The Kogut Programming Language Reference

October 22, 2011
(incomplete yet)

Contents

1 Overview and Terminology

A claim qualified with "*by convention*" is intended to be true, but its truth ultimately depends on all parts of the program complying with the convention, so formally it is guaranteed to be true only for standard types.

An operation qualified with "*a hint*" may influence performance or other informal properties of the program, but does not have an observable effect that a programmer should rely on.

1.1 Execution

Execution of Kogut code manipulates *objects*. A *value* is a pointer to an object.

When execution *completes*, it either *succeeds*, or *fails* with an exception which is a value. Unless specified otherwise, when execution of some syntactic form includes execution of its parts, as soon as one of the parts fails, the whole form fails as well with the same exception.

A *static* error is an error which is detected at compile time and which makes code invalid. Errors detected when code is executed are called *dynamic* errors and are reported with exceptions.

1.2 Types

Every object is classified by its direct *type*. Kogut is dynamically typed, which means that the direct type of the value of an expression is generally not known statically.

A type name used as a generic noun means an object of that type. For example an *int* means an object of type INT.

A type is associated with a list of its direct supertypes. A *supertype* of a type is the type itself or its proper supertype. A *proper supertype* of a type is a supertype of its direct supertype or OBJECT. An object has type T when T is a supertype of the direct type of the object.

T_1 is called a *subtype* of T_2 when T_2 is a supertype of T_1.

The direct type of an object never changes. New subtype relationships between existing types may be declared during execution of the program, but they cannot be removed.

An *abstract* type is a type which is not a direct type of any object (and there is no way to create objects of that type).

A *final* type is a type which may not have any further subtypes declared. A *concrete* type is a final type with no proper subtypes.

A concrete type T is a *singleton* type when there is a single object having type T (and there is no way to create other such objects).

The *constructor* of a type T is a function which returns an object of type T and which is the only way to create objects of type T.

1.3 Supertype ordering

The set of supertypes of a type T is ordered, forming its *supertype list.* The order, as described in [BCH+96], is computed as follows.

Partial orders are gathered:

- For each direct supertype of T, its supertype list, computed recursively by this algorithm.
- The list of direct supertypes of T, with OBJECT added at the end.

The first element of the supertype list of T is T itself. Then, as long as any partial orders remain, the first element of each partial order in turn is examined, searching for a type which does not appear as a nonfirst element of any partial order. If there is no such type, computing the supertype list fails, and the supertype hierarchy is said to be *inconsistent* (this happens when two supertypes of T yield conflicting orders of some further supertypes). Otherwise the first type found becomes the next element of the supertype list being built, it gets removed from all partial orders which contain it (this is always the first element), partial orders which became empty are discarded, and building the list continues.

1.4 Mutability

A *mutable* object is an object which directly contains state which can change during the lifetime of the object. The opposite is called *immutable.* An immutable object can still refer to other mutable objects.

This is not a formal property of an object, because whether given state is considered a part of the given object, or a part of one of its subobjects, is an aspect of the presentation which is not directly observable by the program. It is a useful characterization though, under a specific assuption about which subobjects are exposed and which are considered implementation details of other objects.

1.5 Object identity

Objects have *identity*, which means that it is meaningful to ask whether two values refer to the same object; this relation is called physical equality.

Object identity is the invisible property of any object which allows to decide that. A newly created object is distinguishable from all previously existing objects by having a unique identity.

For many immutable types of objects however, like numbers, lists, or strings, the identity of objects used to represent them is mostly irrelevant (except in rare circumstances like serialization, where checking object identity and maintaining object sharing is essential for avoiding a possible exponential explosion of the object graph size). Generally it does not matter whether two lists were created independently or it is really a single list referred to from two places; it only matters whether they have the same elements. Types of objects with mostly irrelevant identity are called *value types*, as opposed to *object types* whose objects have a meaningful and useful identity.

All mutable objects necessarily have object types, because the consequences of mutation depend on the amount of object sharing. Some immutable objects have object types too however, which means that there is a guarantee that a given value is represented by a unique object. This is especially true for singleton types.

1.6 Threads

Program execution consists of concurrently executing *threads*. A thread may be *active* or *completed*. A completed thread either *succeeded* or *failed* executing its body. An active thread is either *running*, or *waiting* for an event which can be triggered by another thread or is external to the program.

1.7 Syntax basics

The fundamental syntactic forms are expressions, patterns, and definitions.

An *expression* denotes a computation which *evaluates* to a value, if it succeeds.

A *pattern* denotes a computation which examines a value called the *subject* being *matched*, and either *accepts* it or *rejects*, if it succeeds. An accepting pattern may bind some names in the context where the pattern is used.

Unless specified otherwise, when matching against a pattern includes matching against its subpatterns, as soon as one of the submatches rejects its subject, the whole match rejects its subject as well. Names bound by the subpatterns are visible in the code executed after the submatch, and become bound by the whole pattern if it eventually accepts its subject.

A *definition* denotes a computation which may bind some names.

A *statement* is a definition formed by an expression. It evaluates the expression and ignores its result; it does not bind any names.

An important case of an expression is an *application* of a value to a list of values. Consequences of an application depend on the nature of the object being applied.

To *enter* a subexpression means to evaluate it and return its value as the value of the whole expression. To *tail-call* an object with some values as arguments means to enter the application of the object to the arguments. It is expected that the memory which has been implicitly allocated for the caller's execution state (a stack frame) is dealloated before evaluation of the subexpression being entered. Such subexpression is said to be in a *tail position* with respect to the whole expression.

To enter a sequence of definitions means to evaluate them in order, binding in the current scope names they introduce. If the last definition is a statement, the expression which forms it is entered instead of being merely executed, otherwise Null is returned.

Unless specified otherwise, when a form needs values of some of its subforms which are interpreted as expressions, they are evaluated left to right.

1.8 References

Names are bound to *references*, which are objects which support operations like getting and setting their value. The set of supported operations and their interpretation depends on the type of the reference.

The most fundamental type of a reference is a *constant*. Getting the value of a constant always returns the same value, and setting it is not supported.

For the brevity of presentation, to bind a name to something other than a reference means to bind it to a constant with the given value.

Mutable variables support getting and setting the value. Getting the value returns the most recently set value, or the initial value if the variable has not been set yet.

Dynamic variables are associated with *cells*, where a cell behaves like a mutable variable. Getting or setting the value of a dynamic variable gets or sets the value of its current cell. Which cell is current depends on what code accesses the variable. Dynamic variables support local rebinding, which creates a new cell and makes it current only for the given code. The *dynamic environment* of a place is the set of cells which are current for the place.

1.9 Functions

Function definitions and function expressions produce *functions*. When a function is applied, it matches arguments against its parameter patterns and enters definitions in its body.

An ordinary function has a single definition. A *generic function* is implemented separately for different combinations of argument types.

Technically, a function definition consists of several cases, selected by matching arguments against separate sequences of parameter patterns. Regular cases specify the behavior of the function directly. A *generic case* uses some of the parameters, called the *keys*, to find the actual *method*, which is a specialization of the original function to particular argument types. The *arity* of a generic case is either a fixed number of parameters or the lower bound of the number of parameters. A generic function is a function which has at least one generic case.

When $T_1, \ldots, T_n$ are the types of the keys, a method defined for $T'_1, \ldots, T'_n$ is *applicable* if every T'_i is a supertype of T_i. If there is no applicable method, the dispatch fails with 'NoMethod function arity arguments', where function is the symbol of the generic function, a negative arity encodes the arity being at least $-1-$arity. and arguments is the list of key arguments. Otherwise the *best* applicable method is tail-called with the same arguments as the original function. A method defined for $T'_1, \ldots, T'_n$ is better than a method defined for $T''_1, \ldots, T''_n$ if the first T'_i after their common prefix comes earlier than T''_i in the supertype list of T_i.

1.10 Scoping

Several kinds of forms introduce *scopes* associated with some of their subforms. A scope is a region of code where names can be defined.

Statically, a name can be referred to from any place within its scope, except where it is shadowed by a definition of the same name in some inner scope.

Dynamically, a reference can be used only after its definition has been executed, otherwise trying to access the reference fails with 'NotDefined name', where name is the symbol of the reference. This implies that a function may refer to references defined below, as long as the function is not applied before these definitions are executed.

Bound names can be marked as *public* or *private*. This classification is used by some constructs which introduce scopes. The default is **public** unless specified otherwise.

1.11 Object fields

Some objects include *fields*, which are references defined in the scope of the object, accessible from the outside of the object if they are public.

(re)	same as re (grouping)
$re_1\ re_2$	re_1 followed by re_2
$re_1 \mid re_2$	re_1 or re_2
re^*	zero or more occurrences of re
re^+	one or more occurrences of re
$re^?$	zero or one occurrence of re
“a”	this string
U+$xxxx$	this character, given by Unicode code point
“a” . . . “z”	a character from the range of code points
⟨Cat⟩	a character with Unicode category Cat
until_{re}	any fragment which does not include a match for re

Table 1: Notation of lexical rules

Applying an object with fields to a symbol and some arguments, tail-calls its reference named by the symbol to the other arguments. If there is no field with the given name, the application fails with ‘NoField object field’.

Unless specified otherwise, applying an object which was not intended to being applied at all, fails with ‘NotAFunction object’.

1.12 Syntax

In order to facilitate macros, the Kogut syntax is defined in three stages rather than the usual two:

1. Lexical syntax: a sequence of characters is transformed to a sequence of tokens.
2. Abstract grammar: the sequence of tokens is transformed to an abstract syntax tree.
3. Concrete grammar: the abstract syntax tree is interpreted as a structure of expressions, patterns, and definitions.

2 Lexical Syntax

The notation used to describe lexical rules is described in table 1.

The lexical rules are specified in tables 2 and 3. For convenience, “ASCII:” notes remind the intersections of the given sets of characters with ASCII.

The source text is split into fragments which match TOKEN or IGNORE. At each point the longest match is used. The sequence of tokens is used for further parsing.

space	::=	⟨Zs⟩ | `U+0009`
	ASCII:	“ ” | `U+0009`
newLine	::=	`U+000A` | `U+000D` | `U+000D U+000A`
nameFirst	::=	⟨Lu⟩ | ⟨Ll⟩ | ⟨Lt⟩ | ⟨Lm⟩ | ⟨Lo⟩ | ⟨Nl⟩ |
		`U+2118` | `U+212E` | `U+309B` | `U+309C`
	ASCII:	“A” . . . “Z” | “a” . . . “z”
nameRest	::=	nameFirst | ⟨Mn⟩ | ⟨Mc⟩ | ⟨Nd⟩ | ⟨Pc⟩ | “'” |
		`U+1369` . . . `U+1371`
	ASCII:	“A” . . . “Z” | “a” . . . “z” | “0” . . . “9” | “_” | “'”
digit	::=	“0” . . . “9”
hexDigit	::=	“0” . . . “9” | “A” . . . “F” | “a” . . . “f”
octDigit	::=	“0” . . . “7”
binDigit	::=	“0” | “1”
escape	::=	“"” | “'” | “\” |
		“a” | “b” | “t” | “n” | “v” | “f” | “r” | “s” |
		digit^{+} “;” |
		“x” hexDigit^{+} “;” |
		“o” octDigit^{+} “;” |
		space* newLine space*
name	::=	nameFirst nameRest* |
		“_” nameRest^{+} |
		“'” (until$_{“'” \mid “\backslash”}$ | “\” escape)* “'”
intLit	::=	digit^{+} |
		(“0x” | “0X”) hexDigit^{+} |
		(“0o” | “0O”) octDigit^{+} |
		(“0b” | “0B”) binDigit^{+}
exponent	::=	(“e” | “E”) (“+” | “-”)$^{?}$ digit^{+}
floatLit	::=	digit^{+} “.” digit* exponent$^{?}$ |
		“.” digit^{+} exponent$^{?}$
		digit^{+} exponent |
stringLit	::=	“"” (until$_{“"” \mid “\backslash”}$ | “\” escape)* “"”
symbolLit	::=	“#” name
literal	::=	intLit | floatLit | stringLit | symbolLit

Table 2: Lexical rules, part 1

label:	::=	name “`:`”
.field	::=	“`.`” name
@intLit	::=	“`@`” intLit
%op	::=	“`%`” name
op%	::=	name “`%`”
operator	::=	“`@`” | “`*`” | “`/`” | “`+`” | “`-`” | “`==`” | “`~=`” | “`<`” | “`>`” | “`<=`” | “`>=`” | %op | op%
punctuation	::=	“`(`” | “`)`” | “`[`” | “`]`” | “`{`” | “`}`” | “`?`” | “`=>`” | “`...`” | “`!`” | “`^`” | “`~`” | “`->`” | “`\`” | “`&`” | “`|`” | “`=`” | “`,`” | “`;`”
token	::=	name | “`_`” | literal | “`()`” | label: | .field | @intLit | operator | punctuation
lineComment	::=	“`//`” $\text{until}_{\text{newLine}}$
comment	::=	“`/*`” $\text{until}_{\text{“/*”} \mid \text{“*/”}}$ (comment $\text{until}_{\text{“/*”} \mid \text{“*/”}}$)* “`*/`”
ignore	::=	space | newLine | lineComment | comment

Table 3: Lexical rules, part 2

“\"”	“"”
“\'”	“'”
“\\”	“\”
“\a”	U+0007
“\b”	U+0008
“\t”	U+0009
“\n”	U+000A
“\v”	U+000B
“\f”	U+000C
“\r”	U+000D
“\s”	“ ”
“\” DIGIT$^+$ “;”	decimal character code
“\x” HEXDIGIT$^+$ “;”	hexadecimal character code
“\o” OCTDIGIT$^+$ “;”	octal character code
“\” SPACE* NEWLINE SPACE*	ignored
“_” SPACE* NEWLINE	ignored

Table 4: Escape sequences in strings and quoted names

Literals represent constant values. An integer literal denotes an int with the given value. A float literal denotes the application of the value of the dynamic variable DefaultReal to a ratio or int with the given value.

In a string literal any character except “"”, “\”, and NEWLINE stands for itself; NEWLINE stands for U+000A; and escape sequences are interpreted according to table 4.

Quoted symbol names are processed like string literals, except that the demiliters are “'” instead of “"”. Quoted names are just a mechanism of inserting arbitrary characters into names; the meaning of an unquoted name is the same as the meaning of the corresponding quoted name.

A name starting with “_” is associated with a private symbol (see section 5.14 on page 64), unique for the given module. Other names are associated with the corresponding public symbols.

Rationale: The syntax of unquoted names is consistent with Unicode recommendations for identifier syntax, with the addition of “'” as a nonfirst character. □

Rationale: A separate @INTLIT token lets an expression like ‘array@0.field’ be interpreted as the likely intended ‘array@(0).field’ rather than ‘array@(0.) field’. □

3 Abstract Grammar

The grammar is specified in tables 5 and 6, using a notation similar to the lexical syntax in table 1, except that it works on the level of tokens instead of characters.

For presentation brevity, the grammar has been left ambiguous. An unambiguous context-free grammar can be obtained by adding variants of appropriate symbols while applying the following constraints:

- *param* may not begin with '`{`' nor '`=>`'.
- The *body* after '`=>`' extends to the right as far as possible, i.e. unparenthesized '`=>`' may only appear in the rightmost subterm before '`)`', '`]`', '`}`', or the end of input.

In this document '$\rightarrow$' and '$\Rightarrow$' denote '$->$' and '$=>$' respectively.

Some forms are merely syntactic sugar, as specified in table 7.

Parentheses are used for grouping, they are not retained in the AST.

Since rewriting of the above syntactic sugar happens on the AST level, the left-to-right evaluation rule applies to the result of the rewriting, even if the rewriting reorders subexpressions.

4 Concrete Grammar

The concept of signals, the rules of signal blocking and synchronous mode are explained in section 5.26 on page 70.

The notation for describing the meaning of syntactic forms uses the following convention: a line consisting of '$\cdots$' by itself means that the previous line can be repeated any number of times, including zero. The '$_i$' indices at metavariables in a repeated line stand for successive integers in the repetitions.

4.1 Expressions

An expression '*app* = *value*', when *app* is a syntactic application, means *app* with *value* appended as the last argument.

4.1.1 References

Evaluating the name of a reference applies the reference object to no arguments, which by convention gets the current value of the reference.

An expression '**ref** *name*' evaluates to the reference itself.

$atom$	::=	NAME	reference
	|	‘_’	wildcard pattern
	|	LITERAL	literal
	|	‘()’	no arguments
	|	‘(’ $expr$ ‘)’	grouping
	|	‘[’ arg^* ‘]’	list
	|	(‘?’ $param^*)^?$ ‘{’ $body$ ‘}’	lambda abstraction
	|	(‘?’ $param^*)^?$ ‘=>’ $body$	lambda abstraction
$post_1$	::=	$post_1$ .FIELD	field
	|	$post_1$ ‘@’ $atom$	indexing
	|	$post_1$ @INTLit	indexing
	|	$atom$	
$post_2$	::=	$post_1$ ‘...’	multiple values
	|	$post_1$ ‘!’	dispatched parameter
	|	$post_1$ ‘^’ $post_1$	specialized to type
	|	$post_1$	
arg	::=	‘~’ arg	logical negation
	|	LABEL: arg	pair with a symbol
	|	$post_2$	
$param$	::=	arg	
$apply_1$	::=	$arg\ arg^+$	application
	|	arg	
$apply_\rightarrow$	::=	$apply_\rightarrow^?$ ‘->’ $arg\ arg^*$	application
	|	$apply_1$	
$apply_L$	::=	$apply_L^?$ ‘~’$^?$ %OP $apply_\rightarrow$	application
	|	$apply_\rightarrow$	
$apply_R$	::=	$apply_\rightarrow$ ‘~’$^?$ OP% $apply_R$	application
	|	$apply_\rightarrow$	
$apply_2$	::=	$apply_L^?$ ‘~’$^?$ %OP $apply_\rightarrow$	application
	|	$apply_\rightarrow$ ‘~’$^?$ OP% $apply_R$	application
	|	$apply_\rightarrow$	

Table 5: Abstract grammar, part 1

mul	::=	*mul* ‘`*`’ *apply*$_2$	multiplication
	|	*mul*$^?$ ‘`/`’ *apply*$_2$	division
	|	*apply*$_2$	
add	::=	*add* ‘`+`’ *mul*	addition
	|	*add*$^?$ ‘`-`’ *mul*	subtraction
	|	*mul*	
rel	::=	*add*$^?$ ‘`==`’ *add*	equal
	|	*add*$^?$ ‘`~=`’ *add*	inequal
	|	*add*$^?$ ‘`<`’ *add*	less than
	|	*add*$^?$ ‘`>`’ *add*	greater than
	|	*add*$^?$ ‘`<=`’ *add*	less than or equal
	|	*add*$^?$ ‘`>=`’ *add*	greater than or equal
	|	*add*	
cons	::=	*rel* ‘`\`’ *cons*	prepend to a list
	|	*rel*	
and	::=	*cons* ‘`&`’ *and*	conjunction
	|	*cons*	
or	::=	*and* ‘`|`’ *or*	alternative
	|	*and*	
assign	::=	*or* ‘`=`’ *assign*	assignment
	|	*or*	
pair	::=	*assign* ‘`,`’ *pair*	pair
	|	*assign*	
expr	::=	(*pair* ‘`;`’)$^+$ *pair*$^?$	local definitions
	|	*pair*	
body	::=	(*pair* ‘`;`’)* *pair*$^?$	definitions

Table 6: Abstract grammar, part 2

$\{body\}$	$\equiv$	$?\{body\}$
$?param^* \Rightarrow body$	$\equiv$	$?param^*\ \{body\}$
$\Rightarrow body$	$\equiv$	$?\{body\}$
$obj.field$	$\equiv$	$obj\ \#field$
$label$:$expr$	$\equiv$	$\#label,\ expr$
$arg_1 \rightarrow fun\ arg^*$	$\equiv$	$fun\ arg_1\ arg^*$
arg_1 %op arg_2	$\equiv$	$op\ arg_1\ arg_2$
arg_1 ~%op arg_2	$\equiv$	~$op\ arg_1\ arg_2$
arg_1 op% arg_2	$\equiv$	$op\ arg_1\ arg_2$
arg_1 ~op% arg_2	$\equiv$	~$op\ arg_1\ arg_2$
$\rightarrow fun\ arg^*$	$\equiv$	fun _ arg^*
%op arg_2	$\equiv$	op _ arg_2
~%op arg_2	$\equiv$	~op _ arg_2
$==\ arg_2$	$\equiv$	_ $==\ arg_2$
$\sim=\ arg_2$	$\equiv$	_ $\sim=\ arg_2$
$<\ arg_2$	$\equiv$	_ $<\ arg_2$
$>\ arg_2$	$\equiv$	_ $>\ arg_2$
$<=\ arg_2$	$\equiv$	_ $<=\ arg_2$
$>=\ arg_2$	$\equiv$	_ $>=\ arg_2$

Table 7: Syntactic sugar in the abstract grammar

An expression '*name* = *value*' means '(**ref** *name*) *value*', which by convention sets the new value of the reference if the reference supports that.

4.1.2 Application

The syntax of object application, 'fun arg^+', is interpreted specially when fun is an identifier belonging to a fixed set of special expression operators: **case**, **function**, **handle**, **if**, **ifDefined**, **local**, **loop**, **match**, **method**, **module**, **receive**, **ref**, **struct**, **try**.

Otherwise this syntax denotes the application of object fun to arguments arg^+, modified as follows.

Occurrences of '()' among arguments are removed. This syntax allows to express application of an object to the empty list of arguments, since fun alone denotes the object itself and could not be used for this purpose.

An argument followed by '...' must evaluate to a list, and elements of the list are spliced in the place of the argument as separate arguments.

If '_' is used as the function and/or some of the arguments, the expression denotes a *partial application*. Elements which are not '_' are evaluated immediately. The value of the partial application is a function with the same number of parameters as there are '_'s, which performs a tail-call to the appropriate combination of the previously computed values and the parameters supplied later.

A partial application may include at most one occurrence of '_...', which passes an arbitrary number of parameters to the eventual application.

If fun is preceded by '~', the negation is applied to the result of the application. In the case of a partial application it is the result of the resulting function which is negated, rather than the function itself.

Some other expression forms, whose evaluation begins with evaluating some of their arguments, also support multiple values at some argument positions, or partial application at some arguments. In particular '*name* = *value*' supports multiple values and partial application at *value*. All special expression operators support operator negation.

An expression '**ref** *app*', when *app* is an object application, means '*app* = _...'.

4.1.3 Literals

A literal always denotes the same value, given literally.

4.1.4 Lists

An expression ‘[arg^*]’ creates a list with the given elements. This expression supports multiple values and partial application.

Example: An expression ‘[x...]’ has the same value as x, except that it ensures that x is a list. □

4.1.5 Local definitions

A sequence of ‘;’-separated definitions, optionally followed by ‘;’, with at least one ‘;’, denotes an expression which introduces a local scope and enters the definitions.

4.1.6 Unnamed functions

An expression ‘?$param^*$ {*body*}’ denotes a function of type UNNAMED_FUNCTION. When the function is applied, it introduces a local scope, matches arguments of the application against patterns $param^*$, and enters definitions in *body*. If arguments are rejected by parameter patterns, the function fails with ‘BadArguments arguments Null min max’, where min and max are the bounds of acceptable arity (Null means no limit).

An expression:

```
function [
   param_i* {body_i}
   ...
]
```

denotes a function which introduces a local scope and matches the arguments against $param_i^*$ in order. As soon as an accepting case is found, definitions in the corresponding $body_i$ are entered. If every case rejects the arguments, the function fails as above.

In a **function** expression, the name **again** inside parameters and bodies is bound to the function itself.

If ‘*label*:’ is added before the ‘[’, the name *label* is used instead of **again**.

Other forms:

function ?$param^*$ {*body*} ≡
  function [$param^*$ {*body*}]
function *label*:?$param^*$ {*body*} ≡
  function *label*:[$param^*$ {*body*}]

4.1.7 Operators which denote function application

Some operators denote applications of particular standard functions:

$dict$@key	$\equiv$	'@' $dict\ key$
$dict$@$key = value$	$\equiv$	'@' $dict\ key\ value$
~$expr$	$\equiv$	'~' $expr$
$expr_1$ * $expr_2$	$\equiv$	'*' $expr_1\ expr_2$
$expr_1$ / $expr_2$	$\equiv$	'/' $expr_1\ expr_2$
/$expr_2$	$\equiv$	'/' $expr_2$
$expr_1 + expr_2$	$\equiv$	'+' $expr_1\ expr_2$
$expr_1 - expr_2$	$\equiv$	'−' $expr_1\ expr_2$
$-expr_2$	$\equiv$	'−' $expr_2$
$expr_1 == expr_2$	$\equiv$	'==' $expr_1\ expr_2$
$expr_1$ ~= $expr_2$	$\equiv$	'~=' $expr_1\ expr_2$
$expr_1 < expr_2$	$\equiv$	'<' $expr_1\ expr_2$
$expr_1 > expr_2$	$\equiv$	'>' $expr_1\ expr_2$
$expr_1 <= expr_2$	$\equiv$	'<=' $expr_1\ expr_2$
$expr_1 >= expr_2$	$\equiv$	'>=' $expr_1\ expr_2$
$first \setminus rest$	$\equiv$	'\\\\' $first\ rest$
$left, right$	$\equiv$	',' $left\ right$

The expression '*dict*@*key* = *value*' supports multiple values at *value*, and '*first* \ *rest*' supports multiple values at *first*. All of the above operators except '~*expr*' support partial application at all arguments. '**ref** *dict*@*key*' means '**ref** ('@' *dict key*)'.

The '@' operator is a generic function. Its 2-argument methods are expected to implement dictionary lookup and sequence indexing, and 3-argument methods implement setting/adding a value at a particular key in a dictionary, or setting an element of a mutable sequence.

The '~' function is defined in section 5.2 on page 32.

Arithmetic functions are defined in section 5.11.2 on page 42 and in section 5.11.3 on page 49.

The '\\' operator puts all arguments except the last one in a list, and appends the value of the last argument which must be a list.

The ',' function is defined in section 5.13 on page 63.

4.1.8 Conditionals

An expression:

if [

$cond_i$ $\{body_i\}$
…
]

evaluates $cond_i$ in order. As soon as a true condition is found, a local scope is introduced and definitions in the corresponding $body_i$ are entered. If all conditions are false, **if** fails with AllConditionsFalse.

'_' used as a condition in **if** means True.

Other forms:

if $cond$ $\{then\}$ $\{else\}$	$\equiv$	**if** [$cond$ $\{then\}$ _ $\{else\}$]
if $cond$ $\{then\}$	$\equiv$	**if** $cond$ $\{then\}$ {}
$expr_1$ & $expr_2$	$\equiv$	**if** $expr_1$ $\{expr_2\}$ {False}
$expr_1 \mid expr_2$	$\equiv$	**if** $expr_1$ {True} $\{expr_2\}$

The forms '**if** $cond$ $\{then\}$ $\{else\}$' and '**if** $cond$ $\{then\}$' support partial application at $cond$.

4.1.9 Case selection

An expression:

case $subj^*$ [
pat_i^* $\{body_i\}$
…
]

evaluates $subj^*$ and matches their values against pat_i^* in order. As soon as an accepting case is found, definitions in the corresponding $body_i$ are entered. If every case rejects $subj^*$, **case** fails with 'NoMatch subjects'.

Other forms:

case $subj^*$?pat^* $\{then\}$ $\{else\}$ $\equiv$
case $subj^*$ [pat^* $\{then\}$ _... $\{else\}$]
case $subj^*$?pat^* $\{then\}$ $\equiv$
case $subj^*$?pat^* $\{then\}$ {}

4.1.10 Loop

An expression:

loop arg^* [
pat_i^* $\{body_i\}$
…

]

is similar to the corresponding **case** expression, except that the name **again** inside patterns and bodies is bound to the function which evaluates the **loop** again with the arguments of the function instead of arg^*.

If '*label*:' is added before the '[', the name *label* is used instead of **again**.

Other forms:

loop arg^* ?pat^* {$body$}	≡	**loop** arg^* [pat^* {$body$}]
loop arg^* $label$:?pat^* {$body$}	≡	**loop** arg^* $label$:[pat^* {$body$}]

Example: Write elements of a list in a human-friendly form:

```
case list [
   [] {}
   [x] {Write x}
   [x y] {Write x "␣and␣" y}
   _ {
      loop list [
         [x] {Write "and␣" x}
         (x\xs) {Write x ",␣"; again xs}
      ]
   }
];
```

□

4.1.11 Local rebinding

An expression '**local** $reference$ *value body*' means '(**ref** $reference$) *value body*'.

Note: Usually $reference$ is a dynamic variable, and *body* is a nullary function. In this case the expression evaluates *body* with this $reference$ locally rebound to a new cell with this initial *value*. □

4.1.12 Exception catching

An expression:

try fun
   pat_i {$body_i$}
   ...
   done

evaluates 'fun()'. If it succeeds, *done* is tail-called with the result. Otherwise, if it fails, the exception is matched against pat_i in order. As soon as an accepting case is found, definitions in the corresponding *body*$_i$ are executed. If every case rejects the exception, **try** fails with the same exception.

If *done* is omitted, Identity is assumed.

During matching of *pat*$_i$ and evaluating *body*$_i$, signals are blocked once more than outside **try**.

If 'handled:' is added before '{*body*$_i$}', the signal blocking state is restored to the state outside **try** before entering *body*$_i$. A *body*$_i$ is in a tail position with respect to **try** if it is marked as 'handled:'.

4.1.13 Signal handling

An expression:

```
handle [
    pat_i {body_i}
    ...
] body
```

means:

```
local Signal (let super = Signal; function [
    pat_i {body_i}
    ...
    signal {super signal}
]) body
```

An expression:

```
handle [
    pat_i {body_i}
    ...
]
```

means:

```
Signal = (let super = Signal; function [
    pat_i {body_i}
    ...
    signal {super signal}
])
```

In either case **function** does not make the name **again** visible in patterns and bodies.

Other forms:

handle ?*param** {*body*$_1$} *body*$_2$	≡	**handle** [*param** {*body*$_1$}] *body*$_2$
handle ?*param** {*body*$_1$}	≡	**handle** [*param** {*body*$_1$}]

4.1.14 Unnamed structures

An expression '**struct** $\{defs\}$' introduces a local scope, executes the definitions, and evaluates to an object of type STRUCT with fields being public references introduced by the definitions.

The object is created as an incomplete object at the beginning, and is available in its own scope under the name **this**. Applying an incomplete object fails with 'IncompleteObject type'. When the definitions succeed, the object gets completed.

4.2 Patterns

4.2.1 Pattern lists

In some syntactic forms a list of values is matched against a list of patterns.

At most one pattern may be followed by '...'. If there is no such pattern, the number of values must be the same as the number of patterns, and then the values are matched against the corresponding patterns.

If there is a pattern of the form 'pat_i...', the number of values must be greater than or equal to the number of the other paterns in order for the patterns to match. Other patterns correspond to single values taken from the beginning and the end of the list, according to the pattern positions, while pat_i corresponds to the list of the remaining values.

4.2.2 References

A pattern *name* binds *name* to this value.

A pattern '_' binds no names.

A pattern '**ref** *name*' binds *name* to the value itself, treated as a reference.

A pattern '**var** *name*' binds *name* to a new mutable variable with this initial value.

A pattern '**dynamic** *name*' binds *name* to a new dynamic variable with this initial value of the default cell.

4.2.3 Visibility

A pattern '**private** *pat*' means *pat*, except that names it binds are marked as private, unless overridden by inner **public** declarations. Similarly, '**public** *pat*' marks names as public.

Forms which bind a name also accept '**private** *name*' and '**public** *name*' which override the visibility.

4.2.4 Predicates

The syntax of object application, 'fun arg^+', is interpreted specially when fun is an identifier belonging to a fixed set of special pattern operators: **define**, **dynamic**, **if**, **ifDefined**, **match**, **maybe**, **ofType**, **private**, **public**, **ref**, **set**, **var**, **where**, **with**.

Otherwise '$pat_1 \rightarrow fun$ arg^*' matches the subject against pat_1. Then the subject gets accepted if the result of the application of fun to the subject and arg^* is true.

A pattern '$pat_1 \rightarrow$**if** *cond*' matches the subject against pat_1. Then the subject gets accepted if *cond* is true.

4.2.5 Sequencing several matches

A pattern 'pat_1 & pat_2' matches the subject against pat_1, and then against pat_2.

A pattern '$pat_1 \rightarrow$**match** $subj_2$ pat_2' matches the subject against pat_1, and then matches $subj_2$ against pat_2.

A pattern '$pat_1 \rightarrow$**where** fun arg^* pat_2' matches the subject against pat_1, and then matches the result of the application of fun to the subject and arg^* against pat_2.

A pattern '$pat_1 \rightarrow$**maybe** fun arg^* pat_2' means:
'$pat_1 \rightarrow$**where** fun arg^* {[]} [_] (pat_2_)'.

Note: This relies the convention of some functions, such as Get, returning an optional result by accepting a parameter list ending with absent present, and entering 'absent()' or 'present result' depending on whether the result is present. In this case the result is matched against pat_2 if present. □

4.2.6 Checking types

A pattern '$pat \rightarrow$**ofType** $expr^+$' matches the subject against pat. Then, if '$subject \rightarrow$HasType $expr^+$' is False, matching fails with 'BadType $subject$ (',' $expr^+$)'.

4.2.7 Extracting components of compound values

A pattern '[pat^*]' accepts the subject if it is a list and its elements match patterns pat^*.

A pattern '$left$, $right$' accepts the subject if it is a pair and its elements match patterns $left$ and $right$.

A pattern:

pat_0→**with**
    $label_i$:pat_i
    . . .

matches the subject against pat_0, and then for each case matches the result of the application of the subject to the symbol #$label_i$ against pat_i.

4.2.8 Alternative matches

A pattern 'pat_1 | pat_2' matches the subject against pat_1. If it is rejected, it is matched against pat_2.

The whole pattern binds names which are common to pat_1 and pat_2. A common name is statically treated as a constant if it was a constant in both branches.

4.3 Definitions

4.3.1 Assignment

A definition '*app* = *value*', when *app* is a syntactic application, means *app* with *value* appended as the last argument. This substitution is performed before other rules.

4.3.2 Statements

A definition is interpreted specially if it has the syntax of object application, '*fun* arg^+', and *fun* is an identifier belonging to a fixed set of special definition operators: **def**, **defined**, **dynamic**, **export**, **extend**, **feature**, **forward**, **ifDefined**, **include**, **lazy**, **let**, **private**, **public**, **reexport**, **subtypes**, **type**, **use**, **var**.

Otherwise the definition is a statement. It is interpreted as an expression to be executed, its value is ignored, and it binds no names.

4.3.3 References

A definition '**let** *pat* = *value*' matches *value* against *pat*. If it is rejected, the definition fails with 'NoMatch [value]'. This form violates the left-to-right evaluation rule: *value* is evaluated before subexpressions of *pat*.

A definition '**dynamic** *name*' creates a dynamic variable which is initially not associated with any cell: in places where it is not locally rebound, getting or setting its value fails with 'NotDefined variable', where variable is the symbol of *name*.

A definition '**lazy** *name* = *expr*' creates a lazy variable. Lazy variables support getting the value only. Getting it for the first time evaluates *expr*, then stores and returns its value, or stores and propagates the exception if *expr* failed. Getting the lazy variable later returns the stored value immediately or fails immediately. Getting it while another thread is currently evaluating it blocks the getting thread until the outcome becomes known. Getting it while the same thread is currently evaluating it fails immediately with RecursiveLock. The dynamic environment of *expr* corresponds to the definition of the variable, rather than to the place of evaluating *expr* for the first time.

A definition '**forward** *name*' creates a forward variable. Its value can only be set once; an attempt to set it again fails with 'AlreadyDefined variable', where variable is the symbol of *name*. Until it becomes set, an attempt to get its value fails with 'NotDefined variable'.

Other forms:

ref *name* = *value*	≡	**let** (**ref** *name*) = *value*
var *name* = *value*	≡	**let** (**var** *name*) = *value*
var *name*	≡	**var** *name* = Null
dynamic *name* = *value*	≡	**let** (**dynamic** *name*) = *value*

4.3.4 Named functions

A definition:

```
def name [
    param*_i {body_i}
    ...
]
```

binds *name* to a function of type NAMED_FUNCTION. When the function is applied, it introduces a local scope and matches the arguments against $param_i^*$ in order. As soon as an accepting case is found, definitions the corresponding $body_i$ are entered. If every case rejects the arguments, the function fails with 'BadArguments arguments function min max', where function is the symbol of *name*, and min and max are the bounds of acceptable arity (Null means no limit).

The name **again** inside parameters and bodies is bound to the function itself.

If '*label:*' is added before the '[', the name *label* is used instead of **again**.

Cases where some of $param_i^*$ end with '!' are generic cases, and the marked parameters are the keys. A generic case must have an empty $body_i$, and if it includes a parameter ending with '...', all the keys must precede it.

Other forms:

$$\textsf{\textbf{def}}\ \mathit{name}\ \mathit{param}^*\ \{\mathit{body}\} \quad \equiv \quad \textsf{\textbf{def}}\ \mathit{name}\ [\mathit{param}^*\ \{\mathit{body}\}]$$

4.3.5 Methods

A definition:

method *function* [
  pat_i^* {$body_i$}
  ...
]

adds a method to a generic function.

The *function* expression must evaluate to a named function. There must be at least one case. Some of the patterns in the first case may have the form '$pat^\frown type$', where the *type* expression must evaluate to a tuple of types. These patterns are the keys. If the first case includes a parameter ending with '...', all the keys must precede it.

If some of the key types are tuples longer than one element, then a method with the same definition is added for every combination of types taken from these tuples.

The generic case to which the method is added is the first generic case of *function* which matches the common arity of the method. The common arity is a fixed arity if all method cases have the same fixed arity, or the lower bound of arities accepted by all method cases otherwise.

A matching arity means that all lengths of the argument list which could match the method arity must also match the generic case arity. If no matching generic case is found, **method** fails with 'NoGeneric function arity', where function is the symbol of the generic function, and a negative arity encodes the arity being at least $-1-$arity.

The position of each key of the method definition must correspond to a key in the generic case, otherwise **method** fails with 'NotDispatched function arity parameter', where parameter is the parameter position, counting from 0. If a key in the generic case does not correspond to a key in the method definition, OBJECT is assumed as the type.

If the generic case already contains a method with the same sequence of key types, **method** fails with 'MethodConflict function arity types', where types is a list of key types.

If a generic function is being applied and the best applicable method is found at supertypes of the types of the key arguments, it is added at exact types of the key arguments, and thus other methods may conflict with a method added this way.

The name **again** inside parameters and bodies is bound to the method itself.

The name **super** inside parameters and bodies is bound to a function which tail-calls the next applicable method of the same generic case for the arguments of **super**, i.e. the best applicable method which is worse than the method being defined. If there is no next applicable method, **super** fails with 'NoSuperMethod function arity arguments after', where after is a list of the key types of the method being defined. If the method being defined would not be an applicable method for the arguments of **super**, it must have been either because the arity does not match, in which case **super** fails with 'BadArguments arguments function min max', where min and max are the bounds of acceptable arity (Null means no limit), or because some of the key types of the method being defined is not a supertype of the corresponding key argument of **super**, in which case **super** fails with 'NotSupertype subtype supertype'.

Other forms:

method *function param** {*body*} ≡
method *function* [*param** {*body*}]

4.3.6 Types

Below *arg** is a list of arguments of one of the following forms:

- 'is:*type*', where *type* is an expression which evaluates to a type
- 'final:*bool*', where *bool* is an expression which evaluates to a bool

There must be at most one 'final:*bool*' argument, which specifies whether the type is final. The default is False for abstract types, and True for nonabstract types.

A scope is optionally associated with an *implicit supertype*, absent by default.

When a new type is defined, its direct supertypes include:

1. the optional implicit supertype
2. the explicit supertypes (specified with is:*type* arguments)
3. an optional extra supertype, depending on the definition syntax

An attempt to declare a subtype of a final type fails with ‘TypeIsFinal type’.

A definition:

type *name arg**

creates an abstract type named *name*.

A definition:

type *name arg** {*defs*}

creates an abstract type named *name*. It is set as the implicit supertype, definitions from *defs* are executed, and the implicit supertype is restored to its previous state. If the type is specified as final, its final status is applied after *defs* succeed.

A definition:

subtypes *type* {*defs*}

evaluates *type* as an expression, and its value must be a type. It is set as the implicit supertype, definitions from *defs* are included, and the implicit supertype is restored to its previous state.

A definition:

type *name arg** *con*

where *con* is a name, creates a singleton type named *name*, with a single value named *con*. The extra supertype is SINGLETON. This definition implicitly defines a method:

method SingletonName _^*name* {*symbol*};

where *symbol* is the symbol of *con*.

A definition:

type *name arg** *con field**

where *con* and *field*s are names, creates a *record type* named *name*, with a constructor function named *con*. Objects of this type contain fields with the specified names. When the constructor is applied, it returns an object of this type, with fields given by arguments of the constructor. A record type is a value type, which means that the constructor is allowed to return some existing object with the given value. The extra supertype is RECORD. This definition implicitly defines methods:

method RecordConstructor _^*name* {*con*};
method RecordFields _^*name* {[*symbol**]};

where *symbol** are symbols of *field**.

A definition:

type $name$ arg^* con $\{defs\}$

where con is a name, creates a singleton type named $name$, with a single value named con with explicitly specified behavior. A new scope is introduced, definitions from $defs$ are executed, and public references they introduce become fields of the object.

A definition:

type $name$ arg^* con [
    $param_i^*$ $\{defs_i\}$
    $\ldots$
]

where con is a name, creates an type with a constructor function named con. When the constructor is applied, it introduces a local scope, matches arguments against $param_i^*$ like in **def**, and executes definitions from the appropriate $defs_i$. Public reference introduced by the matching $param_i^*$ and $defs_i$ become fields of the object.

The object of the type being defined is created as an incomplete object before executing its definitions, and is available in its own scope under the name **this**. Applying an incomplete object fails with 'IncompleteObject type'. When the definitions succeed, the object gets completed.

Other forms (in the second case $param^*$ must not be all unqualified names, because it would mean a record type):

type $name$ arg^* con $param^*$ $\{defs\}$ $\equiv$
  type $name$ arg^* con $[param^*$ $\{defs\}]$
type $name$ arg^* con $param^*$ $\equiv$
  type $name$ arg^* con $param^*$ $\{\}$

4.3.7 Visibility

A definition '**private** $\{defs\}$' executes $defs$, except that names they bind are marked as private, unless overridden by inner **public** declarations. Similarly, '**public** $\{defs\}$' marks names as public.

5 Prelude

The Prelude module is imported implicitly to each module which does not import it explicitly in its first definition after the module header.

In the descriptions below '$\cdots$' replaces implementation details, which are often primitive constructs not expressible in terms of others.

By convention type names are written with all caps, with words separated by "_", and the constructor of a type has the same name as the type, but written with capitalized words without separators. Definitions complying to this naming convention are not explained in this respect.

5.1 Null

Unless specified otherwise, a function which is executed for side effects and has no interesting result returns Null. Null is also used as a general mark of the absence of a more meaningful value.

```
type NULL Null;
```

5.2 Booleans

When a value is interpreted as a condition, True is considered true, and False is false. Any other value is an error.

```
type BOOL final:True {
    type TRUE True;
    type FALSE False;
};

def '~' x {if x {False} {True}};
```

5.3 Equality

```
def IsSame x y {···};
```

IsSame is called *physical equality*, and tests if its arguments point to the same object, or in other words compares object identity.

```
def Is [
    x (y→if (x %IsSame y)) {True}
    x (y→if (Type x ~%IsSame Type y)) {False}
    x! y {}
];
method Is _ _ {False};
```

Is is called *strict equality*, and by convention tests if its arguments represent the same abstract value.

For many purposes this is the smallest useful equivalence relation. For object types this is the same relation as physical equality, which is also the default for types which do not specialize it themselves. Any way of distinguishing objects which are not strictly equal usually involves physical equality.

Strict equality on floating point numbers differs from arithmetic equality: 0.0 ~%Is (−0.0) & Float Null %Is Float Null.

Note: Since Is dispatches to the generic case only after checking that the arguments are not physically equal, a method like **method** Is _^X _ {False} means that values of type X are strictly equal whenever they are physically equal. □

5.4 Hashing

```
def Hash obj! {};
let MaxHash = 1→BitShift (···) − 1;
def CombineHash x y {···};
```

Hash returns an integer corresponding to the given value, which is used by the implementation of some dictionaries and sets to quickly partition values into separate buckets. Values which are strictly equal must yield equal hashes, and values which are not strictly equal are likely to yield different hashes. Objects which suport Is do not necessarily support Hash.

MaxHash and CombineHash are defined for convenience of writing custom hashing functions. Hash values will be BitAnd'ed with MaxHash; Hash is allowed to return any integer, but bits above MaxHash will be discarded. CombineHash can be used to compute a hash of a structured value from hashes of its components.

5.5 Ordering

```
def '==' x! y! {};
def '~=' x y {~(x == y)};
def '<' x! y! {};
def '>' x y {y < x};
def '<=' x! y! {};
method '<=' x y {~(y < x)};
def '>=' x y {y <= x};
```

These relations define total orders (with exceptions) over those subsets of values which traditionally have a natural total order defined. Numbers are compared in the standard way, with standard floating point peculiarities: 0.0 == −0.0 & Float Null ~= Float Null. Characters are compared by code points, pairs and sequences are compared lexicograpically.

'==' is called *arithmetic equality.* It is generally compatible with '<' and '<=', except that complex numbers have '==' defined but not '<' nor '<=', along with the mathematical tradition.

Numbers of different types which represent the same mathematical values compare as arithmetically equal, even though they are not strictly equal.

```
def Min [
    x {x}
    x! y! {}
    x y zs... {Min x y→Min zs...}
];
method Min x y {if (y < x) {y} {x}};
def Max [
    x {x}
    x! y! {}
    x y zs... {Max x y→Max zs...}
];
method Max x y {if (y < x) {x} {y}};
```

By convention, Min returns the smallest of the arguments, and Max returns the largest, except that for numbers they behave like an arithmetic operation regarding mixed-type arguments, −0.0, and Float Null.

```
def IsInRange [
    x max {again x 0 max}
    x min max {x >= min & x < max}
];
def IsBetween x min max {x >= min & x <= max};
```

5.6 Freeing resources

```
def Close obj! {};
method Close _ {};
```

Close frees resources associated with the given object if there are resources which can be freed explicitly, and depending on the type it may perform other actions which are usually performed as the last thing to be done with the object. This may render the object unusable, possibly with other objects which are considered parts of this object.

By default Close does nothing. By convention it is safe to use Close several times on the same object.

5.7 Simple functions

```
def Identity x {x};

def Ignore _... {};

def Apply f xs... {f xs...};
```

5.8 Registered lists

A registered-list is a sequence of values which supports unregistering values using a key obtained at their registration.

```
type REGISTERED_LIST is:SEQUENCE RegisteredList() {···};
type REGISTERED_KEY ···;

def Register list! value {};
method Register list^REGISTERED_LIST value {···};
```

By convention, 'Register list value' inserts value at the beginning of list, and returns a key (which has type REGISTERED_KEY in the case of a registered-list).

```
method Close key^REGISTERED_KEY {···};
```

The Close method of a registered-key unregisters the associated occurrence of the value from the appropriate registered-list.

A registered-list can be traversed using standard functions for traversing sequences.

A registered-list can be modified while it is being traversed, possibly from another thread. In this case traversal uses a snapshot of the contents from the start of the traversal.

5.9 Exceptions

```
def Fail exn {···};
```

'Fail exn' fails with exception exn.

The main categories of exceptions can be distinguished with supertypes:

type PROGRAM_ERROR;
type EXTERNAL_ERROR;
type RESOURCE_ERROR;
type EXIT;
type SPECIAL_RESULT;

Program errors are manifestations of bugs in the program. A well-written program should prevent them before they happen.

External errors reflect failures reported by the operating system or bad data sent from outside of the program.

Resource errors mean that the system is unable to provide enough resources for the program to continue, possibly because the data given to the program are too large.

Exits are exceptions used to abort a subcomputation.

Special results allow a callback function to communicate a special condition to its caller, if any object returned as the normal result is understood in some regular way. They are generally caught immediately after they are thrown.

5.9.1 Program errors

subtypes PROGRAM_ERROR {

type BAD_TYPE BadType object type;

object was found when a value of the given type was expected.

type NOT_A_FUNCTION NotAFunction object;

Applying object to arguments is meaningless.

type NO_FIELD NoField object field;

object does not have a field with name field (a symbol).

type NO_MATCH NoMatch objects;

objects (a list) match no pattern(s) in a case matching.

type ALL_CONDITIONS_FALSE AllConditionsFalse;

All conditions of an **if** are False.

type BAD_ARGUMENTS BadArguments arguments function min max;

Bad arguments have been passed to function (a symbol or Null); the function expects at least min and at most max arguments (Null means no limit).

type NOT_SETTABLE NotSettable object field;

field (a symbol) of object can only be read, not written.

type UNKNOWN_KEYWORDS UnknownKeywords keywords;

These keywords (a list of symbols) are unknown to the callee.

type NOT_DEFINED NotDefined name;

Definition of this name (a symbol) has not been reached yet.

type ALREADY_DEFINED AlreadyDefined name;

name can be assigned only once.

type LOCK_NOT_LOCKED LockNotLocked;

An attempt to unlock a lock which has not been locked by the current thread.

type INCOMPLETE_OBJECT IncompleteObject type;

An attempt to use an object of this type before its definition completed.

type RECURSIVE_LOCK RecursiveLock;

A thread tries to take the same lock again.

type DEADLOCK Deadlock;

No threads can run.

type OBJECT_CLOSED ObjectClosed type;

An attempt to use an object of this type after it has been closed.

type OBJECT_LOCKED ObjectLocked type;

An attempt to use an object of this type while it is in use by another entity

type SIGNALS_UNBLOCKED SignalsUnblocked;

An attempt to unblock signals when there are already unblocked.

type ALREADY_EXITED AlreadyExited;

An attempt to exit from a scope which has already completed.

type BAD_ARGUMENT BadArgument value what range;

value, denoting what (a description string), should have the form of range (a description string).

type OUT_OF_RANGE OutOfRange value what min max;

value, denoting what (a description string), should be in the range between min and max (inclusive).

type NOT_SINGLE_CHAR NotSingleChar value;

value is a string with a length other than 1.

type TIME_OUT_OF_RANGE TimeOutOfRange value;

value denotes time too far in the past or future.

type ARITHMETIC_ERROR ArithmeticError function arguments;

arguments (a list) do not belong to the domain of this arithmetic function (a symbol).

type EMPTY_COLLECTION EmptyCollection;

A collection is empty.

type INDEX_OUT_OF_RANGE IndexOutOfRange index size;

An attempt to index a sequence using this index, while the sequence has only this size.

type NEGATIVE_SIZE NegativeSize size;

size was specified as a size, but it is negative.

type BAD_SIZE BadSize size expected;

A collection has this size instead of expected size.

type KEY_NOT_FOUND KeyNotFound key;

key is not in a dictionary.

type KEY_CONFLICT KeyConflict key;

key is already present in a dictionary.

type INCONSISTENT_SUPERTYPES InconsistentSupertypes type;

Linearization of supertypes of type is impossible because some supertypes include the same super-supertypes in a different order.

type NOT_SUPERTYPE NotSupertype subtype supertype;

The type given as supertype is not a supertype of subtype.

type TOP_SUPERTYPE TopSupertype;

There can be no supertypes of the OBJECT type.

type NO_METHOD NoMethod function arity arguments;

The generic case with this arity of this function (a symbol) is not defined for the key arguments. A negative arity encodes the arity being at least $-1-$arity.

type NO_SUPER_METHOD NoSuperMethod function arity arguments after;

The generic case with this arity of this function (a symbol) does not have a next method for the key arguments after types given as after (a list of types). A negative arity encodes the arity being at least $-1-$arity.

type NO_GENERIC NoGeneric function arity;

This generic function (a symbol) does not have a generic case of this arity. A negative arity encodes the arity being at least $-1-$arity.

type NOT_DISPATCHED NotDispatched function arity parameter;

The generic case with this arity of this function (a symbol) is not dispatched on this parameter position, counting from 0. A negative arity encodes the arity being at least $-1-$arity.

type METHOD_CONFLICT MethodConflict function arity types;

The generic case with this arity of this function (a symbol) has been already defined for these types (a list of types). A negative arity encodes the arity being at least $-1-$arity.

type IMPOSSIBLE_ERROR ImpossibleError message;

Program has reached a place which was thought to be unreachable; message explains the reason.

};

5.9.2 External errors

subtypes EXTERNAL_ERROR {

type IO_ERROR IOError code;

The operating system has reported an I/O failure. On Unix code is an `errno` value.

type END_OF_STREAM EndOfStream;

An unexpected end of stream has been reached.

};

5.9.3 Resource errors

subtypes RESOURCE_ERROR {

type NOT_SUPPORTED NotSupported what;

A functionality described by what is not supported in the current environment.

type OUT_OF_MEMORY OutOfMemory;

The program might soon run out of memory.

type STACK_OVERFLOW StackOverflow;

The current execution stack is too deep and might soon run out of memory.

type ARITHMETIC_OVERFLOW ArithmeticOverflow;

An arithmetic result is too big.

};

5.9.4 Exits

type PROGRAM_EXIT ProgramExit status;

A request to exit the program with the given status communicated to the process which has started the program (an integer in the range 0 to 255; often 0 indicates success and other values indicate errors).

subtypes EXIT {

type THREAD_EXIT ThreadExit;

A request to abort the current thread.

type THREAD_EXIT_AT_FORK_PROCESS ThreadExitAtForkProcess;

A request to abort the current thread because it should not exist in a child process.

type THREAD_KICKED ThreadKicked;

The current thread was unreachable and has been kicked in order to finish cleanly.

type THREAD_KILLED ThreadKilled;

The thread had to disappear in a child process for technical reasons; this is an exception reported as the exception it failed with.

```
type SYSTEM_SIGNAL SystemSignal code;
```

Represents a signal sent between processes by the operating system. On Unix code is the signal number.

```
};
```

5.10 Exiting the program

```
def ExitProgram [
   () {again 0}
   status {Fail (ProgramExit status)}
];
```

ExitProgram requests the program to exit by failing with a PROGRAM_EXIT exception.

```
def ExitProgramNow [
   () {again 0}
   status {···}
];
```

ExitProgramNow aborts the process without any cleanup.

When an exception leaves the toplevel of the program, some exception types are treated specially:

- PROGRAM_EXIT sets the status code that the process will exit with.
- SYSTEM_SIGNAL interrupts the process by an operating system signal given by the exception if possible.

Exceptions of other types are passed to the unhandled exception handler:

```
let AtUnhandledException = RegisteredList();
```

The first element of AtUnhandledException is removed and applied to the exception. The handler should return an integer being the status to exit the process with. If it fails, the next handler has a chance. At the end of the list there is a default handler supplied by the implementation which presents the exception to the user. It may also present a stack trace.

```
type SOURCE_LOC SourceLoc file row column context;
```

```
def StackTrace() {···};
```

StackTrace extracts a trace of source locations of the called functions, if supported by the implementation. This information is provided for debugging purposes and may not be exact. StackTrace returns a list of events, ordered chronologically, where every element has one of the following forms:

- loc:sourceLoc — this source location was visited; a source location is represented by a value of the SOURCE_LOC type.
- exn:exception — execution failed with exception.

When the program completes, including a possible unhandled exception handler, program exit handlers are executed with signals blocked:

```
let AtExit = RegisteredList();
```

While AtExit is not empty, the first element of AtExit is removed and applied to no arguments. If it fails, the exception is ignored.

5.11 Numbers

5.11.1 Abstract types

Numeric types can have supertypes which indicate the domains of the numbers their objects represent. The supertypes allow for default implementations of various arithmetic operations, including arithmetic between numbers of mixed types.

```
type NUMBER;
type COMPLEX is:NUMBER;
type REAL is:COMPLEX;
type RATIONAL is:REAL;
type INTEGER is:RATIONAL;
```

REAL includes also infinities of both signs, and COMPLEX includes also numbers with an infinite real and/or imaginary part.

5.11.2 Integers

There is a standard type of integer numbers:

```
type INT is:INTEGER ···;
```

Note: Ints are not arbitrarily limited to the range of a machine word. □

```
def Int x! opts... {};
method Int x^INT {x};
```

By convention, Int converts an object to an int or an exact infinity. If it is a number, the real part of its value is truncated towards zero.

```
method Is x^INT y {···};
method Hash x^INT {···};

method '==' x^INT y^INT {x %Is y};
method '<' x^INT y^INT {···};
method '<=' x^INT y^INT {···};

method Min x^INT y^INT {if (y < x) {y} {x}};
method Max x^INT y^INT {if (y < x) {x} {y}};
```

Arithmetic operations below are also defined for some quantities which are not numbers.

```
def '+' [
   () {0}
   x {x}
   x! y! {}
   x y zs... {'+' (x+y) zs...}
];
method '+' x^INT y^INT {···};
```

By convention, adds the arguments.

```
def '−' [
   x! y! {}
   y! {}
];
method '−' x^INT y^INT {···};
method '−' y^INT {···};
```

By convention, subtracts two numbers or negates one number.

```
def Abs x! {};
method Abs x^REAL {if (x < 0) {−x} {x}};
```

By convention, returns the absolute value or magnitude of a number.

```
def Signum x! {};
method Signum x^REAL {
   if [
      (x > 0) {1}
```

```
        (x < 0) {−1}
        _ {x}
    ]
};
```

By convention, projects a number to the unit circle, i.e. divides it by its absolute value (except that Signum 0 == 0).

```
def '*' [
    () {1}
    x {x}
    x! y! {}
    x y zs... {'*' (x*y) zs...}
];
method '*' x^INT y^INT {···};
```

By convention, multiplies the arguments.

```
def Sqr x! {};
method Sqr x^COMPLEX {x * x};
```

By convention, squares a number.

```
def Quot x! y! {};
method Quot x^INT y^INT {···};

def Rem x! y! {};
method Rem x^INT y^INT {···};

def QuotRem [
    x {x}
    x! y! {}
    x d ds... {
        let (q, rs) = QuotRem x ds...;
        let (q', r') = QuotRem q d;
        q', r', rs
    }
];
method QuotRem x^INT y^INT {x %Quot y, x %Rem y};

def Div x! y! {};
method Div x^INT y^INT {···};

def Mod x! y! {};
method Mod x^INT y^INT {···};
```

```
def DivMod [
    x {x}
    x! y! {}
    x d ds... {
        let (q, rs) = DivMod x ds...;
        let (q', r') = DivMod q d;
        q', r', rs
    }
];
method DivMod x^INT y^INT {x %Div y, x %Mod y};
```

By convention, Quot and Div return the integer quotient of two real numbers, Rem and Mod return the remainder, and QuotRem and DivMod return both as a pair.

QuotRem and DivMod are extended to an arbitrary nonzero number of arguments, such that the first argument is divided by the second argument (the integer quotient is returned), the remainder is returned by the next argument (the integer quotient is returned) etc. The returned values are put in a tuple.

Two rounding modes of integer division are provided: Quot, Rem, and QuotRem round the quotient towards zero (the remainder has the sign of the dividend), and Div, Mod, and DivMod round the quotient down (the remainder has the sign of the divisor). Example:

		Quot	Rem	Div	Mod
123	10	12	3	12	3
−123	10	−12	−3	−13	7
123	−10	−12	3	−13	−7
−123	−10	12	−3	12	−3

```
def IsDivisible x! y! {};
method IsDivisible x^REAL y^REAL {x %Rem y == 0};
```

By convention, tests if the first real number is divisible by the second.

```
def IsEven x! {};
method IsEven x^INTEGER {x %Rem 2 == 0};
def IsOdd x {~IsEven x};
```

By convention, tests if an integer number is even or odd respectively.

```
def Floor [
    x! {}
    x! y! {}
];
method Floor x^REAL {x %Div 1};
method Floor x^REAL y^REAL {if (y == 0) {x} {x %Div y * y}};

def Ceiling [
    x! {}
    x! y! {}
];
method Ceiling x^REAL {−(−x) %Div 1};
method Ceiling x^REAL y^REAL {
    if (y == 0) {x} {−(−x) %Div y * y}
};

def Trunc [
    x! {}
    x! y! {}
];
method Trunc x^REAL {x %Quot 1};
method Trunc x^REAL y^REAL {if (y == 0) {x} {x %Quot y * y}};

def Round [
    x! {}
    x! y! {}
];
method Round x^REAL {
    let (q, r) = x %DivMod 1;
    let rr = r * 2;
    if (rr < 1 | rr == 1 & IsEven q) {q} {q + 1}
};
method Round x^REAL y^REAL {
    if (y == 0) {x} ⇒
    let y' = Abs y;
    let (q, r) = x %DivMod y';
    let rr = r * 2;
    if (rr < y' | rr == y' & IsEven q) {q * y'} {(q + 1) * y'}
};

Each [Floor Ceiling Trunc Round] ?f {
    method f x^INTEGER {x};
};
```

Four rounding functions of real numbers are provided. By convention, Floor rounds down, Ceiling rounds up, Trunc rounds towards zero, and Round rounds in whichever direction is closer, with the preference to the even quotient in the case of a tie.

A single argument is rounded to an integer. With two arguments the first argument is rounded to a multiple of the second one, except that if the second argument is 0, the first is returned unchanged.

```
def GCD [
   () {0}
   x {x}
   x! y! {}
   x y zs... {GCD x y→GCD zs...}
];
method GCD x^RATIONAL y^RATIONAL {
   loop x y [
      a 0 {a}
      a b {again b (a %Rem b)}
   ]→Abs
};
```

By convention, GCD returns the greatest common divisor of the arguments.

For the purposes of bit operations, integers are represented in binary, in the two's complement convention with an infinite width for negative integers.

```
def BitNot x! {};
method BitNot x^INT {···};
```

By convention, BitNot complements every bit of the argument.

```
def BitAnd [
   () {−1}
   x {x}
   x! y! {}
   x y zs... {BitAnd (x %BitAnd y) zs...}
];
method BitAnd x^INT y^INT {···};
```

By convention, BitAnd returns the bitwise conjunction of the arguments.

```
def BitAndNot x! y! {};
method BitAndNot x^INTEGER y^INTEGER {x %BitAnd BitNot y};
```

By convention, 'x %BitAndNot y' is 'x %BitAnd BitNot y'.

```
def BitOr [
   () {0}
```

```
    x {x}
    x! y! {}
    x y zs... {BitOr (x %BitOr y) zs...}
];
method BitOr x^INT y^INT {···};
```

By convention, BitOr returns the bitwise disjunction of the arguments.

```
def BitXor x! y! {};
method BitXor x^INT y^INT {···};
```

By convention, BitXor returns the bitwise exclusive disjunction of the arguments.

```
def BitShift x! shift {};
method BitShift x^INT shift {···};
```

By convention, 'x→BitShift shift' shifts x by the amount of bits specified by shift: left if shift > 0 (zero bits are shifted in), or right if shift < 0 (the bits shifted out are discarded).

```
def TestBit x! bit {};
method TestBit x^INT bit {(x %BitAnd 1→BitShift bit) ~%Is 0};
```

By convention, 'x %TestBit bit' returns True if the bit of x with the index bit is 1.

```
def SetBit x! bit {};
method SetBit x^INT bit {x %BitOr 1→BitShift bit};
```

By convention, x %SetBit bit returns x changed by setting the bit with the index bit to 1.

```
def ClearBit x! bit {};
method ClearBit x^INT bit {x %BitAndNot 1→BitShift bit};
```

By convention, 'x %ClearBit bit' returns x changed by setting the bit with the index bit to 0.

```
def SizeInBits x! {};
method SizeInBits x^INT {···};
```

By convention, 'SizeInBits x' returns the smallest number of bits needed to represent x (0 for 0, and for negative numbers 'SizeInBits x' is 'SizeInBits (BitNot x)').

```
def CountBits x! {};
method CountBits x^INT {···};
```

By convention, ‘CountBits x’ returns the count of set bits of x (Inf for negative numbers).

```
def FindBit [
    x absent {again x 0 absent Identity}
    x (begin→HasType INT) absent {again x begin absent Identity}
    x absent present {again x 0 absent present}
    x! begin absent present {}
];
method FindBit x^INT begin absent present {···};
```

By convention, ‘FindBit x begin absent present’ finds the index of the next set bit of x, starting the search from begin (the found index is begin if this bit is already set). It returns ‘present bit’ if the bit is found, or ‘absent()’ otherwise.

```
Each [
    '==' '<' '<=' Min Max '+' '−' '*' Quot Rem QuotRem Div Mod
    DivMod BitAnd BitOr BitXor
] ?f {
    method f x^INTEGER y^INTEGER {f (Int x) (Int y)};
};

Each ['−' BitNot SizeInBits CountBits] ?f {
    method f x^INTEGER {f (Int x)};
};

Each [BitShift TestBit SetBit ClearBit] ?f {
    method f x^INTEGER bit {Int x→f bit};
};

method FindBit x^INTEGER begin absent present {
  Int x0>FindBit beg8h absent present
};
```

5.11.3 Rationals

```
type RATIO is:RATIONAL (private RatioCon) num den {};
```

Ratios represent exact quotients num/den, where num and den are ints, they have no nontrivial common divisors, and den > 1.

```
def Ratio x! opts... {};
method Ratio x^(INT,RATIO) {x};
method Ratio x^INTEGER {Int x};
```

By convention, Ratio converts an object to a ratio, an int, or an exact infinity. If it is a number, the real part of its value is used.

```
method Int x^RATIO {x.num %Quot x.den};
method Int x^RATIONAL {Int (Ratio x)};

def Numerator x! {};
method Numerator x^INT {x};
method Numerator x^INTEGER {Int x};
method Numerator x^RATIO {x.num};
method Numerator x^RATIONAL {Numerator (Ratio x)};

def Denominator x! {};
method Denominator _^(INT,INTEGER) {1};
method Denominator x^RATIO {x.den};
method Denominator x^RATIONAL {Denominator (Ratio x)};
```

By convention, Numerator and Denominator return the numerator and denominator of the given rational number, such that they have no nontrivial common divisors, and the denominator is greater than 0.

```
def '/' [
   x! y! {}
   y! {}
];
```

By convention, '/' divides two numbers or returns the reciprocal of one number.

```
method '/' x^INT y^INT {
   if (y %Is 0) {Fail (ArithmeticError #'/' [x 0])} ⇒
   let (q, r) = x %QuotRem y;
   if (r %Is 0) {q} ⇒
   let d = GCD y r;
   let num = x %Quot d;
   let den = y %Quot d;
   if (den > 0) {RatioCon num den} {RatioCon (−num) (−den)}
};
method '/' y^INT {
   case y [
      (1 | −1) {y}
```

```
            0 {Fail (ArithmeticError #'/' [0])}
            _ {if (y > 0) {RatioCon 1 y} {RatioCon (−1) (−y)}}
        ]
    };
    method '/' x^INTEGER y^INTEGER {Int x / Int y};
    method '/' y^INTEGER {/Int y};
```

Division of ints returns an int or a ratio, depending on whether the result is a whole number.

```
    method Is x^RATIO y {x.num %Is y.num & x.den %Is y.den};
    method Hash x^RATIO {Hash x.num %CombineHash Hash x.den};

    method '==' _^INT _^RATIO {False};
    method '==' _^RATIO _^INT {False};
    method '==' x^RATIO y^RATIO {x.num == y.num & x.den == y.den};

    Each ['<' '<='] ?f {
        method f x^INT y^RATIO {f (x*y.den) y.num};
        method f x^RATIO y^INT {f x.num (y*x.den)};
        method f x^RATIO y^RATIO {f (x.num*y.den) (y.num*x.den)};
    };

    Each ['==' '<' '<='] ?f {
        method f x^REAL y^REAL {f (Ratio x) (Ratio y)};
    };

    method Min x^INT y^RATIO {if (y < x) {y} {x}};
    method Max x^INT y^RATIO {if (y < x) {x} {y}};
    method Min x^RATIO y^INT {if (y < x) {y} {x}};
    method Max x^RATIO y^INT {if (y < x) {x} {y}};
    method Min x^RATIO y^RATIO {if (y < x) {y} {x}};
    method Max x^RATIO y^RATIO {if (y < x) {x} {y}};

    method '+' x^INT y^RATIO {RatioCon (x * y.den + y.num) y.den};
    method '−' x^INT y^RATIO {RatioCon (x * y.den − y.num) y.den};
    method '+' x^RATIO y^INT {RatioCon (x.num + y * x.den) x.den};
    method '−' x^RATIO y^INT {RatioCon (x.num − y * x.den) x.den};
    method '+' x^RATIO y^RATIO {
      (x.num * y.den + y.num * x.den) / (x.den * y.den)
    };
    method '−' x^RATIO y^RATIO {
      (x.num * y.den − y.num * x.den) / (x.den * y.den)
    };
```

```
method '−' y^RATIO {RatioCon (−y.num) y.den};

method '*' x^INT y^RATIO {(x * y.num) / y.den};
method '/' x^INT y^RATIO {(x * y.den) / y.num};
method '*' x^RATIO y^INT {(x.num * y) / x.den};
method '/' x^RATIO y^INT {x.num / (x.den * y)};
method '*' x^RATIO y^RATIO {(x.num * y.num) / (x.den * y.den)};
method '/' x^RATIO y^RATIO {(x.num * y.den) / (x.den * y.num)};

method '/' y^RATIO {
    case y.num [
        1 {y.den}
        (−1) {−y.den}
        (>0) {RatioCon y.den y.num}
        _ {RatioCon (−y.den) (−y.num)}
    }
};

method Quot x^INT y^RATIO {(x * y.den) %Quot y.num};
method Div x^INT y^RATIO {(x * y.den) %Div y.num};
method Quot x^RATIO y^INT {x.num %Quot (x.den * y)};
method Div x^RATIO y^INT {x.num %Div (x.den * y)};
method Quot x^RATIO y^RATIO {(x.num * y.den) %Quot (x.den * y.num)};
method Div x^RATIO y^RATIO {(x.num * y.den) %Div (x.den * y.num)};

method Rem x^INT y^RATIO {((x * y.den) %Rem y.num) / y.den};
method Mod x^INT y^RATIO {((x * y.den) %Mod y.num) / y.den};
method Rem x^RATIO y^INT {(x.num %Rem (x.den * y)) / x.den};
method Mod x^RATIO y^INT {(x.num %Mod (x.den * y)) / x.den};
method Rem x^RATIO y^RATIO {
  ((x.num * y.den) %Rem (x.den * y.num)) / (x.den * y.den)
};
method Mod x^RATIO y^RATIO {
    ((x.num * y.den) %Mod (x.den * y.num)) / (x.den * y.den)
};

method QuotRem x^INT y^RATIO {
    let (q, r) = (x * y.den) %QuotRem y.num;
    q, r / y.den
};
method DivMod x^INT y^RATIO {
    let (q, r) = (x * y.den) %DivMod y.num;
    q, r / y.den
};
method QuotRem x^RATIO y^INT {
    let (q, r) = x.num %QuotRem (x.den * y);
    q, r / x.den
};
method DivMod x^RATIO y^INT {
    let (q, r) = x.num %DivMod (x.den * y);
    q, r / x.den
};
method QuotRem x^RATIO y^RATIO {
    let (q, r) = (x.num * y.den) %QuotRem (x.den * y.num);
    q, r / (x.den * y.den)
};
method DivMod x^RATIO y^RATIO {
    let (q, r) = (x.num * y.den) %DivMod (x.den * y.num);
    q, r / (x.den * y.den)
};

method Floor x^RATIO {x.num %Div x.den};
method Ceiling x^RATIO {−(−x.num) %Div x.den};
method Trunc x^RATIO {x.num %Quot x.den};
method Round x^RATIO {
    let (q, r) = x.num %DivMod x.den;
    let rr = r * 2;
    if (rr < xden | rr == xden & IsEven q) {q} {q + 1}
};

Each [Min Max '+' '−' '*' '/' Quot Rem QuotRem Div Mod DivMod] ?f {
    method f x^INTEGER y^RATIO {f (Int x) y};
    method f x^RATIO y^INTEGER {f x (Int y)};
    method f x^RATIONAL y^RATIONAL {f (Ratio x) (Ratio y)};
};

Each ['−' '/'] ?f {
    method f y^RATIONAL {f (Ratio y)};
};

// The algorithm for Rationalize is based on ratize.scm in SLIB.
private def FindRatio a b {
    let f = Floor a;
    let n = Int f;
    if [
        (f == a) {n, 1}
        (f < Floor b) {n+1, 1}
        _ {
            let (x, y) = FindRatio (/(b−f)) (/(a−f));
            y+n*x, x
        }
    ]
};
private def RatioBetween a b {
    if [
        (a > 0) {let (x, y) = FindRatio a b; x / y}
        (b < 0) {let (x, y) = FindRatio (−b) (−a); −x / y}
        _ {0}
    ]
};
def Rationalize x e {
    let a = x − e;
    let b = x + e;
    if [
        (a < b) {RatioBetween a b}
        (a == b) {Ratio a}
        _ {RatioBetween b a}
    ]
};
```

'Rationalize x e' finds the simplest rational number close to x, with the maximum error of e.

5.11.4 Infinities

```
type INF is:REAL Inf;
type NEG_INF is:REAL NegInf;
```

There are two exact infinities: positive Inf and negative NegInf. Inf is considered larger than any other object, and NegInf is smaller than any other object.

```
method '==' _^INF _^INF {True};
method '==' _^INF _^(NEG_INF,REAL,OBJECT) {False};
method '==' _^NEG_INF _^(INF,REAL,OBJECT) {False};
method '==' _^NEG_INF _^NEG_INF {True};
method '==' _^(REAL,OBJECT) _^(INF,NEG_INF) {False};
method '<' _^INF _^(INF,NEG_INF,REAL,OBJECT) {False};
method '<' _^NEG_INF _^(INF,REAL,OBJECT) {True};
method '<' _^NEG_INF _^NEG_INF {False};
method '<' _^(REAL,OBJECT) _^INF {True};
method '<' _^(REAL,OBJECT) _^NEG_INF {False};

method '<=' _^INF _^INF {True};
method '<=' _^INF _^(NEG_INF,REAL,OBJECT) {False};
method '<=' _^NEG_INF _^(INF,NEG_INF,REAL,OBJECT) {True};
method '<=' _^(REAL,OBJECT) _^INF {True};
method '<=' _^(REAL,OBJECT) _^NEG_INF {False};
```

Note: Separate methods specialized for REAL in addition to OBJECT are needed to make them preferred over methods with two arguments of type REAL which are defined elsewhere. □

```
method Min _^INF y^(INF,NEG_INF,REAL,OBJECT) {y};
method Min _^NEG_INF _^(INF,NEG_INF,REAL,OBJECT) {NegInf};
method Min x^(REAL,OBJECT) _^INF {x};
method Min _^(REAL,OBJECT) _^NEG_INF {NegInf};

method Max _^INF _^(INF,NEG_INF,REAL,OBJECT) {Inf};
method Max _^NEG_INF y^(INF,NEG_INF,REAL,OBJECT) {y};
method Max _^(REAL,OBJECT) _^INF {Inf};
method Max x^(REAL,OBJECT) _^NEG_INF {x};

method Int x^(INF,NEG_INF) {x};
method Ratio x^(INF,NEG_INF) {x};

method '+' x^INF _^(INF,REAL) {x};
method '+' x^INF y^NEG_INF {Fail (ArithmeticError #'+' [x y])};
method '+' x^NEG_INF _^(NEG_INF,REAL) {x};
method '+' x^NEG_INF y^INF {Fail (ArithmeticError #'+' [x y])};
method '+' _^REAL y^(INF,NEG_INF) {y};

method '-' x^INF y^INF {Fail (ArithmeticError #'-' [x y])};
method '-' x^INF _^(NEG_INF,REAL) {x};
method '-' x^NEG_INF _^(INF,REAL) {x};
method '-' x^NEG_INF y^NEG_INF {Fail (ArithmeticError #'-' [x y])};
method '-' _^REAL y^(INF,NEG_INF) {-y};

method '-' _^INF {NegInf};
method '-' _^NEG_INF {Inf};
method '*' _^INF y^(INF,NEG_INF) {y};
method '*' _^NEG_INF y^(INF,NEG_INF) {−y};
method '*' x^(INF,NEG_INF) y^REAL {
    if [
        (y > 0) {x}
        (y < 0) {−x}
        _ {Fail (ArithmeticError #'*' [x y])}
    ]
};
method '*' x^REAL y^(INF,NEG_INF) {
    if [
        (x > 0) {y}
        (x < 0) {−y}
        _ {Fail (ArithmeticError #'*' [x y])}
    ]
};

method Quot x^(INF,NEG_INF) y^(INF,NEG_INF) {
    Fail (ArithmeticError #Quot [x y])
};
method Quot x^(INF,NEG_INF) y^REAL {
    if [
        (y > 0) {x}
        (y < 0) {−x}
        _ {Fail (ArithmeticError #Quot [x y])}
    ]
};
method Quot _^REAL _^(INF,NEG_INF) {0};

method Rem x^(INF,NEG_INF) y^(INF,NEG_INF,REAL) {
    Fail (ArithmeticError #Rem [x y])
};
method Rem x^REAL _^(INF,NEG_INF) {x};

method QuotRem x^(INF,NEG_INF) y^(INF,NEG_INF,REAL) {
    Fail (ArithmeticError #QuotRem [x y])
};
method QuotRem x^REAL _^(INF,NEG_INF) {0, x};

method Div x^(INF,NEG_INF) y^(INF,NEG_INF) {
    Fail (ArithmeticError #Div [x y])
};
```

```
method Div x^(INF,NEG_INF) y^REAL {
    if [
        (y > 0) {x}
        (y < 0) {−x}
        _ {Fail (ArithmeticError #Div [x y])}
    ]
};
method Div x^REAL _^INF {if (x < 0) {−1} {0}};
method Div x^REAL _^NEG_INF {if (x > 0) {−1} {0}};

method Mod x^(INF,NEG_INF) y^(INF,NEG_INF,REAL) {
    Fail (ArithmeticError #Mod [x y])
};
method Mod x^REAL _^INF {if (x < 0) {Inf} {x}};
method Mod x^REAL _^NEG_INF {if (x > 0) {NegInf} {x}};

method DivMod x^(INF,NEG_INF) y^(INF,NEG_INF,REAL) {
    Fail (ArithmeticError #DivMod [x y])
};
method DivMod x^REAL _^INF {
    if (x < 0) {−1, Inf} {0, x}
};
method DivMod x^REAL _^NEG_INF {
    if (x > 0) {−1, NegInf} {0, x}
};

method IsDivisible x^(INF,NEG_INF) y^(INF,NEG_INF,REAL) {
    Fail (ArithmeticError #IsDivisible [x y])
};
method IsDivisible x^REAL _^(INF,NEG_INF) {x == 0};

method Floor x^(INF,NEG_INF) {x};
method Floor x^(INF,NEG_INF) _^REAL {x};

method Ceiling x^(INF,NEG_INF) {x};
method Ceiling x^(INF,NEG_INF) _^REAL {x};

method Trunc x^(INF,NEG_INF) {x};
method Trunc x^(INF,NEG_INF) _^REAL {x};

method Round x^(INF,NEG_INF) {x};
method Round x^(INF,NEG_INF) _^REAL {x};
method '/' x^(INF,NEG_INF) y^(INF,NEG_INF) {
    Fail (ArithmeticError #'/' [x y])
};
method '/' x^(INF,NEG_INF) y^REAL {
    if [
        (y > 0) {x}
        (y < 0) {−x}
        _ {Fail (ArithmeticError #'/' [x y])}
    ]
};
method '/' _^REAL _^(INF,NEG_INF) {0};

method '/' _^(INF,NEG_INF) {0};
```

5.11.5 Floats

```
type FLOAT is:REAL ···;
```

Floats represent numbers in some IEEE binary floating point format.

```
def IsFinite x! {};
method IsFinite _^(INT,RATIO) {True};
method IsFinite _^(INF,NEG_INF) {False};
method IsFinite x^FLOAT {···};
```

IsFinite tests if a real number is finite, i.e. not an infinity and not a NaN.

```
def IsInfinite x! {};
method IsInfinite _^(INT,RATIO) {False};
method IsInfinite _^(INF,NEG_INF) {True};
method IsInfinite x^FLOAT {···};
```

IsInfinite tests if a real number is an infinity.

```
def IsNaN x! {};
method IsNaN _^(INT,RATIO,INF,NEG_INF) {False};
method IsNaN x^FLOAT {···};
```

IsInfinite tests if a real number is a NaN.

```
def SignBit x! {};
method SignBit x^INT {x < 0};
method SignBit x^RATIO {x.num < 0};
method SignBit _^INF {False};
```

method SignBit _^NEG_INF {True};
method SignBit x^FLOAT {· · ·};

SignBit tests if a real number is negative, where 0 and 0.0 are considered nonnegative, and −0.0 is considered negative.

def Float x! opts... {};
method Float x^INT {· · ·};
method Float x^INTEGER {Float (Int x)};
method Float x^RATIO {· · ·};
method Float x^RATIONAL {Float (DefaultRational x)};
method Float _^INF {· · ·};
method Float _^NEG_INF {· · ·};
method Float _^NULL {· · ·};
method Float x^FLOAT {x};

By convention, Float converts an object to a float. If it is a number, the real part of its value is used. 'Float Null' is NaN.

dynamic DefaultReal = Float;

DefaultReal stores a numeric conversion function, which is used by certain operations in order to choose the representation of a real number. In particular it specifies the type of literals with a decimal point, and the type used to perform certain arithmetic operations which are not explicitly defined for the given combination of types of arguments.

method Int x^FLOAT {· · ·};
method Int x^REAL {Int (DefaultReal x)};

Int for non-finite numbers returns Inf, NegInf, or Null.

method Ratio x^FLOAT {· · ·};
method Ratio x^REAL {Ratio (DefaultReal x)};

Ratio for non-finite numbers returns Inf, NegInf, or Null.

def DecodeFloat x! {};
method DecodeFloat x^FLOAT {· · ·};

'DecodeFloat x' converts a number x to a triple sign, mant, exp, where sign is a bool, mant is an int, and exp is an int or Inf, such that x is equal to mant times 2 to the power of exp, negated if sign is True.

If x is zero, mant is 0 and exp is −Inf. If x is infinite, mant is positive and exp is Inf. If x is NaN, mant is 0 and exp is Inf.

def ScaleReal x! exp {};
method ScaleReal x^FLOAT exp {· · ·};
method ScaleReal x^REAL exp {ScaleReal (DefaultReal x) exp};

‘ScaleReal mant exp’ multiplies a real number mant by 2 to the power of exp, which must be an int, Inf, or −Inf.

If exp is Inf, the result is infinity with the sign of mant, or NaN if mant is 0.

```
method Is x^FLOAT y {···};
method Hash x^FLOAT {···};

method '==' x^FLOAT y^FLOAT {···};
method '<' x^FLOAT y^FLOAT {···};
method '<=' x^FLOAT y^FLOAT {···};

Each ['==' '<' '<='] ?f {
    method f x^INT y^FLOAT {~IsNaN y & f x (Ratio y)};
    method f x^FLOAT y^INT {~IsNaN x & f (Ratio x) y};
    method f x^INTEGER y^FLOAT {f (Int x) y};
    method f x^FLOAT y^INTEGER {f x (Int y)};
    method f x^RATIO y^FLOAT {~IsNaN y & f x (Ratio y)};
    method f x^FLOAT y^RATIO {~IsNan x & f (Ratio x) y};
    method f x^RATIONAL y^FLOAT {f (DefaultRational x) y};
    method f x^FLOAT y^RATIONAL {f x (DefaultRational y)};
};

method Min x^INT y^FLOAT {···};
method Max x^INT y^FLOAT {···};
method Min x^FLOAT y^INT {···};
method Max x^FLOAT y^INT {···};
method Min x^FLOAT y^FLOAT {···};
method Max x^FLOAT y^FLOAT {···};

method '+' x^INT y^FLOAT {···};
method '−' x^INT y^FLOAT {···};
method '+' x^FLOAT y^INT {···};
method '−' x^FLOAT y^INT {···};
method '+' x^FLOAT y^FLOAT {···};
method '−' x^FLOAT y^FLOAT {···};

method '−' y^FLOAT {···};
method Abs x^FLOAT {if (SignBit x) {−x} {x}};
method Signum x^FLOAT {···};

method '*' x^INT y^FLOAT {···};
method '/' x^INT y^FLOAT {···};
method '*' x^FLOAT y^INT {···};
method '/' x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #'/' [x 0])} ⇒
    ...
};
method '*' x^FLOAT y^FLOAT {···};
method '/' x^FLOAT y^FLOAT {···};
```

qqlindexmet/INF

```
method '/' _^INF y^FLOAT {
    if [
        (IsNaN y) {Float Null}
        (SignBit y) {Float NegInf}
        _ {Float Inf}
    ]
};
method '/' _^NEG_INF y^FLOAT {
    if [
        (IsNaN y) {Float Null}
        (SignBit y) {Float Inf}
        _ {Float NegInf}
    ]
};

method '/' y^FLOAT {···};

method Quot x^INT y^FLOAT {···};
method Quot x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #Quot [x 0])} ⇒
    ...
};
method Quot x^(INF,NEG_INF) y^FLOAT {x/y};
method Quot x^FLOAT y^FLOAT {···};

method Rem x^INT y^FLOAT {···};
method Rem x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #Rem [x 0])} ⇒
    ...
};
method Rem x^FLOAT y^FLOAT {···};
method QuotRem x^INT y^FLOAT {···};
method QuotRem x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #QuotRem [x 0])} ⇒
    ...
};
method QuotRem x^FLOAT y^FLOAT {···};

method Div x^INT y^FLOAT {···};
method Div x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #Div [x 0])} ⇒
    ...
};
method Div x^(INF,NEG_INF) y^FLOAT {x/y};
method Div x^FLOAT y^FLOAT {···};

method Mod x^INT y^FLOAT {···];
method Mod x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #Mod [x 0])} ⇒
    ...
};
method Mod x^FLOAT y^FLOAT {···};

method DivMod x^INT y^FLOAT {···};
method DivMod x^FLOAT y^INT {
    if (y %Is 0) {Fail (ArithmeticError #DivMod [x 0])} ⇒
    ...
};
method DivMod x^FLOAT y^FLOAT {···};

method Floor x^FLOAT {···};
method Floor x^FLOAT y^FLOAT {···};

method Ceiling x^FLOAT {···};
method Ceiling x^FLOAT y^FLOAT {···};

method Trunc x^FLOAT {···};
method Trunc x^FLOAT y^FLOAT {···};

method Round x^FLOAT {···};
method Round x^FLOAT y^FLOAT {···};
```

```
Each [Min Max '+' '−' '*' '/' Quot Rem QuotRem Div Mod DivMod] ?f {
    method f x^INTEGER y^FLOAT {f (Int x) y};
    method f x^FLOAT y^INTEGER {f x (Int y)};
    method f x^RATIONAL y^FLOAT {f (Float x) y};
    method f x^FLOAT y^RATIONAL {f x (Float y)};
    method f x^REAL y^REAL {let real = DefaultReal; f (real x) (real y)};
};

Each ['−' '/'] ?f {
    method f y^REAL {f (DefaultReal y)};
};
```

5.12 Characters

TODO

5.13 Pairs

A generic record type with two fields:

```
type PAIR (private Pair) left right;
```

The pair constructor is private because it is available through the ‘,’ syntax and the ',' function.

```
def ',' [
    () {Null}
    x {x}
    x xs... {Pair x (again xs...)}
];
```

',' returns a *tuple*tuple of an arbitrary number of elements, expressed as a structure of nested pairs, where the right element of every pair except the last one is the next pair. A tuple of one element is the element itself, and a tuple of no elements is Null.

Note: Tuples are primarily used when the number of elements is known. A tuple may also be used to hold an unknown number of elements if the elements are known to not be tuples themselves; this is sometimes used instead of putting them in a list when the most common case is a single element. □

```
method '==' (left1, right1)^PAIR (left2, right2)^PAIR {
    left1 == left2 & right1 == right2
};
method '<' (left1, right1)^PAIR (left2, right2)^PAIR {
    if (left1 ~= left2) {left1 < left2} {right1 < right2}
};
method '<=' (left1, right1)^PAIR (left2, right2)^PAIR {
    if (left1 ~= left2) {left1 < left2} {right1 <= right2}
};
```

5.14 Symbols

Symbols are similar to strings, with different performance properties. A symbol has a *name* which is a string. Symbols are used instead of strings when it is more important to distinguish them from one another than to examine the particular character sequences used to spell their names. In particular symbols are used to access fields of objects.

Symbols can be public or private. The difference is that there can be only one public symbol with the given name, which can be obtained from the name.

```
type SYMBOL ···;
```

A symbol has the following fields:

- name — the name of the symbol, which is a string
- hash — the hash of the symbol, returned by Hash
- **public** — whether the symbol is public

```
method Is _^SYMBOL _ {False};
method Hash sym^SYMBOL {sym.hash};
method String sym^SYMBOL {sym.name};

def Symbol (name→ofType STRING) {···};
def PrivateSymbol (name→ofType STRING) {···};
```

'Symbol name' returns the public symbol with the given name. The same symbol is returned each time for the same name.

'PrivateSymbol name' returns a new private symbol with the given name. It is distinct from all other symbols existing so far.

5.15 Types

```
type TYPE ···;
```

A type has the following fields:

- name — the symbol of the type, for informational purposes (it is not necessarily unique).
- hash — an integer for hashing.
- supertypes — a list of direct supertypes.

```
def Type object {···};
```

'Type object' is the type of object.

```
method Is _^TYPE _ {False};
method Hash type^TYPE {type.hash};
```

```
def AllSupertypes type {···};
```

'AllSupertypes type' returns the supertype list of type, excluding type itself.

```
def IsSubtype sub super {
    if (Type sub ~%Is TYPE) {Fail (BadType sub TYPE)};
    if (Type super ~%Is TYPE) {Fail (BadType super TYPE)};
    sub %Is super | super→IsIn (AllSupertypes sub)
};
```

'sub %IsSubtype super' tests if sub is a subtype of super.

```
def HasType object types... {
    Some types (IsSubtype (Type object) _)
};
```

'object→HasType types...' tests if object has type T for any T belonging to types.

```
def DeclareSupertype [
    sub super {···}
    sub super before {···}
];
```

'DeclareSupertype sub super' adds super as a direct supertype of sub, at the last position. 'DeclareSupertype sub super before' adds super as a direct supertype of sub, at the position before the first occurrence of before, which must already have been a supertype of sub, otherwise DeclareSupertype fails with 'NotSupertype sub before'.

An attempt to declare a supertype of OBJECT fails with TopSupertype.

Declaring a supertype of T changes the supertype lists of subtypes of T. This may make some of the supertype lists inconsistent, which happens when two supertypes of T' yield conflicting orders of some further supertypes. The conflict does not have to be detected immediately. Computing the supertype list of a type with inconsistent supertypes fails with 'InconsistentSupertypes type'.

def NewType name supertypes $\{\cdots\}$;

'NewType name supertypes' creates a new type with the given name (a symbol) and supertypes being the initial direct supertype list. It returns a pair of the new type and a function which wraps any object in an object of the new type, preserving its behavior when it is applied to arguments.

5.16 Mutable Variables

The type of mutable variables:

type VAR $\cdots$;

5.17 Dynamic variables

The type of dynamic variables:

type DYNAMIC $\cdots$;

An initially unbound dynamic variable created with '**dynamic** *name*' is associated with the symbol of *name*, which appears in the exception thrown when the variable is accessed without being bound. It is also possible to create an initially unbound dynamic variable with the symbol known dynamically:

def NewDynamic name $\{\cdots\}$;

A bound dynamic variable created with '**dynamic** $name = value$' is not associated with any symbol, and thus a function for creating such variable is not provided.

The function:

def AttachDynamic fun $\{\cdots\}$;

saves the current dynamic environment and returns a function which behaves like fun, except that it is executed in the saved dynamic enviromnent, ignoring the dynamic environment of its caller.

5.18 Lazy variables

The type of lazy variables:

```
type LAZY ···;
```

During computation of the value of a lazy variable, BlockSignals is applied. Waiting for a lazy variable to have its value computed by another thread is not a signal handling point.

5.19 Forward variables

The type of forward variables:

```
type FORWARD ···;
```

A forward variable created with '**forward** *name*' is associated with the symbol of *name*, which appears in the exception thrown when the variable is accessed before being set. It is also possible to create a forward variable with the symbol known dynamically:

```
def NewForward name {···};
```

5.20 Functions

Named functions are produced by **def** and **type** (as the constructor in the latter case). Unnamed functions are produced by '?', **function**, **loop**, **method**, partial application, and other constructs.

```
type FUNCTION {
   type NAMED_FUNCTION final:True ···;
   type UNNAMED_FUNCTION final:True ···;
};
```

```
method Is _^(NAMED_FUNCTION,UNNAMED_FUNCTION) _ {False};
```

```
def FunctionName fun {···};
```

'FunctionName fun' returns the name of fun if it is a named function, and Null otherwise.

```
def GenericCase arity keys {···};
def ApplyGenericCase fun generic args {···};
def NamedFunction [
   name fun {···}
   name fun generics {···}
];
def UnnamedFunction fun {···};
def DefineMethod fun arity keys types method {···};
def DefineMethodSuper fun arity keys types methodFun {···};
```

TODO

5.21 Singletons

Singleton types have an implicit supertype and an implicitly defined method of SingletonName. See section 4.3.6 on page 29 for details.

```
type SINGLETON;

def SingletonName obj! {};

method Hash obj^SINGLETON {(Type obj).hash};
```

5.22 Records

Record types have an implicit supertype and implicitly defined methods of RecordConstructor and RecordFields. See section 4.3.6 on page 29 for details.

```
type RECORD;

def RecordConstructor obj! {};
def RecordFields obj! {};

method Is x^RECORD y {
    loop (RecordFields x) [
        (field\fields) {x field %Is y field & again fields}
        [] {True}
    ]
};

method Hash obj^RECORD {
    loop (Type obj).hash (RecordFields obj) [
        acc (field\fields) {again (acc %CombineHash Hash (obj field)) fields}
        acc [] {acc}
    ]
};
```

By convention, ‘Change obj changes...’, where changes is a list of (field, value) pairs, returns a similar object as obj, except that the listed fields have the given values. Change is primarily supported by record types.

```
def Change obj! changes... {};

method Change obj^RECORD changes... {
    let (fields→ofType LIST) = RecordFields obj;
    (RecordConstructor obj)
        (changes→Fold (fields→Map (obj _)) ?values (field, value) {
            loop values fields [
                (v\vs) (f\fs) {
                    if (f %IsSame field) {value \ vs} {v \ again vs fs}
                }
                [] [] {
                    Fail (if (field→HasType SYMBOL) {NoField obj field}
                        {NotAFunction obj})
                }
            ]
        })...
};
```

5.23 Modules

TODO

5.24 Keywords

TODO

5.25 Time

A simple representation of the point in time as the number of ticks since the epoch is used for internal time measurements and computing delays. For a human-oriented representation of calendar dates, see section 7 on page 95.

```
type TIME (private TimeCon) ticks {};
let TicksPerSecond = 1000000000;
```

Ticks of a time are nanoseconds since 1970-01-01 00:00:00 UTC.

Note: In theory they should include leap seconds, but since Unix treats leap seconds by adjusting the speed of its internal timer, in practice the time might not include leap seconds. □

```
method Is time1^TIME time2 {time1.ticks %Is time2.ticks};
method Hash time^TIME {Hash time.ticks};
method '==' time1^TIME time2^TIME {time1.ticks == time2.ticks};
method '<' time1^TIME time2^TIME {time1.ticks < time2.ticks};
method '<=' time1^TIME time2^TIME {time1.ticks <= time2.ticks};

method '+' time^TIME seconds^REAL {
    let (ticks→ofType INT) = time.ticks + Int (seconds * TicksPerSecond);
    TimeCon ticks
};
method '−' time^TIME seconds^REAL {
    let (ticks→ofType INT) = time.ticks − Int (seconds * TicksPerSecond);
    TimeCon ticks
};
method '−' time1^TIME time2^TIME {
    (time1.ticks − time2.ticks) / TicksPerSecond
};
```

Time supports adjustment by addition or subtraction of the number of seconds, represented by a real number. The difference between two times is the number of seconds, represented by a ratio or int.

```
def Time [
    () {TimeCon (···)}
    time! {}
];
```

Time() returns the current time. By convention, Time time converts time from another representation to type TIME.

```
method Time ticks^INT {TimeCon ticks};
```

Time can convert the number of ticks since epoch to a time.

```
def AbsoluteTime timeout! {};
method AbsoluteTime time^TIME {time};
method AbsoluteTime seconds^REAL {Time() + seconds};
```

By convention, AbsoluteTime timeout converts a possibly relative time to the corresponding absolute time of type TIME. If timeout has type TIME, it is returned unchanged. If it has type REAL, it is treated as the number of seconds since now.

5.26 Signals

A *signal* is a value sent to a thread by another thread or from outside of the program. The thread reacts to a signal by applying its *signal handler* to the signal value, and resuming normal execution if the handler succeeded, or propagating the exception if it failed. This reaction can happen immediately, or it can be delayed until the thread reaches a convenient point, depending on the signal blocking state of the thread.

The blocking state of a thread is encoded as a pair $(all, async)$, where *all* is the number of reasons why all signals should be blocked, and *async* is the number of reasons why signals should be blocked in regions other than signal handling points.

This infinite space of states yields only three different direct effects on the reaction to signals:

$(0, \quad 0)$	signals can be handled at any time (the default)
$(0, \quad > 0)$	signals are handled only in signal handling points
$(> 0, \quad \_)$	signals are blocked

The following primitives cover useful transitions of the blocking state:

BlockSignals	$all = all + 1$
UnblockSignals	$all = all - 1$
BlockAsyncSignals	$async = async + 1$
UnblockAsyncSignals	$async = async - 1$
BlockSyncSignals	$all = all + 1;\ async = async - 1$
UnblockSyncSignals	$all = all - 1;\ async = async + 1$

They are scoped, in the sense that they take body as an argument, and they change the blocking state as specified during evaluation of ‘body()’, restoring it afterwards. Trying to make *all* or *async* negative is an error.

The signal handler itself is executed with signals blocked.

```
def DefaultSignalHandler signal! {};
method DefaultSignalHandler signal {Fail signal};
method DefaultSignalHandler signal^SYSTEM_SIGNAL {···};

dynamic Signal = DefaultSignalHandler;
```

The DefaultSignalHandler method of SYSTEM_SIGNAL emulates the operating system’s default reaction to a system signal.

The current signal handler is stored in a dynamic variable Signal.

5.27 Bracketing resource usage

A common pattern of dealing with a resource which requires or recommends explicit freeing, is to bracket a piece of code with an action which obtains the resource at the beginning, and an action which frees it at the end. The closing action is executed no matter whether the main body succeeds or fails.

```
def Using [
    obtain body {again obtain Close Close body}
    obtain release body {again obtain release release body}
    obtain commit rollback body {
        BlockSignals ⇒
        let resource = UnblockSyncSignals obtain;
        try {
            UnblockSignals ⇒
            body resource
        } exn {
            rollback resource;
            Fail exn
        } ?result {
            commit resource;
            result
        }
    }
];

def Ensure [
    release body {again {} release release body}
    obtain release body {again obtain release release body}
    obtain commit rollback body {
        BlockSignals ⇒
        UnblockSyncSignals obtain;
        try {
            UnblockSignals body
        } exn {
            rollback();
            Fail exn
        } ?result {
            commit();
            result
        }
    }
];
```

Using is used for resources represented by objects. In Ensure the resource is not materialized as an object, but is implicit in state changes by the provided actions.

The obtain part is executed in synchronous mode. The release part is executed with signals blocked. The body part is executed with the same signal blocking state as outside the whole construct.

5.28 Mutexes

A mutex allows to guard related concurrent operations, so that if a thread has to temporarily violate invariants of a data structure in order to update it, other threads do not observe this structure at an inappropriate time.

```
type MUTEX Mutex() {···};

def Lock mutex body {···};
def Unlock mutex body {···};
def LockRead mutex body {···};
def UnlockRead mutex body {···};
def LockUpdate mutex body {···};
def UnlockUpdate mutex body {···};
def LockWriteUpdating mutex body {···};
def UnlockWriteUpdating mutex body {···};
```

A mutex is either *unlocked*, or *locked for writing* by a thread, or *locked for reading* by some threads, or *locked for updating* by a thread and at the same time possibly locked for reading by some threads. A thread which has locked a mutex for updating can upgrade the lock to writing without unlocking it; conversely, a thread which has locked a mutex for writing can downgrade it to updating.

The mutex ensures that the constraints above are met by letting a thread wait if it attempts to perform an operation on a mutex which would violate the constraints if done immediately.

If a thread already has locked the mutex in a way which conflicts with the currently attempted operation, the operation fails with RecursiveLock instead of futile waiting.

A read lock is not granted if some thread is waiting for a write lock and the calling thread does not already have a read lock, even if only read locks are currently held. This avoids starvation of writers by multiple readers.

All locking and unlocking operations are scoped: they change the state of the mutex, execute the body of the operation, and change the state of the mutex in the opposite way.

An attempt to unlock, upgrade, or downgrade a mutex which is not locked by the current thread in the appropriate mode fails with LockNotLocked.

Locking operations apply BlockSignals during execution of the body, and unlocking operations apply UnblockSignals. Upgrading and downgrading do not change the signal blocking state but require signals to be blocked. These operations are not signal handling points.

While a thread is waiting for a lock after completing the body of a an unlocking operation, if a signal is handled and the signal handler fails, the thread continues to wait for the lock with signals blocked, and propagates the exception from the signal handler after it obtains the lock.

5.29 Materialized object identity

For any object, an associated object id can be obtained, which provides equality, hashing, and ordering of the identity of the original object.

```
type OBJECT_ID ···;
method Is _^OBJECT_ID _ {False};
method Hash id^OBJECT_ID {···};
method '==' id1^OBJECT_ID id2^OBJECT_ID {id1 %IsSame id2};
method '<' id1^OBJECT_ID id2^OBJECT_ID {···};
method '<=' id1^OBJECT_ID id2^OBJECT_ID {···};

def ObjectId [
    () {···}
    key {···}
];
```

'ObjectId key' returns the object id associated with key. Each application of ObjectId to the same key returns the same object id, as long as the previous object id is alive. An object id is not kept alive by its key.

Object ids are hashable and ordered. Ordering between object ids is arbitrary but consistent. If an object id dies and another object id is created for the same key, the hash and the ordering of the new object id are not necessarily the same as previously.

'ObjectId()' returns a new object id not associated with any key.

5.30 Weak references

A weak reference allows to refer to an object in a way which does not keep it alive.

```
type DEAD_WEAK_REFS DeadWeakRefs() {···};

type WEAK_REF ···;
def WeakRef [
    key {WeakPair key key}
    key dead data... {WeakPair key key dead data...}
];
def WeakPair [
    key value {···}
    key value dead data... {···}
];

def EachDeadWeakRef dead body {···};
```

A dead-weak-refs is a collection of dead weak references. Initially it is empty.

WeakPair returns a new weak reference with the given key and value, which maintains the following association: as long as the weak reference and key are alive, value is kept alive too. A weak reference by itself does not keep its key nor value alive.

WeakRef is a variant of WeakPair where key and value are the same.

For a given weakRef being a weak reference, 'weakRef absent present' enters 'present value' if the association is active, or 'absent()' otherwise. 'weakRef absent' means 'weakRef absent Identity', and 'weakRef()' means 'weakRef {Null}'.

A weak reference is also optionally associated with a dead-weak-refs and some data. If the key dies before its weak reference dies, and there is an associated dead-weak-refs, the weak reference is added to the dead-weak-refs.

'EachDeadWeakRef dead body' for a dead being a dead-weak-refs, for each contained weakRef with data, executes 'body weakRef data...' and ignores the result.

Rationale: EachDeadWeakRef is typically used to remove dead weak references from the container they are stored in, such as a weak dictionary. □

5.31 Lost threads

If a thread is going to be dead, i.e. it is not running and the garbage collector determines that any objects which could wake it up are unreachable, and if it responds to signals, it is kept alive and a ThreadKicked signal is sent to it.

Rationale: Such thread may hold resources which are freed by the continuation of the thread execution. □

If several threads are going to be dead at the same time, they might be kicked together, even if kicking only some of them would make others reachable.

If no threads are running or waiting for an external event, and the thread receiving system signals (see section 6.8 on page 90) is not waiting on SleepForever or **receive**, a garbage collection is forced.

Note: In other words, for the purposes of detection of lost threads, merely having signals unblocked is not considered to be waiting for an external event, unless the thread is waiting specifically for signals and for nothing else. □

Note: A garbage collection might help by kicking some threads or triggering finalization. □

If garbage collection did not resolve the situation, the configuration is considered a program error. In this case, if the thread receiving system signals responds to signals, a Deadlock signal is sent to it, otherwise the program terminates with a fatal error.

Rationale: This configuration is considered an error because it is likely not intended, even though technically it is not a deadlock: a system signal can interrupt a thread waiting in some other way. □

5.32 Collections

The following kinds of objects are called *collections*:

- *sequences* of *elements* arranged in some order, with possible duplicates, indexed with ints starting from 0
- *sets* of unique *elements*
- *dictionaries* mapping unique *keys* to *values*

Elements of dictionaries are pairs of a key and the corresponding value.

Collection types can have abstract supertypes, which provide default methods of various generic functions:

```
type COLLECTION;
type SEQUENCE is:COLLECTION;
type FLAT_SEQUENCE is:SEQUENCE;
type SET is:COLLECTION;
type DICTIONARY is:COLLECTION;
```

Flat sequences are a subset of sequences for which it can be expected that iteration by indexing with integers in the range of the size is efficient.

5.32.1 Iteration

Iterators are used to provide a sequence of elements, one by one. An iterator maintains the state needed to determine the remaining elements.

When an iterator is applied to two arguments: 'iter absent present', it enters 'absent()' if there are no more elements, or 'present elem' with the next element, advancing the state past it. It is correct to call an iterator again after it reports no more elements, and it should report no more elements each time.

Iteration over a set or a dictionary provides elements in an arbitrary order, unless a particular set or dictionary type specifies otherwise.

A *parallel iterator* provides elements of several sequences in parallel: it enters 'absent()' if any sequence runs out, or enters 'present elems...' with several elements corresponding to the sequences. An iterator is a special case of a parallel iterator with a single sequence.

```
def Iterate [
   () {?_absent present {present()}}
   coll! {}
   colls... {
      let iters = Map colls Iterate;
      ?absent present {
         loop iters present [
            (it\its) cont {
               it absent ?elem ⇒
               again its ?elems... ⇒
               cont elem elems...
            }
            [] cont {cont()}
         ]
      }
   }
];
```

'Iterate colls...' returns a new parallel iterator over the given collections.

```
let EmptyIterator = ?absent _present {absent()};
```

EmptyIterator is an iterator over an empty sequence.

5.32.2 Lists

The basic type of a finite sequence is LIST, with subtypes NIL for the empty list, and CONS for a nonempty list consisting of the *first* element and a list of the *rest* of elements. Lists are immutable. Despite dynamic typing the rest of a list is always a list, which is enforced at the time of construction. Lists are generally used when the number of elements is not known statically, especially when they are in some sense homogeneous.

```
type LIST is:SEQUENCE final:True {
    type CONS (private UnsafeCons) first rest {};
    type NIL (private Nil);
};
```

Cons and nil constructors are private because they are available through the '[*elem**]' syntax, the '*first* \ *rest*' syntax, and the '\\\\' function.

```
def '\\' [
    (list→ofType LIST) {list}
    x xs... list {UnsafeCons x (again xs... list)}
];
```

'\\\\' is the generalization of the cons constructor: it prepends any number of elements to the given list.

```
def List [
    () {[]}
    [] {[]}
    coll {again coll []}
    coll1! coll2 {}
    coll rest... {List coll (List rest...)}
];
```

'List colls....' returns a list with elements of the concatenated colls.

```
method List list^LIST rest {[list... (List rest)...]};

method List coll^COLLECTION rest {
    let iter = Iterate coll;
    loop {
        iter {List rest} ?elem ⇒
        elem \ again()
    }
};

method Is list1^LIST list2 {
    loop list1 list2 [
        (x\xs) (y\ys) {x %Is y & again xs ys}
        [] [] {True}
        _ _ {False}
    ]
};
```

```
method Hash list^LIST {
    loop 0 list [
        acc (elem\rest) {again (acc %CombineHash Hash elem) rest}
        acc [] {acc}
    ]
};

method '==' list1^LIST list2^LIST {
    loop list1 list2 [
        (x\xs) (y\ys) {x == y & again xs ys}
        [] [] {True}
        _ _ {False}
    ]
};

method '<' list1^LIST list2^LIST {
    loop list1 list2 [
        (x\xs) (y\ys) {if (x ~= y) {x < y} {again xs ys}}
        _ [] {False}
        [] _ {True}
    ]
};

method '<=' list1^LIST list2^LIST {
    loop list1 list2 [
        (x\xs) (y\ys) {if (x ~= y) {x < y} {again xs ys}}
        [] _ {True}
        _ [] {False}
    ]
};

method Iterate list^LIST {···};
```

5.32.3 Generators

A generator is a sequence defined by its iterator.

```
type GENERATOR is:SEQUENCE Generate (private newIter) {
    extend newIter;
};
```

'Generate newIter' returns a generator which evaluates 'newIter()' to obtain an iterator.

```
method Iterate gen^GENERATOR {gen()};

let EmptyGenerator = Generate {EmptyIterator};
```

EmptyGenerator is a generator of an empty sequence.

```
def Generator [
    () {EmptyGenerator}
    coll {
        if (coll→HasType GENERATOR) {coll} ⇒
        Generate {Iterate coll}
    }
    colls... {
        Generate {
            var collIter = colls;
            var iter = EmptyIterator;
            function ?absent present {
                iter {
                    case collIter [
                        (coll\restColls) {
                            collIter = restColls;
                            iter = Iterate coll;
                            again absent present
                        }
                        _ {absent()}
                    ]
                } present
            }
        }
    }
];
```

'Generator colls....' returns a generator with elements of the concatenated colls.

```
method Is gen1^GENERATOR gen2 {gen1 %IsEqual gen2};

method Hash gen^GENERATOR {
    gen→Fold 0 ?acc elem {acc %CombineHash Hash elem}
};

method '==' gen1^GENERATOR gen2^GENERATOR {IsEqual gen1 gen2 '=='};
method '<' gen1^GENERATOR gen2^GENERATOR {
    let iter1 = gen1();
    let iter2 = gen2();
    loop {
        iter2 {False} ?elem2 ⇒
        iter1 {True} ?elem1 ⇒
        if (elem1 ~= elem2) {elem1 < elem2} ⇒
        again()
    }
};

method '<=' gen1^GENERATOR gen2^GENERATOR {
    let iter1 = gen1();
    let iter2 = gen2();
    loop {
        iter1 {True} ?elem1 ⇒
        iter2 {False} ?elem2 ⇒
        if (elem1 ~= elem2) {elem1 < elem2} ⇒
        again()
    }
};

def Collect body {
    Generate {
        var absent;
        var present;
        var cont = {
            body {absent()} ?elem next {cont = next; present elem}
        };
        ?(set absent) (set present) {cont()}
    }
};
```

'Collect body' returns a generator which enters 'body end give' when iterated. This application should give elements by entering 'give elem next' for an element elem, where next is a nullary function which continues giving elements, and finally it should enter 'end()'.

Example: A generator which gives 3 elements:

```
let ABC = Collect ?end give {
    give "A" ⇒
    give "B" ⇒
    give "C" end
};
```

□

5.32.4 First and last element

The concept of the first element is extended to dictionaries and sets to mean the first element in the iteration order, i.e. an arbitrary element, unless a particular set or dictionary type specifies otherwise.

```
def First coll! {};
```

By convention, 'First coll' returns the first element of coll.

```
method First seq^FLAT_SEQUENCE {
   if (IsEmpty seq) {Fail EmptyCollection} ⇒
   seq@0
};

method First coll^COLLECTION {
   let iter = Iterate coll;
   iter {Fail EmptyCollection} Identity
};

def Last seq! {};
```

By convention, 'Last seq' returns the last element of seq.

```
method Last seq^FLAT_SEQUENCE {
   let (size→ofType INT) = Size seq;
   if (size > 0) {seq@(size − 1)} ⇒
   if (size == 0) {Fail EmptyCollection} ⇒
   Fail (NegativeSize size)
};

method Last seq^SEQUENCE {
   let iter = Iterate seq;
   iter {Fail EmptyCollection} (function ?last {iter {last} again})
};

def GetFirst [
   coll absent {again coll absent Identity}
   coll! absent present {}
];
```

By convention, 'GetFirst coll absent present' enters 'present elem' with elem being the first element of coll, or enters 'absent()' if coll is empty.

```
method GetFirst seq^FLAT_SEQUENCE absent present {
    if (IsEmpty seq) {absent()} {present seq@0}
};

method GetFirst coll^COLLECTION absent present {
    let iter = Iterate coll;
    iter absent present
};

def GetLast [
    seq absent {again seq absent Identity}
    seq! absent present {}
];
```

By convention, ‘GetLast seq absent present’ enters ‘present elem’ with elem being the last element of seq, or enters ‘absent()’ if seq is empty.

```
method GetLast seq^FLAT_SEQUENCE absent present {
    let (size→ofType INT) = Size seq;
    if (size > 0) {present seq@(size − 1)} ⇒
    if (size == 0) {absent()} ⇒
    Fail (NegativeSize size)
};

method GetLast seq^SEQUENCE absent present {
    let iter = IterateBack seq;
    iter absent present
};

def AddFirst seq! elem {};
```

By convention, ‘AddFirst seq elem’ returns a sequence like seq, with elem added before the first element.

```
method AddFirst list^LIST elem {elem\list};

method AddFirst seq^SEQUENCE elem {AddPart seq 0 [elem]};

def AddLast seq! elem {};
```

By convention, ‘AddLast seq elem’ returns a sequence like seq, with elem added after the last element.

```
method AddLast seq^SEQUENCE elem {AddPart seq [elem]};

def RemoveFirstGen coll! {};
```

By convention, 'RemoveFirstGen coll' returns a generator with elements of coll except the first element.

```
method RemoveFirstGen seq^FLAT_SEQUENCE {PartGen seq 1};

method RemoveFirstGen coll^COLLECTION {
    Generate {
        let iter = Iterate coll;
        iter {EmptyIterator} ?_ ⇒
        iter
    }
};

def RemoveLastGen seq! {};
```

By convention, 'RemoveLastGen seq' returns a generator with elements of seq except the last element.

```
method RemoveLastGen seq^FLAT_SEQUENCE {PartGen seq 0 (Size seq − 1)};

method RemoveLastGen seq^SEQUENCE {
    Generate {
        let iter = Iterate seq;
        iter {EmptyIterator} ?(var elem) ⇒
        ?absent present {
            iter absent ?next ⇒
            let prev = elem;
            elem = next;
            present prev
        }
    }
};

def RemoveFirst coll! {};
```

By convention, 'RemoveFirst coll' returns a collection like coll, with the first element removed.

```
method RemoveFirst list^CONS {list.rest};
method RemoveFirst list^NIL {list};

method RemoveFirst seq^FLAT_SEQUENCE {Part seq 1};

method RemoveFirst seq^SEQUENCE {RemoveFirstGen seq→Like seq};
method RemoveFirst set^SET {GetFirst set {set} (Remove set _)};
```

```
method RemoveFirst dict^DICTIONARY {
   GetFirst dict {dict} ?(key, _) {Remove dict key}
};
```

```
def RemoveLast seq! {};
```

By convention, ‘RemoveLast seq’ returns a sequence like seq, with the last element removed.

```
method RemoveLast seq^FLAT_SEQUENCE {Part seq 0 (Size seq − 1)};
```

```
method RemoveLast seq^SEQUENCE {RemoveLastGen seq→Like seq};
```

```
def DoAddFirst seq! elem {};
```

By convention, ‘DoAddFirst seq elem’ adds elem before the first element of seq.

```
method DoAddFirst seq^SEQUENCE elem {
   DoAddPart seq 0 [elem]
};
```

```
def DoAddLast seq! elem {};
```

By convention, ‘DoAddLast seq elem’ adds elem after the last element of seq.

```
method DoAddLast seq^SEQUENCE elem {
   DoAddPart seq [elem]
};
```

```
def DoRemoveFirst coll! {};
```

By convention, ‘DoRemoveFirst coll’ removes the first element of coll, or does nothing if it was already empty.

```
method DoRemoveFirst seq^SEQUENCE {
   DoRemovePart seq 0 1
};
```

```
method DoRemoveFirst set^SET {
   GetFirst set {} (DoRemove set _)
};
```

```
method DoRemoveFirst dict^DICTIONARY {
    GetFirst dict {} ?(key, _) {DoRemove dict key}
};
```

```
def DoRemoveLast seq! {};
```

By convention, ‘DoRemoveLast seq’ removes the last element of seq, or does nothing if it was already empty.

```
method DoRemoveLast seq^SEQUENCE {
    DoRemovePart seq (Size seq − 1)
};
```

```
def DoCutFirst coll! {};
```

By convention, ‘DoCutFirst coll’ removes and returns the first element of coll, or fails with EMPTY_COLLECTION if it was already empty.

```
method DoCutFirst seq^SEQUENCE {
    let elem = First seq;
    DoRemoveFirst seq;
    elem
};

method DoCutFirst set^SET {
    let elem = First set;
    DoRemove set elem;
    elem
};

method DoCutFirst dict^DICTIONARY {
    let (elem & (key, _)) = First dict;
    DoRemove dict key;
    elem
};
```

```
def DoCutLast seq! {};
```

By convention, ‘DoCutLast seq’ removes and returns the last element of seq, or fails with EMPTY_COLLECTION if it was already empty.

```
method DoCutLast seq^SEQUENCE {
    let elem = Last seq;
    DoRemoveLast seq;
    elem
};
```

```
def TryCutFirst [
    coll absent {again coll absent Identity}
    coll! absent present {}
];
```

By convention, ‘TryCutFirst coll absent present’ removes the first element of coll and enters ‘present elem’ with elem being that element, or enters ‘absent()’ if coll was already empty.

```
method TryCutFirst coll^COLLECTION absent present {
    GetFirst coll absent ?value ⇒
    DoRemoveFirst coll;
    present value
};
```

```
def TryCutLast [
    seq absent {again seq absent Identity}
    seq! absent present {}
];
```

By convention, ‘TryCutLast seq absent present’ removes the last element of coll and enters ‘present elem’ with elem being that element, or enters ‘absent()’ if coll was already empty.

```
method TryCutLast seq^SEQUENCE absent present {
    GetLast seq absent ?value ⇒
    DoRemoveLast seq;
    present value
};
```

5.32.5 Strings

A value of the STRING type represents a piece of text, expressed with a sequence of Unicode code points (with codes between `U+0000` and `U+10FFFF`). An element of a string is called a *character*, but it has no separate type: characters are represented with strings of length 1. Strings are immutable.

TODO

5.33 Showing data as strings

TODO

5.34 I/O streams

TODO

5.35 Serialization

TODO

6 Threads

Multithreading facilities are exported by the Threads module.

Functions with names beginning with Can are designed to be used with event queues: see section 6.12 on page 92.

6.1 Exceptions related to threads

subtypes PROGRAM_ERROR {

type NO_CONDITION_PREDICATE NoConditionPredicate;

An attempt to Wait without specifying a predicate on a condition which does not have an associated predicate either.

};

6.2 Starting threads

```
type THREAD ···;

def ThreadBlockSignals body {···};
def Thread body {
  ThreadBlockSignals {UnblockSignals body}
};
```

Threads are represented by handles of the THREAD type. Thread creates a new thread executing ‘body()’ with signals initially unblocked, and returns its handle. ThreadBlockSignals differs from Thread by starting the thread with signals blocked once.

The *main thread* is the initial thread created to run the program.

ref CurrentThread = {···};

CurrentThread is the handle of the calling thread.

6.3 Conditions

```
type CONDITION ···;
def Condition [
    mutex {again mutex {Fail NoConditionPredicate}}
    mutex pred {···}
];

def Wait [
    condition {again condition (···)}
    condition pred {···}
];
def Notify condition {···};
def Notify1 condition {···};
```

A condition is associated with a mutex and a predicate.

Wait must be called when the current thread has locked the mutex associated with condition, otherwise it fails with LockNotLocked. Wait uses the specified pred, or the predicate associated with condition if none was given.

Wait checks whether the predicate is true. While the predicate is false, Wait unlocks the mutex, waits until condition is notified or until a batch of signals is processed or until a spurious wakeup, relocks the mutex, and checks the predicate again. Waiting for notification is a signal handling point, while relocking behaves with respect to signals like in Unlock. A spurious wakeup is allowed to happen for no apparent reason.

Notify wakes up all threads waiting on condition. Notify1 wakes up some thread waiting on condition, if there is any.

6.4 Waiting for threads

```
def WaitForThread [
    thread {again thread Fail Identity}
    thread failed {again thread failed Identity}
    thread failed finished {···}
];
```

WaitForThread suspends the current thread until the given thread completes. This is a signal handling point.

When thread has finished, ‘finished value’ is entered with the value of the thread body, ‘failed exn’ is entered with the exception it failed with.

6.5 Yielding the processor

def Yield() $\{\cdots\}$;

Yield is a hint that allowing other threads to run now is a good idea for performance.

6.6 Sleeping forever

def SleepForever() $\{\cdots\}$;

SleepForever is the same as waiting for a condition with a predicate which is always false, except that when the thread which receives system signals is sleeping forever and no other thread can run, this does not result in a Deadlock signal.

6.7 Sending signals

def SignalThread thread signal $\{\cdots\}$;
def CancelThread thread {SignalThread thread ThreadExit};

SignalThread sends the given signal to the given thread. It returns immediately, even if the thread has signals blocked. If the thread has already finished, the signal is lost.

6.8 System signals

System signals are signals sent from outside of the program or by the internal language mechanisms.

ref SystemSignalHandler = $\cdots$;

One of the threads is designated to receive system signals. Initially this is the main thread. The handle of this thread can be obtained by SystemSignalHandler or changed by 'SystemSignalHandler = thread'.

If the thread receiving system signals completes, or a thread which has completed is designated as the thread receiving system signals, the main thread becomes the thread receiving system signals.

System signals include values of type SYSTEM_SIGNAL, which reflect signals sent between processes by the operating system.

OutOfMemory might be sent if the garbage collector determines that the program might soon run out of memory. Running out of memory is a fatal error.

Note: OutOfMemory gives a chance for the program to be informed that it should reduce its memory consumption, to reduce the probability of a fatal out of memory error. Since it is impossible to force a program to use less memory, and since it is unspecified how much memory is needed by various language constructs, including those which might be used when handling this signal, it does not prevent a fatal error. This signal is not even guaranteed to be delivered before running out of memory. □

6.9 Action signals

```
type ACTION_SIGNAL ActionSignal (var action);

method Close signal⌒ACTION_SIGNAL {
    signal.action = {}
};

method DefaultSignalHandler signal⌒ACTION_SIGNAL {
    signal.action()
};
```

6.10 Boxes

A box is a thread-aware container for an optional value.

```
type BOX Box [
    () {···}
    (private value) {···}
];
```

Box() returns a new empty box, and Box value returns a new box containing value.

```
def Put (box→ofType BOX) value {···};
```

Put box value puts value into box, or waits until another thread takes the value if the box is already full.

```
def Take (box→ofType BOX) {···};
```

Take box takes and returns the value from box, leaving the box empty, or waits until another thread puts a value if the box is already empty.

def CanPut (box→**ofType** BOX) {· · ·};

CanPut box returns when Put box Null would return, but without changing the contents of box.

def CanTake (box→**ofType** BOX) {· · ·};

CanTake box returns when Take box would return, but without changing the contents of box.

6.11 Queues

A queue is a thread-aware unbounded buffer.

type QUEUE Queue() {· · ·};

Queue() returns a new empty queue.

def PutLast (queue→**ofType** QUEUE) elem {· · ·};

PutLast queue elem adds elem as the last element of queue.

def TakeFirst (queue→**ofType** QUEUE) {· · ·};

TakeFirst queue takes and returns the first element of queue, removing it from the queue, or waits until another thread puts an element if the queue is already empty.

def CanTakeFirst (queue→**ofType** QUEUE) {· · ·};

CanTakeFirst queue returns when TakeFirst queue would return, but without changing the contents of queue.

6.12 Event queues

An event queue allows to wait for several events concurrently.

type EVENT_QUEUE EventQueue() {· · ·};

EventQueue() returns a new empty event queue.

type EVENT · · ·;

def RegisterEvent (queue→**ofType** EVENT_QUEUE) action {· · ·};

RegisterEvent queue action starts executing ‘action()’ in the background, and registers the thread executing it in queue. The action should wait for some event and return a value which identifies the event.

RegisterEvent returns an event object which can be used to unregister this event and cancel its action prematurely.

```
type READY_EVENT is:SPECIAL_RESULT ReadyEvent commit rollback;
def EventReady [
    commit {again commit Ignore}
    commit rollback {Fail (ReadyEvent commit rollback)}
];
```

If consuming an event has an observable effect, the function representing the event should only wait until the event is ready, without consuming it, and then apply EventReady to a nullary function which consumes the event and returns its value. This function will be called when this event is chosen by TakeEvent.

```
type NOT_READY_EVENT is:SPECIAL_RESULT NotReadyEvent continue;
def EventNotReady continue {Fail (NotReadyEvent continue)};
```

It might happen that the function being the argument of EventReady determines that the event is no longer ready this time (e.g. some other process has just grabbed the item from a shared queue). In this case this function may apply the special function EventNotReady to a nullary function which tells how to continue waiting, which lets TakeEvent start looking for another event.

```
method Close event^EVENT {···};
```

Close unregisters an event and cancel its action.

```
def TakeEvent (queue→ofType EVENT_QUEUE) {···};
```

TakeEvent waits until some registered action from the queue is ready, and returns its result. A given action fires only once.

```
def CanTakeEvent (queue→ofType EVENT_QUEUE) {···};
```

CanTakeEvent queue returns when TakeEvent queue would return, but without changing the contents of queue.

```
method Close queue^EVENT_QUEUE {···};
```

Close cancels all registered actions.

6.13 Finalizers

A *finalizer* is an action to be executed some time after the given object *dies*, i.e. ceases to be *alive*.

The set of alive objects is the smallest set which includes global references, alive threads, and with the following property: if an object belongs to the set, then all references contained in the object belong to the set too.

An *alive thread* is a thread which is strongly alive or responds to signals. A *strongly alive thread* is a thread which is either an alive object, or is running, or is waiting for an event outside of the program (such as I/O, timeout, or a system signal).

A thread is considered to contain *alive local references* of active scopes of the code executed by the thread. If a local reference will be accessed, it is alive; if it will not be accessed, but it belongs to an active scope, it is unspecified whether it is alive.

```
def Finalizer key close {···};
```

Finalizer establishes the following association: some time after the key object dies, 'close()' will be executed in a finalization thread created automatically to run finalizers, with signals blocked. If close fails, any exception is ignored.

```
type FINALIZER ···;
method Close finalizer^FINALIZER {···};
```

Finalizer returns a control object of the FINALIZER type which can be used to trigger finalization explicitly. Its Close method deactivates the association and executes 'close()' immediately, with signals blocked, unless it has already been executed (either automatically or explicitly), in which case Close does nothing. If close fails, the exception is propagated.

If the closing action is being executed by some other thread at the time Close is used, Close waits until the closing action completes. If it is already being executed by the same thread, Close fails with RecursiveLock.

The key object keeps close alive. The close function should not refer to key, otherwise key would never die. The lifetime of the finalizer object has no influence on the association.

```
def Touch [
   _ {}
   result _ {result}
];
```

Touch does nothing, but ensures that the object pointed to by the last argument of Touch is alive at the given point of the program.

Rationale: This is necessary in the following scenario:

```
def Operation obj {
   RawOperation obj.raw
   →Touch obj
};
```

if obj has a finalizer which makes obj.raw invalid, to ensure that obj is not finalized during RawOperation. □

6.14 Kicking weakly alive threads

TODO

6.15 Explicit garbage collection

```
def GarbageCollect() {···};
def MaybeGarbageCollect() {···};
def WantGarbageCollect() {···};
```

GarbageCollect is a hint that performing *garbage collection* now, i.e. determining which objects are unreachable and freeing their resources, is a good idea, as this could avoid garbage collection pauses later or reduce memory consumption. Garbage collection may take some time.

MaybeGarbageCollect is a hint that performing garbage collection or a partial garbage collection now is a good idea if enough garbage can be estimated to have been accumulated since last garbage collection. MaybeGarbageCollect doesnt bother to perform a full garbage collection each time if it is called too often.

WantGarbageCollect tests if MaybeGarbageCollect would choose to perform garbage collection now.

6.16 Exiting a program with several threads

The Threads module installs an AtExit handler which sets SystemSignalHandler to the main thread, cancels all other threads, waits for them to complete, performs garbage collection, and while there exist some other threads (they might have been created by garbage collection), cancels them, waits for them to complete, and performs garbage collection. Waiting for all other threads to complete is done in synchronous mode with signals unblocked.

6.17 Running code with a time limit

TODO

7 Calendar

TODO

References

[BCH^{+}96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Conference on Object-Oriented Programming Systems Languages and Applications*, pages 69–82, 1996.

Index

An *italic* page number means a less important reference.