

The Parallel Conditional

S W Postma and N C K Phillips

Department of Computer Science, University of Natal, P O Box 375, Pietermaritzburg, 3200

Abstract

The parallel conditional is a new but natural programming language construct. It is particularly suited to evaluation on parallel machines and generalizes other well known conditionals.

Received November 1987, Accepted August 1988

1. Introduction

The programming languages in common use at any time tend to reflect the architecture of the computers that are in use at that time. Thus the long use of the inherently sequential and deterministic von Neumann architecture machines has led to the widespread use of languages that are inherently sequential and deterministic. However, the advent of relatively affordable parallel machines has greatly stimulated interest in parallel programming languages, and in individual language constructs that are intended for parallel evaluation. Our present purpose is to explore one such new construct, the parallel conditional.

The parallel conditional is a natural generalisation of both LISP's COND and Dijkstra's guarded conditional that is suited to parallel processing. It arose while designing the new language QUADLISP and experience with it in this context suggested that it would be worthwhile to make an independent study of it, and of its relationship with other conditionals.

Given n predicate-expression pairs (P_i, X_i) a conditional selects an X_i according to some condition formulated in terms of the P_i . We investigate the guarded conditional of Dijkstra [1], the COND of LISP, and a new construct, the parallel conditional. In each case, following Dijkstra, the P_i are called guards.

2. Conditionals and Evaluation Strategies

Dijkstra's conditional selects non-deterministically some X_i where P_i is true, provided that all P_j are defined. If there is no such P_i or if some P_j is undefined, his conditional "aborts" - which we shall take to mean "becomes undefined". Dijkstra's notation for his conditional is

if $P_1 \rightarrow X_1$ [] ... [] $P_n \rightarrow X_n$ **fi**

and we call this expression DC (for Dijkstra's Conditional).

LISP's COND conditional selects the X_i such that P_i is true and P_j is false for $1 \leq j < i$. If there is no

such P_i then the value of the COND is undefined. Note that if COND selects X_i then P_j may be undefined for $j > i$. The LISP notation for the COND conditional is $(\text{COND } (P_1 X_1) \dots (P_n X_n))$, and we call this expression LC (for LISP's Conditional).

The parallel conditional is the simplest of the three conditionals to describe: it selects non-deterministically an X_i such that P_i is true, although all P_j need not be defined, and is undefined if there is no such true P_i . Our notation for the parallel conditional is $\{P_1 \rightarrow X_1, \dots, P_n \rightarrow X_n\}$, and we shall call this expression PC (for parallel Conditional). An essential difference between these conditionals lies in the evaluation strategies that are appropriate for the guards P_1, \dots, P_n in each case.

For PC it is appropriate to evaluate the P_j in parallel, and to return X_i , where P_i is a guard which evaluates to true. We intend non-determinism to be the "don't care" variety, so that it would be legal to return X_i where P_i is the first guard to evaluate to true. Note that the remaining guards may be true or false or even undefined: their attempted evaluation would be terminated on selecting X_i .

For LC the sequence P_1, \dots, P_n should be evaluated sequentially from the left until a true P_i is reached. X_i is then returned, and the remaining P_j are not evaluated (and could even be undefined).

For DC the guards can be evaluated either in parallel or sequentially, but since it is required that they all be defined, their complete evaluation should be attempted before selecting an X_i .

PC is defined if some P_i is true; LC is defined if some P_i is true and where P_j are defined for $j < i$; DC is defined if some P_i is true and all P_j are defined.

3. Correspondence with Three Valued Regular Logics

$\{P_1 \rightarrow X_1, P_2 \rightarrow X_2\}$ is defined precisely when P_1 **por** P_2 is true, where **por** is the three-valued disjunction defined by the following truth table (# denoting undefined):

		P ₂		
		t	f	#
P ₁ por P ₂ :	t	t	t	t
	f	t	f	#
	#	t	#	#

(COND (P₁X₁) (P₂X₂)) is defined precisely when P₁ cor P₂ is true, where cor is defined by:

		P ₂		
		t	f	#
P ₁ cor P ₂ :	t	t	t	t
	f	t	f	#
	#	#	#	#

if P₁ → X₁ [] P₂ → X₂ fi is defined precisely when P₁ wor P₂ is true, where wor is defined by:

		P ₂		
		t	f	#
P ₁ wor P ₂ :	t	t	t	#
	f	t	f	#
	#	#	#	#

por, cor and wor are commonly called the parallel, conditional and weak three-valued disjunctions. Each of these, together with three-valued negation, generates one of the four regular three-valued logics of Kleene [2]. (The fourth regular three-valued logic is a trivial symmetric variation of the one that corresponds to cor.) The regular logics are the only three-valued extensions of two-valued logic that have the property that the propositional connectives are partial recursive predicates when applied to partial recursive arguments.

A more striking relationship between conditionals and the regular three-valued logics will appear in the next section, but firstly we clarify what we mean by "undefined".

The various meanings of "undefined" are manifest, and trying to be quite precise about them leads to sticky philosophical problems that we wish to avoid. In this article we have three situations in mind. Firstly, an expression may be defined for some values of sub-expressions, and undefined for others. For example, 4/c is undefined when c has value zero. Secondly, an expression is undefined if attempting to evaluate it results in a non-terminating process. A famous example of this from the lambda calculus is the expression (λx.(xx)λx.(xx)). Lastly, if we are working in the context of three-valued logic, a partial predicate P is said to be undefined at (x₁, ..., x_n) if (x₁, ..., x_n) is not in the domain of P, and in this case the truth-value of P(x₁, ..., x_n) is #.

4. Weakest Precondition Semantics for Conditionals

Weakest precondition semantics, invented by Dijkstra and described in [1], can be used to illuminate both the differences between the conditionals and their correspondence with regular logics.

Let Q,R be predicates, X be a computer process. Then Q{X}R is the assertion: if X is started with Q true, then X terminates, and on termination R is true. The weakest precondition for R to be true after executing X is defined to be the weakest Q such that Q{X}R, and is denoted wp(X,R). Weakest preconditions are total predicates. Thus for any X, wp(X,false) is false (here false is the constant predicate whose value is f), while if X is a non-terminating process then, for any R, wp(X,R) is false. The semantics of a process X can be determined by specifying how wp(X,R) is constructed for any R.

Assuming that all the guards, the P_i, are total, Dijkstra defined wp(DC,R) to be

$$(P_1 \text{ or } \dots \text{ or } P_n) \text{ and } (P_1 \rightarrow \text{wp}(X_1, R)) \text{ and } \dots \text{ and } (P_n \rightarrow \text{wp}(X_n, R))$$

where the logical connectives are as usual in two-valued logic.

We are interested in the case that the guards are partial predicates, especially since this is particularly appropriate to the conditionals LC and PC. In this case Dijkstra suggests that the above wp(DC,R) "be prefixed, with a cand, by the requirement that the initial state lies in the domain of all the guards." (cand is the and of the regular logic determined by negation and cor). Since the use of cand already involves three-valued logic, we look for an alternative to Dijkstra's suggestion that will allow firstly, expressing "the initial state lies in the domain of all the guards" by using logical connectives, and secondly can be modified easily to produce wp's for the other conditionals. Our solution is to define wp(DC,R) to be

$$(P_1 \text{ wor } \dots \text{ wor } P_n) \hat{\wedge} (P_1 \Rightarrow \text{wp}(X_1, R)) \hat{\wedge} \dots \hat{\wedge} (P_n \Rightarrow \text{wp}(X_n, R))$$

where wor is weak disjunction and $\hat{\wedge}$ and \Rightarrow have the tables below.

		P $\hat{\wedge}$ Q			P \Rightarrow Q		
		Q			Q		
		t	f	#	t	f	#
P	t	t	f	f	t	t	f
	f	f	f	f	f	t	t
	#	f	f	f	#	t	t

$\hat{\wedge}$ and \Rightarrow are respectively conjunction and implication

in Bochvar's exterior logic which is discussed in Rescher [3]. \wedge yields f if one of its arguments is $\#$, so our expression for $wp(DC,R)$ is a total predicate and has value f when one or more of the guards is undefined. In the case that all guards are total, our $wp(DC,R)$ is equivalent to Dijkstra's.

Some reflection on the informal semantics for LISP's COND reveals that $wp(LC,R)$ can be defined to be

$$(P_1 \text{ cor } \dots \text{ cor } P_n) \wedge (P_1 \Rightarrow wp(X_1,R) \wedge \dots \wedge (P_n \Rightarrow wp(X_n,R))).$$

From our earlier discussion of the informal semantics of the parallel conditional it should be clear that an appropriate expression for $wp(PC,R)$ is

$$(P_1 \text{ por } \dots \text{ por } P_n) \wedge (P_1 \Rightarrow wp(X_1,R) \wedge \dots \wedge (P_n \Rightarrow wp(X_n,R))).$$

The weakest preconditions show clearly how the three conditionals correspond to the three regular logics.

5. Relative Strength of the Parallel Conditional

Several constructs, including the other conditionals, can be defined in terms of the parallel conditional and negation, so the parallel conditional is relatively strong. We list some results.

1. The parallel conditional and negation can define the regular logics. To show this we need only define **por**, **cor** and **wor**.

$$P \text{ por } Q = \{P \rightarrow t, Q \rightarrow t, \sim P \rightarrow \{Q \rightarrow t, \sim Q \rightarrow f\}, \sim Q \rightarrow \{P \rightarrow t, \sim P \rightarrow f\}\}$$

$$P \text{ cor } Q = \{P \rightarrow t, \sim P \rightarrow \{Q \rightarrow t, \sim Q \rightarrow f\}\}$$

$$P \text{ wor } Q = \{P \rightarrow \{Q \rightarrow t, \sim Q \rightarrow t\}, Q \rightarrow \{P \rightarrow t, \sim P \rightarrow t\}\} \sim P \rightarrow \{Q \rightarrow t, \sim Q \rightarrow f\}, \sim Q \rightarrow \{P \rightarrow t, \sim P \rightarrow f\}$$

2. The parallel conditional and negation can define if-then-else. For, define **if P then X else Y** to be $\{P \rightarrow X, \sim P \rightarrow Y\}$.

3. The parallel conditional and negation can define LISP's COND. For, define $(\text{COND } (P_1 X_1))$ to be $\{P_1, X_1\}$ and $(\text{COND } (P_1 X_1) \dots (P_n X_n))$ to be $\{P_1 \rightarrow X_1, \sim P_1 \rightarrow (\text{COND } (P_2 X_2) \dots (P_n X_n))\}$ if $n > 1$.

4. The parallel conditional and negation can define **if $P_1 \rightarrow X_1 \square \dots \square P_n \rightarrow X_n$ fi**. For, define **if $P_1 \rightarrow X_1$ fi** to be $\{P_1 \rightarrow X_1\}$, **if $P_1 \sim X_1 \square P_2 \rightarrow X_2$ fi** to be $\{P_1 \rightarrow \{P_2 \rightarrow X_2, \sim P_2 \rightarrow X_1\}, P_2 \rightarrow \{P_1 \rightarrow X_2, \sim P_1 \rightarrow X_2\}\}$, and if $n > 2$, **if $P_1 \rightarrow X_1 \square \dots \square P_n \rightarrow X_n$ fi** to be $\{P_1 \rightarrow Y_1, \dots, P_n \rightarrow Y_n\}$

where, for $1 \leq i \leq n$, Y_i is the expression which results on removing $P_i \rightarrow X_i$ and $\sim P_i \rightarrow X_i$ from **if $P_1 \rightarrow X_1 \square \dots \square P_n \rightarrow X_n$ fi**.

5. The parallel conditional can select one of X_1, \dots, X_n non-deterministically. For, $\{t \rightarrow X_1, \dots, t \rightarrow X_n\}$ does this selection.

Finally, it must be mentioned that several other parallel non-deterministic operators have been proposed in the literature. It is difficult to quantify the relative strength of these operators, since rarely is one obtainable from another without using considerable additional machinery. For example, consider the ambiguous function **amb** of McCarthy [4]. **amb**(x,y) selects non-deterministically one of x,y if both are defined, otherwise whichever is defined, but is itself undefined if neither is defined. In the case that all of X_1, \dots, X_n are defined, **amb**(X_1, \dots, X_n) is $\{t \rightarrow X_1, \dots, t \rightarrow X_n\}$. But if the X_i are possibly all undefined, **amb**(X_1, \dots, X_n) is $\{\text{is-defined}(X_1) \rightarrow X_1, \dots, \text{is-defined}(X_n) \rightarrow X_n\}$, where **is-defined**(X)= t if X is defined, f otherwise. Since the **is-defined** predicate is in general not computable, this relationship between **amb** and the strong conditional is not very illuminating. On the other hand, if **amb** and local scoping were available in LISP, then consider

$$(\text{let}(j(\text{amb}(\text{if } P_1 \text{ then } 1 \text{ else LOOP}) \dots (\text{if } P_n \text{ then } n \text{ else LOOP})))) (\text{COND } ((\text{equal } j \ 1) X_1) \dots (\text{equal } j \ n) X_n)))$$

where LOOP is a non-terminating LISP process. This would have the effect of $\{P_1 \rightarrow X_1, \dots, P_n \rightarrow X_n\}$, but again this is not very illuminating, especially since we have used COND, which we have shown can itself be considered to be a special case of the parallel conditional.

6. Implementation

By distributing processes to parallel processors, a useful result can be obtained even if some of the individual processes might abort in error or loop forever. The straightforward way to implement the parallel conditional is to distribute the evaluation of the guards to parallel processors. From the implementation point of view the only interesting problem is how to ensure clean termination of the evaluation of remaining guards after a P_i has been found to be true. Such problems have been extensively studied and a variety of solutions can be found in Brinch Hansen [5], Hoare [6] and elsewhere.

We are implementing the parallel conditional in an extension of LISP called QUADLISP [7].

7. Conclusion

The parallel conditional proves to be a powerful unifying concept. In programming languages it has the same role as the (unbounded) minimisation operator in recursive function theory – it is used to specify partial functions. The concept can thus be used to simplify the definition of the semantics of programming languages in readily understood concepts, and also used to simplify the implementation of language processors by requiring fewer basic implementation routines.

References

- [1] E W Dijkstra, [1976], *A Discipline of Programming*, Prentice-Hall, New Jersey.
- [2] S C Kleene, [1952], *Introduction to metamathematics*, North-Holland, Amsterdam.
- [3] W Rescher, [1969], *Manyvalued Logic*, McGraw-Hill, New York.
- [4] J McCarthy, [1963], A Basis for a Mathematical Theory of Computation, in *Computer Programming and Formal Systems*, North-Holland, 33-70.
- [5] P Brinch Hansen, [1978], Distributed processes: A Concurrent Programming Concept, *Comm. ACM*, **21** (11), 934-941.
- [8] C A R Hoare, [1978], Communicating Sequential Processes, *Comm. ACM*, **21** (8), 888-877.
- [7] S W Postma, [1985], Introduction to Quadlisp/88, Technical Report, Dept of Computer Science, University of Natal, Pietermaritzburg.