A Brief Guide to Linear Janus

Clifford Tse

February 18, 1992

Abstract

Linear Janus, a subset of the more general framework of linear concurrent constraint (cc) languages, is designed in conformance with the actor computation paradigm. In many respects, Linear Janus is a lower level language than previous cc languages, such as Janus. In Linear Janus, the user is given more control over the course of execution; thus the language is suitable for system programming. In this document, we will informally explain the semantics and syntax of Linear Janus and also give some examples of its use. Please be reminded that this document is informal. A companion document will give a formal description of the language. Most of the material in this document is derived from notes written during discussions with Saraswat.

1 Introduction

Linear cc languages have been introduced by Saraswat as a refinement of cc languages. The crucial difference between the linear cc and cc frameworks is the underlying logics. cc languages are founded on classical intuitionistic logic while linear cc languages are founded on the linear intuitionistic logic proposed by Girard. The move to linear logic allows a logical treatment of indeterminacy, which was not possible in the intuitionistic setting. Linear logic (LL), which subsumes classical and intuitionistic logic, provides a finer control of logical deduction which makes it ideal for modeling many computational phenomena. The most important concept of linear logic is the notion of accounting. Unlike classical logics, the order and multiplicity of logic formulae in LL are carefully accounted for in each deduction. For this documentation, an understanding of linear logic is not crucial, interested readers are referred to Girard's exposition on the subject.

Linear cc is a general framework which encompasses a whole family of languages. Linear Janus is a member that exploits the actor model of computation. Unlike Janus, Linear Janus does not have restriction on the number of occurrences of a variable and it provides many-to-many communications, a type inference system and better abstraction facilities.

2 Core Linear Janus

In this documentation, we will only present Core Linear Janus which is a semantically well defined but syntactically impoverished core of Linear Janus. Straight forward extensions can be built on top of Core Linear Janus.

The basic syntax of linear Janus is as follows, informally:

(agent)	A	::=	**1**
			| x:t
			| A, A
			| x ^ A
			| c $\rightarrow$ A
			| x:m $\multimap$ A
			| ! x:m $\multimap$ A
(constraint)	c	::=	BUILT-IN-CONSTRAINT (such as $3 < 4$)
(term)	t	::=	TERM (e.g. `[C | D]`, see Section: Linear Janus Data Types)
	m	::=	Quantified TERM (e.g. `[\C | \D]`)
(variables)	x,y,z	::=	

In the syntax of Linear Janus, there is no category of programs, only the category of agents. An agent's behavior is defined by the formula "`X:\M`$\multimap$`A`", where "$\multimap$" is linear implication. A statement like "`X:\M`$\multimap$`A`" is to be read as "remove a message `N` from `X` and then behave like `A` with `N` substituted for free occurrences of `M`". A pattern can be used in the place of `M`, for example, "`X: [\C | \D]` $\multimap$ `A`", which is to be read as "remove a message `N` (a cons-cell) from `X`, then behave like `A` with (free occurrences of) `C` substituted by `car` of `N` and (free occurrences of) `D` substituted by `cdr` of `N`. A reusable agent is defined by using "!". "`!X:\M`$\multimap$`A`" defines an agent which can be used more than once[1]. "$\rightarrow$" is the *blocking-ask* operator. "`C`$\rightarrow$`A`" is to be read as "given the current configurations, *if* `C` is true, *then* do `A`, where `C` is a primitive test, such as `A < 4`". ":" is the channel operator. "`X:M`" is to be read as "send the message `M` to channel `X`". "^" is the existential quantifier. "`X ^ A`" is to be read as "if there exists `X` such that `A` is true" or "create a *local* variable `X` and use it in `A`". "," is parallel composition. "`A, B`" means a parallel composition of `A` and `B`. "\" is the local universal quantifier. "`x:\y`$\multimap$`A`" is to be read as "$\forall$`y.x:y` $\multimap$`A`", i.e. the quantification is localized. The "`1`" agent does nothing.

A more complete BNF syntax of Linear Janus can be found in the Appendix.

3 Primitive Types of Linear Janus

Unlike other `cc` languages, Linear Janus is an implicitly, statically and polymorphically typed language. Linear Janus defines a number of primitive types and derived types. User can also define

[1] The readings for most logic operators in Linear Janus are derived from linear logic. In linear logic, a logic formula can be used only once, unless it is quantified by the *exponential* or *of course* operator "!", in which case it is reusable.

datatype using abstractions (not described in this document).

Linear Janus primitive types consists of INT, REAL, BOOL, CHAR, RECORD, and ARRAY. The three derived types are STRING, TUPLE, and LIST. A string is an array of CHAR; a TUPLE is an implicitly named record; a LIST is a record with 2 fields, `car` and `cdr`.

A formal description of the type system can be found in the Appendix.

3.1 Linear Janus Data Types

3.1.1 Primitive Types: INT, REAL, BOOL, CHAR

Each primitive type has the usually primitive operations, as well as a type test operator. For example, for INT, we have +, -, *, /, etc., and also `int?(A)` which tests if `A` is an INT.

3.1.2 Data Structures: Array, Tuple and Record

Array

array?(A)	: datatype test
array(N, T)	: Create an array of N elements with initial value T
A.i	: i^{th} element of A, index starting from 0
|A|	: size of A
A< i..j >	: a subarray composed of, inclusively, i^{th} to j^{th} elements
A+B	: concatenate A and B

Record

A record is a named collection of items. Each of the term can be of different types.

record?(A)	: datatype test
A{field$_1$: value$_1$, ..., field$_n$: value$_n$}	: construct a record A with n fields
A.field	: select $field$ from record A

Tuple

A tuple is a derived type of Record. It is an ordered collection of items. The field of a tuple are implicitly indexed by natural numbers which represent the position of the fields in the tuple.

tuple?(A)	: datatype test
{A, B}	: a tuple of two elements: A and B
{}	: an empty tuple

For example, if `M` matches the tuple {`A, B`}, then `M.0 = A` and `M.1 = B`.

List

A list is also a derived type of Record. A list is a record with two fields; the first is `car` and the second is `cdr` of the list respectively.

list?(A)	: datatype test
[A,B,C]	: A list of three elements A, B, C, in that order
|A|	: length of A
[A | B]	: cons A, B
[]	: an empty list
A+B	: append A, B
A++B	: append A, [B]

For example, if `L = [A | B]` is a list, then `L.first = A` and `L.rest = B`.

3.2 Operators

Arithmetic Operators And Comparisons

+	: adds
-	: negative/minus
*	: times
/	: divides
**	: exponential

=	: equal
>	: greater
<	: less
~=	: not equal
>=	: ge
<=	: le

4 Linear Janus Programming Examples

Channels are one of the primitive entities in Linear Janus. Using channels, various computations can be expressed easily. Agents communicates with each other by sending messages to and/or receiving messages from channels. Each channel is an unordered set (or a bag). In other words, messages in a channel are not in any particular order. Channels are asynchronous and many-to-many communication links. Any number of agents can send messages to and receive messages from the same channel. A message sent on a channel can be received only once, though the receiving agent may choose to put it back.

Please be reminded that "`C:M`" means "send message `M` to channel `C`"; "`C:\M` $\multimap$ `A`" means "remove a message `N` from channel `C` and then behave like `A` with `N` substituted for free occurrences of `M` in `A`". Also, we may write "`X(`t_1`, ..., `t_n`)`" for "`X:{`t_1`, ..., `t_n`}`" or vice versa.

We will first show some examples of different channel structures:

```
/* case 1 */
/* C is channel of numbers */
C:1, C:2, C:3, C:4
```

Since "," is the parallel composition operator, the contents of `C` will be a bag of 4 numbers, namely 1, 2, 3, 4, in some arbitrary order.

However, if we want to enforce some ordering for messages, we can collect the messages in a list and then send the list to a channel. For example:

```
/* case 2 */
/* C is a channel of lists of numbers */
C:[1, 2, 3, 4]
```

In this case, `C` will contain a list of 4 elements.

However, this is not very efficient since the sending agent has to construct the whole list before sending and the receiving agent has to fetch the whole list before processing any elements. For example, if agent `A` is producing some elements, and agent `B` is waiting to process the elements produced by `A`, in the same order in which the elements are produced, then using the above method, `B` has to wait till `A` finishes. A better way is to allow `B` to process an element as soon as it is produced by `A`. To achieve this, we can establish a channel `C` between `A` and `B`; now when `A` produces an element e, it sends the element e and another channel `D` to `C`; when `B` receives a message form `C` in the form [e | `D`], it processes e and waits for the next message at channel `D`. In essence, `A` sends to `C` a message (a cons-cell) which contains a value and a continuation channel to which further value will be communicated. This is a very common idiom in Linear Janus programming (and in concurrent logic programming). We will call this kind of channels "channel-with-continuation", or simply *c-continuation*. For example:

```
/* case 3 */
/* C is c-continuation of numbers */
C:[1 | C1], C1:[2 | C3], C3:[3 | C4], C4:[4 | C5], C5:[]
```

Of course, we can also construct a channel whose messages are channels or cons-cells of channels, such as:

```
/* case 4 */
/* C is channel of cons-cell of c-continuation of numbers */
C:[C1 | C2], C1:[2 | C3], C2:[3 | C4], C3:[], C4:[]
```

Henceforth, we will write "`C:[1 | C1], C1:[2 | C2], C2:[3 | C3], C3:[]`" as "`C:[1 | C1:[ 2 | C2:[3 | C3:[]]]]`", or simply "`C:[1 | [ 2 | [3]]]`", when the channel labels are unimportant. Also, unspecified channel can be assumed to be empty.

Also, we will use "`@X`" as a term to mean "a message on channel `X`". In other words, an agent `A` with an occurrence of `@X` inside it is to be thought of as the agent "`X:\M→ A[M/@X]`". We will sometimes refer to "`Y:@X`" as "forwarding a message of `X` to `Y`". "`!Y:@X`" means forward *all* messages of `X` to `Y`, if there is no other agents read `X`.

4.1 Simple Programs

There are many different ways to express a program with different degrees of efficiency and concurrency, by using different data structures and channel structures. Some of them may not be obvious to naive users. We will try to illustrate some examples below.

Append

There are a number of different ways to write Append in Linear Janus, each with a slightly different behavior.

```
/* case 1 */
/* inputs L1, L2 are lists */
/* output O is a channel to which we will send our result (a list) */

! Append(\L1, \L2, \O)
   -o (L1=[] -> O:L2,
       L1=[\A | \B] -> C^(O:[A|@C], Append(B, L2, C)))

/* or, using multi-clause style */
! Append([], \L2, \O) -o O:L2
! Append([\A | \B], \L2, \O) -o C^(O:[A|@C], Append(B, L2, C))
```

```
/* invocation */
/* Initial Configuration ==> Final Configuration */

        Append([1, 2, 3], [4, 5, 6], O) ==> O:[1, 2, 3, 4, 5, 6]
```

This is a naive version. Append goes down the first list, creating a new list with the elements; when it reaches the end, it uses the second list as the next element; then it returns the whole list. (Remember that `O:[A | @C]` in Append is short for `C:\M`$\multimap$`O:[A | M]`.)

There are a number of ways to improve Append's performance. For example, we don't need to have the whole lists before starting the append. In other words, we can use c-continuation inputs, which gives:

```
/* case 2 */
/* input L1 is c-continuations of list */
/* input L2 is channel of lists */
/* output O is a channel to which we will send our result (a list) */

! Append(\L1, \L2, \O)
   -o (L1 :\M
        -o (M=[] -> O:@L2,
            M=[\A | \B] -> C^(O:[A|@C], Append(B, L2, C))))

/* invocation */

        L1:[1 | L2:[2 | L3: [3 | L4:[]]]], L5:[4, 5, 6], Append(L1, L5, O)
     ==>
        O:[1, 2, 3, 4, 5, 6]
```

Now, Append reads the first message from `L1`, which is a cons-cell `[A | B]`, where `A` is an element and `B` is a continuation; it recursively calls itself with the continuation `B`; it also cons `A` with `C`, which is the result of the recursive call.

However, this does not improve the performance much, because it still constructs the whole list before returning. We can improve performance further by returning the result before finishing, by constructing a c-continuation result:

```
/* case 3 */
/* inputs L1, L2 are c-continuations */
/* output O is a c-continuation to which we will send our result */

! Append(\L1, \L2, \O)
   -o (L1 :\M
```

```
        -o (M=[] -> O:@L2,
            M=[\A | \B] -> C^(O:[A|C], Append(B, L2, C))))

/* invocation */
/* C1..C3 are newly created channel, in C^(O:[A|C], Append(B, L2, C)) */

      L1:[1 | L2:[2 | L3: [3 | L4:[]]]],
      L5:[4 | L6:[5 | L7: [6 | L8: ... ]]], Append(L1, L5, O)
   ==>
      O:[1 | C1:[2 | C2:[3 | C3:[4 | L6:[5 | L7:[6 | L8: ... ]]]]]]
```

In other words, when Append gets the first elements of the first list, it returns that to channel `O` and also a continuation. So, as soon as we gets one element from the first list, we send it to channel `O`. In this way, the *partial* result can be used by some other agent, before Append finishes.

Or

In Linear Janus, the user is given the control of the execution of programs. Using the finer control, a program can exploit features not accessible in other languages. For example, if we want to define an agent Or, we can define it as:

```
/* inputs A, B are channels, each gives a bool */
/* output C is a channel to which  we will send the result of (A or B) */

! Or(\A, \B, \C)
   -o A: \M1
     -o B: \M2
       -o (M1=false -> (M2=false -> C: false, [*] -> C:True),
           [*] -> C: True)
```

"`[*]`" is to be read as "`else`. For example, "$\mathtt{c}_1 \rightarrow \mathtt{A}_1$, $\mathtt{c}_2 \rightarrow \mathtt{A}_2$, `[*]` $\rightarrow \mathtt{A}_3$" is read as "if $\mathtt{c}_1$ is `true`, do $\mathtt{A}_1$; if $\mathtt{c}_2$ is `true`, do $\mathtt{A}_2$; if both $\mathtt{c}_1$ and $\mathtt{c}_2$ are `false`, do $\mathtt{A}_3$". "`[*]` $\rightarrow$ `A`" is allowed only in a conjunction where all other conjuncts are of the form "`c` $\rightarrow$ `A`".

However, this Or is not efficient. If `V1` is `true`, output is true regardless of the value of `V2`. So, a better Or would be:

```
/* _ is the black hole operator */
! Or(\A, \B, \C)
   -o A: \M
     -o (M=True -> (C:true, _:@B),
         [*] -> C:@B)
```

"`_`" is the black hole that ignores everything it receives. So, `_:@B` only serves to consume a message from `B`, to maintain the balance of channels. This Or looks reasonable. However, it assumes that the value from the A will be available first. So, if it turns out that the value from B is available first, this program does not help at all. We can write a better Or that does not need to worry about the order in which the values arrives:

```
! Or(\A,\B,\C)
   -o D^(D:@A, D:@B,
         (D:\M
             -o (M=true -> (C:true, _:@D),
                 [*]-> C:@D)))
```

So, A and B *forward* their values to D. When the first values arrives at D, it is checked to see if it is **true**; if yes, the output is **true**, else the next value is forwarded to the output.

In the above example, we, in fact, show how many-to-one communication can be done in Linear Janus: agents simply send messages to the same channel.

4.2 Using Channels as First Class Functions

First class functions are commonplace in high level programming. In Linear Janus, channels play the role of first class functions. Instead of passing a function pointer as an argument, we pass the *channel to a function* as an argument. For example, the Map function can be written as follows:

```
/* Map */
/* F is channel which receives 2-tuples */
/* L is a channel with continuation */
/* O is a channel with continuation */

! Map(\F, \L, \O)
   -o (L :\M
        -o (M=[] -> O:[],
            M=[\A | \B] -> C^D^(F:{A, C}, Map(F, B, D), O:[@C|D])))
```

In above, F is a channel which can be connected to some function which expects a 2-tuple, where the first element is the argument to the function, and second element is a channel to which the result from the function is to be returned.

Notice that Map will return the result to O after it receives a result from C. If we write O:[C | D] instead, Map will immediately return the result to O. However, the car of the result will be a channel where the result of applying the function will be returned, the cdr of the result will be a continuation, whose structure is the same as O.

For example, if we want to increment all the elements of a list by 1, we can define Inc as follows:

```
! Inc(\C) -o C :{\V, \O} -o (O:V+1, Inc(C))
```

Here, C receives a message, which is cons-cell of V and O, and it returns the value V+1 to O. Notice that Inc recursively calls itself to handle further messages. However, we can get rid of the tail recursive calls by writing:

```
! Inc(\C) -o ! C :{\V, \O} -o O:V+1
```

Inside Inc, we define the behavior of C as "`! C:{\V, O} -o O:V1+`" which causes further messages to be handled, i.e. C acts like a server awaiting messages (using "!" in this way can be tricky and is explained more carefully in the Counter_creator example below). A typical invocation would be:

```
Inc(I), Map(I, L, O), L:[1 | [2]] ==> Inc(I), O:[2 | [3]]
```

Notice that, in Map, the three agents in (`F:{A, C}, Map(F, B, D), O:[@C|D]`) will run in parallel. Since a channel is a bag, the sharing of channel F between two agents (`F:{A, C}` and `Map(F, B, D)`) may cause a race condition. For example, in the pathological case:

```
! Dumb(\C) -o Dumber(C, 0)
! Dumber(\C, \N) -o C:{\V, \O} -o (O:N+1, Dumber(C, N+1))
```

Dumber, upon receiving a message, will return a number which depends on the order arrival. The following situations may arise:

```
/* case 1 */
Dumb(I), Map(I, L, O), L:[1 | [2]] ==> Dumb(I), O:[0 | [1]]

/* case 2 */
Dumb(I), Map(I, L, O), L:[1 | [2]] ==> Dumb(I), O:[1 | [0]]
```

In other words, sharing a channel with local states can causes race conditions.

Interestingly, Inc and Dumb once invoked will stay around, until the scope in which they are invoked exits.

4.3 Divide And Conquer Algorithms

Compute GCD of an Array of Numbers.

```
/* Gcd */
/* input A is an array of numbers */
/* output Res is channel which we send an array */
! Gcd(\A, \Res)
   -o (|A|=1 -> Res:A.0,
        Temp^(Gcd(A<0..|A|/2>, Temp),
              Gcd(A<|A|/2+1...|A|-1>, Temp),
              Gcd_I(@Temp, @Temp, Res)))

/* compute the GCD of two numbers, V1, V2 */
! Gcd_I(\V1, \V2, \Res)
   -o (V1 = V2 -> Res:V1,
       V1 > V2 -> Gcd_I(V1 % V2, V2, Res),
       V1 < V2 -> Gcd_I(V1, V2 % V1, Res))
```

Notice that in Gcd we re-use Temp as a result channel to the two recursive calls to Gcd with the two sub-arrays. We can do this because we don't care about the order of in which the two results are returned to Temp. The last clause of Gcd, `Gcd(@Temp, @Temp, Res)`, is short for: "`Temp:\M1`$\multimap$ `Temp:\M2` $\multimap$ `Gcd(M1, M2, Res)`" or "`Temp:\M1` $\multimap$`Temp:\M2` $\multimap$ `Gcd(M2, M1, Res)`", i.e. the substitution is indeterministic.

Linear Janus also allows nested definitions which are lexically scoped. So, we can have local definitions, such as:

```
/* same as above, but using nested syntax */
! Gcd(\A, \Res)
   -o Gcd_I^(! Gcd_I(\V1, \V2, \Res)
                -o (V1 = V2 -> Res:V1,
                    V1 > V2 -> Gcd_I(V1 % V2, V2, Res),
                    V1 < V2 -> Gcd_I(V1, V2 % V1, Res)),
             (|A|=1 -> Res:A.0,
                  Temp^(Gcd(A<0..|A|/2>, Temp),
                        Gcd(A<|A|/2+1...|A|-1>, Temp),
                        Gcd_I(@Temp, @Temp, Res))))
```

Now, Gcd_I is local to Gcd and is invisible to agents outside the scope of Gcd. Using lexical scoping, we can also avoid passing arguments. In the above example, Res is not changed in Gcd_I, so we can avoid passing it, for example:

```
/* same as above, using variable scoping for Res  */
! Gcd(\A, \Res)
   -o Gcd_I^(! Gcd_I(V1, V2)
                -o (V1 = V2 -> Res:V1, /* <--- Res */
                    V1 > V2 -> Gcd_I(V1 % V2, V2),
```

```
        V1 < V2 -> Gcd_I(V1, V2 % V1)),
(|A|=1 -> Res:A.0,
      Temp^(Gcd(A<0..|A|/2>, Temp),
            Gcd(A<|A|/2+1...|A|-1>, Temp),
            Gcd_I(@Temp, @Temp))))
```

The scoping rules for Linear Janus are every similar to those of Scheme. A form like "`X^A`" means `X` is visable only in `A` (similar to Scheme scoping using "`let`"). Not only variables but also agents can be lexically scoped.

4.4 Higher Order Functions

Higher order functions are another commonplace of modern programming languages. In Linear Janus, we can also have higher order agents via channels. For example:

```
/* Adder */
! Adder(\M, \X)
     -o ! X:(\N, \Y) -o Y:M+N
```

As the name suggests, Adder adds two numbers together. The behavior of Adder is as follows:

Adder receives two arguments `M` and `X`; `M` is the first operand of the addition; however, `X`, instead of being the second operand, is an agent whose behavior is defined in Adder as "`!X:(\N,\Y)` ⊸`Y:M+N`"; so, `X` receives two arguments, `M` and `Y`; `X` send the value of `N+M` to `Y`.

Now, given Adder, we can define other agents, like:

```
Adder(5, Add5)
```

Now, Add5 is the input channel of an agent that, when given a number `N` and a channel `Y`, will send to `Y` the value of `5+N`. (Adder is similar to the λ-term λ`x.`λ`y.(+ x y)`; Add5 is similar to `((`λ`x.`λy`.(+ x y)) 5)`.)

4.5 Creating Agents and Hiding Private Data

In the previous section, we show how an agent can create another agent. However, we can extend the idea further, creating a agent with private (local) state. Before showing the next example, we introduce a short hand "`?X`" to mean "*examine* the value of channel `X` without removing it form the channel". In other words, an agent `A` with an occurrence of "`?X`" is to be thought of as the agent "`X:\M` ⊸ `(A[M/?X], X:M)`".

Now, say, we want to create a number of agents each with their own private counters. We can do so by defining each agent separately. But, a better way is by using Counter_creator as follows:

```
/* Creating a counter, with initial value 0 */
! Counter_creator : \Counter
   -o Value ^ (Value:0,
              !Counter : inc -o Value:@Value+1,
              !Counter : dec -o Value:@Value-1,
              !Counter : val.\Res -o Res:?Value)
```

Counter_creator takes an argument, `Counter`; it creates a local variable `Value`, initialized to 0 and then defines the behavior of `Counter`. Now, since `Value` is existentially quantified (i.e. each invocation will give a new `Value`), each `Counter` has its own private copy. Now, to create, increment, decrement and examine a counter can be done by:

```
/* to create a counter */
Counter_creator: C

/* to increment the counter */
C: inc

/* to decrement the counter */
C: dec

/* to read the value of a counter (result will return at R) */
C: val.R
```

In Counter_creator, we use an existentially quantified variable, `Value`, to encapsulate a local state. This idea is exactly like taking closure in Scheme and is a very clear way to implement agents with local state.

Also, in `Counter_creator`, new behaviors of `Counter` are defined (`!Counter:...`). If `Counter` already has some other behaviors, then these new behaviors will be added on top of its old behaviors.

Tower of Hanoi

```
/* read *A* as a string composed of symbol A  */
/* Hanoi */
/* input N is a number */
/* output Z is a channel of strings */
! Hanoi(N, Z) -o Hanoi(N, *A*, *B*, *C*, Z)
! Hanoi(N, \A, \B, \C, \Z)
   -o (N=1 -> Z:A + *->* + B,
```

```
    [*] -> (Hanoi(N-1, A, C, B, Z1),
            Hanoi(1, A, B, C, Z2),
            Hanoi(N-1, C, B, A, Z3),
            Z:@Z1+@Z2+@Z3))
```

As in Prolog, functions with the same name but different arities are different functions, i.e. Hanoi/2 is a different from Hanoi/5.

4.6 Data Parallel Computation (not finalized yet)

Return a list to `Res` which is a list with elements from `F` to `T`.

```
/* Iota */
/* inputs F, T are numbers, such that F < T (not enforced) */
/* outputs to Res is a list */
! Iota(\F, \T, \Res)
   -o F<=T -> R^(Iota(F+1, T, R), Res:[F | @R])

/* a more efficient one */

/* Iota */
/* inputs F, T are numbers, such that F < T (not enforced) */
/* outputs to Res is c-continuation */
! Iota(\F, \T, \Res)
   -o R^(Iota(F+1, T, R), Res:[F | R])
```

However, if we want to have an array with elements from `F` to `T`, we can initialize all the elements of the array in parallel. We proposal the following notation: "`\i:l..m | A`" is to be read as "`A[l / i], A[l+1 / i], ..., A[m / i]`", or, "for each `j` from `l` to `m`, do `A[j/i]`". For example:

```
/* for an array */
/* <a, b, c> constructs an array with elements a, b, c */
! Iota(\F, \T, \Res) -o (F<T -> Res:<\i:0..(T-F) | <i> = F+i>)

/* for a list */
/* [a, b, c] constructs a list with elements a, b, c */
! Iota(\F, \T, \Res) -o (F<T -> Res:[\i:0..(T-F) | F+i])

/* initialize each element by applying some arbitrary function to the index */
/* F takes a tuple, the first is the index, the second is a channel */
/* F returns to B a tuple, the index (x) and a value (M) */
/* notice that we need the index, x, to be returned from B */
```

```
! Fill_Array(A, F)
   -o B^(\i:0..|A| | F:{i, B}, B:{\x, \M}, A.x = M)
```

5 Concurrency and Indeterminacy

In the above examples, careful readers should realize that Linear Janus is a concurrent and indeterministic language. Concurrency arises from parallel execution of multiple agents; indeterminacy arises from the unordering of messages in a channel, and non-disjoint definitions and conditionals.

Linear Janus is a fairly fine-grained languages. It provides asynchronous communication and also a simple way for synchronization. A *message send* is non-blocking and always succeeds, while a *message read* blocks until a message arrives. It also provides mechanisms for lexical scoping, data hiding, dynamic agent creations and reconfigurations. Various kinds of programming idioms can be expressed clearly in Linear Janus. The examples given above is merely the tip of the iceberg.

6 BNF Syntax for Linear Janus (not complete)

```
(agent)            A        ::=  1
                                 | x “:” t
                                 | A “,” A
                                 | x “^” A
                                 | c “→” A
                                 | x “:” m “⊸” A
                                 | “!” x “:” m “⊸” A “.”
(constraint)       c        ::=  term cop term | term “=” m
                                 | “int?(” term “)”
                                 | “real?(” term “)”
                                 | “char?(” term “)”
                                 | “bool?(” term “)”
                                 | “array?(” term “)”
                                 | “tuple?(” term “)”
                                 | “list?(” term “)”
(variable)         x,y,z    ::=  Symbolic-Constant
(term)             t        ::=  char | iterm | rterm | dterm
(matching term)    m        ::=  “\”x | “[” m “|” m “]” | “{” m { “, m } “}”
(arithematics)     aop      ::=  + | - | * | / | **
(comparison)       cop      ::=  = | < | > | ~= | <= | =<

(integer term)     iterm    ::=  Integer-Constant | INTEGERS
                                 | iterm aop iterm
                                 | “|” dterm “|”
                                 | aterm “.” iterm
                                 | variable
(real term)        rterm    ::=  Real-Constant | REALS | iterm | rterm aop rterm

(datatype term)    dterm    ::=  aterm | tterm | lterm

(array-term)       aterm    ::=  “<” term, { “,” term } “>”
                                 | variable “<” iterm “..” iterm “>”
                                 | aterm “+” aterm
                                 | variable

(tuple-term)       tterm    ::=  “{}” | “{” term {“,” term} “}”

(list-term)        lterm    ::=  “[” term, { “,” term} “]”
                                 | “[” term “|” term “]”
                                 | lterm “+” lterm
                                 | lterm “++” term
                                 | “[]”
                                 | variable
```

6.1 Operational Behavior of Agents

6.1.1 Logic of Linear Janus

The logic underlying Linear Janus is basically the $(\otimes, \exists, \forall, \multimap, !)$ fragment of linear logic. In the following, please note the rules for "!" are non-standard.

Structural Rule

$$(Exchange)\ \frac{\Gamma, A, B, \Delta \vdash D}{\Gamma, B, A, \Delta \vdash D}$$

Logic (using one-sided rules, with negation reduced to leaves atomic formulae)

(Identity)	$A^{\perp}, A \vdash$	(Cut)	$\frac{\Gamma,A\vdash \qquad \Delta,A^{\perp}\vdash}{\Gamma,\Delta\vdash}$
$(\mathbf{1})$	$\frac{\Gamma\vdash}{\Gamma,\mathbf{1}\vdash}$	$(\perp)$	$\perp\ \vdash$
$(\otimes)$	$\frac{\Gamma,A,B\vdash}{\Gamma,A\otimes B\vdash}$		
$(\& - 1)$	$\frac{\Gamma,A\vdash}{\Gamma,A\&B\vdash}$	$(\& - 2)$	$\frac{\Gamma,B\vdash}{\Gamma,A\&B\vdash}$
$(\oplus)$	$\frac{\Gamma,A\vdash \qquad \Gamma,B\vdash}{\Gamma,A\oplus B\vdash}$		
$(\multimap)$	$\frac{\Gamma,A^{\perp}\vdash \qquad \Delta,B\vdash}{\Gamma,A\multimap B,\Delta\vdash}$		
$(!)$	$\frac{\Gamma,A,!A\vdash}{\Gamma,!A\vdash}$	$(?)$	$\frac{F_1,\ldots,F_n,A,A\otimes F_1\otimes\ldots\otimes F_n\multimap\mathbf{1}\vdash}{F_1,\ldots,F_n,?A\vdash}$
$(\exists_L *)$	$\frac{\Gamma,A\vdash}{\Gamma,\exists xA\vdash}$		
$(\forall_L)$	$\frac{\Gamma,A[t/x]\vdash}{\Gamma,\forall xA\vdash}$		

*: X is not free in lower sequent.

Theorem 1: Cuts can be eliminated in all proofs not using non-logical axioms.

Theorem 2: $\vdash$!A $\multimap$A$\otimes$!A, $\vdash$ A$\otimes$!A $\multimap$A.

6.2 Operational Semantics of Linear Janus

Let Γ and Δ be some configurations, i.e. multisets of agents, $A_1, ..., A_n$. "$\equiv_V$" is an equivalence relation in the set of variables V. "$\mathtt{var}(\Gamma)$" is the set of all (visible) variables in Γ.

Structural Rules

$$(\Gamma, A, B, \Delta) \equiv_V (\Gamma, B, A, \Delta)$$

$$(\Gamma, A \otimes B) \equiv_V (\Gamma, A, B)$$

$$(\Gamma, !A) \equiv_V (\Gamma, !A, A)$$

$$(\Gamma, x\hat{\ }A) \equiv_V (\Gamma, A) \qquad (X \notin V \cup \mathbf{var}(\Gamma))$$

α-renaming is assumed, i.e. $x\hat{\ }A \equiv_V y\hat{\ }A[y/x]$, y is free for x in A.

$$(\Gamma, x : t, \forall \bar{y}.x : s \multimap A) \longrightarrow_V (\Gamma, A[\bar{u}/\bar{y}]) \qquad where\ s[\bar{u}/\bar{y}] = t$$

$$(\Gamma, c \to A) \longrightarrow_V (\Gamma, A) \qquad (\mathcal{C} \models c)$$

$$(\Gamma, c \to A) \longrightarrow_V (\Gamma, A) \qquad (\mathcal{C} \not\models c)$$

$$\frac{\Gamma \equiv_V \Gamma' \longrightarrow_V \Delta' \equiv_V \Delta}{\Gamma \longrightarrow_V \Delta}$$

Operational Semantics

$$\begin{array}{rcll} \theta\,[[\ \mathrm{A}\ ]] & = & \{\ \ \mathrm{B} & \mid \mathrm{A} \longrightarrow_V^* \mathrm{B} \not\longrightarrow,\ \mathrm{V} = \mathtt{var}(\mathrm{A})\ \} \\ & & \cup\ \{\ \ \bot & \mid \exists \text{ an infinite sequence of execution} : \mathrm{A} \longrightarrow_V^* \mathrm{A}_1 \longrightarrow_V^* \mathrm{A}_2 \longrightarrow_V^* ...\} \end{array}$$

Theorem 3: (Completeness of Linear Janus for atomic consequents)

$$\forall A \in FV(V).\ A \vdash x : t \otimes \top\ \underline{\mathtt{iff}}\ \exists B.A \longrightarrow_V^* B\ \underline{\mathtt{and}}\ x : t \in B$$

7 Linear Janus Type System (Not complete)

The type system of Linear Janus is static, implicit and polymorphic. The design of type system is in line with Milner's polymorphic type system for ML.

Linear Janus type system uses type scheme. Each built-in operator has a type associated with it. Each user-defined function also has a principle type which is the most general type possible given the body of the function. For example:

$$Append(\backslash L1, \backslash L2, \backslash O) : list(\alpha) \rightarrow list(\alpha) \rightarrow list(\alpha).$$

Below, we will give a formal/partial presentation of Linear Janus's type system devised by Saraswat.

In the following, lowercase Greek letters, such as σ, γ, and τ, are type variables; uppercase Greek letters, such as Γ, and Δ, are configurations; uppercase and lowercase English letters, such as, x, y, z, X, Y, Z are channels; and M is message.

Since Linear Janus functions do not return a value, we use the notation $A_y : \tau$ to mean the channel y of agent A will have type τ.

$$(Id) \qquad \Gamma, X : \tau \vdash (X : \backslash M \multimap Y : M)_Y : \tau$$

$$(Cut) \qquad \frac{\Gamma \vdash A_x : \tau \qquad \Delta, x : \tau \vdash B_y : \gamma}{\Gamma, \Delta \vdash (\exists x.A, B)_y : \gamma}$$

$$(Mix) \qquad \frac{\Gamma \vdash A_x : \tau \qquad \Gamma \vdash B_y : \tau}{\Gamma \vdash (A[z/x], B[z/y])_z : \tau}$$

$$(I) \qquad \frac{\Gamma \vdash A_x : \tau}{\Gamma \vdash (!A)_x : \tau}$$

Functions

(These are used only in the old syntax.)

$$(Func-1) \qquad \frac{\Gamma, x : \tau \vdash A_y : \gamma}{\Gamma \vdash (z : \backslash x . \backslash y \multimap A)_z : \tau \rightarrow \gamma}$$

$$(Func-2) \qquad \frac{\Gamma \vdash A_y : \tau \rightarrow \gamma, \Delta \vdash B_x : \tau}{\Gamma, \Delta \vdash (\exists x, y.\ y : x.z, A, B)_z : \gamma}$$

Products

$$(P-1) \qquad \frac{\Gamma \vdash A_x : \tau \qquad \Delta \vdash B_y : \tau}{\Gamma, \Delta \vdash (\exists x \exists y.\ z :< x, y >, A, B)_z : \gamma \times \tau}$$

$$(P-2) \qquad \frac{\Gamma \vdash A_x : \gamma \times \tau \qquad \Delta, y : \gamma, z : \tau \vdash B_r : \sigma}{\Gamma, \Delta \vdash (\exists x. A, x :< \backslash y, \backslash z > \multimap B)_r : \sigma}$$

To emphasize the differences between channels and values in the following, we use $\hat{}\tau$ to mean a channel of type τ, and $\underline{\texttt{val}}\ \tau$ to mean a value of type τ.

Channel

$$(C-1) \qquad \frac{\Gamma, M : \tau \vdash A_x : \tau'}{\Gamma, Y : \hat{}\tau \vdash (Y : \backslash M \multimap A)_x : \tau'}$$

$$(C-2) \qquad \Gamma, X : \tau \vdash (Y : X)_y : \hat{}\tau$$

Values

$$(V-1) \qquad \Gamma, X : \underline{\texttt{val}}\ \tau \vdash (Y : X)_Y : \tau$$

$$(V-2) \qquad \frac{\Gamma, x : \underline{\texttt{val}}\ \tau \vdash A_x : \tau}{\Gamma, Y : \tau \vdash (Y : \backslash M \multimap A)_x : \tau'}$$

Terms

$$(T-1) \qquad \frac{\Gamma \vdash t : \underline{\texttt{val}}\ \tau \qquad \Delta, x : \underline{\texttt{val}}\ \tau \vdash A_z : \tau'}{\Gamma, \Delta \vdash A[t/x]_z : \tau'}$$

Primitive types

Each built-in operator has an associate type. For example:

$$\begin{array}{lll} X : \underline{\texttt{val}}\ int, Y : \underline{\texttt{val}}\ int & \vdash X + Y & : \underline{\texttt{val}}\ int \\ & \vdash X - Y & : \underline{\texttt{val}}\ int \\ & \vdash X * Y & : \underline{\texttt{val}}\ int \\ & \vdash X / Y & : \underline{\texttt{val}}\ int \\ & & etc. \end{array}$$

Recursion

TBD

7.0.1 Theorem

TBD

7.0.2 Type Checking Algorithm

TBD

7.0.3 An Example of Type Checking

TBD