NOVI SAD J. MATH.
VOL. 28, NO. 2, 1998, 173-186

LINDA AS AN ABSTRACT DATA TYPE FOR CONCURRENT PROGRAMMING

Zoran Budimac, Dragan Mašulović
Institute of Mathematics, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: {zjb,masul}@unsim.im.ns.ac.yu

Abstract

In this paper we show how Linda (a parallel programming paradigm) can be implemented as an abstract data type. This approach enables Linda to be used as a concurrent programming paradigm. Therefore, Linda applications and Linda itself become available on single-processor machines and more portable.

AMS Mathematics Subject Classification (1991): 68N25
Key words and phrases: Linda, parallel programing, concurent programming

1. Introduction

Linda is a parallel programming paradigm. When injected into an existing sequential programming language in a suitable way, a parallel programming language for the target parallel platform is obtained.

Rather than constructing a new compiler for a specific parallel platform, we show how to implement Linda as an abstract data type, on top of existing process synchronization mechanisms. We obtain implementations of Linda on a high level of abstraction because they almost completely rely on

the underlying process synchronization mechanisms provided by the operating system and/or the programming language. Linda implemented as an abstract data type offers several advantages:

- Linda becomes another (alternative) mechanism for process synchronization, (besides condition variables, semaphores, etc.) as well as a concurrent programming paradigm;
- Linda becomes more portable: to implement Linda on a new platform, it suffices to reimplement it using synchronization mechanisms of a new operating system (it is reasonable to assume that synchronization mechanisms on the new platform are based upon the underlying processor architecture and, therefore, are expected to be efficient);
- Linda applications become more portable as well: it is possible to develop and test Linda applications using Linda as an abstract data type and port it afterwards to any truly parallel architecture.

We show how to implement Linda as an abstract data type using monitors and condition variables (which are supported by the programming language). Since monitors are programming language constructs rather than constructs of the operating system, a solution presented in this paper can be directly implemented in some languages (e. g. Modula-2, Concurrent Euclid, Modula-3, Java, ...). The same approach can be used straightforwardly to implement Linda using less abstract programming language constructs and mechanisms for process synchronization (see e. g. [4] p. 72, for equivalences among the most popular process synchronization mechanisms).

The rest of the paper is organized as follows: the following two sections briefly overview the concepts of monitors, condition variables and the Linda paradigm. The fourth and the fifth section present abstract data types which implement tuples and Linda itself. The sixth section discusses the usage and possible further work on this implementation. The seventh section concludes the paper.

2. Monitors and Condition Variables

A monitor is a software module i. e., a collection of procedures and data. Monitor exports some identifiers and only those are visible outside the mon-

itor's body. That way, monitor (as any other module) hides the implementation of important data structures from its environment. Hidden data structures can be accessed only through exported procedures. Actually, the only true difference between monitors and modules is that only one monitor procedure can be active at any instant. If a process calls a monitor procedure while another monitor procedure is active, the calling process is blocked. It will resume when currently active monitor procedure has blocked or has finished its execution. This feature (together with information hiding) makes monitors suitable for implementation of mutual exclusion and/or critical regions of running processes. Monitors are programming language constructs and are usually regarded as a synchronization mechanism of higher level.

Condition variables usually go along with monitors. Condition variables can be dynamically created, can be waited on and signaled on (i. e., "announced"). When a process waits on a condition variable c, it is immediately blocked and other waiting processes are immediately allowed to enter the critical section. When a process signals on a condition variable c, one (and only one) of the processes waiting on c shall be awakened. The process signaling on some condition variable is blocked and that enables some waiting process to enter the monitor.

To summarize: process blocks when

- it calls a monitor procedure while other monitor procedure is being executed, or
- it waits on a condition variable, or
- it signals a condition variable.

After issuing 'wait' or 'signal', the process leaves the monitor hence enabling some waiting process to enter the critical section.

Monitors and condition variables are standard part of Modula-2, Concurrent Euclid and Java, and can be easily simulated in Modula-3 (condition variables are called signals in Modula-2). The behavior of monitors and condition variables can be successfully implemented using other available synchronization mechanisms.

3. Linda—A Brief Overview

Linda [1] is an explicitly-parallel programming paradigm designed to support:

- asynchronous communication between processes,
- dynamic allocation of processes,
- efficient distribution of algorithm and data, and
- independence of underlying hardware (i. e. number of processing elements and the topology).

It is a "software injection" that consists of six primitive instructions: `in`, `inp`, `rd`, `rdp`, `out`, and `eval` which are injected into a sequential language. A language L with the injection is usually referred to as L-Linda.

The entire available memory of the system behaves as a unique continuous piece of memory, called the tuple space. Tuple space is a bag whose elements are tuples of data of different lengths. The fields of a tuple need not be of the same type, but are required to be of a simple type. A process (Linda-worker in jargon) manipulates the tuple space via primitive instructions.

`out` adds a tuple to the tuple space. E. g. `out("GRAPH", k + 1, g(k), TRUE)` adds the quadruple `("GRAPH", k + 1, g(k), TRUE)` to the tuple space.

Instructions `in` and `rd` browse the tuple space and look for the tuple that matches the template supplied as the argument to the instruction. A template is a tuple with two kinds of fields: actual fields and formal fields. Actual fields are represented by expressions, while formal fields are represented by special syntax constructs (usually `?variable`). Both instructions block the current process and wait for the tuple that matches the template to appear in the tuple space. The tuple matches the template iff they are of the same length, values at positions of actual fields are equal, while values at positions of formal fields are of the same (simple) type. As soon as such a tuple appears, the process awakens, copies the corresponding values from the retrieved tuple into variables at formal fields' positions and proceeds. There is a slight difference between `in` and `rd`: after the match has been found, `in` removes the

matching tuple from the tuple space, while `rd` leaves it there. Examples: `in("GRAPH", ?n, 2k + 1, ?OK)`, `rd("STACK", id, ?val, ?next)`.

Instructions `inp` and `rdp` are modifications of `in` and `rd`, respectively, which do not wait for the tuple to appear in the tuple space. They browse the tuple space only once. If the match is found they behave as their analogons, but if the match is not found, the failure is somehow reported and the execution proceeds.

Instruction `eval` is a modification of `out`. It forks a new process which evaluates its argument and then performs `out`. After that the process disintegrates. `eval` is Linda's way to dynamically create processes.

4. Tuple as an Abstract Data Type

For the sake of simplicity, at the level of implementation we assume no distinction between tuples and templates. The same abstract data is used in both cases. A tuple/template is represented by a list of cells. E. g. the template generated by Linda primitive `in(i, ?b, r)` (where `i`, `b` and `r` are, respectively, an integer, a boolean and a real variable) is shown in figure 1.

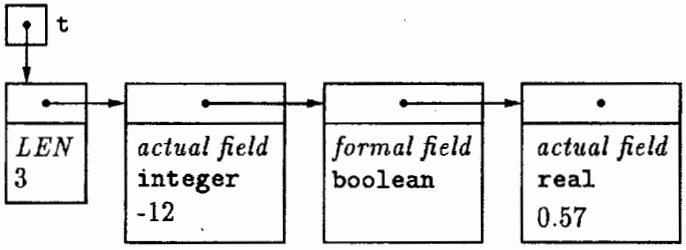

Figure 1: Representation of a template

Module `Tuple` implements the abstract data type `Tuple.T` and exports operations to manipulate tuples. The interface of the module `Tuple` follows (in Modula-2 syntax):

```
DEFINITION MODULE Tuple;
IMPORT String;

TYPE T;    (* tuple/template data type *)

TYPE FormalType = (* the type of a formal field in a template *)
```

```
    (shInt, int, lngInt, card, lngCard, real, lngReal, set, ch,
    str, bool);

PROCEDURE New(): T;                          (* create a new tuple *)
PROCEDURE Kill(VAR t: T);                    (* kill t *)
PROCEDURE IsEmpty(t: T): BOOLEAN;            (* is t empty? *)
PROCEDURE IsTemplate(t: T): BOOLEAN;         (* is t a template? *)
PROCEDURE TheyMatch(t1, t2: T): BOOLEAN;     (* do t1 and t2 match? *)
PROCEDURE DoMatch(t1, t2: T);                (* match t1 and t2 *)

(* Adding new field 'v' to the existing tuple 't' *)
(* NB: There is one procedure per simple type     *)
PROCEDURE AddShInt  (t: T; v: SHORTINT)   : T;
PROCEDURE AddInt    (t: T; v: INTEGER)    : T;
PROCEDURE AddLngInt (t: T; v: LONGINT)    : T;
PROCEDURE AddCard   (t: T; v: CARDINAL)   : T;
PROCEDURE AddLngCard(t: T; v: LONGCARD)   : T;
PROCEDURE AddReal   (t: T; v: REAL)       : T;
PROCEDURE AddLngReal(t: T; v: LONGREAL)   : T;
PROCEDURE AddSet    (t: T; v: BITSET)     : T;
PROCEDURE AddCh     (t: T; v: CHAR)       : T;
PROCEDURE AddStr    (t: T; v: String.T)   : T;
PROCEDURE AddBool   (t: T; v: BOOLEAN)    : T;
PROCEDURE AddFormal (t: T; ty: FormalType): T;

(* Get the field 'v' from position 'pos' of tuple 't' *)
(* NB: There is one procedure per simple type        *)
PROCEDURE GetShInt  (t: T; pos: CARDINAL; VAR v: SHORTINT);
PROCEDURE GetInt    (t: T; pos: CARDINAL; VAR v: INTEGER);
PROCEDURE GetLngInt (t: T; pos: CARDINAL; VAR v: LONGINT);
PROCEDURE GetCard   (t: T; pos: CARDINAL; VAR v: CARDINAL);
PROCEDURE GetLngCard(t: T; pos: CARDINAL; VAR v: LONGCARD);
PROCEDURE GetReal   (t: T; pos: CARDINAL; VAR v: REAL);
PROCEDURE GetLngReal(t: T; pos: CARDINAL; VAR v: LONGREAL);
PROCEDURE GetSet    (t: T; pos: CARDINAL; VAR v: BITSET);
PROCEDURE GetCh     (t: T; pos: CARDINAL; VAR v: CHAR);
PROCEDURE GetStr    (t: T; pos: CARDINAL; VAR v: String.T);
PROCEDURE GetBool   (t: T; pos: CARDINAL; VAR v: BOOLEAN);

END Tuple.
```

Operations `New` and `Add...` are used to construct a tuple (or a template). Operations `Get...` are used to retrieve information from tuples after match-

ing. Matching tuples with templates reduces to substitution of cells which represent formal fields in the template by copies of corresponding cells of the tuple. Thus, if `t` were a template and `t1` were a tuple, after `DoMatch(t, t1)` `t` would become a tuple. As an example, consider the following sequence of instructions which has the effect of `in(i, ?b, r)` (Modula-2 syntax):

```
VAR
    t: Tuple.T;
    i: INTEGER;
    b: BOOLEAN;
    r: REAL;
BEGIN
    t := Tuple.New();
    t := Tuple.AddInt(t, i);
    t := Tuple.AddFormal(t, Tuple.bool);
    t := Tuple.AddReal(t, r);
    Linda.In(t);
    Tuple.GetBool(t, 2, b)
END
```

5. Linda as an Abstract Data Type

Linda can be implemented as a monitor `Linda` which exports the following five procedures (we use a Modula-2 like pseudolanguage):

```
DEFINITION MONITOR Linda;
IMPORT Tuple;

VAR DoneP: BOOLEAN;

PROCEDURE Out(t: Tuple.T);
PROCEDURE In (t: Tuple.T);
PROCEDURE Inp(t: Tuple.T);
PROCEDURE Rd (t: Tuple.T);
PROCEDURE Rdp(t: Tuple.T);
PROCEDURE NewWorker(script: PROC; memSize: LONGCARD);

END Linda.
```

The tuple space is implemented as a list of tuples and is encapsulated in module `Linda`. Two more lists are required for efficient synchronization:

`WaitingIn` the list of templates which are waiting for an `in`, and

`WaitingRd` the list of templates which are waiting for a `rd`.

Truly speaking, the latter two lists are lists of pairs $\langle P, C \rangle$, where P is a template and C is a condition variable to be signaled as soon as the template P finds its match. The signal awakens the process that has been waiting for the match. The corresponding declarations in a Modula-2 like pseudo-language are:

```
VAR
    TupleSpace: LIST OF Tuple.T;
    WaitingIn:  LIST OF PAIR(Tuple.T, CondVar.T);
    WaitingRd:  LIST OF PAIR(Tuple.T, CondVar.T);
```

We shall now describe the implementation of `in`, `rd` and `out` in a pseudo-language. The implementation is a straightforward translation of standard behaviour of these instructions.

```
PROCEDURE In(VAR p: Tuple.T);
VAR
    t: Tuple.T;
    c: CondVar.T;
BEGIN
    (* Look for p in TupleSpace *)
    FOREACH t IN TupleSpace DO
        IF Tuple.TheyMatch(p, t) THEN
            Tuple.DoMatch(p, t);
            REMOVE t FROM TupleSpace;
            RETURN
        END
    END;

    (* If not found, the process has to to wait for it *)
    c := CondVar.New();
    ADD PAIR(p, c) TO WaitingIn;
```

```
    CondVar.Wait(c)
END In;


PROCEDURE Rd(VAR p: Tuple.T);
VAR
    t: Tuple.T;
    c: CondVar.T;
BEGIN
    (* Look for p in TupleSpace *)
    FOREACH t IN TupleSpace DO
        IF Tuple.TheyMatch(p, t) THEN
            Tuple.DoMatch(p, t);
            RETURN
        END
    END;

    (* If not found, the process has to to wait for it *)
    c := CondVar.New();
    ADD PAIR(p, c) TO WaitingRd;
    CondVar.Wait(c)
END Rd;


PROCEDURE Out(t: Tuple.T);
VAR
    p: Tuple.T;
    c: CondVar.T;
BEGIN
    (* Awake all the processes waiting for t in WaitingRd *)
    FOREACH PAIR(p, c) IN WaitingRd DO
        IF Tuple.TheyMatch(p, t) THEN
            Tuple.DoMatch(p, t);
            REMOVE PAIR(p, c) FROM WaitingRd;
            CondVar.Signal(c)
        END
    END;

    (* Awake only the first process waiting for t in WaitingIn *)
```

```
    FOREACH PAIR(p, c) IN WaitingIn DO
        IF Tuple.TheyMatch(p, t) THEN
            Tuple.DoMatch(p, t);
            REMOVE PAIR(p, c) FROM WaitingIn;
            CondVar.Signal(c);
            RETURN
        END
    END;

    (* If no process waits for t in WaitingIn, add the tuple to
       TupleSpace *)
    ADD t TO TupleSpace
END Out;
```

Note that `Linda.In` and `Linda.Rd` add the template they are looking for to an appropriate list if they cannot find matches in the `TupleSpace`, even if the same template already exists in the list. That way each process waiting for `in` or `rd` has its own template and its own condition variable.

The implementation of Linda primitives `inp` and `rdp` is similar to the implementation of `in` and `rd`, respectively. The only difference is that in case they do not manage to find a match during one scan of the `TupleSpace`, they signalize the failure through the global variable `Linda.DoneP` instead of waiting for the signal. The effect of Linda primitive `eval` is obtained with the help of `Linda.NewWorker` and `Linda.Out`. For example, the following sequence of Modula-2 instructions has the effect of `eval(1, f(i), j+1)`:

```
VAR
    intVar1, intVar2: INTEGER;

PROCEDURE f(x: INTEGER): INTEGER;
    ...
END f;

PROCEDURE Eval1;
VAR
    t: Tuple.T;
BEGIN
    t := Tuple.New();
    t := Tuple.AddInt(t, 1);
```

```
    t := Tuple.AddInt(t, f(intVar1));
    t := Tuple.AddInt(t, intVar2);
    Linda.Out(t)
END Eval1;

BEGIN
    ...
    intVar1 := i;
    intVar2 := j;
    Linda.NewWorker(Eval1, 1000);
    ...
END
```

`Linda.NewWorker` can be easily implemented as a direct translation to the underlying facility that creates new processes.

6. Implementations, Usage and Further Work

An application that employs Linda uses the module `Tuple` to build tuples and then the monitor `Linda` to manipulate created tuples. The full potential is obtained if the modules are regarded as a support to a translator from a higher level Linda programming language (e. g. Modula-2-Linda or Modula-3-Linda). Such a language should support Linda primitives as syntax constructs. Each construct, then, is to be translated to a sequence of appropriate procedure calls. For example, the following Modula-2-Linda excerpt

```
PROCEDURE DoSomething;
VAR
    u, N: INTEGER;
BEGIN
    N := 0;
    WHILE INP("data" ?u) DO
        INC(N);
        EVAL("res", N, F(u))
    END;
    ProcessResults
END DoSomething;
```

translates to

```
IMPORT Tuple, Linda, String;

VAR
    intVar1, intVar2: INTEGER;

PROCEDURE Eval1;
VAR
    t: Tuple.T;
BEGIN
    t := Tuple.New();
    t := Tuple.AddStr(t, String.New("res"));
    t := Tuple.AddInt(t, intVar1);
    t := Tuple.AddInt(t, F(intVar2));
    Linda.Out(t)
END Eval1;

PROCEDURE DoSomething;
VAR
    u, N: INTEGER;
    t: Tuple.T;
BEGIN
    N := 0;
    LOOP
        t := Tuple.New();
        t := Tuple.AddStr(t, String.New("data"));
        t := Tuple.AddFormal(t, Tuple.int);
        Linda.Inp(t);
    IF NOT Tuple.DoneP THEN EXIT END;
        Tuple.GetInt(t, 2, u);
        INC(N);
        intVar1 := N;
        intVar2 := u;
        Linda.NewWorker(Eval1, 1000)
    END;
    ProcessResults
END DoSomething;
```

Ideas presented in this paper are implemented both in Modula-2 and Modula-3. Implementation in Modula-3 supports multi-tasking based on Modula-3 threading facilities. Modula-3 multi-threading and mutex constructs have been used to implement the `eval` primitive as well as the `in/out` control of the tuple space.

Implementation in Modula-2 supports multi-tasking as well. The implementation relies on the multi-tasking support for Modula-2 under DOS developed at the Institute of Mathematics, University of Novi Sad.

Many other improvements to the ideas presented in this paper are possible. To enhance the efficiency of the Linda Kernel, tuples and templates could be organized into binary search trees, sorted with respect to the tuple dimensions. Priorities among the processes could also be introduced, so that processes with higher priority are to be awakened before others.

7. Conclusion

We proposed an implementation of the Linda paradigm as an abstract data type. Mutual exclusion and process synchronization needed in the Linda Kernel are implemented using monitors and condition variables, respectively. Having in mind the semantics of these synchronization primitives, similar implementations could be built using other synchronization primitives (semaphores, rendezvous, etc.)

Although Linda is about to celebrate her thirteenth birthday, her simplicity and elegance are still attractive [2, 3, 5]. Papers [2] and [3] present implementations of Linda that rely on powerful concepts of the respective languages. The implementation presented in this paper has above all other things a great educational value—it brings the Linda paradigm down to the cheapest uni-processor machines. The simple design carried out in a high level programming language may serve as a good example to Computer Science students with special interest in operating systems. The existence of the implementation of this kind is important because it enhances the portability of Linda implementations as well as Linda applications. Linda can now be regarded even as a concurrent programming mechanism like monitors, semaphores and so on.

Acknowledgements

The authors would like to acknowledge Vladimir Blagojević for the implementation of the multi-tasking library for Modula-2 under DOS, Miljan Milinković for the implementation of Linda as an abstract data type in Modula-2, and Srdjan Mladjenović for the implementation of Linda as an abstract data type in Modula-3.

References

[1] Ahuja S. et al., Linda and friends, IEEE Computer, pp. 26–34, August 1986.

[2] Jellinghaus J., Eiffel Linda: an object-oriented Linda dialect, SIGPLAN Notices, Vol. 25, No. 12 (1990), pp. 70–84.

[3] Ledru P., Space: Implementation of a Linda System in Java, SIGPLAN Notices, Vol. 33, No. 8 (1998), pp. 48–50.

[4] Tanenbaum A. S., Operating Systems—Design and Implementation, Prentice Hall Int. 1987

[5] Yuen C. K., Wong W. F., A self interpreter for BaLinda Lisp, SIGPLAN Notices, Vol. 25, No. 5 (1990), pp. 39–58.

Received by the editors July 8, 1998.