Expression(⊔ (("," MoreKeywordArguments)))
⊔ (Expression "," MoreArguments)):MoreArguments▮
(KeywordArgument
⊔ (KeywordArgument
"," MoreKeywordArguments)):MoreKeywordArguments▮
(Keyword "⌸" Expression):KeywordArgument▮
(Identifier"["ArgumentDeclarations "]" ":" Type "→"
Preparations"▮"):Definition▮

[11] ? solves the infamous "dangling else" problem [Abrahams 1966].

[12] (test:Expression "?" ExpressionCases"?"):Expression▮
(ExpressionCase ⊔ MoreExpressionCases):ExpressionCases▮
(ExpressionCase ⊔ (ExpressionCase "," MoreExpressionCases)
⊔ ExpressionElseCases):MoreExpressionCases▮

```
    ⟪ ⊔ ExpressionElseCase  ⊔ ⟪ExpressionElseCase
          "," MoreExpressionElseCases⟫⟫:ExpressionElseCases ▮
    ⟪ExpressionElseCase
      ⊔ ⟪ExpressionElseCase
          "," MoreExpressionElseCases ⟫⟫:MoreExpressionElseCases▮
    ⟪ ⟪"else" ":" Preparations⟫
      ⊔ ⟪"else" Pattern ":" Preparations⟫⟫:ExpressionElseCase ▮
      // The else case is executed only if the patterns before
            //   the else case do not match the value of test.
    ⟪Pattern ":" Preparations⟫:ExpressionCase ▮
13  ⟪"(" Preparations Expression ")"⟫:CompoundExpression
    Binding⊑Preparation ▮
      // A let binding is a preparation
    ⟪ ⟫:Preparations
    ⟪Binding"←" Expression ⟪"●" ⊔ "," ⟫ Preparations⟫⟫:Preparations
    ⟪Expression⟪"●" ⊔ "||"⟫ Preparations⟫⟫:Preparations
    ⟪Pattern "←" Expression⟫:Binding ▮
14  ⟪recipient:Expression
          "." MessageName "[" Arguments "]"⟫:NamedMessageSend ▮
     NamedMessageSend ⊑Expression ▮
          // Recipient is sent message MessageName with Arguments
     MessageName⊑Word ▮
    ⟪"Interface"  Identifier▷  "with"
        MessageHandlerSignatures"▮"⟫:InterfaceDefinition ▮
     InterfaceDefinition ⊑Definition ▮
    ⟪ ⊔ MoreMessageHandlerSignatures⟫⟫
                                          :MessageHandlerSignatures ▮
    ⟪MessageHandlerSignature
          ⟪ ⊔ MoreMessageHandlerSignatures⟫⟫
                                      :MoreMessageHandlerSignatures ▮
    ⟪MessageName"[" ArgumentTypes"]"  ⟪ "↦"⊔ "|••>"⟫
        returnType:Type⟫:MessageHandlerSignature ▮
     MessageHandlerSignature ⊑Expression ▮
15  equivalent to [1, ∀[2, 3]:Integer, 4]:Integer
16  ⟪"[" ComponentExpressions"]"
          ⟪ ⊔ ⟪":" aType:Type⟫
            ⊔ ⟪"⁝" "[" someTypes:Types"]"⟫⟫⟫ :Expression
        // An ordered list with elements ComponentExpressions
         // ⁝ means uniformly of type aType
         // ⁞ means each element is of the
              // corresponding type in someTypes
    ⟪ ⊔ MoreComponentExpressions⟫:ComponentExpressions ▮
```

```
((( ⊔ "∀") Expression) ⊔ (( ⊔ "∀") Expression
    "," MoreComponentExpressions)):MoreComponentExpressions▮
("[" TypeExpressions "]"):TypeExpression▮
( ⊔ MoreTypeExpressions):TypeExpressions▮
(TypeExpression ⊔ (TypeExpression "," MoreTypeExpressions))
                                        :MoreTypeExpressions▮
```

[17] Equivalent to the following:

```
Define Reverse◁aType▷▪[aList:[aType⊛]]:[aType⊛] ≡
  aList ⍰
    [ ] ∶ [ ],
    [first, ∀rest] ∶ [∀rest, first] ⍰▮
```

[18]
```
("_"):UnderscorePattern▮
UnderscorePattern⊑Pattern▮
Identifier⊑Pattern▮
(Pattern "thatIs" Expression):ThatIs▮
ThatIs⊑Pattern▮
("ρ" Expression):Pattern▮
("[" ComponentPatterns "]"):Pattern▮
  // A pattern that matches a list whose elements match
    // ComponentPatterns
( ⊔ MoreComponentPatterns):ComponentPatterns▮
(Pattern ⊔ ( "∀"Pattern )
          ⊔ (Pattern "," MoreComponentPatterns))
                                  :MoreComponentPatterns▮
```

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

[20] 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.

[21]
```
("Actor" ConstructorDeclaration ActorBody):Expression▮
  //  The above expression creates an Actor with
            // declarations for variables and message handlers
  ( ⊔ ( "uses" ConstructorList "|" ))
  ( ⊔ "management" Expression)
  MessageHandlers
  InterfaceImplementations):ActorBody▮
(Identifier "◁" ParametersDeclarations "▷"
    ( ⊔ ("[" ArgumentDeclarations "]"))) :ConstructorDeclaration▮
```

```
⟪Constructor "," MoreConstructors "|"⟫:ConstructurList❙
⟪Constructor
    ⊔ ⟪Constructor "," MoreConstructors⟫⟫:MoreConstructors❙
⟪ ⊔ "locals" LocalsDeclarations  "|" ⟫:LocalsDeclaration❙
⟪QueuesDeclaration LocalsDeclaration⟫:Declaration❙


⟪LocalDeclaration
          "," MoreLocalDeclarations⟫:MoreLocalDeclarations❙
⟪Identifier "←" Expression⟫:IdentifierDeclaration❙
IdentifierDeclaration⊑LocalDeclaration❙
⟪Variable ":=" Expression InstanceVariableAQualifications⟫
                                                :VariableDeclaration❙
VariableDeclaration⊑LocalDeclaration❙
Variable⊑Word❙
InstanceVariableQualifications⊑ InstanceQualifications❙
⟪ ⊔ InstanceVariableQualification
    ⊔⟪ InstanceVariableQualification
          InstanceVariableQualifications⟫
                                  :InstanceIVariableQualifications❙
 "nonpersistent"⊑InstanceVariableQualification ❙
          // A nonpersistent variable must be Nullable,
               // and can be nulled out before a message is received
⟪ "queues" QueueName⟫:QueueDeclaratoins❙
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❙
```

```
⟪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❙
```

[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] `(("Ⓟ" anExpression:Expression( ⊔ ("sponsor" Expression)))`
`: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.
```

[25] *cf.* [Crahen 2002, Amborn 2004, Miller, et. al. 2011]

[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

[30] ⟪"**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*"ᵴ" *ExpressionsContinuation*⟫:*ContinuationCase*▮
⟪ ⊔ *MoreContinuationElseCases*⟫:*ContinuationElseCases*▮
⟪*ContinuationElseCase*
⊔ ⟪*ContinuationElseCase* "," *MoreContinuationElseCases*⟫⟫
:*MoreContinuationElseCases*▮
⟪⟪"**else**" "ᵴ" *ExpressionsContinuation*⟫
⊔ ⟪"**else**" *Pattern*"ᵴ" *ExpressionsContinuation*⟫⟫
:*ContinuationElseCase*▮
⟪*Continuation*⟫:*ExpressionsContinuation*▮
⟪*preparation*⟪"," ⊔ "●"⟫ *MoreExpressionsContinuation*⟫⟫
:*ExpressionsContinuation*▮
⟪⟪*Continuation*⟫⟫
⊔ ⟪*Expression*","*MoreExpressionsContinuation*⟫⟫
: *MoreExpressionsContinuation*▮

[31] Equivalent to the following:

Define Fringe[aTree:Tree]:[String⊛]
aTree ⍰
Leaf[aString] ᵴ [aString],
Fork[left, right] ᵴ
[∀Fringe.[left], ∀Fringe.[right]] ?▮

[32] Equivalent to the following:

Fringe.[Fork[Leaf["The"]↑Tree₈Leaf["boy"]↑Tree]↑Tree]

[33] Swiss cheese was called "serializers" in the literature.

[34]
```
⟪"▪▪" 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 */
```

[35] 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 writing := False,
           numberReading := 0|
  implements ReadersWriter using
    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)§❙
```

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

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

[38] ⟪"(" *Antecedents*
"**enqueue**" *QueueExpression* ⟪ ⊔ "**backout**" *Preparations*⟫
Continuation ")"⟫:*Continuation*▮
/*
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* ⊔ *MoreContinuationCases*⟫:*ContinuationCases*▮
⟪*ContinuationCase* ⊔
⟪*ContinuationCase* "," *MoreContinuationCases*⟫
⊔ *ContinuationElseCases*⟫:*MoreContinuationCases*▮
⟪ ⊔ *ContinuationElseCase* ⊔
⟪*ContinuationElseCase* "," *MoreContinuationElseCases*⟫⟫
:*ContinuationElseCases*▮
⟪*ContinuationElseCase*
⊔ ⟪*ContinuationElseCase* "," *MoreContinuationElseCases*⟫⟫
:*MoreContinuationElseCases*▮
⟪⟪"**else**" ":" *ContinuationList*⟫
⊔ ⟪"**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*
⟪ ⊔ ⟪ "**permit**" aQueue:*Expression*⟫⟫
⟪ ⊔ ⟪"↺" *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*
⟪ ⊔ ⟪"**also**" *VariableAssignments*⟫⟫⟫:*PermitAlso*▮

The following can be used temporarily release the cheese:

⟪"**Hole**" *Expression*⟫:*Continuation*❙
/*
1. Leave the cheese
2. The response is the result of evaluating *Expression* */

⟪"**(**" *Antecedents*
hole *Expression* ⟪ ⊔ ⟪ "↺" *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* */

⟪"**(**" *Antecedents*
hole *Expression*
⟪ ⊔ ⟪ "**returned**" *ContinuationCases* "□"⟫⟫
⟪ ⊔ ⟪ "**threw**" *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* "▷" *Preparations*⟫
:*ParameterizedDefinition*❙
ParameterizedDefinition⊑*Definition*❙
// Parameterize definition with *ParametersDeclarations*❙
⟪ ⊔ *MoreParameterDeclarations*⟫:*ParametersDeclarations*❙
⟪*ParameterDeclaration*
⊔ ⟪*ParameterDeclaration*
"," *MoreParameterDeclarations*⟫⟫
:*MoreParameterDeclarations*❙
⟪*Identifier*⟪ ⊔ *Qualifier*⟫⟫:*ParameterDeclaration*❙
⟪ ⊔ ⟪"**extends**" *Type*⟫⟫:*TypeQualifier*❙
⟪*Identifier* "◁" *Parameters* "▷"⟫:*TypeExpression* ❙
⟪*Identifier* ⊔ ⟪ ⊔ ⟪*Identifier* "," *Parameters*⟫⟫:*Parameters*❙

[42] ⟪"**Discrimination**" *Identifier MoreTypeDiscriminations* "❙"⟫
:*Definition*❙

```
⟪Identifier⊔ ⟪Identifier ","MoreTypeDiscriminations⟫⟫
                                        :MoreTypeDiscriminations▮
⟪Expression"↓" Type⟫:Expression▮
        // Discriminate to be of Type if possible.
         // Otherwise, an exception is thrown.
⟪Expression "↓?" Type⟫:Expression▮
        // If Expression 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.
⟪ "⋒⋒↓" StructurePattern⟫:Pattern▮
        // Matching Actor must be discrimination that
            // can downed as StructurePattern which matches
```

[43] Equivalent to the following:

```
(x ← 3,
 TrieFork◁Integer▷[Terminal◁Integer▷[x]↑Trie◁Integer▷,
                  Terminal◁Integer▷[x+1]↑Trie◁Integer▷])▮
```

[44]
```
⟪Identifier"[" Arguments"]"⟫:Expression▮
 ⟪Identifier "[" Patterns"]"⟫:Pattern▮
```

[45]
```
⟪"Nullable" Expression⟫:Expression▮
 ⟪"◎" Expression⟫:Expression▮
        // reduce Expression if not null.
         // Otherwise, an exception is thrown.
 ⟪"⋒◎" 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. /*

```
(("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)
                                                    :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. */
[47] ((test:Expression "precondition" Preparations):Expression
    // test must evaluate to True or an exception is thrown
  ((test:Expression "precondition" ExpressionsContinuation)
                                                    :Continuation
    // test must evaluate to True or  an exception is thrown
  ((value:Expression "postcondition" pre:Expression):Expression
      // The expression pre must evaluate to True when sent value
        //  or an exception is thrown
[48] ° is a reserved postfix operator for degrees of angle
[49] Using parameterized procedures like the ones below can improve the
  simplicity and effectiveness of types by comparison with other approaches
[50] 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
[51] 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
```

```
52 (("Structure" Identifier "[" FieldDeclarations "]"
     (⊔(("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
  (⊔ MoreFieldDeclarations):FieldDeclarations▮
  (((SimpleFieldDeclaration
                                 (⊔(("," MoreNamedFieldDeclarations))))
        ⊔(SimpleFieldDeclaration
              "," MoreFieldDeclarations)):MoreFieldDeclarations▮
  (((Identifier(⊔"default" Expression)):SimpleFieldDeclaration▮
  (NamedFieldDeclaration
    ⊔(NamedFieldDeclaration
         "," MoreNamedFieldDeclarations))
                                          :MoreNamedFieldDeclarations▮
  (FieldName
       (":⌸" ⊔ ":⌸") SimpleFieldDeclaration))
                                          :NamedFieldDeclaration▮
  FieldName⊑QualifiedName▮
     // ":⌸" is used for assignable fields.
  ((⊔Identifier) ActorBody):StructureImplementation▮
 (Expression "[[" FieldName"]]" ):FieldSelector▮
       // FieldName of Expression which must be a structure
  FieldSelector⊑Expression▮
  (StructureName "[" FieldExpressions "]" ):StructureExpression▮
  StructureExpression⊑Expression▮
  (⊔ MoreFieldExpressions):FieldExpressions▮
  (((SimpleFieldExpression(⊔(("," MoreNamedFieldExpressions)))
      ⊔(SimpleFieldExpression
              "," MoreFieldExpressions)):MoreFieldExpressions▮
  (NamedFieldExpression
     ⊔(NamedFieldExpression
              "," MoreNamedFieldExpressions))
                                          :MoreNamedFieldExpressions▮
```

```
⦇FieldName
      ⦇"▤" ⊔ ":▤"⦈ SimpleFieldExprression⦈⦈
                                                  :NamedFieldExpression▮
⦇StructureName "[" FieldPatterns "]" ⦈:StructurePattern▮
StructurePattern⊑Pattern▮
⦇ ⊔ MoreFieldPatterns⦈:FieldPatterns▮
⦇⦇SimpleFieldPattern⦇ ⊔ ⦇"," MoreNamedFieldPatterns⦈⦈⦈
   ⊔ ⦇ SimpleFieldPattern "," MoreFieldPatterns⦈⦈
                                                  :MoreFieldPatterns▮
⦇NamedFieldPattern
 ⊔ ⦇ NamedFieldPattern
          "," MoreNamedFieldPatterns⦈⦈
                                                  :MoreNamedFieldPatterns▮
⦇FieldName ⦇"▤" ⊔ ":▤"⦈ SimpleFieldExprression⦈⦈
                                                  :NamedFieldPattern▮
```

[53] ⦇"{" ComponentExpressions "}"⦈:Expression▷▮
    // A set of Actors without duplicates
⦇"{" ComponentPatterns "}"⦈:Pattern▮

[54] ⦇"{|" ComponentExpressions "|}"⦈:Expression▮
    // A multiset of the Actors with possible duplicates
⦇"{|" ComponentPatterns "|}"⦈:Pattern▮

[55] ⦇"[" Expressions "]" "↠" Expression⦈:Expression▮
⦇"{|" ComponentPatterns "|}"⦈:Pattern▮

[56] 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 "|↔>"

[57] ⦇"**Encrypt**" Expression⦈:Expression
  Define ⦇"**Encrypt**" anExpression:Expression⦈ ≡
      ⦇aType "▪" "**encrypt**" "[" anExpression "]"⦈

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

```
Actor f▪[ ]:Future◁BottomLessFuture▷ → Future f▪▮
Define BottomLessFuture  ([ ] ↦ BottomLessFuture)▮
```

[59] (("**Future**" aValue:*Expression* (⊔ ("**sponsor**" *Expression*)))
:*Expression*▮
// A future for aValue.
("◎" *Expression*):*Expression*▮
// Reduce a future

[60] A **Postpone** expression does not begin execution of Expression$_1$ 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

[61] ("(" *MoreGrammers* ")"):*Grammar*▮
("("*Grammar*"⊔"*Grammar*")"):*Grammar*▮
(*ReservedWord* (⊔ *StartsWithIdentifier*)):*StartsWithReserved*▮
StartsWithReserved⊑*MoreGrammers*▮
(*Identifier* (⊔ *StartsWithReserved*))):*StartsWithIdentifier*▮
StartsWithIdentifier⊑*MoreGrammers*▮
("\ "" *Word* "\""):*ReservedWord*▮
// The use of \ escapes the next character in a string so
// that "\"" has just one character that is ".
(*Grammar*":" *GrammarIdentifier* "▮"):*Judgment*▮
(*Identifier*"⊑"*Identifier*"▮"): *Judgment*▮

[62] Equivalent to the following:

Loop OneOfTen.[n:**Integer** ← 10]:**Integer** **is**
n=1 ⧄ **True** ⁝ P.[ ],
False ⁝ ⓅP.[ ] **either** ⓅOneOfTen.[n−1]] ?▮

[63] ("**Loop**" LoopName:*Identifier* "." "[" *Initializers* "]" ":" *Type*)
"**is**" *Expression*):*Expression*▮
(⊔ *MoreInitializers*):*Initializers*▮
(*Initializer* ⊔ (*Initializer* "," *MoreInitializers*))
:*MoreInitializers*▮
(*Identifier* "←" *Expression*):*Initializer*▮

[64] The implementation below requires careful optimization.

[65] ("“" *ComponentExpressions* "”"):*Expression*▮
("“" *ComponentPatterns* "”"):*Pattern*▮

[66] (recipient:*Expression*"." message:*Expression*):*Expression*▮
// Send recipient the message

[67] The implementation below can be highly inefficient.

[68] (("**Atomic**" aLocation:*Expression*
"**compare**" comparison:*Expression*
"**update**" update:*Expression* "**⦁**"
"**updated**" "**:**"
compareIdentical:*ExpressionsContinuation* ","
"**notUpdated**" "**:**"
compareNotIdenticial:*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.

[69] ((*Identifier* "**`**" *Qualifier*)):*QualifiedName*▮
QualifiedName⊑*Expression*▮
Identifier⊑*QualifiedName*▮

((*Identifier*⊔((*Identifier* "**`**" *Qualifier*))):*Qualifier*▮

[70] (("**Enumeration**" *Identifier*
MoreEnumerationNames "**|**")):*Definition*▮
((*EnumerationName*
⊔((*EnumerationName*
"**,**" *MoreEnumerationNames*))):*MoreEnumerationNames*▮
EnumerationName⊑*Word*▮

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

[72] Declarations provide version number, encoding, schemas, *etc.*

[73] ((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*▮

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

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

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

[77] Execution can proceed differently depending on how sets fit into computer storage units.

[78] /* 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]:

```
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! */

[79] For example, the message could be of type

Message◁**DepositOnlyAccount**, **deposit**[**Euro**] ↦ **Void**▷

where

Interface **DepositOnlyAccount** **restricts** **Account** **with** **deposit**[**Euro**] ↦ **Void**

[80] the device may have *no* access to anAccount or **Account**

[81] the device may have *no* access to anAccount, x, **et1**, or **Account**

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

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

[84] // **acquire** message received running myActivity

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

[86] Not to be confused with \0 which is the null character or with \o which is ⌀.

[87] Not to be confused with \0 which is the null character or with \o which is ℴ.
[88] Not to be confused with \p which is ¶
[89] Used in type specifications for interfaces.
[90] Used in message handlers.
[91] Used to bind identifiers.
[92] Not to be confused with \P which is Ⓟ
[93] Not to be confused with \0 which is the null character or with \O which is ◎.
[94] Used in patterns.
[95] Used in structures.
[96] Used in one-way message passing.