A Characterization of Lazy and Eager Semantic Solutions to the Linda Predicates Ambiguity Problem

Marc L. Smith [*]
Computer Science Department, Colby College
Waterville, ME 04901-8858, USA
mlsmith@colby.edu; voice: 207.872.3672; fax: 207.872.3801

Charles E. Hughes [†]
School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816-2362, USA
ceh@cs.ucf.edu; voice: 407.823.2762; fax: 407.823.5419

March 5, 2003

Abstract

The problems associated with Linda predicate operations `inp()` and `rdp()` are well known but not always well understood. One problem, from a purely academic standpoint, is that of semantic ambiguity in the case of failure. Despite this problem, commercial tuple space implementations all provide some version of the Linda predicate operations, and thus, problems concerning safety and liveness arise. Solutions to address these problems range from proposing extensions to an existing model of computation (CSP) to proposing alternative definitions for the Linda predicate operations themselves. While these solutions may at first appear unrelated, it is possible to relate these two disparate approaches in the context of lazy and eager semantics. The benefits of this characterization include a clearer understanding of the Linda predicates ambiguity problem, two respective solutions to this problem, and the importance of considering both lazy and eager semantic perspectives as techniques for problem solving in parallel and distributed systems.

Keywords: Linda predicates, semantic ambiguity, CSP, lazy, eager

1 Introduction

2 Background

This section presents the background information required for characterizing the two solutions to the Linda predicates ambiguity problem presented in Section 3. First, Section 2.1 gives an overview of the Linda and Tuple Space model for parallel and distributed computation. Next, Section 2.2 describes the Linda predicate operations, including the associated ambiguity problem. Finally, Section 2.3 introduces the terminology and implications of lazy and eager semantics.

[*]Presenting author.
[†]Partially supported by the National Science Foundation under grant EIA 9986051.

2.1 Linda and Tuple Space

The tuple space model and Linda language are due to Gelernter [5]. Linda is distinct from pure message passing-based models (e.g., Actors [2]). Unlike message passing models, tuple space exhibits what Gelernter called communication orthogonality, referring to interprocess communications decoupled in destination, space, and time. The tuple space model is especially relevant to discussion of concurrency due to the current popularity of commercial tuple space implementations, such as Sun's JavaSpaces [4] and IBM's T Spaces [10].

Linda is not a complete programming language; it is a communication and coordination language. Linda is intended to augment existing computational languages with its coordination primitives to form comprehensive parallel and distributed programming languages. The Linda coordination primitives are `rd`(), `in`(), `out`(), and `eval`(). The idea is that multiple Linda processes share a common space, called a tuple space, through which the processes are able to communicate and coordinate using Linda primitives.

A tuple space may be viewed as a container of tuples, where a tuple is simply a group of values. A tuple is considered active if one or more of its values is currently being computed, and passive if all of its values have been computed. A Linda primitive manipulates tuple space according to the template specified in its argument. Templates represent tuples in a Linda program. A template extends the notion of tuple by distinguishing its passive values as either *formal* or *actual*, where formal values, or *formals*, represent typed wildcards for matching. Primitives `rd`() and `in`() are synchronous, or blocking operations; `out`() and `eval`() are asynchronous.

The `rd`() and `in`() primitives attempt to find a tuple in tuple space that matches their template. If successful, these primitives return a copy of the matching tuple by replacing any formals with actuals in their template. In addition, the `in`() primitive, in the case of a match, removes the matching tuple from tuple space. In the case of multiple matching tuples, a nondeterministic choice determines which tuple the `rd`() or `in`() operation returns. If no match is found, these operations block until such time as a match is found. The `out`() operation places a tuple in tuple space. This tuple is a copy of the operation's template. Primitives `rd`(), `in`(), and `out`() all operate on passive tuples.

All Linda processes reside as value-yielding computations within the active tuples in tuple space. Any Linda process can create new Linda processes through the `eval`() primitive. Execution of the `eval`() operation places an active tuple in tuple space, copied from the template. When a process completes, it replaces itself within its respective tuple with the value resulting from its computation. When all processes within a tuple replace themselves with values, the formerly active tuple becomes passive. Only passive tuples are visible for matching by the `rd`() and `in`() primitives; thus active tuples are invisible.

Communication orthogonality refers to three desirable attributes that seem particularly well-suited for distributed computing. Tuple space acts as a conduit for the generation, use, and consumption of information between distributed processes. First, unlike message passing systems, where a sender must typically specify a message's recipient, information generators do not need to know who their consumers will be, nor do information consumers need to know who generated the information they consume. Gelernter called this attribute *destination decoupling*. Next, since tuples are addressed associatively, through matching, tuple space is a platform independent shared memory. Gelernter called this attribute *space decoupling*. Finally, tuples may be generated long before their consumers exist, and tuples may be copied or consumed long after their generators cease to exist. Gelernter called this attribute *time decoupling*. Distinct from both pure shared memory and message passing paradigms, Gelernter dubbed Linda and Tuple space to be a form of *generative communication*.

2.2 Predicates and Ambiguity

In addition to the four primitives `rd`(), `in`(), `out`(), and `eval`(), the Linda definition once included predicate versions of `rd`() and `in`(). Unlike the `rd`() and `in`() primitives, predicate operations `rdp`() and `inp`() were nonblocking primitives. The goal was to provide tuple matching capabilities without the possibility of blocking. The Linda predicate operations seemed like a useful idea, but their meaning proved to be semantically ambiguous, and they were subsequently removed from the formal Linda definition.

Predicate operations `rdp()` and `inp()` attempt to match tuples for copy or removal from tuple space. A successful operation returns the value one (1) and the matched tuple in the form of a template. A failure, rather than blocking, returns the value zero (0) with no changes to the template. When a match is successful, no ambiguity exists. It is not clear, however, what it means when a predicate operation returns a zero.

The ambiguity of the Linda predicate operations is subtle and, in general, not well understood; it is a consequence of reasoning about concurrency through an arbitrary interleaving of tuple space interactions. Jensen noted that when a predicate operation returns zero, "only if every existing process is captured in an interaction point does the operation make sense." [8]. For a complete discussion of such an interaction point in tuple space, see Smith *et al.* [9]. For more in general about reasoning with interleaved traces, see Hoare's seminal work in Communicating Sequential Processes (CSP) [6].

Briefly, the meaning of a failed predicate operation breaks down in the presence of concurrency expressed as an arbitrary interleaving of atomic events. This breakdown in meaning is due to the restriction of representing the history of a computation as a total ordering of atomic events. More specifically, within the context of a sequential event trace, one cannot distinguish the intermediate points between concurrent interleavings from those of events recorded sequentially. Reasoning about computation with a sequential event trace leads to ambiguity for failed Linda predicate operations.

2.3 Lazy versus Eager Semantics

Lazy and eager (sometimes, *normal-order* and *applicative-order*, respectively) are terms typically used to describe the semantics of particular programming languages, especially in the context of when procedure arguments are evaluated. But, one may refer more generally to expression evaluation, and not be restricted solely to procedure arguments, when referring to lazy and eager evaluation. There are many facets of lazy and eager evaluation, but for the purposes of this paper, we will differentiate the respective meanings of lazy and eager by focusing on the temporal aspects of their definitions. For a more complete discussion of lazy and eager semantics, see Abelson and Sussman [1].

Briefly, lazy evaluation involves delaying the evaluation of an expression, e, until just before its result is needed to ensure continued computational progress. The opposite of lazy evaluation is eager evaluation, which requires the evaluation of e as soon as it is possible to do so. From a temporal perspective, the only difference between lazy and eager evaluation is *when* e gets evaluated.

Notice that e need not evaluate to some native value, and could, in general, be a data structure, such as a list or tree. In the case of potentially infinite data structures (e.g., a stream of the natural numbers), eager evaluation is not feasible, as it would lead to divergence. In such cases, lazy evaluation offers a viable alternative by supporting lazy data structures. For example, a lazy list could be represented by its first element, and the promise to evaluate the rest of the list, should more elements ever be needed. Now, from a temporal standpoint, the difference between lazy and eager evaluation might not only be when, but *if* e gets evaluated. The point is, the choice between lazy and eager semantics in expression evaluation potentially affects the meanings of expressions themselves.

3 A Tale of Two Solutions

This section presents two solutions to the Linda predicates ambiguity problem; one by Jacob and Wood [7], the other by Smith, *et al.* [9]. One thing both solutions have in common is their CSP foundation for reasoning about the meaning of the Linda predicate operations. These solutions are not new, and on the surface, the approaches they take appear orthogonal to each other. It is possible, however, to characterize one of these approaches as *lazy*, and the other, *eager*. Section 3.1 presents the lazy solution by Jacob and Wood. Section3.2 presents the eager solution by Smith, *et al.*.

3.1 A Lazy Solution

Jacob and Wood's goal to define "a principled semantics for inp" arises in part out of frustration that, despite citing seven sources giving "informal specifications" for Linda predicate operations, only one specification was deemed "useful"! Furthermore, the one useful specification, by Carriero and Gelernter [3], had little or nothing to say about the meaning of a failed predicate operation. We reproduce the quote here.

> If and only if it can be shown that, irrespective of relative process speeds, a matching tuple must have been added to tuple space before the execution of inp, and cannot have been withdrawn by any other process until the inp is complete, the predicate operations are *guaranteed* to find a matching tuple.

Jacob and Wood set out to define, unambiguously, what failure *should* mean for `inp()`. They reasoned that `inp()` should fail only in situations where it could be proved `in()` would never find a matching tuple. Such a situation occurs only in the presence of deadlock. In short, they changed the meaning of `inp()`, and issued the following warning: "`inp()` is *not* a 'non-blocking' version of `in()` — it will block, as `in()`, until a matching tuple is retrieved, *or* until deadlock is detected."

How is Jacob and Wood's new definition for `inp()` lazy? In the case of finding a matching tuple in tuple space, it's not. However, from a temporal perspective, `inp()` no longer returns an indication of failure right away. It waits. In fact, it could wait a very long time, i.e., block. So long as there is no matching tuple in tuple space, the only way this version of `inp()` ever returns a value is if deadlock is detected. Since `inp()` is capable of blocking, and only returns an indication of failure in the presence of deadlock – the last possible moment of computational progress – from a temporal perspective, we could characterize Jacob and Wood's `inp()` as lazy.

There are many more implications of this new version of `inp()`. For more information, see Jacob and Wood [7].

3.2 An Eager Solution

Working from Jensen's [8] operational semantics for Linda and Tuple Space, and his observation about failed predicates and interaction points in tuple space, Smith *et al.* [9] chose a different path to disambiguate the Linda predicate operations. Namely, since the source of the ambiguity was known to be the result of constructing a computation's history from the sequential interleaving of concurrent events (as specified by CSP), we sought to extend the CSP's notion of a trace (history).

Our extensions to CSP include unordered and ordered parallel events as primitives for constructing a computation's history, and multiple, possibly imperfect, views. Taken together, these extensions to CSP constitute important aspects of View-Centric Reasoning (VCR). VCR's parallel event traces permit disambiguating the original definitions of `inp()` and `rdp()`. Parallel events, by their nature, capture every existing process involved in an interaction point in tuple space. Thus, parallel events permit unambiguous interpretation of the meaning of failed Linda predicate operations. Rather than introduce new definitions for the Linda predicate operations, we extended the model used to reason about their meaning. For a more complete discussion, see Smith, *et al.* [**?**].

How is Gelernter's original definition of `inp()` eager? Simply put, from a temporal perspective, `inp()` always returns a value immediately, no matter whether the operation was a success or failure. Gelernter's `inp()` was intended to be a non-blocking version of `in()` (similarly for `rdp()` and `rd()`), and thus could always have been considered eager. What's new is that it no longer needs to be considered ambiguous in the case of failure.

4 Conclusions

We presented two previously unrelated solutions to the Linda predicates ambiguity problem, and characterized them in terms of lazy and eager semantics. The benefits of this characterization are threefold. First, we identified a unifying perspective from which to reason about the meaning of Linda predicate operations. Second, we have a semantic basis for classifying and comparing at least two seemingly disparate approaches to solving the problem of Linda predicate ambiguity. Third, in a broader sense, we are encouraged to revisit, through the lens of lazy versus eager semantics, issues of safety and liveness in parallel and distributed applications.

5 Acknowledgements

The eager solution to the Linda predicates ambiguity problem was developed in a previous collaboration between the authors and Dr. Rebecca J. Parsons. The insights into lazy and eager semantics, which permitted the revelations that led to this paper, were also developed during the same collaboration.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, second edition, 1996.

[2] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1986.

[3] N. Carriero and D. Gelernter. *How to Write Parallel Programs: a First Course*. MIT Press, 1990.

[4] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. The Jini Technology Series. Addison Wesley, 1999.

[5] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1), Jan. 1985.

[6] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice-Hall International, UK, Ltd., UK, 1985.

[7] J. L. Jacob and A. M. Wood. A principled semantics for inp. In A. Porto and G.-C. Roman, editors, *Coordination Languages and Models*, volume 1906 of *Lecture Notes in Computer Science*, pages 51–65, Berlin, Germany, 2000. Springer Verlag. Coordination 2000: Proceedings of 4th International Conference.

[8] K. K. Jensen. *Towards a Multiple Tuple Space Model*. PhD thesis, Aalborg University, Nov. 1994. http://www.cs.auc.dk/research/FS/teaching/PhD/mts.abstract.html.

[9] M. L. Smith, R. J. Parsons, and C. E. Hughes. View-centric reasoning for linda and tuple space computation. In J. S. Pascoe, P. H. Welch, R. J. Loader, and V. S. Sunderam, editors, *Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering Series*, pages 223–254, Amsterdam, 2002. IOS Press.

[10] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.